TESIS DOCTORAL

MODELO DE CALIDAD PARA EL *SOFTWARE* ORIENTADO A OBJETOS

presentada en la

FACULTAD DE INFORMÁTICA

de la

UNIVERSIDAD POLITÉCNICA DE MADRID

para la obtención del

GRADO DE DOCTOR EN INFORMÁTICA

Autor: José Luis Fuertes Castro

Licenciado en Informática por la

Universidad Politécnica de Madrid

Directores: César Montes Gracia

Fernando Alonso Amo

Madrid, enero de 2002

AGRADECIMIENTOS

Estoy muy agradecido a todos aquéllos que, de una u otra manera, me han ayudado y apoyado durante la larga preparación de este trabajo.

En primer lugar, quisiera comenzar por mis directores, Fernando y César. Fue su ánimo, perseverancia y capacidad de trabajo lo que me impulsaba a continuar, aunque nunca estaré a su altura. ¡Gracias! Ah, y disculpad por la cantidad de medidas que tuvisteis que digerir *en la dura labor de lectura y corrección de la memoria. Prometo no volver a hacerlo.*

En segundo lugar, a mis compañeros y amigos de la Facultad. Con muchos he compartido (y sufrido) momentos de trabajo en las respectivas tesis. Algunos la terminasteis antes que yo. Otros seguís trabajando en ella. Ánimo, que todo llega: lógicamente, lo sé. En cualquier caso, muchas gracias a todos, pues siempre me ayudasteis cuando lo necesité e, incluso, muchos me aportasteis interesantes sugerencias y comentarios: Ángel Lucas, Angélica, César, Chema, Jacinto, Javier, Loïc, Nacho, Paloma, Rafa, Sonia...

En tercer lugar, al resto de mis amigos. Por fin, he terminado y espero volver a frecuentar vuestra compañía como antiguamente. Muchos también me echasteis una mano cuando lo necesité. Gracias Alberto, Ángel, Antonio, Carlos, Esther, Katrin, Mar, Pascual, Susana...

En cuarto lugar, al resto de las personas que han colaborado en alguna parte de la tesis, por pequeña que sea. Gracias a Javier, José Luis, José Luis, José, Salva...

En último lugar, pero no por ello menos importante, a CETTICO. La tesis nació y creció ahí, gracias al apoyo de todos vosotros.

Seguro que se me olvida alguien (la memoria no es lo mío), pero para eso están los puntos suspensivos.

A todos, amigos, muchas gracias. Como dijo Fiessinder, "el número de los amigos está en relación inversa de su valor". Aunque sois pocos, estabais ahí cuando os necesité.

RESUMEN

El *software* ha obtenido en la actualidad una gran importancia en todos los ámbitos de la vida cotidiana. Es indudable que la calidad del *software* juega un papel fundamental en todo desarrollo informático, aunque en ocasiones no se le presta la suficiente atención, quizás debido a los relativamente escasos trabajos relacionados con este tema desarrollados hasta la fecha.

En el presente trabajo, se plantea la necesidad de un modelo de calidad completo. Para cubrir esta necesidad se presenta un nuevo **modelo de calidad**, obtenido tras un estudio pormenorizado de los modelos de calidad existentes, centrado en el paradigma orientado a objetos. Este modelo de calidad muestra cómo la calidad del *software* se descompone en una serie de factores y éstos, a su vez, se descomponen en un conjunto de criterios medibles utilizando medidas. El modelo incluye un amplio conjunto de **medidas**, diseñadas especialmente para su aplicación dentro del paradigma orientado a objetos. Para completar el modelo, se ha diseñado un sencillo **método de aplicación** de este modelo de calidad para que pueda ser utilizado de una forma simple por los desarrolladores de sistemas informáticos orientados a objetos.

El modelo de calidad definido se ha validado realizando un juego de experimentos. Estos experimentos han consistido en la aplicación del modelo sobre una serie de desarrollos orientados a objetos. Los resultados obtenidos han demostrado su utilidad práctica para determinar tanto la calidad global de los sistemas, como para identificar aquellas partes del sistema susceptibles de ser mejoradas.

Con este trabajo, se llena un importante hueco existente en esta área, pues, en primer lugar, no existen modelos de calidad completos para la orientación a objetos. En segundo lugar, aunque hay medidas para la orientación a objetos, no se han asociado a los atributos que determinan la calidad del *software*, por lo que su utilidad, tal cual fueron definidas, resulta bastante cuestionable. Para finalizar, nunca se ha asociado un modelo de calidad con una método de aplicación, por lo que su utilidad quedaba considerablemente mermada, quedando a expensas de la habilidad y experiencia del Ingeniero del *Software* que lo utilizara.

ABSTRACT

El *software* ha obtenido en la actualidad una gran importancia en todos los ámbitos de la vida cotidiana. Es indudable que la calidad del *software* juega un papel fundamental en todo desarrollo informático, aunque en ocasiones no se le presta la suficiente atención, quizás debido a los relativamente escasos trabajos relacionados con este tema desarrollados hasta la fecha.

En el presente trabajo, se plantea la necesidad de un modelo de calidad completo. Para cubrir esta necesidad se presenta un nuevo **quality model**, obtenido tras un estudio pormenorizado de los modelos de calidad existentes, centrado en el paradigma orientado a objetos. Este modelo de calidad muestra cómo la calidad del *software* se descompone en una serie de factores y éstos, a su vez, se descomponen en un conjunto de criterios medibles utilizando medidas. El modelo incluye un amplio conjunto de **measures**, diseñadas especialmente para su aplicación dentro del paradigma orientado a objetos. Para completar el modelo, se ha diseñado un sencillo **método de aplicación** de este modelo de calidad para que pueda ser utilizado de una forma simple por los desarrolladores de sistemas informáticos orientados a objetos.

El modelo de calidad definido se ha validado realizando un juego de experimentos. Estos experimentos han consistido en la aplicación del modelo sobre una serie de desarrollos orientados a objetos. Los resultados obtenidos han demostrado su utilidad práctica para determinar tanto la calidad global de los sistemas, como para identificar aquellas partes del sistema susceptibles de ser mejoradas.

Con este trabajo, se llena un importante hueco existente en esta área, pues, en primer lugar, no existen modelos de calidad completos para la orientación a objetos. En segundo lugar, aunque hay medidas para la orientación a objetos, no se han asociado a los atributos que determinan la calidad del *software*, por lo que su utilidad, tal cual fueron definidas, resulta bastante cuestionable. Para finalizar, nunca se ha asociado un modelo de calidad con una método de aplicación, por lo que su utilidad quedaba considerablemente mermada, quedando a expensas de la habilidad y experiencia del Ingeniero del *Software* que lo utilizara.

ÍNDICE

No me importa si algo es barato o caro.
Solamente me preocupa si es bueno.
Si es lo suficientemente bueno,
el público pagará dinero por ello.

— Walt Disney

("Walt Disney, An American Original",
Bob Thomas, Simon & Schuster, 1976)

1. INTRODUCCIÓN

1. INTRODUCCIÓN

Toda persona relacionada, en una u otra medida, con el mundo de la informática se habrá podido dar cuenta que el *software* no es perfecto y, en algunas ocasiones, posee defectos que pueden originar un funcionamiento inadecuado con un importante costo económico. Seguidamente, se relatan algunos "desastres informáticos" causados por defectos en los programas.

> En 1983, la compañía de seguros *Blue Cross & Blue Shield* de Wisconsin encargó un sistema informático de unos 200 millones de dólares. Estuvo listo en 18 meses... pero no funcionaba: por un error en la introducción de un dato, el ordenador envió cientos de cheques a la ficticia aldea de nombre *None*. Además, durante su primer año de vida, el sistema desembolsó 60 millones de dólares en pagos incorrectos o cheques duplicados. [Rothfeder, 88]
>
> En 1987, el *First Interstate Bank* de California informó que un inexplicable error informático causó un día de retraso en el procesamiento de transacciones por unos mil millones de dólares. El problema afectó a unos 3 millones de cheques y depósitos que fueron rechazados por el ordenador. Los empleados trabajaron toda la noche para corregir el problema y las transacciones fueron procesadas al día siguiente. [Neumann, 88]
>
> En 1987, la compañía de investigaciones médicas *International Data Corp.* sufrió una auténtica pesadilla cuando, durante el primer día de funcionamiento de su nuevo sistema informático, perdió información perteneciente a miles de pruebas médicas cruciales. El *software* no fue capaz de manejar el gran volumen de datos almacenados. Afortunadamente, la pérdida no fue permanente, pero llevó 30 días reconstruir todos los archivos. [Rothfeder, 88]
>
> En 1988, el *Australian Commonwealth Bank* proporcionó un respiro a muchos de sus clientes: les dobló su sueldo. Para empeorar más las cosas, el sistema informático procesaba cada transacción dos veces. El ordenador tuvo que ser controlado manualmente. [Neumann, 88]
>
> La nave espacial Galileo pasó alrededor de Venus en su camino hacia Júpiter. La cámara electrónica empezó a disparar fotografías descontroladamente siguiendo las instrucciones del *software*. La cámara tomó 16 fotos de Venus y entonces, durante cinco horas, mientras la nave completaba su primera maniobra, el obturador empezó a abrirse y cerrarse. 450 veces se abrió para observar el planeta y 450 veces se cerró rápidamente como si se hubiera ofendido por la vista contemplada. Esa misma tarde, los ingenieros de la NASA, en el *Jet Propulsion Laboratory* de Pasadena, corrigieron el *software* y la máquina tomó 38 preciosas fotografías más de Venus. [Arthur, 93]

Seguro que la mayoría de estos relatos habrán interesado al lector, que también disfrutará con los que abren los capítulos 2, 3 y 7. Incluso, alguno de estos desastres informáticos, habrá hecho asomar en su rostro una leve sonrisa, si bien otros le habrán producido cierta preocupación, puesto que hay que tener en cuenta que todos ellos son completamente verídicos, salvo, evidentemente, uno de ellos (el lector podrá detectarlo sin problemas), que pertenece a una ciencia-ficción no demasiado alejada de la realidad. Afortunadamente, los errores informáticos habituales no suelen resultar tan catastróficos como los que aquí se han presentado a modo ilustrativo, aunque las causas de un fallo pueden ser de lo más diversas [Pfleeger, 98]: especificaciones incorrectas o incompletas, un análisis equivocado, un diseño con fallos o un código con errores.

Los problemas que puede plantear el *software* están ahí, agazapados, esperando solamente a que entre en acción la ley de Murphy ("Si algo puede ir mal, irá mal" [Bloch, 77]) y algo inexplicable ocurra. Todo el mundo puede sentir el problema, pero no lo pueden definir. Muchos Ingenieros del *Software* sugieren que hay una crisis, pero no saben cómo resolverla. "–¡El problema es la calidad! –suelen exclamar. Tonterías, la calidad es la solución al problema" [Arthur, 93]. Evidentemente, el desarrollo con un cierto nivel de calidad puede ayudar a evitar esta aclamada crisis del *software*.

Muchos sistemas no han podido ser finalizados por los riesgos asumidos, y éstos podían haber sido evitados si se hubieran encontrado mejores formas para llamar la atención de los desarrolladores acerca de la existencia de dichos riesgos [Boehm, 89]. El general George S. Patton dijo en una ocasión: "Asumid riesgos calculados. Lo cual no significa que haya que ser un temerario". Como sugiere la cita, asumir un riesgo no es necesariamente malo. De hecho, en el combate es beneficioso, suponiendo que la persona que asume el riesgo reconozca lo que está haciendo y que planifique por adelantado los posibles problemas que pudieran surgir. El secreto estriba en mantener a Murphy y sus *gremlins* bajo control [Edgar, 82].

Durante las pasadas décadas, los conceptos relacionados con el desarrollo del *software* sufrieron una importante revolución. En los años 70 se comenzó la transición de una actividad artística, poco entendida y anárquica hacia una disciplina cuidadosamente controlada, metódica y predecible. El vehículo para este progreso fue la creciente popularidad de la Ingeniería del *Software*, la cual causó que un campo, que comenzó con pocas o ninguna técnica de diseño, con programas sin estructurar y poco fiables, evolucionara hasta la creación de un amplio número de técnicas que permiten mejorar y controlar los resultados obtenidos durante el desarrollo. Como ejemplo, puede citarse la transformación que sufrieron los mecanismos de abstracción conforme se mejoraba la comprensión de las diferentes técnicas de construcción de programas, según [Shaw, 84]. Uno de los avances más significativos en éstas técnicas lo supuso la aparición de los subprogramas y de los paradigmas imperativo y funcional, aunque posteriormente se comprendió su limitada adecuación para proporcionar a los programadores las abstracciones necesarias para manejar los problemas de un tamaño en continuo crecimiento, como afirma [Rising, 94]. El soporte necesario en los paradigmas y lenguajes de programación para el gradual aumento de los niveles de abstracción aparecieron en primer lugar en las construcciones de Simula [Dahl, 70] y estructuras similares se encontraron posteriormente en los paquetes de Ada [ANSI, 83] y las clases de Smalltalk [Goldberg, 83] y C++ [Stroustrup, 86], derivando en el paradigma orientado a objetos.

El término calidad es un concepto compuesto; hay muchas facetas de la calidad, muchos temas diferentes que tienen que ver con ella. Algunos son obvios y sencillos; otros son sutiles y complicados. Unos son comúnmente aceptados; otros son simplemente comprendidos [Burrill, 80]. El concepto de **calidad** del *software* se refiere, principalmente, al hecho de **especificar, analizar, diseñar e implementar *software* que cumpla plenamente los requerimientos del usuario, sin errores y con un funcionamiento adecuado**. Ello involucra a todas las etapas del ciclo de vida y, por tanto, tiene relación con los diversos métodos y herramientas que pueden utilizarse para alcanzar una buena calidad [Smith, 89].

Uno de los principales objetivos de la Ingeniería del *Software* es mejorar la calidad de los productos informáticos. Para tal fin se han ideado diferentes técnicas y métodos que intentan cumplir dicho objetivo. No obstante, hay que tener en cuenta que la historia del *software* no es tan extensa como la de otros productos industriales y, por tanto, las técnicas de control de la calidad no se encuentran todavía bien establecidas, al contrario de lo que ocurre con otras técnicas de gestión del *software*. Incluso, puede afirmarse que, hasta no hace mucho, no existían técnicas probadas que mostraran cuantitativamente que un programa era mejor que otro funcionalmente equivalente [Henry, 90a].

Últimamente se han dedicado muchos esfuerzos para intentar reducir el coste del *software*. Sin embargo, no se han destinado tantos recursos para intentar gestionar y aumentar la calidad de los programas. Incluso hoy en día, la mayor parte del *software* tiene a menudo un número significativo de defectos [Jones, 01]. Con la creciente complejidad del *software* desarrollado se puede comprender la necesidad de controlar la calidad de los sistemas, siendo necesario saber cómo especificarla y medirla, de tal forma que se pueda asegurar que el sistema cumple con todos sus objetivos. Para ello, **resulta necesaria la incorporación de un conjunto de medidas del *software*** que permitan recopilar información útil acerca de los programas. Uno de los roles de las medidas consiste en ser capaces de determinar si se ha alcanzado la calidad [Tian, 95]. La importancia de las medidas para proporcionar una ingeniería más disciplinada capaz de especificar, predecir y evaluar la calidad del *software* está ampliamente reconocida [Bologna, 88].

Las medidas del *software* son esenciales para comprender, gestionar y controlar el proceso de desarrollo. Una caracterización cuantitativa de varios aspectos de este proceso incluye una profunda comprensión de las actividades de desarrollo de *software* y sus interrelaciones [Pfleeger, 92]. Las medidas permiten a los ingenieros cuantificar la fiabilidad y el rendimiento, aislando los atributos del proceso y del producto que influyen sobre éstos y, por tanto, demostrar cómo influyen los cambios del proceso y del producto en estos atributos [Khoshgoftaar, 94b].

Existen muchos usos posibles para las medidas. Pueden usarse como retroalimentación a los programadores, como guía en las pruebas o en la estimación de requisitos de mantenimiento [Curtis, 79b]. Pero, en términos generales, se pueden emplear para obtener una información cuantitativa y reproducible del desarrollo realizado [Liggesmeyer, 95]. La comunidad científica ha realizado gran cantidad de trabajo en el terreno de las métricas, siguiendo diversas líneas de desarrollo. Se han publicado gran número de medidas, principalmente para los paradigmas imperativo y orientado a objetos, aunque rara es la ocasión en la que queda establecida su utilidad en la práctica. En cualquier caso, y debido principalmente a la amplia y diversa información que se podría extraer de ellas, diversos autores [Buckley, 89] [Zuse, 89] han planteado la necesidad de obtener un completo conjunto de medidas del *software* **útiles**. Cuando se presenta una medida, debe indicarse qué mide, para qué sirve, su significado y precisar una relación clara con el atributo del *software* que mida.

Esta necesidad resulta aún más palpable al considerar el paradigma orientado a objetos, que ha introducido nuevas características, no disponibles en su conjunto en los paradigmas tradicionales, como el concepto de clase, la herencia y el polimorfismo

paramétrico, dinámico y de sobrecarga [Lejter, 92]. La creciente popularización de la orientación a objetos en el desarrollo de sistemas ha infundido numerosas ventajas y beneficios, amén de inspirar nuevas áreas de investigación. Una de estas áreas, que representa un interés fundamental para el futuro de este paradigma, la constituye el problema de cómo determinar la calidad de un sistema, además de cómo medirla y mejorarla [Gillibrand, 98]. Este problema no resulta en absoluto trivial, pues involucra un buen conocimiento de la tecnología orientada a objetos, así como de los atributos que influyen en la calidad del *software*. A pesar de la imperiosa necesidad para resolver este asunto, en comparación con los paradigmas clásicos, se ha realizado muy poco trabajo en el tema de la gestión del desarrollo de *software* orientado a objetos. Una de las claves en dicha gestión estriba en la **habilidad de medir y registrar los atributos relevantes** de los productos y procesos *software* de una forma coherente y estructurada. La disponibilidad de las medidas ayuda a los gestores a controlar las actividades del ciclo de vida, contribuyendo al objetivo general de la calidad del *software*. No obstante, hasta la fecha, no se han publicado suficientes medidas útiles y apropiadas para la orientación a objetos. De igual forma, tampoco se ha publicado todavía ningún modelo de calidad completo para este paradigma. Además, para lograr el fin buscado, es necesario disponer de datos y experiencia de los proyectos, además de obtener una serie de medidas adecuadas tanto para el producto como para el proceso [Roberts, 93].

Hay muchas características del producto o del proceso que representan un importante papel para determinar la calidad [Evanco, 94]. Los resultados obtenidos con un conjunto de medidas han demostrado ser superiores a los resultados obtenidos con cada una de esas medidas por separado [Zage, 93]. Por ello, se deben intentar identificar los atributos fundamentales que influyen en la calidad del *software*, así como la forma de medirlos adecuadamente. Un aspecto primordial, que también debe tenerse en cuenta en cualquier intento de construir un modelo de calidad, es que el *software* no manifiesta directamente sus atributos de calidad, es decir, no son medibles de forma inmediata. En su lugar, exhibe características que contribuyen a dichos atributos de calidad (los favorecen), así como otras características que influyen negativamente en ellos (los penalizan), siendo todas estas características medibles más o menos fácilmente de forma directa. Por ello, los modelos de calidad suelen dividir ésta en atributos que, a su vez, se descomponen hasta llegar a una serie de características para las que puede definirse algún tipo de medida.

No obstante todo lo dicho, un modelo de calidad por sí solo tampoco tiene mucho sentido. Es así mismo necesario diseñar un **método para su correcta aplicación**, de tal forma que el estudio detenido de los resultados obtenidos permita mejorar el proceso *software* tanto personal como corporativo [Alonso, 98a].

En el trabajo que se presenta en los sucesivos capítulos se intentará describir cómo se da una solución a estos temas, describiéndose un **nuevo modelo de calidad**, obtenido tras un estudio pormenorizado de los modelos de calidad existentes, junto con un amplio **conjunto de medidas**, diseñadas especialmente para su aplicación dentro del paradigma orientado a objetos, y, para finalizar, un **método de aplicación** de este modelo de calidad a los desarrollos de sistemas informáticos orientados a objetos. De esta forma, se viene a llenar un hueco importante existente en esta área, pues puede afirmarse que, en primer lugar, no existen modelos de calidad completos para la

orientación a objetos. En segundo lugar, aunque sí se han publicado medidas de aplicación en la orientación a objetos, éstas no se han asociado a los atributos que determinan la calidad del *software*, por lo que su utilidad, tal cual fueron definidas, resulta bastante cuestionable. Y por último, nunca se ha asociado un modelo de calidad con un método de aplicación, por lo que su utilidad quedaba considerablemente mermada, dependiendo de la habilidad y experiencia del Ingeniero del *Software* que lo utilizara.

El trabajo se completa con una validación empírica del modelo de calidad mediante una serie de casos de prueba. Estos casos consisten, principalmente, en la aplicación del modelo de calidad sobre diversos programas desarrollados empleando la tecnología orientada a objetos e implementados con los lenguajes de programación C++, Object Pascal y Java.

A las 14:21 de un 15 de enero, hora punta en el tráfico telefónico, el conmutador 4ESS de la red telefónica de Manhattan detectó un pequeño fallo en su hardware. *El 4ESS se desconectó de la red tras, amablemente, notificar a los conmutadores vecinos su desconexión. En pocos segundos, el conmutador volvió a funcionar, lo que notificó también a sus vecinos. Pero éstos estaban todavía procesando el primer mensaje cuando recibieron el segundo. Esto les confundió, se dieron cuenta de este desconcierto y se desconectaron de la red para realizar un auto-chequeo y reinicializarse, no sin antes notificar a todos sus vecinos su desconexión. Como una epidemia de gripe, los mensajes se distribuyeron por todo Estados Unidos. Tan pronto como un conmutador se desconectaba y se recuperaba, recibía un aluvión de mensajes de sus compañeros que le hacían volver a fallar. En el centro de operaciones, los técnicos veían cómo las líneas de comunicaciones del mapa se ponían rojas por todo el país partiendo de Manhattan, mientras que los ingenieros se veían* negros *siguiendo todos los procedimientos estándar para la solución de problemas, los no estándar y los improvisados. Pero nada funcionó. Un minúsculo error en la versión de diciembre del* software *del conmutador 4ESS causó el caos. AT&T volvió temporalmente a la versión anterior e instaló la versión corregida cinco días después. Pero el daño ya estaba hecho.*

[Arthur, 93]

2. ESTADO DE LA CUESTIÓN

2. ESTADO DE LA CUESTIÓN

En el presente capítulo se comentará el estado de la cuestión relacionado con los temas tratados en este trabajo, y se estructurará como sigue: tras una introducción donde se comentarán algunos aspectos generales y definiciones de la calidad del *software*, se expondrán los principales modelos de calidad diseñados hasta la fecha, que han servido como punto de partida para diseñar el que se presenta en este trabajo. Como todo modelo de calidad debe poderse evaluar cuantitativamente, seguidamente, se presentan algunos ejemplos de medidas para los paradigmas tradicionales y orientado a objetos y unas clasificaciones de medidas. Finalmente, se describen algunos aspectos relacionados con la mejora del proceso *software*, asunto también de vital importancia para su integración con el modelo de calidad.

2.1. ASPECTOS DE LA CALIDAD DEL *SOFTWARE*

En la actualidad, puede afirmarse que todavía existe un problema real a la hora de producir *software* de calidad. A menudo, el *software* no verifica las expectativas de calidad previstas. En gran número de ocasiones, se entrega con retraso, incompleto, con un coste superior al previsto, no cumple todos los requisitos de uso y contiene errores. La mayor parte de las metodologías de análisis y diseño se centran en la especificación de requerimientos funcionales. Desafortunadamente, las especificaciones sobre la calidad son a menudo demasiado imprecisas, subjetivas y difíciles de medir [Barnes, 93]. La comunidad de ingenieros del *software* tiene que aprender a definir claramente los objetivos de calidad en términos cuantificables. Muchos proyectos no fracasan por no conseguir los requisitos funcionales, sino por que no se alcanzan dichos requisitos con un nivel adecuado [Gilb, 88].

Con la creciente complejidad de los sistemas *software* que se realizan en la actualidad y la necesidad de desarrollarlos en un corto espacio de tiempo, se está enfatizando la importancia de disponer de un programa de gestión de calidad. En dicho programa, resulta esencial saber cómo especificar y medir la calidad del *software* de tal forma que se puedan asegurar los objetivos del sistema durante todo su ciclo de vida [Cooper, 79].

Existen múltiples razones para explicar por qué el concepto de calidad del *software* está adquiriendo tanta importancia y, básicamente, casi todos tienen un trasfondo económico [González, 93]:

- Por una parte y de forma esencial está el cliente, que paga y usa el *software* y quiere los mejores resultados de su inversión. El cliente, cada día que pasa, va conociendo más no sólo su propio negocio sino la herramienta que utiliza, esto es, el *software*. En consecuencia, crecen sus exigencias en sus requisitos de calidad y afina más en sus contratos.
- Por otra parte, está el desarrollador. Cada vez es menos cierto la máxima que viene a decir que si el cliente no pide o exige actividades de calidad, el suministrador no las realiza, ya que se ahorra el coste de las mismas. Precisamente, uno de los objetivos perseguidos por dichas actividades está directamente relacionado con la disminución de los costes de desarrollo del *software*. Basta pensar en el impacto económico que

tienen las revisiones formales para prevenir los graves problemas que se presentan cuando existe una inadecuada definición de funciones o de interfaces, etc. durante el proceso de diseño. Hay estudios que indican que la relación de costes para corregir un error durante las fases de especificación y análisis es de 10 a 20 veces mayor que si se hace durante las pruebas del sistema, y si se realiza durante la explotación, la relación llega a ser 100 veces superior. Es evidente que si mejoran los costes de desarrollo, además de ofrecer al cliente una mayor confianza en la consecución de la calidad requerida, se le pueden ofrecer mejores precios que la competencia. Por tanto, el efecto de la mejora de la calidad de los procesos de desarrollo *software* constituye un mejor posicionamiento en el mercado a través de una clientela más satisfecha.

Ni que decir tiene que estos conceptos se están aplicando con notable éxito y desde hace ya bastantes años en diversos tipos de industrias, como la farmacéutica, automoción, electrónica, etc. Lo que resulta diferente son algunos métodos específicos de la ingeniería y de la calidad de la industria del *software*, dada la distinta naturaleza de sus productos (productos físicos frente a productos intangibles como son los sistemas informáticos). Por ello, no deben reutilizarse los modos y modelos de actuación de la industria del *hardware* en la del *software* (basta pensar en el poco éxito de la aplicación de los modelos de fiabilidad del *hardware* al *software* [Littlewood, 78] [Cooper, 79] [Buckley, 89]).

Por tanto, el estudio que se va a realizar se va a centrar en la definición de la calidad del *software*, sin entrar en detalles propios del *hardware* ni en técnicas provenientes de otras industrias cuya validez no haya sido probada dentro de esta área.

Pero un problema con la calidad, no restringido a la industria del *software*, lo constituye el propio hecho de definir qué es la calidad.

El Diccionario de la Real Academia Española define calidad como: "Propiedad o conjunto de propiedades inherentes a una cosa, que permiten apreciarla como igual, mejor o peor que las restantes de su especie" [RAE, 92], mientras que la definición de calidad aprobada por la Organización Internacional para la Estandarización es: "Totalidad de propiedades y características de un producto o de un servicio que conlleva su capacidad para satisfacer las necesidades especificadas o implícitas" [ISO, 86] [ISO, 94].

Sin embargo, éstas no pueden considerarse como definiciones útiles, puesto que la primera de ellas es demasiado general y en la segunda se quedan sin resolver ciertos aspectos:

- Las 'necesidades' pueden incluir características no funcionales como 'accesibilidad', 'facilidad de uso', 'mantenibilidad', etc.
- Muchas 'necesidades' se encuentran implícitas, pero para evaluar la calidad de todas las necesidades deben, de alguna forma, ser identificadas y definidas.
- Las 'necesidades' pueden cambiar con el tiempo, como también pueden cambiar las características y el proceso de desarrollo del producto.

En general, las 'necesidades' dependen del cliente (o del usuario), y las 'necesidades' de un cliente pueden ser irrelevantes o, incluso, entrar en conflicto con las de otro cliente. En la industria del *software*, este conflicto suele encontrarse en el *software* multi-objetivo, donde las necesidades que pueda tener un cliente para que su sistema tenga una alta fiabilidad pueden entrar en conflicto con otro que requiera un alto rendimiento. [Kitchenham, 89a]

Por ello, el concepto de calidad resulta complejo y significa distintas cosas para distintas personas. Pueden identificarse cinco puntos de vista diferentes de la calidad [Garvin, 84]:

1. Desde el punto de vista **subyacente**, se equipara calidad con 'excelencia innata'. En este sentido, la calidad no puede definirse con precisión; más que algo medible es una sensación. Como ocurre con la apreciación del arte o de la música, la apreciación de la calidad proviene de la experiencia personal.
2. Desde el punto de vista del **producto**, surge de la economía y está relacionada con los contenidos de un producto. Cuantos más ingredientes o atributos, mejor es su calidad. Esto implica que las altas calidades sólo pueden obtenerse con un alto coste y, como la calidad se determina por características medibles, puede calcularse objetivamente. Este punto de vista tiene relevancia en la industria alimenticia, donde los consumidores saben que deben pagar más por productos con mayor calidad.
3. Desde el punto de vista del **usuario**, puede resumirse en la expresión 'adecuación para el uso' [Juran, 74]. Este punto de vista requiere que exista una lista definitiva de necesidades, que será difícil de obtener si hay muchos usuarios potenciales. Además, requiere que haya una clara distinción entre atributos que contribuyen a las necesidades del usuario y aquellos atributos no imprescindibles pero que favorecen su satisfacción.
4. Desde el punto de vista **industrial**, se resume en la frase 'conforme a los requisitos'[1]. La excelencia de un producto se obtiene cuando se alcanzan los requerimientos al primer intento. Un objetivo primordial es reducir los costes de mantenimiento y reparación, mientras que otro importante objetivo lo constituye la reutilización de la tecnología empleada. De esta forma, significa que el producto debe adecuarse a un conjunto predefinido de requisitos funcionales [Blundell, 97].
5. Desde el punto de vista del **valor del producto**, puede entenderse como la composición de los dos puntos de vista previos. Se define la calidad como la capacidad de proporcionar lo que quiere el cliente a un precio aceptable. Esto conduce a la idea de 'diseño al coste', reflejando el punto de vista del fabricante que necesita llegar a un compromiso entre las necesidades de minimizar los costes de producción y finalizar a tiempo el producto, pero que también ve como una exigencia importante del usuario que el producto resulte económico y, por tanto, competitivo.

Como en otras industrias, en la industria del *software* los puntos de vista más apropiados son la 'adecuación para el uso' y 'conforme a los requisitos'[2]. Las compañías que prefieren este último, tienden a considerar la calidad como una función del coste del trabajo y se concentran en procedimientos destinados a prevenir la introducción de defectos y la detección anticipada de cualquier error introducido (cuanto antes se introduce un error, mayores son los costes para corregirlo [Ward, 91]). Por el contrario, las compañías de *software* que prefieren el primer punto de vista tienden a considerar la calidad como una función de las propiedades finales del producto (que debe incluir

[1] Así define el Departamento de Defensa de EE.UU. el concepto de calidad: "capacidad del producto *software* para satisfacer los requisitos establecidos" [DoD, 88].

[2] Concuerda con la definición de calidad de IEEE, que la define como "grado con el cual el cliente o usuario percibe que el *software* satisface sus expectativas" [IEEE, 90].

todas aquellas características del producto reconocibles como beneficiosas para el usuario [Card, 90]) y utilizan para definir la calidad alguno de los modelos de calidad existentes [Kitchenham, 89a] (como los que se explican en los siguientes apartados).

Desde el punto de vista empresarial, existen diversas razones por las que la calidad resulta un área de preocupación [Sanders, 94]:

- La calidad se ha convertido en un aspecto competitivo. Hasta hace poco, el *software* solía considerarse como un negocio puramente técnico. Hoy ya no se puede confiar únicamente en las funcionalidades de los productos para competir en el vertiginoso campo del *software*. La única forma de diferenciar un producto de otro es por su calidad.
- La calidad es esencial para sobrevivir. Los clientes están demandando calidad demostrable en los productos que adquieren. Si una empresa no la puede proporcionar, sus posibilidades de sobrevivir en un mercado altamente competitivo y cambiante son escasas.
- La calidad es esencial para intervenir en el mercado internacional. El mercado del *software* se ha convertido en un mercado global. La habilidad para demostrar la calidad proporciona credibilidad, incluso a un pequeño negocio, para penetrar en el mercado internacional, repleto de diferentes estándares y certificaciones de calidad.
- La calidad ahorra dinero. Un sistema de calidad proporciona un incremento en la productividad y una reducción permanente en los costos, pues permite reducir los esfuerzos para la corrección de defectos enfatizando la prevención.
- La calidad retiene a los clientes e incrementa los beneficios. La baja calidad a menudo cuesta clientes. Una mejor calidad conduce a la satisfacción de la clientela, aumentando de año en año la cartera de clientes y, por tanto, los beneficios de la empresa.
- La calidad es la pieza clave de los negocios mundiales. El mundo de los negocios de primera clase realiza un énfasis estratégico en la calidad. Esto les permite crecer y sobrepasar a sus competidores.

En resumen, en la calidad influye un compendio de muchos aspectos diferentes, que dependen en gran medida de la persona que los utilice. Por ello, resulta sumamente complicado encontrar una forma objetiva de medir la calidad de un producto que resulte adecuado para todas las situaciones y sea aceptado por todo el mundo.

2.2. MODELOS DE CALIDAD DEL *SOFTWARE*

Conforme la industria del *software* está alcanzando la madurez, se encuentran diversas deficiencias en la producción del *software*. Y el resultado ha sido el desarrollo de sistemas que han tenido un alto coste de mantenimiento o que no han cumplido las necesidades reales de los usuarios.

Todo ello ha llevado a la comunidad científica a buscar un medio de mejorar la calidad del *software* y una forma adecuada para medirla. En ocasiones, hay razones legales para proporcionar un *software* de calidad, particularmente cuando están en juego grandes cantidades de dinero o, incluso, la vida de personas. En otras ocasiones, dar satisfacción a los usuarios del producto [Dekkers, 01] es, simplemente, una cuestión de orgullo propio e imagen de la empresa.

Principalmente, existen dos motivos para querer medir la calidad del software [Watts, 87]:

- En primer lugar, predictivos, en el sentido de que el desarrollador tiene que poder saber si lo que está haciendo va a llevar a cabo correctamente la función prevista de acuerdo a las especificaciones.
- En segundo lugar, una vez que se ha finalizado el *software*, los usuarios querrán saber si cumple con sus requisitos y, en el caso de que existan varios sistemas que resuelven sus problemas, ser capaces de decidir cuál de ellos deberían utilizar.

Desde que empezó a hablarse de la crisis del *software*, se han venido creando diversas técnicas para convertir el desarrollo de *software* en una ingeniería. Sin embargo, hasta la fecha no se han desarrollado totalmente las habilidades necesarias para construir sistemas *software* de alta calidad. Algunas de estas razones son:

- No se ha definido completamente el concepto de 'calidad alta' y se ve de distinta forma desde el punto de vista del desarrollador, del cliente o del usuario.
- Cuando se define el concepto de 'calidad', las definiciones no tienen un significado preciso para un ingeniero.
- Las tareas actuales relacionadas con la calidad se suelen centrar en trabajos de revisión, lo cual no tiene demasiado sentido para un ingeniero.
- La calidad de los grandes sistemas puede llegar a ser algo difuso y poco manejable. Además, el diseño de estos sistemas reales rara vez es óptimo, pues ha de existir un compromiso entre las distintas características que influyen en la calidad [Govindaraj, 98].

Por todo ello, hasta que no se disponga de un modelo de calidad del *software* adecuado, no se avanzará significativamente en dicho terreno [Dromey, 95].

Como se verá posteriormente, puede definirse la tarea de medir como el proceso por el cual se asignan números (o, en general, símbolos con una relación de orden) a atributos de una entidad del mundo real para describirla de acuerdo a reglas claramente definidas [Roberts, 79a] [Finkelstein, 84]. Para esta definición, una entidad puede ser un objeto, un concepto o un evento y un atributo es una característica o propiedad de una entidad. La asignación de estos números debe preservar cualquier observación intuitiva o empírica sobre los atributos y las entidades.

Así, por ejemplo, al medir el peso de una persona, se deben asignar números mayores a las personas más pesadas, aunque estos números sean diferentes al utilizar gramos, libras o arrobas. En muchas situaciones, un atributo puede tener un significado intuitivo distinto para diferentes personas. La forma habitual de resolver este problema es definir un modelo para las entidades que se están midiendo, que reflejará un punto de vista específico. Así, en el caso del modelo del peso de una persona, podría especificar si se le pesa con ropa o no, o si se realiza la medida antes o después de alguna de las comidas del día. Una vez que se ha fijado el modelo, existe un consenso en la relación existente entre los humanos con respecto a su peso. La necesidad de buenos modelos es particularmente relevante en las medidas empleadas en la Ingeniería del *Software* [Fenton, 94a]. Por ejemplo, incluso en algo aparentemente tan sencillo como la medida de la longitud de un programa basándose en las líneas de

código (LOC) [Conte, 86], se requiere un modelo correctamente definido que permita identificar las líneas de código sin ambigüedad (en un experimento [Ford, 93] con 80 personas, se les proporcionó un pequeño programa en lenguaje C (Figura 2.1) para que contaran las líneas de código y hubo hasta nueve respuestas distintas (Figura 2.2), ¡y el programa tan solo tenía 10 líneas físicas!).

```
#define LOWER   0    /* límite inferior de la tabla */
#define UPPER 300    /* límite superior */
#define STEP   20    /* tamaño del paso */

main ()   /* Imprime una tabla de conversión de Fahrenheit a Celsius */
{
   int fahr;
   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
       printf ("%4d %6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32));
}
```

Figura 2.1: Ejemplo de un programa en C para contar sus líneas de código

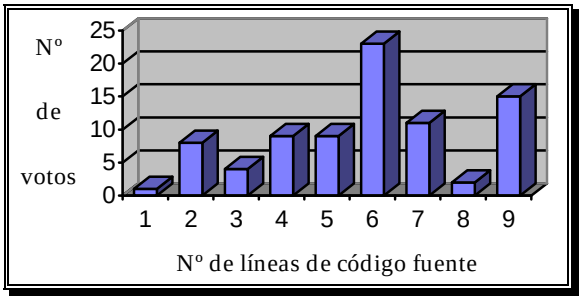


Figura 2.2: Resultados obtenidos en el experimento de la Figura 2.1

Según [Turban, 94] [Schuyler, 96], un modelo es una abstracción o simplificación de la realidad formada por variables y fórmulas matemáticas, que debería contener [Shepperd, 94]:

- una relación entre el mundo real y el modelo, permitiendo su uso en la práctica,
- una definición adecuada de los datos de entrada y salida,
- relaciones entre las entradas y las salidas,
- suposiciones o restricciones acerca del dominio que representan.

Los trabajos realizados en los modelos de calidad del *software* se remontan al artículo de Rubey [Rubey, 68], en el que por primera vez se presentó el concepto de métrica del *software*, y donde se definían algunos atributos del código junto con sus métricas. Los orígenes de este campo incluyen la lista de atributos identificados en [Wulf, 73], así como los estudios para establecer un marco de trabajo conceptual [Kosarajo, 74] [Boehm, 78]. Desde los primeros modelos, como el de Boehm [Boehm, 76] y el de McCall [McCall, 77] [Cavano, 78] (que han sido, además, los de mayor renombre), se intentó cuantificar la calidad del *software* descomponiendo la calidad en una serie de niveles estructurados, denominados, en general, factores, sub-factores y métricas [IEEE, 93]. Lo que diferencia a los distintos modelos es el grafo de calidad (que representa la relación existente entre estos niveles) y, en ocasiones, la cantidad de niveles en los que se descompone la calidad [Hausen, 89].

Normalmente, el primer nivel de un modelo suele contener una serie de atributos externos (aquéllos que sólo pueden cuantificarse por su relación con otras entidades de su entorno) mientras que los niveles intermedios contienen una serie de atributos internos (aquéllos que pueden medirse directamente, dependiendo únicamente del propio producto *software*) [Fenton, 94a]. Se considera que los atributos internos son la clave para mejorar la calidad del *software*, pues los métodos modernos de Ingeniería del *Software* se concentran en éstos, pudiéndose utilizar para el control y mantenimiento de la calidad; además, algunos pueden calcularse en los primeros momentos del desarrollo del sistema y resultan cruciales para evaluar la eficacia de los métodos del *software* [Fenton, 91]. Por último, es necesario medir los atributos internos porque son la base para la medida de los atributos externos [Tian, 92].

2.2.1. MODELOS DE CALIDAD COMPLETOS

Para los paradigmas tradicionales, durante el último cuarto de siglo se han publicado distintos modelos de calidad, estando las principales diferencias en el grafo que los representa.

En los siguientes apartados se presentan algunos de los modelos de calidad más significativos existentes en la actualidad, ordenados cronológicamente. Hay que destacar que el modelo que más relevancia ha tenido hasta el presente, y así se refleja en la literatura relacionada con el tema, ha sido el modelo de McCall, en el que se basan muchos de los modelos publicados.

2.2.1.1. Modelo de Boehm

B. W. Boehm diseñó un árbol que representaba las características de la calidad del *software* (Figura 2.3) cuyo primer nivel reflejaba los usos sobre los que se podía basar la evaluación del *software* [Boehm, 76] [Boehm, 78]. El nivel inferior proporcionaba un conjunto de características primitivas completamente independientes entre sí. Estas primitivas se combinaban en conjuntos de condiciones necesarias que conformaban las características del nivel intermedio. Las características primitivas se podían medir utilizando una serie de medidas.

Se resumen a continuación las explicaciones dadas por Boehm a las características de la calidad de este modelo que debe verificar el código [Boehm, 76].

- Accesibilidad: facilidad en el uso selectivo de sus partes.
- Aumentabilidad: el código puede acoger fácilmente una ampliación de los requisitos de funcionalidad o de almacenamiento de datos.
- Auto-contenido: efectúa todas sus funciones implícitas y explícitas sin requerir nada del exterior.
- Auto-descriptivo: contiene suficiente información para que un lector determine o verifique sus objetivos, suposiciones, restricciones, entradas, salidas, componentes y estado.
- Completitud: se encuentran presentes todas sus partes y cada una está desarrollada totalmente.
- Comprensibilidad: su propósito resulta claro para quien lo inspeccione.

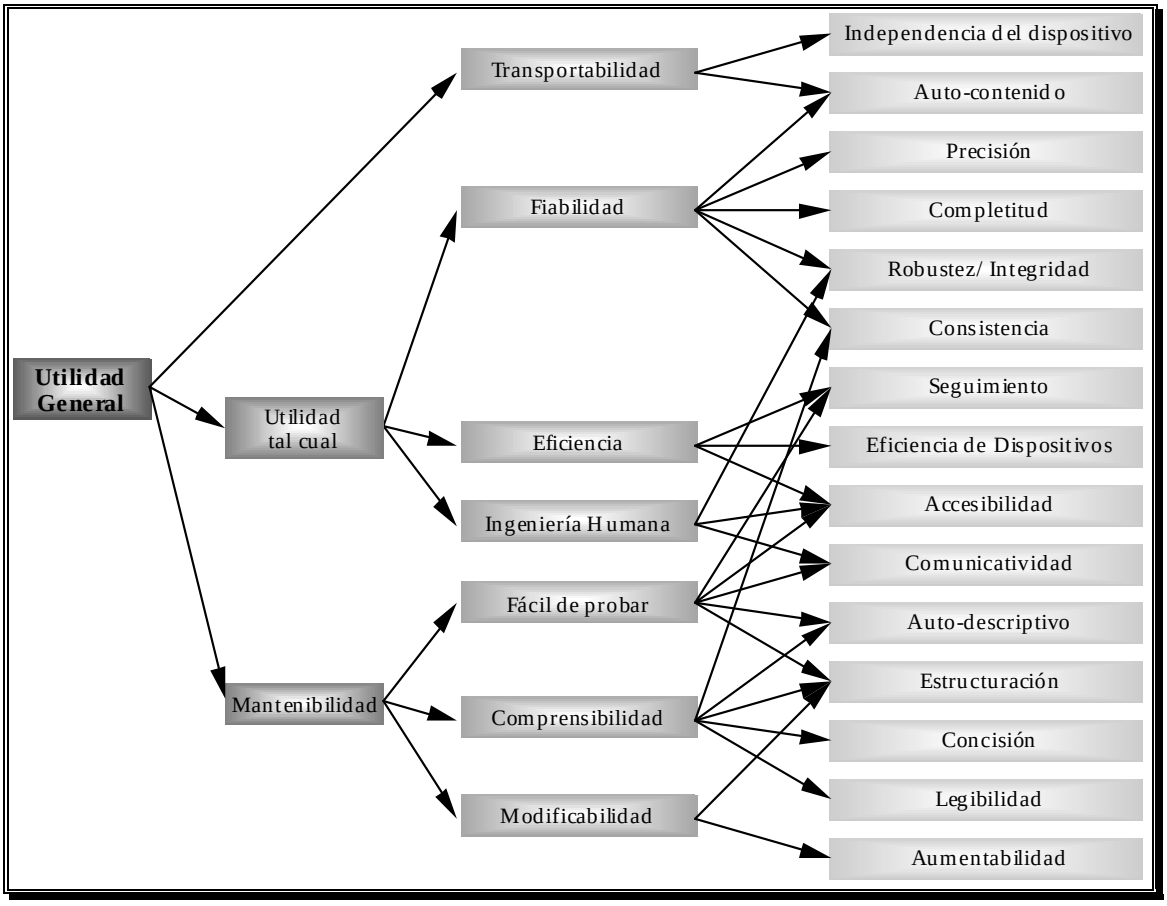


Figura 2.3: Modelo de Calidad de Boehm

- Comunicatividad: facilita la especificación de las entradas y proporciona salidas cuya forma y contenidos son útiles y fáciles de asimilar.
- Concisión: no hay un exceso de información.
- Consistencia: el código presenta consistencia interna si contiene una notación, terminología y simbología uniforme, y presenta consistencia externa si su contenido concuerda con los requerimientos.
- Eficiencia de dispositivos: los dispositivos se utilizan eficientemente.
- Eficiencia: cumple todos sus objetivos sin desperdicio de recursos.
- Estructuración: posee un patrón definitivo de la organización de sus partes interrelacionadas.
- Fácil de probar: facilita el establecimiento de criterios de verificación y favorece la evaluación de su rendimiento.
- Fiabilidad: capacidad de desarrollar todas sus funciones satisfactoriamente.
- Independencia del dispositivo: puede ejecutarse en ordenadores de la misma arquitectura con diferentes configuraciones *hardware* a la usada en el desarrollo.
- Ingeniería humana: cumple todos sus objetivos sin desperdiciar el tiempo ni la energía de los usuarios y sin degradar su moral.
- Legibilidad: puede distinguirse fácilmente su función mediante una lectura del código.
- Mantenibilidad: facilidad de actualización para satisfacer nuevos requerimientos o corregir deficiencias.
- Modificabilidad: facilidad para incorporar cambios una vez que se ha determinado la naturaleza de los cambios deseados.
- Precisión: las salidas tienen la exactitud suficiente.

- Robustez/integridad: capacidad para seguir funcionando aunque se produzcan algunas violaciones de los supuestos asumidos en la especificación.
- Seguimiento: favorece las mediciones sobre el uso del código.
- Transportabilidad: capacidad del sistema para ser manejado bien y sin dificultad en configuraciones y arquitecturas distintas a la del desarrollo.
- Utilidad general: la utilizabilidad, mantenibilidad y transportabilidad son condiciones necesarias para alcanzar la utilidad general.
- Utilizabilidad (utilidad tal cual): es fiable, eficiente y adaptado al usuario humano.

El modelo de Boehm está planteado desde el punto de vista de la utilidad del *producto*. El objetivo perseguido era identificar un conjunto de características de la calidad del *software* y definir una o más medidas para cada una de ellas. En la práctica, su mayor utilidad era para servir de ayuda al programador en lenguaje FORTRAN, que debería utilizarlo tras la codificación. El modelo incorporaba un conjunto de 151 medidas para poderlas aplicar sobre programas implementados en FORTRAN, aunque en realidad, la mayoría de estas medidas consistían en listas de comprobación (*check-lists*). El modelo está diseñado para el paradigma imperativo, pero sin tener estrictamente en cuenta la metodología estructurada (o, como mucho, lo poco estructurado que podía ser FORTRAN en aquella época).

Según comenta el propio Boehm, el modelo no es completo, sino que podía complementarse con herramientas analizadoras de código, además de servir de base para refinamientos y extensiones futuras. A pesar de ser la primera vez que se presentaba un modelo claro y bien definido para manejar los complejos temas relacionados con la calidad del *software*, es uno de los más completos, incluyendo, incluso, una sugerencia del autor para que se empleara el modelo para mejorar el ciclo de vida del *software*.

2.2.1.2. Modelo de McCall

James A. McCall estableció un modelo para obtener una medida de la calidad del *software* [McCall, 77] [Cavano, 78] [McCall, 79]. El modelo se basa en la división jerárquica de la calidad que se muestra en la Figura 2.4. En el nivel superior, se identifican los principales aspectos de la calidad del *software*: **factores**. Por debajo de estos factores (que se agrupan según la visión del gestor mostrada en la Figura 2.5) se encuentran una serie de atributos que, si se encuentran presentes en el *software*, proporcionan las características representadas por los factores. Estos atributos se denominan **criterios**. Por último, se sitúan las **métricas** que proporcionarán información cuantitativa sobre distintos aspectos de los criterios.

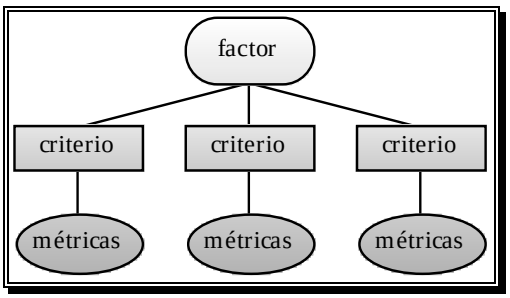


Figura 2.4: Modelo de Calidad de McCall

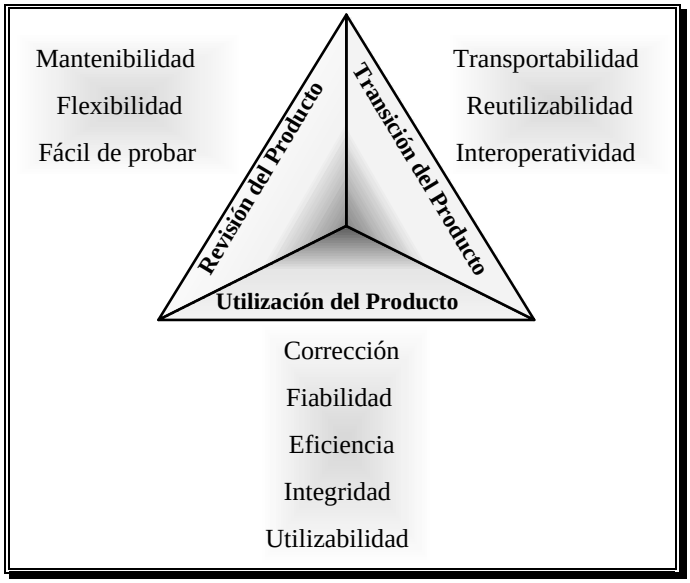


Figura 2.5: Factores de calidad del modelo de McCall

A continuación se muestran las definiciones dadas por el autor a los factores de calidad [Cavano, 78].

- Corrección: característica de un programa que satisface sus especificaciones y cumple con los objetivos del usuario.
- Eficiencia: cantidad de recursos informáticos y código requerido por un programa para desarrollar su función.
- Fácil de probar: esfuerzo necesario para probar un programa para asegurarse que funciona como debe.
- Fiabilidad: cuando un programa lleva a cabo sus funciones con la precisión requerida.
- Flexibilidad: esfuerzo requerido para modificar un programa en funcionamiento.
- Integridad: capacidad para controlar el acceso al *software* o a los datos por personas no autorizadas.
- Interoperatividad: esfuerzo necesario para utilizar un programa en conjunción con otro.
- Mantenibilidad: esfuerzo requerido para localizar y eliminar un error en un programa operativo.
- Reutilizabilidad: posibilidad de utilizar un programa en otras aplicaciones (relacionadas con el alcance de las funciones que efectúa el programa).
- Transportabilidad: esfuerzo necesario para transferir un programa de un entorno con una configuración *hardware* o *software* a otro.
- Utilizabilidad: esfuerzo requerido para aprender, operar, preparar las entradas e interpretar la salida de un programa.

En lo referente a los criterios, en la Tabla 2.1 se muestran sus relaciones con los factores, resumiéndose a continuación su significado.

Factores	Criterios
Corrección	Completitud Consistencia Seguimiento
Eficiencia	Eficiencia de almacenamiento Eficiencia de ejecución
Fácil de probar	Auto-descriptivo Instrumentación Modularidad Simplicidad
Fiabilidad	Consistencia Precisión Tolerancia a errores
Flexibilidad	Auto-descriptivo Expansibilidad Generalidad Modularidad
Integridad	Auditoría de accesos Control de acceso
Interoperatividad	Comunicaciones estándar Datos estándar Modularidad
Mantenibilidad	Auto-descriptivo Concisión Consistencia Modularidad Simplicidad
Reutilizabilidad	Auto-descriptivo Generalidad Independencia de la máquina Independencia del sistema *software* Modularidad
Transportabilidad	Auto-descriptivo Independencia de la máquina Independencia del sistema *software* Modularidad
Utilizabilidad	Comunicatividad Entrenamiento Operatividad

Tabla 2.1: Relaciones entre criterios y factores de McCall

- Auditoría de accesos: proporciona un registro del acceso al *software* y los datos.
- Auto-descriptivo: proporciona una explicación de la implementación de las funciones.
- Completitud: provee la implementación completa de todas las funciones requeridas.
- Comunicaciones estándar: proporciona la utilización de protocolos y rutinas de interfaz estándar.
- Comunicatividad: proporcionan entradas y salidas útiles que pueden asimilarse.
- Concisión: proporciona la implementación de una función con la mínima cantidad de código.
- Consistencia: proporciona un diseño y una notación y técnicas de implementación uniformes.

- Control de acceso: proporciona control del acceso al *software* y los datos.
- Datos estándar: ofrece la utilización de representaciones de datos estándar.
- Eficiencia de almacenamiento: provee los requisitos mínimos de almacenamiento durante el funcionamiento.
- Eficiencia de ejecución: provee el mínimo tiempo de proceso.
- Entrenamiento: proporciona la transición desde el sistema anterior o la familiarización inicial con el nuevo sistema.
- Expansibilidad: facilita la expansión de los requisitos de almacenamiento de los datos o de las funcionalidades.
- Generalidad: proporciona amplitud a las funciones desarrolladas.
- Independencia de la máquina: determina su independencia del *hardware*.
- Independencia del sistema *software*: determina su independencia del entorno *software* (sistema operativo, utilidades, rutinas de entrada/salida, etc.).
- Instrumentación: permite realizar medidas del uso o de identificación de errores.
- Modularidad: provee una estructura de módulos altamente independientes.
- Operatividad: determina los procedimientos relacionados con el funcionamiento y uso del *software*.
- Precisión: ofrece la precisión requerida en los cálculos y en los resultados del programa.
- Seguimiento: proporciona una relación entre los requisitos y la implementación con respecto al desarrollo específico y al entorno operacional.
- Simplicidad: provee la implementación de las funciones de la forma más comprensible posible.
- Tolerancia a errores: proporciona continuidad en el funcionamiento ante circunstancias anormales.

El modelo de McCall tiene dos niveles de aplicación: desde el punto de vista de la gestión del proyecto y desde el punto de vista de la gestión de la calidad. El modelo está pensado inicialmente para los paradigmas tradicionales y, para su evaluación, incorpora un conjunto de medidas normalizadas, la mayoría como listas de comprobación, otras como una cuenta de un atributo concreto y otras tomadas de otros autores. Además, también se otorga un peso a cada medida, obtenido por correlación en datos empíricos obtenidos de programas de las Fuerzas Aéreas Norteamericanas. La principal ventaja de este modelo sobre el precedente es que está diseñado para ser utilizado no sólo sobre el *código*, sino también sobre la *documentación*. Y, lo que es más importante, por primera vez en un modelo de calidad, el autor propone su utilización durante las distintas etapas del ciclo de vida de construcción del *software*, aunque sin detallar cuáles serían los pasos de su aplicación.

Este modelo ha servido como punto de partida de muchos otros trabajos sobre el tema, además de haber sido ampliamente utilizado y referenciado por gran número de investigadores.

2.2.1.3. Modelo de Arthur

L. J. Arthur presentó una pequeña variación al modelo de McCall consistente en añadir tres nuevos criterios, así como en modificar las relaciones existentes entre éstos y los

factores [Arthur, 85]. En la Tabla 2.2 se muestran estas relaciones, describiéndose, a continuación, los nuevos criterios.

Factores	**Criterios**
Corrección	Completitud Consistencia Seguimiento
Eficiencia	Concisión Eficiencia de ejecución Operatividad
Fácil de probar	Auditabilidad Auto-documentado Complejidad Instrumentación Modularidad Simplicidad
Fiabilidad	Complejidad Consistencia Modularidad Precisión Simplicidad Tolerancia a errores
Flexibilidad	Auto-documentado Complejidad Concisión Consistencia Expansibilidad Generalidad Modularidad Simplicidad
Integridad	Auditabilidad Instrumentación Seguridad
Interoperatividad	Comunicaciones estándar Datos estándar Generalidad Modularidad
Mantenibilidad	Auto-documentado Concisión Consistencia Instrumentación Modularidad Simplicidad
Reutilizabilidad	Auto-documentado Generalidad Independencia del *hardware* Independencia del sistema *software* Modularidad
Transportabilidad	Auto-documentado Generalidad Independencia del *hardware* Independencia del sistema *software* Modularidad
Utilizabilidad	Entrenamiento Operatividad

Tabla 2.2: Relaciones entre los criterios y factores de Arthur

- Auditabilidad: son los atributos que facilitan la realización de auditorías sobre el *software*, los datos y sus resultados.
- Complejidad: son aquellos atributos del *software* que disminuyen la claridad de un módulo o del programa.
- Seguridad: son atributos del *software* que proporcionan control y protección, tanto al *software* como a los datos.

Arthur presentó una ligera modificación al árbol de calidad de McCall. Según el autor, el modelo serviría para evaluar las necesidades del sistema y transmitirlas a los programadores. Por primera vez, se aporta la idea de agregar los valores por el árbol, indicando que la calidad es una función de los factores y cada factor, a su vez, es una función de los criterios, aunque no define cómo deben ser estas funciones.

Su modelo de calidad está pensado para utilizarlo solamente sobre el *código* (aplicado a los paradigmas tradicionales). Divide los criterios en atributos medibles, para los que propone unas ideas y medidas sencillas, muchas sin normalizar (y, por tanto, de difícil manejo).

2.2.1.4. Modelo de Gilb

T. Gilb definió una jerarquía de los atributos de un sistema que para él resultaban habitualmente interesantes en el campo de la Ingeniería del *Software* [Gilb, 88]. Gilb llamaba atributo a una medida de la calidad de un sistema. Dicha medida podía variar entre ciertos límites, siendo aún el sistema aceptable para el usuario. El objetivo fundamental del Ingeniero del *Software* sería identificar los atributos críticos y los límites de cada atributo entre los cuales el sistema sigue siendo aceptable. Estos atributos se aplican a la lógica, a los datos, a la documentación, a la interfaz y demás aspectos del *software*.

En la Figura 2.6 se muestran los atributos del modelo, cada uno de los cuales se divide, a su vez, en sub-atributos. Seguidamente, se muestra la definición dada por Gilb para estos atributos y sub-atributos.

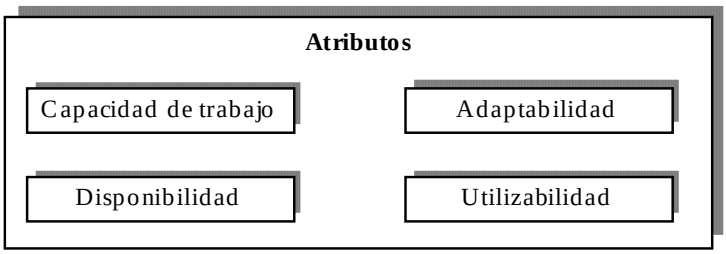


Figura 2.6: Principales atributos del modelo de Gilb

- Capacidad de trabajo: mide la capacidad natural del sistema para realizar su trabajo. Se compone de tres sub-atributos:

  a. Capacidad de almacenamiento: mide la facultad del sistema para almacenar unidades de información de un determinado elemento predefinido.

  b. Capacidad de proceso: mide la facultad de procesar cierta cantidad de datos por unidad de tiempo.

  c. Capacidad de respuesta: mide la habilidad para reaccionar ante un evento.

- Disponibilidad: mide la capacidad del sistema para llevar a cabo de forma útil el trabajo para el cual fue diseñado. Se determina por el conjunto de conceptos mostrados en la Figura 2.7:

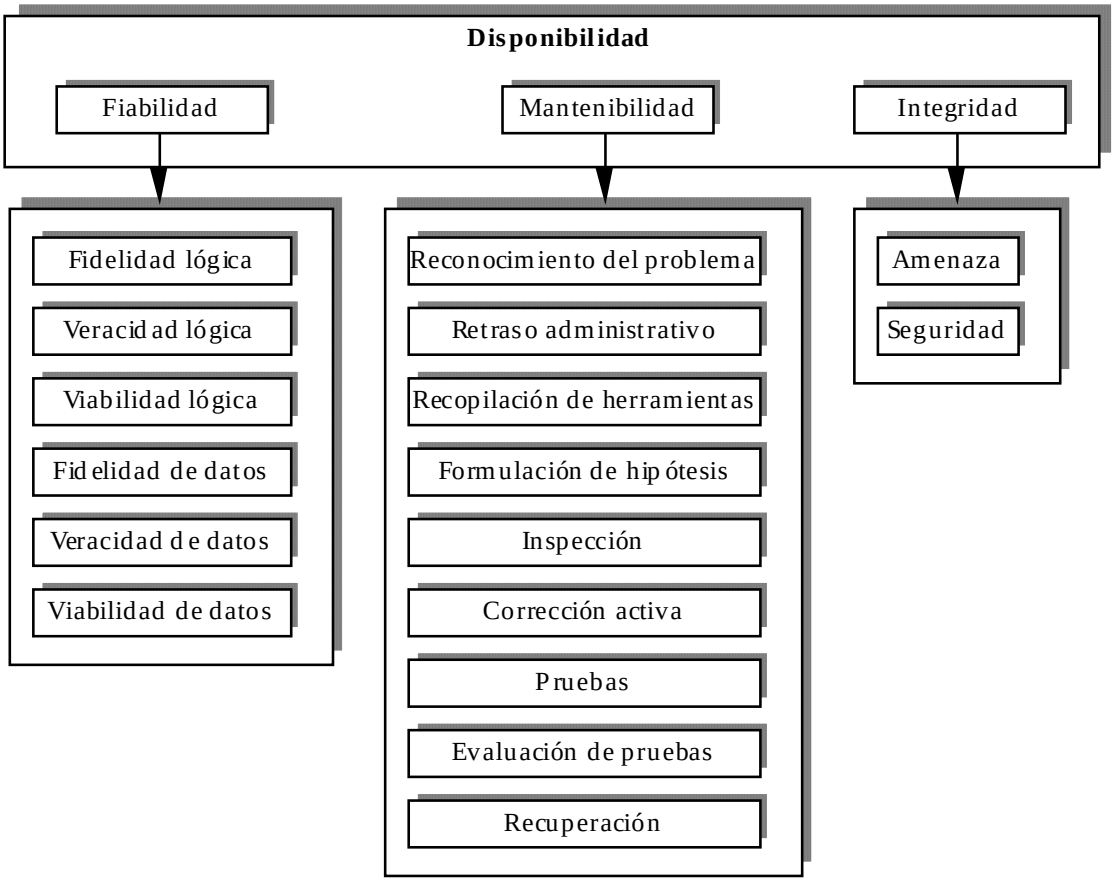


Figura 2.7: Sub-atributos de disponibilidad

a. Fiabilidad: mide el grado en el que el sistema hace lo que debería hacer. Esta medida puede ser especificada de distintas formas:

   ◊ Fidelidad lógica: mide la precisión con la cual se ha implementado un algoritmo dado para un entorno *hardware* y *software* específico.
   ◊ Veracidad lógica: mide la adecuación con la que la implementación de un algoritmo se relaciona con el mundo real, con el que debe interactuar.
   ◊ Viabilidad lógica: mide lo bien que el sistema cumple con las restricciones de diseño en áreas como la velocidad de ejecución, requisitos de espacio o seguridad.
   ◊ Fidelidad de datos: mide la precisión con la que se representa un concepto en un entorno *hardware* y *software* determinado.
   ◊ Veracidad de datos: mide la adecuación con la que los datos representan al mundo real.
   ◊ Viabilidad de datos: mide lo bien que los datos cumplen las restricciones de diseño en áreas como tiempos de recuperación de la información, espacio de almacenamiento o seguridad.

b. Mantenibilidad: mide el tiempo que costaría convertir un sistema no fiable en uno que lo sea. Se divide en varios conceptos, no todos útiles para todos los casos:

   ◊ Reconocimiento del problema: tiempo necesario para identificar la existencia de un fallo en el sistema que necesita ser reparado.
   ◊ Retraso administrativo: tiempo requerido desde la identificación del problema hasta que algo o alguien se pone en marcha para corregir el fallo.
   ◊ Recopilación de herramientas: tiempo necesario para reunir la documentación, analizar los programas, los juegos de pruebas y sus resultados y los ficheros necesarios para analizar la naturaleza del fallo.
   ◊ Formulación de hipótesis de corrección: tiempo necesario para traducir la causa del fallo en una acción adecuada para corregirlo.
   ◊ Inspección: tiempo para estudiar la hipótesis de corrección con el fin de determinar su consistencia y adecuación en relación tanto con el cambio local como con el sistema completo.
   ◊ Corrección activa: tiempo necesario para llevar a cabo la hipótesis de corrección estudiada.
   ◊ Pruebas: tiempo necesario para ejecutar los casos de prueba adecuados para validar que el cambio funciona como se esperaba y que no existen efectos laterales no deseados.
   ◊ Evaluación de pruebas: tiempo necesario para evaluar los resultados de las pruebas.
   ◊ Recuperación: tiempo para recuperarse del fallo y restituir todos los ficheros necesarios.

c. Integridad: medida de la capacidad para sobrevivir a ataques al sistema. Está relacionado con:

   ◊ Amenaza: se mide mediante la probabilidad estimada de que un ataque de un tipo determinado ocurra en un instante dado.
   ◊ Seguridad: puede medirse como la probabilidad de contrarrestar los ataques de un determinado tipo.

♦ Adaptabilidad: es la medida de la capacidad de un sistema para ser cambiado adecuadamente. Se divide en los siguientes sub-atributos:

a. Improbabilidad: mide la eficiencia al hacer pequeñas adaptaciones, cambios y mejoras al sistema.

b. Extensibilidad: mide la facilidad para añadir nuevas características a un sistema existente.

c. Transportabilidad: mide la facilidad para trasladar un sistema de un entorno a otro.

- Utilizabilidad: es la medida de la facilidad con que la gente será capaz y estará motivada para usar el sistema en la práctica. Se divide en los siguientes sub-atributos:

  a. Requisitos de entrada: mide los requisitos humanos para alcanzar éxito en el aprendizaje y manejo del sistema.

  b. Requisitos de aprendizaje: mide los recursos necesarios (principalmente el tiempo) para alcanzar cierto nivel de habilidad con el sistema.

  c. Habilidad de manejo: mide la productividad neta en el tiempo con el nuevo sistema.

  d. Complacencia: mide el grado en el que los usuarios están contentos con el sistema.

Gilb presentó una jerarquía de los atributos de un sistema que podían utilizarse para medir su calidad o para estimar los recursos necesarios para su construcción o mantenimiento. Sus objetivos eran ayudar a los gestores del proyecto, servir de guía para solucionar problemas y detectar zonas de riesgo. En cuanto a las medidas, Gilb se limita a dar algunas ideas generales acerca de las cosas a medir (todas relativas al código), pero sin entrar en detalle y sin proporcionar medida alguna, por lo que no se le puede considerar un modelo de calidad completo. Además, sólo pensó en la aplicación del modelo al finalizar la *implementación* del sistema (o de los prototipos); en las fases anteriores del ciclo de vida sencillamente sugería la realización de inspecciones basadas en listas de comprobación. En lo referente a los atributos, Gilb presentaba una serie de conceptos que, aunque pueden tener relación con la calidad, muchos no son del estilo de los comúnmente aceptados, siendo simplemente algunos (como los de mantenibilidad) una descomposición en sub-atributos para realizar estimaciones de tiempo.

2.2.1.5. Modelo de Deutsch y Willis

Deutsch y Willis publicaron en 1988 una jerarquía similar a la de McCall, aunque incluía nuevos factores y criterios, así como nuevas relaciones entre éstos [Deutsch, 88]. La jerarquía parte de las necesidades del usuario que pueden descomponerse como se muestra en la Figura 2.8.

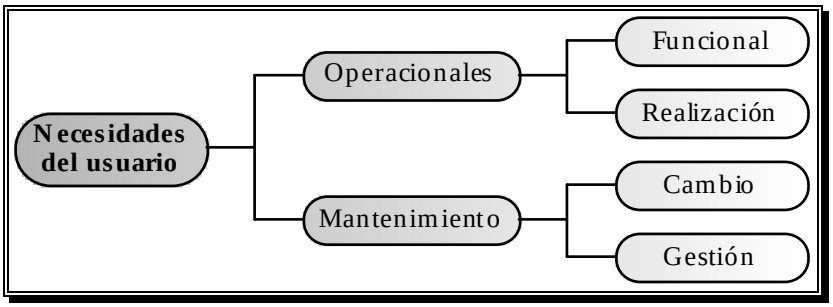


Figura 2.8: Necesidades del usuario

Las necesidades 'operacionales' tienen que ver con la capacidad del *software* para llevar a cabo las tareas que se supone que debe realizar. Las necesidades de 'mantenimiento'

tienen relación con la modificación del *software*, con el fin de ayudar al usuario. Dentro de las necesidades 'operacionales', la 'funcionalidad' trata de lo que hace el *software* al ejecutarse, mientras que la 'realización' trata de lo bien que lo hace. En cuanto a las necesidades de 'mantenimiento', el 'cambio' trata de las modificaciones del *software* para corregir errores, adaptarse a nuevos entornos o añadir nuevas funcionalidades, mientras que la 'gestión' tiene que ver con la planificación para cambiar, controlar versiones, pruebas e instalación.

Necesidades del Usuario	Factores de Calidad
Funcional	Integridad Fiabilidad Supervivencia Utilizabilidad
Realización	Eficiencia Corrección Seguridad Interoperatividad
Cambio	Mantenibilidad Expansibilidad Flexibilidad Transportabilidad Reutilizabilidad
Gestión	Verificabilidad Gestionable

Tabla 2.3: Factores de calidad de Deutsch

Estas necesidades del usuario agrupan a 15 factores de calidad. En la Tabla 2.3 se muestra su relación con las necesidades de los usuarios. Los conceptos innovadores presentados por Deutsch y Willis (en relación con el modelo de McCall) son:

- Fiabilidad: tiene relación con la tasa de fallos en el *software* que lo hacen inutilizable, como la falta de precisión, tiempos de respuesta no apropiados o bloqueos del sistema.
- Flexibilidad: el *software* es flexible si puede adaptarse a diferentes entornos.
- Gestionable: trata de los aspectos administrativos para la modificación del *software*.
- Seguridad: trata de la ausencia de condiciones inseguras del *software*; si un sistema es seguro significa que se puede confiar en él.
- Supervivencia: ante la presencia de un fallo del sistema, las funciones esenciales del *software* deben continuar su ejecución de forma fiable.
- Verificabilidad: indica lo fácil que es verificar que el *software* está funcionando correctamente.

Estos factores se subdividen a su vez en un total de 27 criterios de la calidad del *software*. Dichos criterios muestran las características que debe cumplir todo sistema. En la Figura 2.9 se muestra la relación de estos criterios con los factores. A continuación se explican brevemente los criterios del modelo de Deutsch y Willis no definidos en el de McCall.

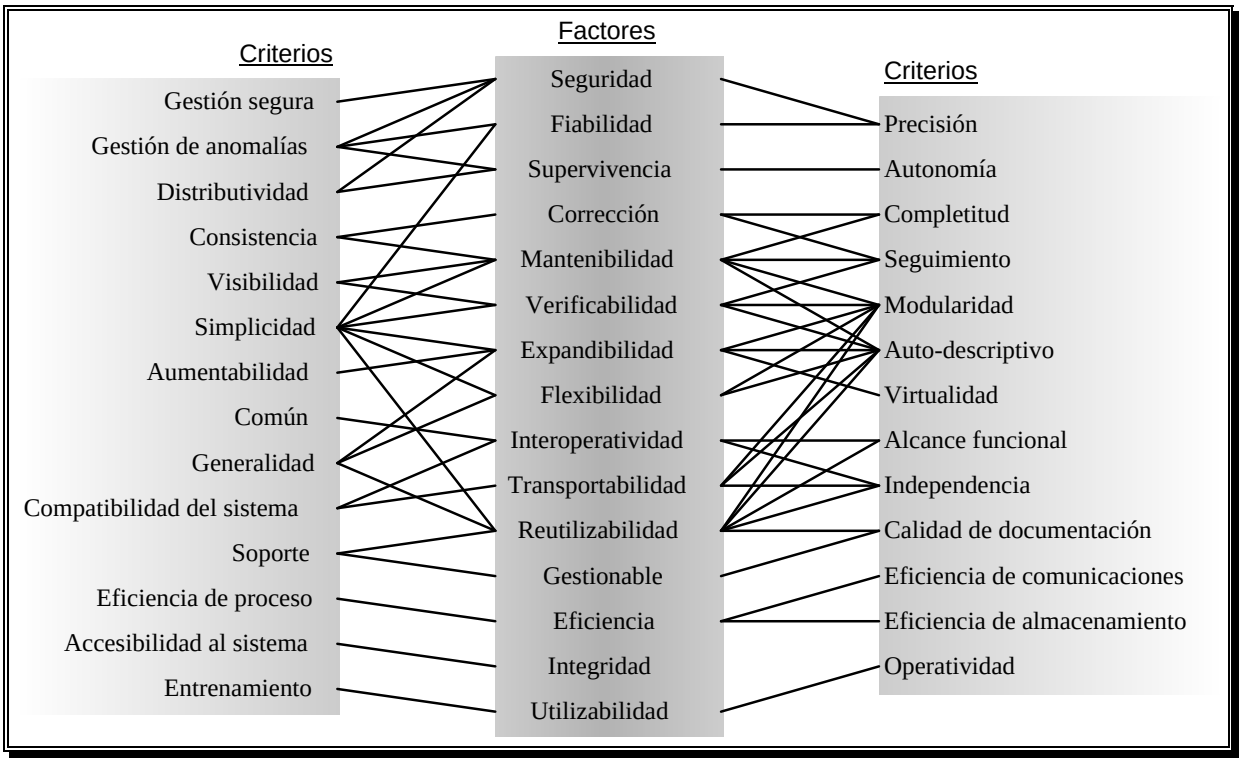


Figura 2.9: Relación entre los criterios y los factores de Deutsch

- Accesibilidad al sistema: unifica al control de acceso y la auditoría de accesos del modelo de McCall.
- Alcance funcional: indica si las funciones tienen una amplio rango de aplicabilidad, es decir, si las funciones son reutilizables en aplicaciones o sistemas similares u otros lugares.
- Aumentabilidad: sencillez para añadir nuevas capacidades funcionales o nuevos datos; es decir, si resulta fácil expandir sus posibilidades sin grandes esfuerzos de rediseño o modificación (similar al concepto de expansibilidad).
- Autonomía: el *software* es autónomo si, en caso de un fallo de sus componentes de interfaz, servicios o dispositivos, puede reconfigurarse automáticamente y proseguir la ejecución.
- Calidad de la documentación: el *software* está bien documentado si una persona puede acceder fácilmente a la información sobre el diseño y sus capacidades y, además, la información es comprensible y completa.
- Compatibilidad del sistema: habilidad de dos o más sistemas para trabajar en armonía.
- Común: si se utilizan estándares reconocidos para representar datos, interfaces con redes de comunicación, interfaces de usuario…
- Distribuido: el *software* está distribuido si sus partes se encuentran localizadas en distintos dispositivos de proceso o almacenamiento.
- Eficiencia de proceso: coincide con eficiencia de ejecución del modelo de McCall.
- Gestión de anomalías: capacidad para detectar y recuperarse de condiciones anómalas en lugar de bloquearse o finalizar la ejecución (similar al concepto de tolerancia a errores).
- Gestión segura: el *software* es seguro si evita las situaciones de peligro.

- Independencia: unifica la independencia del sistema *software* y de la máquina del modelo de McCall.
- Soporte: representa si se incluyen herramientas, bases de datos y procedimientos que facilitan la gestión de cambios después de que el *software* ha sido entregado.
- Virtualidad: capacidad de representar componentes físicos distintos mediante el mismo componente lógico o virtual.
- Visibilidad: el *software* es visible si puede comprenderse su progreso en relación con la codificación y pruebas, lo cual se obtiene comprobando si hay una evidencia objetiva de un funcionamiento adecuado.

El punto de vista del modelo de calidad de Deutsch y Willis se presenta desde las necesidades del usuario, dando, quizás, demasiada importancia al mantenimiento. El trabajo de los autores no incorpora ninguna medida; tan solo da una serie de sugerencias (mediante ejemplos) para mejorar los criterios, basándose en los paradigmas tradicionales. El modelo puede utilizarse para validar los *requisitos* y el *diseño* del sistema, así como durante las *pruebas*, aunque los autores no indican método alguno para su aplicación durante dichas fases del ciclo de vida. En resumen, Deutsch y Willis se han limitado a realizar una nueva descomposición de la calidad, sin completar el modelo con una serie de medidas objetivas.

2.2.1.6. Modelo de Schulmeyer

G. Gordon Schulmeyer publicó una serie de indicadores de la calidad del *software* [Schulmeyer, 90]. Estos indicadores de la calidad no debían ser interpretados como medidas, sino que servirían de base para dichas medidas. Los indicadores serían de ayuda pues podrían dar una idea de la tendencia de la calidad (adecuación a requisitos) en el desarrollo del *software*. El objetivo de estos indicadores era:

- Completitud de las pruebas: mide si se ha completado totalmente el proceso de pruebas, tanto desde la perspectiva del desarrollador como desde la del usuario. El indicador está basado en la estructura del *software* y en los requerimientos (casos de prueba predefinidos o funcionalidades).
- Completitud: proporciona una visión de lo bien que se han traducido los requerimientos del sistema en las especificaciones de requisitos *software*. El indicador se basa en el análisis de requisitos y en el proceso de diseño e inspección del código.
- Densidad de defectos: proporciona una idea de la calidad de la traducción de los requisitos en el diseño e implementación. Está basado en la fase de diseño y en el proceso de inspección del código.
- Densidad de fallos: proporciona una información similar al anterior indicador, aunque, en este caso, el énfasis se centra en ver si los requerimientos se han implementado correctamente en productos *software* que puedan probarse. Este indicador toma su información directamente de los resultados de la fase de pruebas.
- Documentación: se utiliza con el fin de determinar si la documentación del *software* se adecua correctamente a las necesidades que tiene el usuario en cuanto a ayuda y mantenimiento del *software* entregado. La información se obtiene a partir de la documentación y listados fuente de cada uno de los productos desarrollados.

- Estructura del diseño: se utiliza para determinar la simplicidad o claridad del diseño detallado de cada uno de los elementos que intervienen en el desarrollo del *software*. Se basa fundamentalmente en el diseño y las inspecciones de código.
- Suficiencia de las pruebas: permite calcular cómo ha ido el proceso de pruebas y de integración a la hora de detectar errores del *software* antes de la entrega del sistema al usuario, basándose en la predicción del resto de los fallos del *software*. Este indicador obtendrá su información a partir de pruebas y sistemas de calidad de la compañía, empleando diversas técnicas para estimar la predicción del número de fallos.

Schulmeyer se sale de la línea habitual y descompone la calidad en un único nivel de unos pocos indicadores, que pueden emplearse para evaluar la calidad del desarrollo, basándose en los paradigmas tradicionales. Sin embargo, solamente proporciona un total de 20 medidas (con fórmulas normalizadas) que cubren a 4 de los indicadores. De estas 20 medidas, 11 de ellas son aplicables en el diseño. Por tanto, el modelo de calidad de Schulmeyer es de aplicación durante las fases de *diseño* e *implementación*, aunque, debería incluir un mayor número de medidas que, además, deberían abarcar los 7 indicadores. No obstante, sí sugiere que la agregación de las distintas medidas de un indicador debería realizarse mediante una media ponderada, siendo el primer modelo en detallar este tipo de cálculos. Tampoco proporciona una indicación de cómo tendría que aplicarse su modelo para evaluar la calidad del sistema.

2.2.1.7. Modelo de Gillies

A. C. Gillies presentó su modelo de calidad que se diferenciaba de otros trabajos previos en el sentido de que consideraba todos los criterios como un subconjunto de corrección, dividiéndola en corrección funcional y empresarial [Gillies, 92]. Esto requería la incorporación de nuevos criterios asociados con la corrección desde el punto de vista de los negocios. Según el autor, este modo de ver el modelo le permitiría reflejar tanto el punto de vista de calidad del diseñador como el del usuario. En la Figura 2.10 se muestra la jerarquía definida por Gillies y a continuación se definen aquellos criterios nuevos o diferentes en relación con el modelo de McCall.

- Ajuste al plan de tiempo: indica si las entregas del sistema se realizan en los plazos acordados.
- Coste/beneficio: representa si el sistema satisface las especificaciones de coste/beneficio.
- Facilidad de transición: refleja la facilidad y eficiencia para asimilar la información de los sistemas existentes.
- Integridad: representa la capacidad para que los accesos al *software*, al *hardware* o a los datos puedan ser controlados o recuperados en caso de corrupción.
- Mantenibilidad: es el esfuerzo requerido para modificar o ampliar el sistema.
- Seguridad: es la capacidad para controlar el acceso al sistema por personas no autorizadas.
- Uso amigable: indica si el sistema ayuda al operador a llevar a cabo su trabajo.

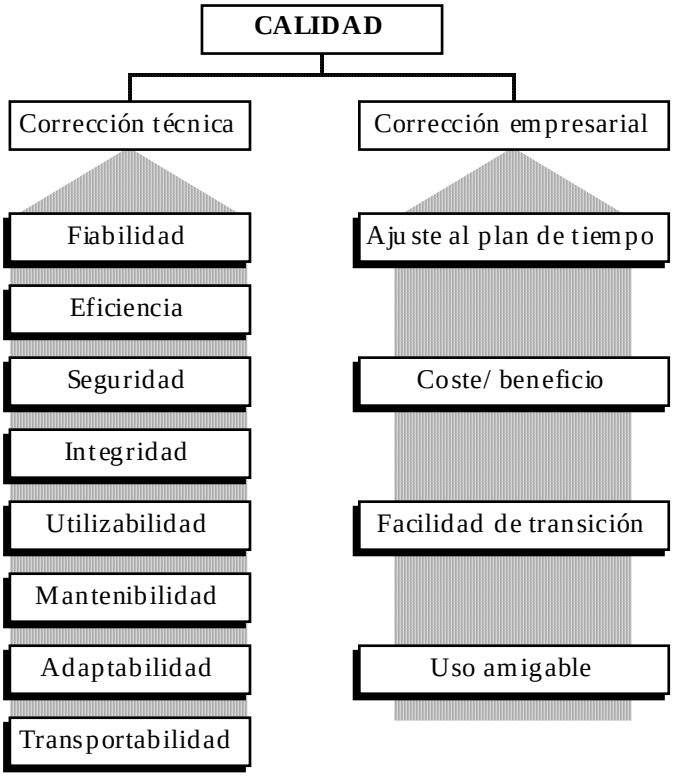


Figura 2.10: Jerarquía de calidad de Gillies

El punto de vista del modelo de Gillies es el de la corrección. No obstante, intenta clasificar todo desde este punto de vista, por lo que algunos conceptos importantes (como la flexibilidad) quedan fuera. Por otro lado, la separación entre los dos tipos de correcciones no queda del todo claro, como en el caso del 'uso amigable'. Proporciona algunas ideas para desarrollar medidas sobre sus criterios, siendo unas para obtener medidas del producto en ejecución y otras para elaborar medidas de costes. Todas estas ideas son para su utilización a posteriori, una vez que se ha finalizado el sistema. El autor no indica forma alguna de agregar los valores ni una forma de aplicación del modelo a un desarrollo.

2.2.1.8. Modelo REBOOT

El modelo de calidad REBOOT [Stålhane, 92], fruto de un proyecto ESPRIT, está basado en la división habitual de la calidad en factores, criterios y métricas, así como en las relaciones entre sí representadas mediante una jerarquía, que se refleja en la Figura 2.11. Pero, junto al modelo de calidad REBOOT, se ha incluido, además, otro modelo de reutilización [Mora, 92], cuya jerarquía se muestra en la Figura 2.12.

Ambos modelos se presentan separados porque, según los autores, hay factores que son importantes para la reutilización y que, en cambio, no tienen ninguna importancia cuando se instala un sistema en un lugar de trabajo estable. A continuación, se definen brevemente los nuevos factores.

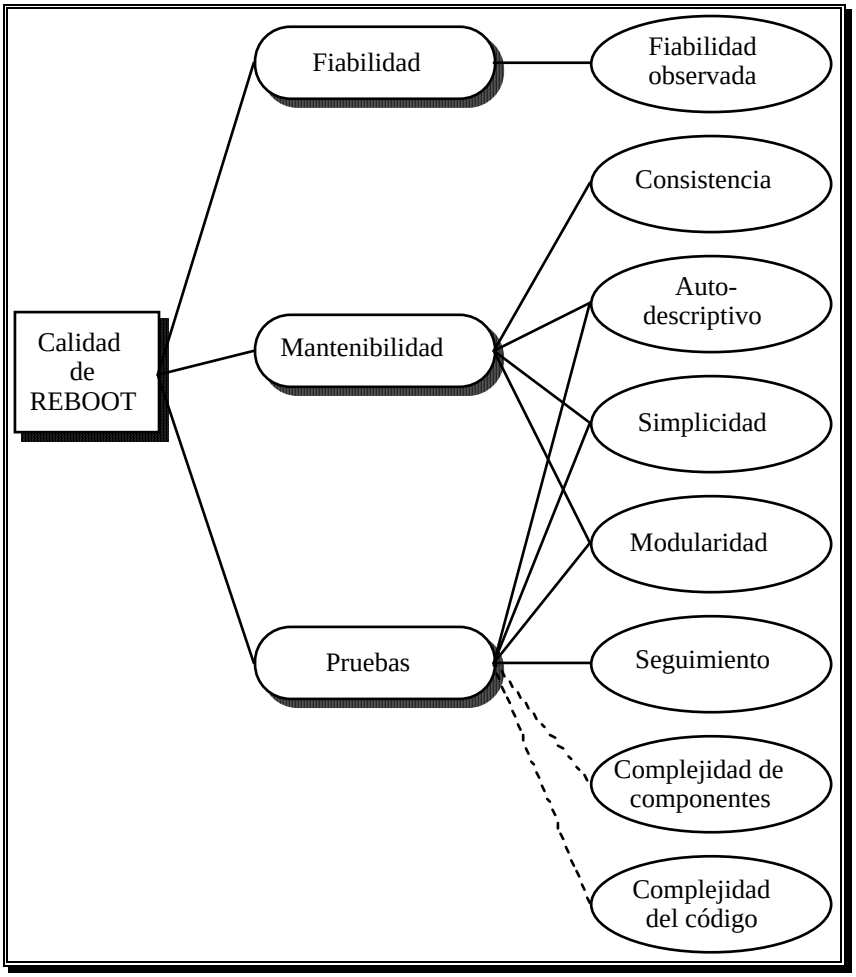


Figura 2.11: Modelo de calidad REBOOT

- Comprensibilidad: denota si el propósito del programa queda claro para todo aquél que lo inspeccione.
- Confianza: probabilidad de que un módulo, programa o sistema lleve a cabo su objetivo satisfactoriamente (sin fallos) en un tiempo determinado y sobre un entorno diferente al que se utilizó para su construcción.
- Flexibilidad: representa la existencia de un rango de opciones disponibles para el programador (cuantas más opciones, mayor flexibilidad).
- Mantenibilidad: facilidad con que se puede llevar a cabo el mantenimiento de una unidad funcional de acuerdo con los requisitos predeterminados.
- Pruebas: la capacidad de *software* para facilitar tanto el establecimiento de los criterios de pruebas como la evaluación de dicho *software* con respecto a estos criterios.

Los modelos fueron diseñados para evaluar un repositorio de componentes para reutilizar. El método propuesto era estimar, en primer lugar, la calidad teórica con ambos modelos y, después, compararlo con los datos observados tras la reutilización. Se aplican en el proyecto para obtener puntos para los criterios a partir de las 19 medidas (no originales) que presentan, siendo algunas de estas medidas listas de comprobación. Las medidas se aplican en el código y, para la agregación de valores, sugieren el uso de medias ponderadas. Aunque la descomposición de la

reutilización es bastante exhaustiva, la realizada para la calidad resulta demasiado superficial.

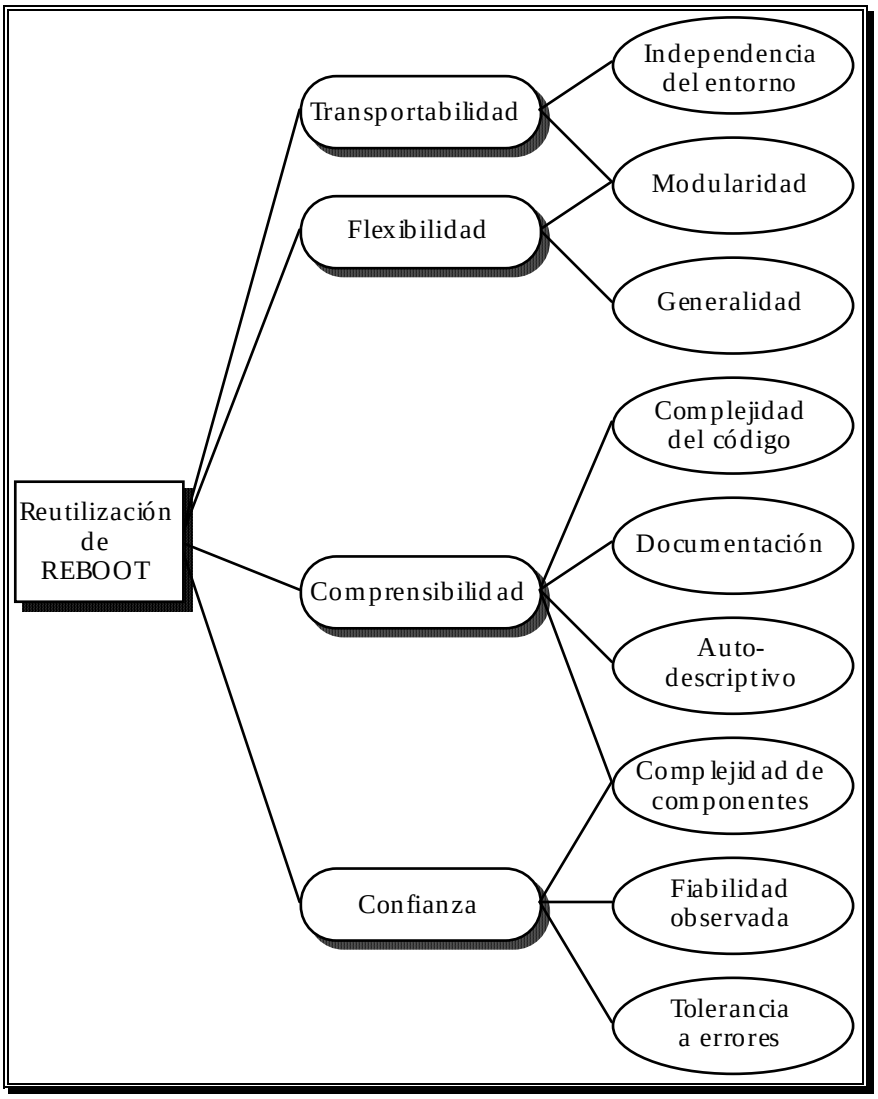


Figura 2.12: Modelo de reutilización

Como puede observarse, es uno de los primeros que indica la forma de aplicación, aunque, como va dirigida a la reutilización se separa ligeramente del objetivo perseguido en este trabajo. Lo realmente novedoso de estos modelos, es que es la primera vez que se tiene en cuenta la existencia de la tecnología orientada a objetos, puesto que son de aplicación tanto para los paradigmas tradicionales como para el orientado a objetos.

2.2.1.9. Modelo de Dromey

Hace pocos años, R. Geoff Dromey publicó un nuevo modelo para la calidad del producto *software* [Dromey, 95]. Su modelo se basa en determinar una serie de *propiedades* de la calidad del *software* y clasificarlas en cuatro *categorías* básicas:

- Corrección: requisitos genéricos mínimos para la corrección.
- Estructuración: aspectos de bajo nivel del diseño interno de los módulos.

- Modularidad: aspectos de alto nivel del diseño de la comunicación entre módulos.
- Descriptivo: diversas formas de especificación/documentación.

A su vez, también define unos *atributos* de calidad de alto nivel que son los habituales en otros trabajos: eficiencia, fiabilidad, funcionalidad, mantenibilidad, reutilizabilidad, transportabilidad y utilizabilidad.

A continuación, se explican las distintas propiedades sugeridas por el autor, agrupadas por categorías.

♦ Corrección:

- Asignación: una variable está asignada si antes de ser utilizada recibe un valor, bien por una sentencia de asignación, una entrada o por ser un parámetro de una función. Lo contrario podría ser causa de una incorrección.
- Completitud: una función posee la propiedad de ser completa si tiene todos los elementos necesarios para definir e implementar la función para que cumpla su rol de forma que no afecte a la fiabilidad o a la funcionalidad.
- Computable: indica si la solución incluye cálculos que están definidos de acuerdo con la teoría estándar de la computación y dentro de los límites definidos por el programa, el lenguaje de programación o la máquina.
- Consistencia: una estructura es consistente si sus propiedades o funcionalidad se conserva tras su uso y si todos sus elementos contribuyen y favorecen su intención u objetivo.
- Inicializado: un bucle está inicializado si todas sus variables se han inicializado antes de entrar al bucle y lo más cerca posible a la entrada al bucle.
- Precisión: una variable o una constante tiene un tipo impreciso cuando su precisión no es suficiente para alcanzar la exactitud requerida en los cálculos en los que interviene.
- Progresivo: cuando en un bucle o en un algoritmo recursivo hay una evidencia clara de que la estructura converge con cada iteración o llamada recursiva.
- Variante: si una guarda define una relación que es congruente y derivable de la función empleada para probar la terminación del bucle.

♦ Estructuración:

- Ajustable: cuando no existen constantes no declaradas y se utiliza el mínimo número de variables con un único objetivo en la solución del problema.
- Directo: un cálculo puede expresarse directamente si la abstracción, selección de la representación y la estructura del cálculo son congruentes con el problema original que se está modelando.
- Efectividad: si se tienen todos los elementos necesarios y solo los elementos necesarios para definir e implementar un algoritmo.
- Estructuración: cuando se siguen las reglas de la programación estructurada.
- Homogeneidad: cuando un algoritmo puede describirse por un invariante donde los principales predicados tienen una estructura conjuntiva.
- No redundancia: cuando se tienen todos los elementos lógicos necesarios y solo los elementos necesarios para definir e implementar un algoritmo.

- Rangos independientes: cuando los límites inferiores y superiores de una estructura no son constantes numéricas o caracteres.
- Resuelto: si las estructuras de control de la implementación concuerdan con la estructura de los datos.
- Utilizado: cuando lo que se define se utiliza dentro de su ámbito.

Atributos	Propiedades
Eficiencia	Directo, Efectividad, No redundancia, Resuelto, Utilizado
Fiabilidad	Asignación, Completitud, Computable, Consistencia, Encapsulación, Especificado, Estructuración, Inicializado, Pobremente acoplado, Precisión, Progresivo, Variante
Funcionalidad	Asignación, Completitud, Computable, Consistencia, Especificado, Estructuración, Inicializado, Precisión, Progresivo, Variante
Mantenibilidad	Ajustable, Auto-descriptivo, Cohesión, Completitud, Consistencia, Directo, Documentado, Efectividad, Encapsulado, Especificado, Estructuración, Generalidad, Homogeneidad, Inicializado, No redundancia, Parametrizado, Pobremente acoplado, Progresivo, Rangos independientes, Resuelto, Utilizado, Variante
Reutilizabilidad	Abstracto, Ajustable, Auto-descriptivo, Cohesión, Consistencia, Documentado, Encapsulación, Especificado, Generalidad, Parametrizado, Pobremente acoplado, Rangos independientes
Transportabilidad	Ajustable, Auto-descriptivo, Cohesión, Consistencia, Documentado, Encapsulación, Especificado, Generalidad, Parametrizado, Pobremente acoplado
Utilizabilidad	Auto-descriptivo, Completitud, Consistencia, Documentado, Efectividad, Especificado

Tabla 2.4: Clasificación de los atributos de alto nivel en sus propiedades

♦ Modularidad:

- Abstracción: un objeto/módulo es una abstracción si no existe un concepto obvio de más alto nivel que lo abarque.
- Cohesión: cuando todos los elementos están vinculados a algún otro, contribuyendo todos a cumplir un objetivo del programa.
- Encapsulación: si un nombre sólo se utiliza en el ámbito en el que ha sido definido.
- Generalidad: un módulo es genérico si sus cálculos son independientes de los tipos de los datos.
- Parametrizado: se considera que un módulo está bien parametrizado cuando sus parámetros son todas las entradas y salidas necesarias y suficientes para determinar un módulo bien definido.
- Pobremente acoplado: cuando los datos de las llamadas a los módulos están acoplados al módulo llamado.

♦ Descriptivo:

- Auto-descriptivo: cuando el propósito, estrategia, intención y propiedades de cada elemento del *software* son claramente evidentes a partir de los nombres de los módulos y los diferentes identificadores tienen sentido y son congruentes dentro del contexto de la aplicación.
- Documentado: cuando se ha definido de forma explícita y precisa el propósito, estrategia, intención y propiedades de cada elemento del *software*.
- Especificado: cuando la funcionalidad se describe mediante precondiciones y postcondiciones.

A su vez, la Tabla 2.4 muestra la relación existente entre los atributos y las propiedades de la calidad.

Dromey presentó un modelo para la calidad del *producto* construido utilizando los paradigmas tradicionales, cuyo objetivo era poder detectar defectos y clasificarlos. No se incluían medidas en el modelo; en su lugar, se propone la realización de inspecciones del código. Así, la forma de utilizar el modelo consiste en valorar los atributos de calidad de alto nivel comprobando las reglas de inspección de código asociadas a dichos atributos. Por tanto, el presente modelo es únicamente de aplicación tras la implementación, aunque el autor sugiere sobre qué componentes del código fuente debe centrarse cada propiedad.

2.2.2. MODELOS DE CALIDAD PARCIALES

Hay autores que en lugar de intentar abordar el complejo problema de definir un modelo completo de calidad, se han centrado en alguno de los factores determinantes de la calidad del *software*. Se presentan a continuación, a modo de ejemplo, algunos de estos estudios.

2.2.2.1. Estabilidad por Yau y Collofello

Stephen S. Yau y James S. Collofello afirmaban que el atributo más influyente al realizar cualquier tipo de modificación en un sistema era la estabilidad de un programa, definiendo estabilidad como la resistencia de un programa a ser cambiado (y que siga funcionando correctamente), debido a efectos de lado tanto locales como globales [Yau, 80]. Define también la estabilidad de un módulo como una medida de la resistencia a los potenciales efectos de lado que puede ocasionar una modificación en un módulo sobre el resto de los módulos del programa. Hay dos aspectos de la estabilidad de un módulo:

- Estabilidad lógica: es una medida de la resistencia al impacto de una modificación sobre otros módulos del programa por consideraciones lógicas.
- Estabilidad de ejecución: es una medida de la resistencia al impacto de una modificación sobre otros módulos del programa por consideraciones de rendimiento.

Aunque para los autores influyen varios atributos en el mantenimiento, consideran que el más importante es el de la estabilidad. El concepto de estabilidad presentado supone una interesante e innovadora aportación de Yau y Collofello. Además, proponen una

complicada medida normalizada, basada en la complejidad ciclomática (véanse los apartados 2.3.1.2 y 2.3.1.6), para medir, sobre el código fuente, la estabilidad lógica de un programa. Por el contrario, confiesan que no han desarrollado ninguna medida para la estabilidad de ejecución. Aunque plantean el modelo únicamente para la *implementación* (según paradigmas tradicionales), su extensión a la fase de diseño no resultaría complicada, puesto que se podría centrar en el análisis de los flujos de datos y de control para obtener una estimación de la estabilidad.

2.2.2.2. Mantenimiento por Rombach

H. Dieter Rombach define mantenimiento del *software* como el rendimiento de aquellas actividades necesarias para mantener operativo a un sistema. En el modelo de mantenimiento del *software* propuesto por Rombach, se descompone el concepto de mantenimiento en los cinco aspectos siguientes [Rombach, 87]:

- Mantenibilidad: representa el impacto de la estructura del *software* en el esfuerzo medio por cada tarea de mantenimiento.
- Comprensibilidad: refleja el impacto de la estructura del *software* en el esfuerzo medio necesario para aislar y entender qué es lo que hay que modificar. Además, refleja el porcentaje de trabajo necesario para realizar una tarea de mantenimiento en comparación con todos los trabajos de mantenimiento.
- Localidad: comprende el impacto de la estructura del *software* en el número medio de módulos cambiados por tarea de mantenimiento, así como en la parte máxima media del esfuerzo empleado para realizar los cambios en un único módulo por cada tarea de mantenimiento.
- Modificabilidad: representa el impacto de la estructura del *software* en el esfuerzo medio por módulo empleado para realizar las correcciones por cada tarea de mantenimiento.
- Reutilizabilidad: refleja el impacto de la estructura del *software* en el porcentaje medio de documentación reutilizada por cada tarea de mantenimiento.

Rombach afirmaba que la estructura del *software* influye en el mantenimiento, por lo que realizó un estudio empírico con 4 pequeños sistemas implementados en un lenguaje propio para obtener su modelo. Incorpora la necesidad de emplear medidas (él usa medidas sin normalizar de la complejidad del código y de la estructura del sistema), aunque no proporciona medidas para todos los aspectos de su modelo. La forma de utilizar el modelo consiste en evaluar las medidas de complejidad aplicadas sobre el sistema, los módulos y el personal de desarrollo. Si estos valores sobrepasan ciertos límites, debe examinarse el sistema para encontrar una justificación. El principal problema de la técnica propuesta es que se basa en evaluar los aspectos del mantenimiento a posteriori, es decir, cuando ya se están realizando tareas de mantenimiento, por lo que su verdadera utilidad resulta cuestionable.

2.2.2.3. Complejidad por González

Para definir su modelo de la complejidad del *software*, Renato R. González [González, 95] se basa en el concepto de "bloque básico" [Aho, 90]: "secuencia de una o más sentencias consecutivas con la propiedad de que no existe explícitamente ninguna

transferencia de control a cualquiera de sus sentencias salvo, quizás, a la primera, ni desde ninguna de sus sentencias salvo, quizás, la última". Define, también, complejidad como una medida de los recursos que deben emplearse en desarrollar, probar, depurar, enseñar a los usuarios y corregir el producto *software*. El autor considera que un programa puede concebirse en tres dominios fundamentales de complejidad: sintáctico, funcional y computacional. Cada uno de ellos tiene igualmente tres magnitudes: tamaño, tiempo y nivel. Utilizando estas ideas, en la Figura 2.13 se presenta la taxonomía de la complejidad del *software* que sugiere González, mostrándose a continuación una explicación breve de los factores de complejidad.

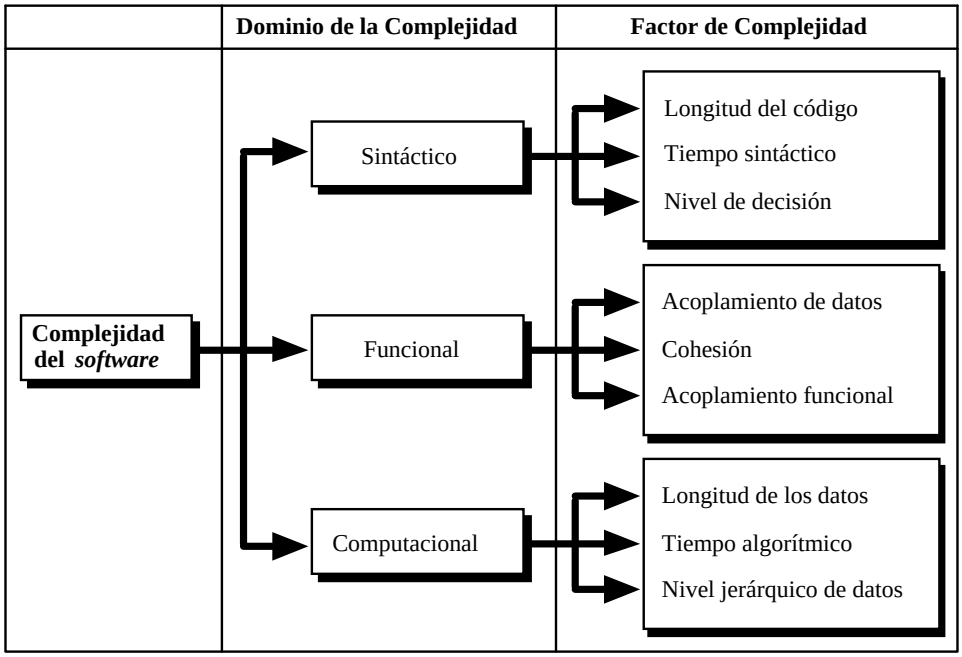


Figura 2.13: Taxonomía de la complejidad de González

- Longitud del código: es la cantidad de código contenido en los bloques básicos basándose en las propiedades del programa sin interpretar u ordenar los componentes de los bloques básicos.
- Tiempo sintáctico: se define mediante el árbol sintáctico de un bloque básico que identifica los componentes importantes de cada construcción sintáctica o expresión. De esta forma, cada nodo no terminal (los operadores) representa el consumo lógico de tiempo.
- Nivel de decisión: define la complejidad lógica del programa o su nivel de anidamiento.
- Acoplamiento de datos: representa la cantidad de información manejada por un bloque básico. Esta información se refiere a diferentes campos de entrada provenientes de otros bloques básicos y a los datos enviados hacia otros bloques básicos, variables globales y estructuras de datos.
- Cohesión: es el grado de interacción entre instrucciones de un mismo bloque básico.
- Acoplamiento funcional: mide la complejidad producida por los caminos de ejecución que forman los bloques básicos en el grafo de control de flujo.

- Longitud de los datos: es el grado de los nodos de datos en una estructura de datos (un grafo donde los nodos son los datos y los arcos representan la relación entre ellos) y es igual al número de arcos que entran y salen de cada nodo en dicho grafo de datos.
- Tiempo algorítmico: es la cantidad de trabajo realizado, medido, por ejemplo, por el número de operaciones básicas efectuadas durante la ejecución de un programa.
- Nivel jerárquico de datos: es la profundidad del grafo de datos representada por el número de caminos distintos y el nivel de los nodos de datos en dicho grafo.

González identificó distintas fuentes para la complejidad del *software*. Emplea medidas basadas en la teoría de la información de Shannon [Shannon, 71] y las estructuras matemáticas discretas del código. Es de utilidad para los paradigmas tradicionales y solamente para la fase de implementación. Como aspecto interesante, utiliza todas las medidas definidas para obtener una única medida para la complejidad.

2.2.2.4. Cohesión y Acoplamiento por Dhama

Harpal Dhama [Dhama, 95] observó que de los trece factores de calidad incluidos en el modelo de calidad desarrollado por Bowen [Bowen, 83] (corrección, fiabilidad, eficiencia, integridad, utilizabilidad, supervivencia, mantenibilidad, verificabilidad, flexibilidad, transportabilidad, reutilizabilidad, interoperatividad y expansibilidad), los ocho últimos dependían de dos propiedades del *software* (cohesión y acoplamiento) con un gran impacto, por lo tanto, en la calidad del *software*. Estas dos propiedades se definen como sigue:

- Cohesión en un módulo se refiere a aquella propiedad del *software* que enlaza juntas las diversas sentencias y otros módulos menores que formen parte de dicho módulo. Es una propiedad positiva intra-módulo que refleja las consideraciones de diseño para integrar los distintos componentes del módulo en una única unidad. Es como el 'pegamento' que mantiene al módulo unido.
- Acoplamiento: es una medida de la interdependencia entre dos módulos. El acoplamiento es una propiedad negativa inter-módulos que refleja el deseo de que los cambios realizados en un módulo afecten a otro módulo lo menos posible.

Las dos propiedades son divididas cada una en cuatro categorías, como se muestra en la Tabla 2.5, que a continuación se resumen:

Cohesión	Acoplamiento
Funcional	Flujo de Control
Flujo de Datos entre Acciones	Flujo de Datos Global
Lógica	del Entorno

Tabla 2.5: Subdivisión de la cohesión y el acoplamiento

- Cohesión funcional: resulta del diseño funcional de un módulo con una sola funcionalidad. Cuanto más definido está el objetivo de un módulo, mayor es su cohesión funcional. Además, es inversamente proporcional a la generalidad de los propósitos funcionales del módulo.

- Cohesión del flujo de datos: describe las interdependencias entre las diferentes sentencias de un módulo dependiendo del procesamiento de los datos. Para un dato dado, se dice que sigue un flujo de datos, cuando tras sufrir una transformación en una sentencia, debe utilizarse en otra sentencia. Entonces, ambas sentencias están cohesionadas por el flujo de datos.
- Cohesión entre acciones: cuando diversas acciones actúan sucesivamente sobre un mismo dato, se dice que hay cohesión entre dichas acciones.
- Cohesión lógica: es habitual que un bloque lógico de sentencias de un programa se ejecuten basándose en una condición lógica. Una estructura de bloque de este tipo se mantiene unida por la decisión lógica común que forma la base para la ejecución del bloque y, por tanto, se dice que entre las sentencias de dicho bloque existe una cohesión lógica.
- Acoplamiento del flujo de control: está causado por los parámetros de la interfaz que determinan la ejecución o se utilizan como código de error o diagnóstico.
- Acoplamiento del flujo de datos: mide la independencia entre dos módulos por causa de los parámetros de entrada y de salida que posee su interfaz.
- Acoplamiento global: es el acoplamiento que se produce entre módulos debido a la utilización de variables globales.
- Acoplamiento del entorno: mide la independencia entre módulos producida al llamar y ser llamado por otros módulos.

En este modelo, Dhama utilizó una medida del código por cada categoría. Su objetivo era construir un modelo cuantitativo de la cohesión y el acoplamiento y utilizarlo para evaluar dichas características de los módulos ya existentes. Todo ello quiere decir que es un modelo a posteriori que se aplica únicamente tras la implementación.

2.3. MEDIDAS DEL *SOFTWARE*

Un modelo de calidad que no incorpore medidas puede constituir un interesante estudio teórico, pero carecerá de utilidad alguna en la práctica. Para que un modelo de calidad se considere completo, debe estar sustentado por un amplio conjunto de medidas que permitan totalmente su evaluación para, de esta manera, poder analizar y estudiar detenidamente los resultados y poder extraer las conclusiones adecuadas. En el apartado anterior (2.2) se ha mostrado una serie de modelos de calidad, algunos de los cuales incluían medidas y otros no. En el presente apartado, se van a introducir unos conceptos generales acerca de las medidas, exponiendo también las medidas más significativas publicadas hasta la fecha.

Nadie discute como hecho evidente la espectacular evolución tecnológica que en los últimos años está experimentando la industria informática: el *hardware* incrementa su potencia mientras que su precio baja; al mismo tiempo aumenta la demanda de nuevas aplicaciones. El tamaño del *software*, por su parte, sigue evolucionando y creciendo, así como el esfuerzo destinado a su mantenimiento. Esto puede traer consigo una serie de problemas, puesto que debe tenerse en consideración que, cuanto más compleja sea la arquitectura de un sistema, hay más posibles formas de que algo vaya mal [Sloman, 98]. Habitualmente, la escena del desarrollo *software* se caracteriza por [Mills, 88]:

- estimaciones de tiempo y coste poco precisas,

- *software* con una pobre calidad,
- tasa de productividad que aumenta más lentamente que la demanda.

Esta serie de problemas son los que tradicionalmente se han englobado dentro de la llamada "crisis del *software*". Esta crisis del *software* debe afrontarse y, en la medida que sea posible, resolverse. Para hacerlo se requieren estimaciones de tiempo y coste precisas, productos de mayor calidad y mejor productividad. Todo esto puede conseguirse mediante una gestión más efectiva del desarrollo de los sistemas que, a su vez, puede facilitarse con la utilización efectiva de las medidas del *software*. No obstante, la gestión de proyectos no es plenamente efectiva pues el desarrollo *software* es extremadamente complejo y se tienen pocas medidas fiables y bien definidas, tanto del producto como del proceso, que guíen y evalúen el desarrollo. Así, la estimación, planificación y control preciso y efectivo resultan un objetivo casi imposible de alcanzar [Rubin, 83]. La mejora del proceso depende de la habilidad para identificar, medir y controlar los parámetros esenciales del proceso de desarrollo, y éste es el objetivo de las medidas del *software*.

Se han propuesto gran cantidad de medidas del *software* [Perlis, 81], aunque raramente se han empleado de una forma regular y metódica. Algunos estudios indican que la implementación y aplicación de un plan de medidas del *software* pueden ayudar a conseguir mejores resultados tanto en corto como en largo plazo [Grady, 87]. Por ello, la resolución de algunos de los problemas de la crisis del *software* pasa por una mejor utilización de las medidas existentes y el desarrollo de mejores medidas.

La mayor parte de las numerosas medidas definidas, surgieron para solucionar los problemas y mejorar los métodos para producir *software* de alta calidad [Mata-Toledo, 92]. Gran parte del trabajo en Ingeniería del *Software* ha reconocido que la reducción de costos y el incremento de la calidad son objetivos compatibles entre sí. Debido a la necesidad de conseguir una buena calidad, diversos estudios han centrado su atención en el desarrollo y validación de conjuntos de medidas. Estas medidas constituyen ayudas útiles en la gestión, herramientas importantes de diseño y elementos básicos en las investigaciones realizadas para mejorar el *software* [Henry, 81a].

Las medidas del *software* han sido un campo de investigación activo desde los años setenta, aunque los progresos para poner en práctica sus resultados se están produciendo muy lentamente. La estimación de proyectos es quizá la única medida puesta en práctica ampliamente debido a las demandas de los clientes. El lento progreso en la utilización en la práctica de las medidas del *software* se debe a que existe una diferencia entre lo que ofrecen los investigadores y lo que necesitan los gestores de proyectos. De hecho, las ideas de los investigadores tienen distintos motivos y criterios de éxito que los usuarios de esas ideas [Card, 91]:

- Desde el punto de vista de la investigación, una medida es apropiada si es lógica y se ha validado. Esto significa que se deriva de alguna teoría o modelo sobre una característica del *software* y que se ha demostrado que está altamente correlada con dicha característica. Claramente, las medidas permiten comprender la esencia del *software* que, después de todo, es lo que se quiere investigar.

- Por otra parte, el rol de las medidas en una organización es guiar a los gestores e ingenieros. Desde este punto de vista, una medida adecuada es aquélla que se utiliza en la práctica y se reconoce que contribuye a tomar mejores decisiones y a realizar acciones más eficientes (es decir, más beneficiosas).

Seguidamente, se van a proporcionar definiciones de medidas y métricas con el fin de aclarar las principales similitudes y diferencias entre estos conceptos.

Según el IEEE [IEEE, 93], puede definirse 'métrica' como una función cuyas entradas son datos del *software* y cuya salida es un valor numérico simple que puede interpretarse como el grado con el que el *software* posee un atributo dado que afecta a su calidad. Sin embargo, esta definición se aproxima más a la definición de medida (no se define el concepto de medida) que a la de métrica, tal como se ve a continuación.

Desde otro punto de vista, se define 'medida' como una forma perfectamente definida de asociar valores numéricos a los atributos de los elementos del *software* [Roberts, 79] y una 'métrica' sería un criterio para determinar la diferencia o distancia entre dos entidades [Ebert, 92]. En la literatura sobre el tema, frecuentemente se utilizan estos dos términos indistintamente, pudiendo causar una pequeña confusión al lector, por lo que aquí, en lo que sigue, se va a adoptar esta última definición, más de acuerdo con sus definiciones matemáticas [Mansfield, 63] [Ford, 93]:

Medida: Sea A un conjunto de objetos físicos o empíricos. Sea B un conjunto de objetos formales, como por ejemplo, números. Una medida μ se define como una aplicación $\mu: A \rightarrow B$.

Métrica: Sea A un conjunto de objetos, sea $\Re$ el conjunto de números reales y sea $m: A \otimes A \rightarrow \Re$ una medida ($\otimes$ denota el producto cartesiano). Entonces m es una métrica si y sólo si satisface las siguientes cuatro propiedades:

1. $m(\alpha, \beta) \geq 0, \forall \alpha, \beta \in A$
2. $m(\alpha, \beta) = 0 \Leftrightarrow \alpha = \beta$
3. $m(\alpha, \beta) = m(\beta, \alpha), \forall \alpha, \beta \in A$
4. $m(\alpha, \gamma) \leq m(\alpha, \beta) + m(\beta, \gamma), \forall \alpha, \beta, \gamma \in A$

Pero, ¿por qué medir? Los ingenieros miden para describir el estado del mundo en un instante determinado, para establecer unos requisitos cuantitativamente y demostrar su adecuación, para controlar el progreso y predecir los resultados del proyecto ingenieril y para analizar costes y beneficios [Ford, 93].

En general, la respuesta a la pregunta viene dada por la afirmación[3]: *You can't control what you can't measure* ("No se puede controlar lo que no se puede medir") [De Marco, 82]. Es decir, se mide porque se necesita conocer algo sobre un atributo o una característica que resulta importante [Mills, 90]. Una métrica es un medio para el

[3] Mucho antes, Lord Kelvin también había afirmado: "Si algo puede ser medido y expresado con números, entonces se sabe algo acerca de ello".

entendimiento de un objeto, considerando que se use una unidad estándar como base para realizar comparaciones entre estas características. Por ejemplo, en la vida cotidiana se usan litros, gramos y metros como métricas básicas para comparar medidas de la capacidad, peso y longitud de muchas cosas. De la misma forma, las medidas del *software* proporcionan una base para comparar los diversos aspectos del desarrollo del *software* necesarios para comprender y evaluar el proceso de desarrollo *software*. Una forma en que las medidas facilitan el entendimiento es haciendo más visibles ciertos aspectos del desarrollo, de tal manera que se pueden responder preguntas sobre la efectividad de las técnicas o herramientas, la productividad de las actividades o la calidad de los productos. Además, permite a los desarrolladores monitorizar los efectos de las actividades y los cambios en cualquier parte del desarrollo, de forma que se pueden tomar decisiones tan pronto como sea posible para controlar el producto final. Por tanto, las medidas son útiles para comprender, para establecer una base para mejorar y para planear, monitorizar y controlar productos, procesos y recursos. [Pfleeger, 95]

Sin embargo, hay que tener en cuenta, a la hora de realizar la implantación de un plan de medidas en un grupo de trabajo, la versión del principio de incertidumbre de Heisenberg[4] aplicada al cálculo de una medida: "La medida de cualquier parámetro de un proyecto y su asociación con una evidencia significativa influirá en la utilidad de dicha medida" [De Marco, 82]. Es decir, si por ejemplo se desea medir la productividad de un programador por el número de líneas de código que produce al día, entonces si dicha persona averigua esta medida, seguramente intentará (consciente o inconscientemente) que su código contenga gran cantidad de líneas, pudiendo llegar incluso a generar líneas en blanco, líneas de comentarios sin sentido o líneas de código inútil (simplemente copiando varias veces, en un lugar innecesario, código ya escrito), desvirtuando, por tanto, la medición. Este comportamiento es también conocido como el efecto Hawthorne[5] [Roesthlisberger, 39].

En resumen, se pueden emplear las medidas para predecir la calidad del producto *software* final así como para evaluar la calidad en cualquier punto del desarrollo [Bowen, 78]. También pueden usarse para detectar aquellas partes del sistema que son susceptibles de fallar [Emam, 01]. O, incluso, pueden utilizarse medidas para comparar sistemas con similares objetivos [Wieringa, 98].

En general, debe tenerse en cuenta el uso que se hace de los resultados de las medidas. Las mediciones proporcionan una ayuda que facilita una línea de razonamiento eficiente acerca del objeto medido, pero no se deben utilizar como una respuesta firme a una pregunta. Incluso las mediciones imperfectas pueden suministrar un entendimiento básico de la calidad de un sistema, las estrategias de diseño empleadas, la principal fuente de problemas, la evolución del sistema... [Briand, 99b].

---

[4] "Es imposible determinar simultáneamente la posición y velocidad exactas de un electrón" [Morcillo, 83].

[5] "La persona que está siendo estudiada ajusta su comportamiento porque está siendo estudiada, de tal forma que resulta imposible conseguir una medida válida". Efectivamente, en general, la gente que sabe que está siendo medida se comporta de una forma diferente que si no estuviera siendo medida [Austin, 97].

Por todo ello, el *software* debe medirse con medidas objetivas, fiables, válidas y adecuadas, pero no de forma subjetiva [Li, 87]. En [Gould, 81] se describe cómo unos científicos del siglo XIX calculaban la capacidad mental humana midiendo el volumen del cerebro. Estos mismos científicos "probaron", de esta forma, que las personas del norte de Europa eran más inteligentes que el resto y que los hombres tenían, en general, más inteligencia que las mujeres. Así, Gould critica no solo la medida sino su interpretación. Las medidas erróneas del *software*, que ha practicado la industria informática durante años, al menos no promueven el racismo (defendido por la medida errónea de la inteligencia descrita por Gould), pero engendra una ingeniería pobre y decisiones inadecuadas en el desarrollo del *software* [Byard, 94].

La idea de que el desarrollo del *software* constituye una ingeniería, está comúnmente aceptada desde hace más de dos décadas, lo cual ha llevado a intentos de predecir las propiedades de los productos y los procesos *software* mediante el uso de medidas. Su éxito se ha visto limitado por la ausencia de modelos adecuados, la dificultad de realizar experimentos controlados y repetibles y lo complicado que resulta comparar los datos obtenidos por distintos investigadores o de distintos sistemas [Churcher, 95b], aunque estos problemas no son específicos de las medidas [Fenton, 94b].

Aunque pueden evitarse algunas dificultades, la aceptación de las medidas como una disciplina de la Ingeniería del *Software* se ha visto afectada por la escasa calidad de algunos de los trabajos experimentales y la ausencia de estandarización de la terminología y las técnicas utilizadas [Churcher, 95b]. Como ejemplo, Churcher cita la escasa utilidad de la familia de medidas de la "Ciencia del *Software*" [Halstead, 77], limitada por la carencia de definiciones precisas de las fórmulas [Lassez, 81] [Shen, 83]. Por ello, debe tratarse de evitar repetir los errores del pasado conforme se avanza en un futuro con sistemas cada vez más complejos.

La amplia experiencia de Victor Basili (lleva trabajando en este campo más de 30 años) en la medida y evaluación de productos y procesos de la Ingeniería del *Software* en diferentes entornos se resume en una serie de principios de las medidas del *software*, en los que los cuatro primeros se centran en el propósito del proceso de medir y el resto se refieren a las medidas y al propio proceso de medir [Basili, 88]:

- Medir es un mecanismo ideal para caracterizar, evaluar, predecir y proporcionar motivación para los diversos aspectos de los procesos de construcción del *software*.
- Las medidas deben aplicarse tanto sobre el **proceso** *software* como en el **producto**.
- Debe estar claramente indicado el **propósito** de cada medida.
- Hay que ver las medidas desde la perspectiva apropiada.
- Se necesitan tanto medidas objetivas como subjetivas.
- La mayor parte de los aspectos del producto y del proceso *software* son demasiados **complicados** para ser identificados por una **única medida**.
- Los entornos de desarrollo y mantenimiento deben estar preparados para las medidas.
- La tarea de medir no se debe limitar a utilizar modelos y medidas tal como han sido definidas en otros entornos.
- El proceso de medida debe ser de arriba hacia abajo, en vez de ser de abajo hacia arriba, para poder definir un conjunto de objetivos operativos, especificar las

medidas apropiadas, permitir interpretaciones y análisis contextuales válidos y proporcionar **retroalimentación** para el aprendizaje y el seguimiento.
- Para cada entorno existe un conjunto característico de medidas que proporcionan la información necesaria con objetivos de definición e interpretación.
- Las medidas deben asociarse con interpretaciones, pero estas interpretaciones deben corresponder con un determinado contexto.
- Se necesitan múltiples mecanismos para la recopilación y validación de datos.
- Para evaluar y comparar proyectos y para llevar a cabo modelos se necesita una **base histórica** de experiencias.
- La base de experiencias debería evolucionar de una base de datos a una base de conocimientos (acompañada de un sistema experto) para formalizar la reutilización de la experiencia.

Tras haber presentado los principales conceptos básicos relacionados con las medidas, seguidamente, se expondrán algunos de los ejemplos más relevantes de medidas publicadas hasta la fecha, tanto para los paradigmas tradicionales como para el paradigma orientado a objetos, para después, finalmente, comentar determinadas clasificaciones de las medidas.

2.3.1. MEDIDAS TRADICIONALES

Numerosos autores han presentado, durante las últimas décadas, medidas que se basaban en el producto y que calculaban algún aspecto relacionado con la calidad de un programa. Evidentemente, en este trabajo no se van a poder describir las innumerables medidas desarrolladas. Como ejemplo, se van a destacar seguidamente algunos de los principales trabajos publicados en esta área y centrados en los lenguajes de programación tradicionales (procedimentales). No se comentan medidas empleadas en la estimación ni medidas dinámicas, puesto que se salen del objetivo del presente trabajo.

2.3.1.1. Ciencia del *Software* por Halstead

Puede decirse que uno de los primeros y más conocidos trabajos realizados en el campo de las medidas del *software* es el fruto de la investigación realizada por Maurice H. Halstead en la Universidad de Purdue, que se publicó en su conocido libro "*Software Science*" [Halstead, 77], en el que se describen una serie de medidas y estimaciones sencillas que pueden ser calculadas fácilmente sobre el código fuente de un programa. Tiene relación con los algoritmos y su implementación, tanto como programas informáticos como instrumentos de la comunicación humana. Trata sólo con aquellas propiedades de los algoritmos que pueden medirse, tanto directa como indirectamente, tanto estática como dinámicamente, y con las relaciones entre estas propiedades que permanecen invariantes en la traducción de un lenguaje a otro. Experimentalmente se detectaron relaciones que parecen gobernar la implementación de los algoritmos con respecto a su longitud, nivel, modularidad, pureza, volumen, contenido de inteligencia, el número de discriminaciones mentales… La notación utilizada, junto con las principales ecuaciones, se muestran, respectivamente, en la Tabla 2.6 y en la Tabla 2.7.

Estas medidas se idearon para su aplicación a posteriori sobre el *código fuente* de un programa. Como son medidas léxicas, la automatización resulta bastante sencilla construyendo un analizador léxico que cuente los tipos de *tokens* necesarios. De esta forma, se pueden evaluar los principales conceptos reflejados por las medidas: dificultad, esfuerzo y contenido de inteligencia.

D	Dificultad	V	Volumen del programa
E	Esfuerzo	V^*	Volumen potencial
I	Contenido de inteligencia	$\hat{V}$	Estimación del volumen del programa
L	Nivel del programa	η	Tamaño del vocabulario
N	Longitud del programa	η_1	Número de operadores únicos
N_1	Total de operadores	η_2	Número de operandos únicos
N_2	Total de operandos	η^*	Tamaño potencial del vocabulario
$\hat{N}$	Estimación de la longitud	η_1^*	Número potencial de operadores únicos
λ	Nivel del lenguaje	η_2^*	Número potencial de operandos únicos

Tabla 2.6: Lista de símbolos usados en la "Ciencia del *Software*"

$\eta = \eta_1 + \eta_2$	$N = N_1 + N_2$	$\eta^* = \eta_1^* + \eta_2^* = 2 + \eta_2^*$
$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$	$V = N \cdot \log_2 \eta$	$V^* = \eta^* \log_2 \eta^*$
$L = \frac{V^*}{V}$	$\hat{L} = \frac{\eta_1^*}{\eta_1}\frac{\eta_2}{N_2} = \frac{2}{\eta_1}\frac{\eta_2}{N_2}$	$E = \frac{V}{L} = \frac{V^2}{V^*}$
$I = \hat{L}V \cong V^*$	$D = \frac{1}{L}$	$\lambda = LV^* = L^2 V$

Tabla 2.7: Principales ecuaciones de la "Ciencia del *Software*"

Se ha escrito mucho, tanto a favor [Baker, 79] [Szulewski, 81] [Shen, 85] como en contra [Baker, 80] [Hamer, 82] [Ince, 89] [Shepperd, 94], sobre el trabajo de Halstead, pero sin duda ha sido el punto de partida para muchas de las investigaciones realizadas en este terreno [Gordon, 79a] [Woodfield, 79] [Christensen, 81] [Basili, 83b] [Bail, 88] [Ramamurthy, 88] [Shaw, 89] [Keller, 90] [Mehndiratta, 90] [Mayer, 92] [Khoshgoftaar, 94a] [Lanning, 94].

En la actualidad, las ecuaciones de la "Ciencia del *Software*", que están basadas en la teoría de la información, siguen siendo válidas, no sólo para los paradigmas tradicionales, sino que pueden adaptarse a otros paradigmas. En lugar de realizar las medidas para el programa completo, se puede realizar su cálculo por módulos. En cambio, estas medidas solamente son de aplicación tras la implementación, no pudiendo ser utilizadas en las fases anteriores del ciclo de vida.

2.3.1.2. Complejidad Ciclomática por McCabe

Un buen número de las medidas publicadas trata de medir la complejidad de un programa. Una de las primeras, y también una de las más referenciadas, es la complejidad ciclomática de Thomas J. McCabe [McCabe, 76] que pretendía obtener un valor numérico que midiera y controlara el número de caminos de ejecución posibles de un programa. Como esto podría resultar imposible para cualquier programa que

tuviera un salto atrás, se definió una medida en función de los caminos básicos, que tomados en conjunto, generarían cada camino posible.

La medida se basa en la teoría de grafos. Suponiendo que se tiene un grafo *G* en el que los nodos (*n*) representan bloques de código donde el flujo es secuencial y los arcos (*e*) corresponden a las bifurcaciones tomadas en el programa, la ecuación que determina la complejidad ciclomática de una grafo (que coincide con el número de zonas definidas por el grafo) es $v(G) = e - n + 2p$, siendo p el número de subgrafos inconexos. Un ejemplo puede verse en la Figura 2.14.

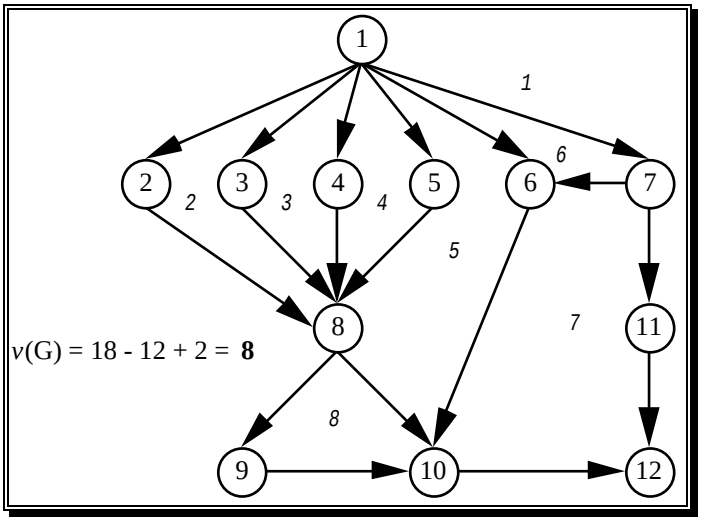


Figura 2.14: Ejemplo de cálculo de la complejidad ciclomática en un grafo

Sin embargo, este cálculo puede resultar realmente tedioso para un programador, por lo que McCabe demostró que dicha fórmula era equivalente a la dada por la ecuación $v(G) = \pi + 1$, siendo π el número de predicados del programa, lo cual es mucho más fácil de calcular e, incluso, de automatizar.

Inicialmente, la medida fue pensada para su aplicación sobre el *código fuente* de programas realizados con lenguajes de programación no estructurados, aunque su extensión a lenguajes estructurados sigue siendo perfectamente válida, incluso para los métodos de los lenguajes orientados a objetos. Además, la medida puede ser ampliada para ser aplicable también sobre el *diseño*, tal como el propio McCabe hizo unos años después [McCabe, 89].

La complejidad ciclomática ha sido igualmente muy comentada por distintos investigadores [Hall, 84] [Ramamurthy, 88] [Samadzadeh, 91] [Mayer, 92] [Henderson-Sellers, 94a] [Lanning, 94] [Takahashi, 97], así como ampliamente utilizada en la realización de un gran número de experimentos, tanto para contrastar los resultados [Hansen, 78] [Harrison, 81] como para refutarlos [Baker, 80] [Shepperd, 88] [Khoshgoftaar, 94a].

2.3.1.3. Complejidad del Control de Flujo por Woodward

Martin R. Woodward y sus colegas presentaron una medida para medir la complejidad del flujo y la ausencia de estructura de un programa [Woodward, 79]. Parte de la idea

de que las sentencias `goto` pueden llevar a programas complejos y poco estructurados, aunque en ciertos lenguajes, como FORTRAN, son imprescindibles. Así, en vez de simplemente contar el número de sentencias `goto` presentes en el programa, definieron el concepto de 'nudo'. Para ilustrar este concepto, lo más adecuado es fijarse en la Figura 2.15 que muestra un fragmento de programa con 4 nudos.

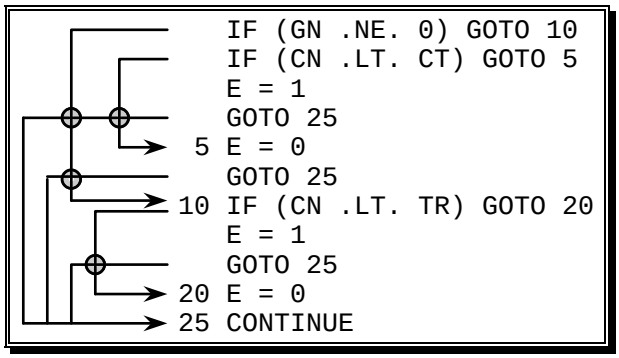


Figura 2.15: Cálculo de los nudos de un programa

La idea parte de dibujar una flecha que parta de cada `goto` y que finalice en el destino de dicho salto. Cada vez que una de estas flechas se cruza con otra se dice que hay un nudo. De una manera más formal (y menos ambigua), se define un nudo como sigue: si un salto de la línea *a* a la línea *b* se representa por el par ordenado de enteros *(a, b)*, entonces el salto *(p, q)* causa un nudo (o se cruza) con respecto al salto *(a, b)* si se cumple una de las dos condiciones:

1. mín (a, b) < mín (p, q) < máx (a, b) y máx (p, q) > máx (a, b)
2. mín (a, b) < máx (p, q) < máx (a, b) y mín (p, q) < mín (a, b)

La medida consiste, pues, en contar el número de nudos que se producen en un programa. Cuanto mayor sea este valor, el programa estará menos estructurado y tendrá un flujo de control más complejo.

Esta medida está pensada para su aplicación sobre el *código fuente* de un programa, por lo que no se podría evaluar en una fase anterior del ciclo de vida.

En la actualidad, con los modernos lenguajes de programación estructurados, que disponen de sentencias de control de flujo, la necesidad del uso de las sentencias de salto ha quedado eliminada, por lo que todo buen programador seguramente no utilizará en su vida un `goto`. Por ello, esta medida ha quedado desfasada, pudiéndose simplificar a meramente contar el número de saltos del programa, cuyo valor debería ser nulo.

2.3.1.4. Medidas del Diseño por Yin y Winchester

Se han definido diversas medidas con el fin de evaluar el diseño de un producto. Uno de los primeros trabajos en este sentido fue el conjunto de tres medidas presentado por Yin y Winchester [Yin, 78] [Yin, 79]. En sus medidas, los autores pretenden evaluar gráficos de diseño, como el presentado en la Figura 2.16. Estos gráficos están divididos

en una serie de niveles numerados desde el 0 (el gráfico de la figura tiene los niveles 0, 1, 2, 3 y 4). Para cada gráfico y para cada nivel, se calculan las siguientes tres variables:

N_i: número de módulos entre el nivel 0 y el nivel *i*
A_i: número de arcos del grafo entre los módulos del nivel 0 al nivel *i*
$T_i = N_i - 1$: número de arcos del árbol entre los módulos del nivel 0 al nivel *i*

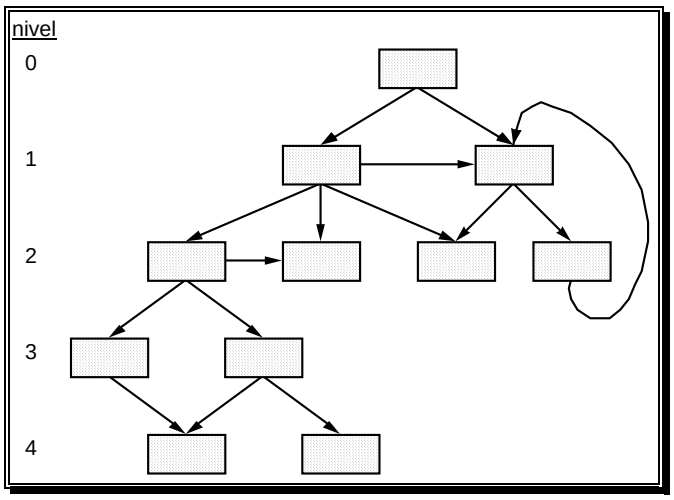


Figura 2.16: Grafo de ejemplo para las medidas de Yin

A continuación se evalúan las siguientes tres medidas, $\forall\ i \neq 0$:

$$C_i = A_i - T_i \qquad R_i = 1 - \frac{T_i}{A_i} = \frac{C_i}{A_i} \qquad D_i = 1 - \frac{T_i - T_{i-1}}{A_i - A_{i-1}}$$

La medida C_i, que es una función monótona no decreciente, indica la fluctuación de los incrementos de T_i frente a A_i. En otras palabras, mide la complejidad del grafo. Un fuerte incremento de C_i de un nivel con respecto al siguiente señala un incremento anormalmente alto de la complejidad y puede sugerir un lugar donde pueden aparecer problemas.

Las medidas R_i y D_i, ambas con valores entre 0 y 1, miden la impureza del árbol en el nivel *i* (el valor 0 representa una estructura ideal en forma de árbol). Es decir, estas medidas crecen conforme el diagrama se aleja de la estructura de árbol. La diferencia entre ambas medidas está en que R_i mide la impureza del árbol del nivel *i* frente al nivel 0, mientras que D_i mide la impureza del nivel *i* frente al nivel *i* – 1.

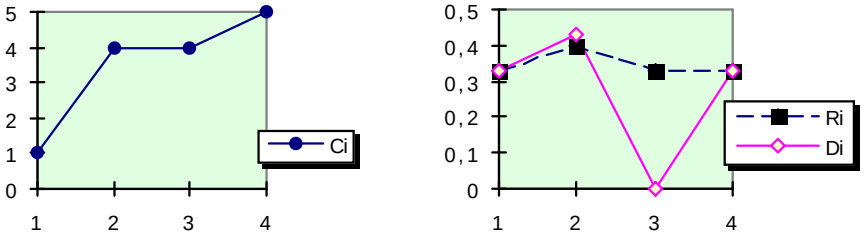


Figura 2.17: Gráficas que resumen los valores obtenidos para C_i, R_i y D_i en el ejemplo de la Figura 2.16

Todos estos valores se pueden representar en forma de gráfica, con lo que visualmente se pueden detectar de forma rápida los posibles problemas que podrían surgir. Así, las gráficas de la Figura 2.17 representan los valores de las tres medidas obtenidos para el ejemplo dado. Analizándolas, se puede detectar un fuerte crecimiento entre C_1 y C_2 (primera gráfica), lo que debería levantar las sospechas por un importante incremento de la complejidad, mientras que entre C_2 y C_3 no se pone de manifiesto el más mínimo problema. Por otra parte, analizando la segunda gráfica, se puede ver que el nivel 2 es el que más se aleja de la estructura de árbol, mientras que el tercero es el que más se acerca.

Si durante el *diseño* se obtiene un grafo de este tipo, las medidas pueden dar una idea somera de su complejidad y, lo que es más importante, ayudar a detectar las zonas sospechosas de ser causa de posibles problemas. Por tanto, las medidas pueden utilizarse a priori (antes de la implementación) para simplificar el diseño, lo cual constituye una importante innovación a la hora de crear medidas del *software*.

2.3.1.5. Complejidad por Oviedo

Enrique I. Oviedo presentó una medida para evaluar la complejidad de un programa basándose en dos submedidas, que miden la complejidad del flujo de control y del flujo de datos del programa [Oviedo, 80].

Para medir el flujo de control, se parte de la construcción, a partir del programa, de un grafo dirigido de control, en la que cada nodo representa un bloque básico (concepto definido en el apartado 2.2.2.3) y cada arista refleja que el control puede fluir del nodo origen al nodo destino (es decir, un organigrama de bloques básicos). De esta forma, la complejidad del flujo de control de un programa con un determinado grafo de control puede obtenerse calculando la cardinalidad del conjunto *E* de aristas del grafo:

$$CF = |E|$$

Para definir el concepto de complejidad del flujo de datos, que refleja que el uso de variables en un programa puede afectar significativamente a la comprensibilidad del programa, primero es necesario explicar una serie de conceptos.

Se llama 'definición' de una variable a cualquier declaración o asignación que se realice sobre dicha variable. Una 'referencia' a una variable ocurre cada vez que se utiliza.

Una definición de variable 'localmente disponible' para un bloque básico es cuando la variable se define en dicho bloque. Una referencia a una variable está 'localmente expuesta' en un bloque si se refiere a una variable no definida en dicho bloque.

Se dice que la definición de una variable en el bloque básico n_i alcanza al bloque n_k si la definición está localmente disponible en el bloque n_i y no lo está en ningún bloque del camino que une n_i con n_k.

Se definen los siguientes conjuntos:

D_i es el conjunto de definiciones localmente disponibles en el bloque n_i
P_i es el conjunto de definiciones que alcanzan n_i

A_i es el conjunto de definiciones de variables disponibles a la salida de n_i
R_i es el conjunto de definiciones de variables que alcanzan el bloque n_i

Entonces, para todos los bloques básicos n_j que son predecesores inmediatos de n_i:

$$R_i = \bigcup_j A_j \qquad A_i = P_i \bigcup D_i$$

Teniendo en cuenta que V_i es el conjunto de variables cuyas referencias están localmente expuestas en el bloque n_i, y que $DEF(v_j)$ representa el número de definiciones de la variable v_j disponibles en el conjunto R_i, la fórmula que mide la complejidad del flujo de datos en el bloque n_i es:

$$DF_j = \sum_{j=1}^{|V_i|} DEF(v_j)$$

Y, por tanto, la medida de la complejidad del flujo de datos es (siendo S el conjunto de todos los bloques básicos excepto el inicial):

$$DF = \sum_{i=1}^{|S|} DF_i$$

Una vez definidos estos dos tipos de complejidades, la complejidad del programa se define como:

$$C = \alpha \cdot CF + \beta \cdot DF$$

siendo α y β unos pesos apropiados definibles, que, en principio, pueden ser la unidad.

Como puede observarse, estas medidas son de aplicación tras la fase de *implementación* y estaban previstas inicialmente para el paradigma imperativo, aunque sin tener en cuenta las llamadas a subprogramas. Su ampliación a otros paradigmas no resultaría demasiado complicada, teniendo simplemente que redefinir algunos de los conceptos.

2.3.1.6. Estabilidad por Yau y Collofello

Yau y Collofello definieron una medida para evaluar la estabilidad lógica de un programa (véase el apartado 2.2.2.1) [Yau, 80]. Antes de obtener la medida final, es necesario definir una serie de conceptos preliminares.

El 'efecto de propagación' de una modificación representa todos los elementos del programa que se ven afectados directa o indirectamente por dicha modificación. Haciendo una comparación, realizar una modificación en el *software* es como lanzar una piedra sobre un estanque tranquilo: una serie de ondas se propagarán por toda su superficie, llegando incluso a afectar a todo el estanque.

El conjunto Z_{ki} contiene las variables de interfaz afectadas por el efecto de propagación como consecuencia de una modificación de la definición de la variable i en el módulo k. El conjunto X_{kj} consta de los módulos que influyen en la variable de interfaz j del módulo k. El conjunto W_{ki} contiene los módulos envueltos en la propagación de

cambios entre módulos como consecuencia de modificar la definición de la variable *i* del módulo *k*, y puede calcularse como sigue:

$$W_{ki} = \bigcup_{j \in Z_{ki}} X_{kj}$$

A continuación, se calcula la 'complejidad lógica de una modificación', que se obtiene para cada definición de variable *i* en cada módulo *k* y se denota mediante LCM_{ki}. Una forma de evaluar este valor es calcular la complejidad (usando cualquier medida de complejidad) de cada módulo *t* (C_t) y seguidamente:

$$LCM_{ki} = \sum_{t \in W_{ki}} C_t$$

Como la estabilidad lógica de un programa se define como la resistencia ante el potencial efecto de propagación lógica de una modificación de la definición de una variable *i* en otros módulos del programa, se debe determinar la probabilidad *P (ki)* de que se seleccione para modificar la definición de una variable *i* de un módulo *k*.

Ahora, se puede calcular el potencial 'efecto de propagación lógica' como una medida del impacto esperado en el programa de una modificación sobre el módulo *k* (V_k es el conjunto de todas las definiciones de variables en el módulo *k*):

$$LRE_k = \sum_{i \in V_k} \left[P(ki) \cdot LCM_{ki} \right]$$

Entonces, la medida para la 'estabilidad lógica' del módulo *k* puede calcularse así:

$$LS_k = \frac{1}{LRE_k}$$

Si ahora se desea medir el potencial 'efecto de propagación lógica' de una modificación sobre un programa, llamando *P (k)* a la probabilidad de que ocurra una modificación sobre el módulo *k* y siendo *n* el número de módulos del programa, se puede emplear la fórmula:

$$LREP = \sum_{k=1}^{n} \left[P(k) \cdot LRE_k \right]$$

Finalmente, la medida de la 'estabilidad lógica' de un programa será:

$$LSP = \frac{1}{LREP}$$

Como puede comprobarse, la medida de la estabilidad lógica resulta bastante complicada de calcular, interviniendo también otras medidas, como la complejidad. Esta medida está planteada para su aplicación a posteriori tras la *implementación*, es decir, no se puede utilizar para predecir la estabilidad en las anteriores fases del ciclo de vida.

2.3.1.7. Flujo de Información por Henry y Kafura

Sallie Henry y Dennis Kafura diseñaron una medida del flujo de información que tenía en cuenta toda la información que fluye a y desde un módulo [Henry, 81a]. La medida, de muy fácil cálculo, precisa solamente de tres definiciones previas:

La complejidad interna (C_{ip}) de un módulo *p* se define como el número de líneas de código (LOC) de dicho módulo, aunque podría mejorarse empleando la longitud de Halstead u otras medidas de complejidad, como la ciclomática.

El *fan-in* de un módulo *A* es el número de flujos de datos locales entrando al módulo más el número de estructuras de datos de las que el módulo *A* obtiene información.

El *fan-out* de un módulo *A* es el número de flujos de datos locales saliendo del módulo más el número de estructuras de datos que modifica el módulo *A*.

De esta forma, se define la complejidad del flujo de información de un módulo *p* (C_p) como:

$$C_p = C_{ip} \cdot (\text{fan-in} \cdot \text{fan-out})^2$$

La principal ventaja de esta sencilla medida es que es posible utilizarla tras la fase de *diseño* y, aunque estaba inicialmente pensada para los paradigmas tradicionales, cualquier paradigma que disponga de módulos (subprogramas) podría utilizarla.

2.3.2. MEDIDAS ORIENTADAS A OBJETOS

Aunque la Orientación a Objetos es como un río repleto de afluentes, la fuente de la programación orientada a objetos se remonta a finales de los años 60 con los trabajos de Dahl y Nygaard que sembraron la semilla con su introducción al lenguaje Simula67 y con el Smalltalk, desarrollado en los laboratorios de Xerox en Palo Alto en los años 70 [Archer, 95b]. La programación orientada a objetos no tuvo demasiado interés, considerándose como una curiosidad, hasta los años 80 [Page-Jones, 92], con trabajos como los desarrollados por [Booch, 86] [Cox, 86] y [Stroustrup, 88a]. En concreto, fue tomando importancia al ir convirtiéndose el C++ en uno de los lenguajes más empleados en el desarrollo de nuevos sistemas [Churcher, 95b]. El diseño y el análisis orientado a objetos hicieron sus primeras apariciones significativas a finales de los 80 [Booch, 91]. Durante la mayor parte de las dos primeras décadas de coexistencia entre la programación orientada a objetos y la estructurada, ha habido poca interacción entre los investigadores de ambos paradigmas.

La orientación a objetos constituye un paradigma de desarrollo del *software* de gran trascendencia para la producción de *software* eficiente y barato. Este paradigma, en el que el análisis, diseño y programación configuran las fases fundamentales del ciclo de vida de un sistema informático, ha sentado las bases para establecer la estructura metodológica de los años 90 y se presenta como paradigmática en el desarrollo de aplicaciones. [Alonso, 95]

La orientación a objetos, que consiste en una forma relativamente nueva de pensar en el *software* basándose en abstracciones existentes en el mundo real [Rumbaugh, 91], puede definirse como "una disciplina de ingeniería de desarrollo y modelado de *software* que permite construir más fácilmente sistemas complejos a partir de componentes individuales" [Khoshafian, 90].

Entre los muchos beneficios reconocidos de los sistemas orientados a objetos están un rápido desarrollo, mayor calidad, mantenimiento más sencillo, costes reducidos, fácil extensibilidad, mejores estructuras de información y un aumento de la adaptabilidad [Taylor, 90], además de mejorar substancialmente la productividad, aunque todos estos beneficios, que son comúnmente aceptados, deberían verificarse empíricamente [Lewis, 91]. Una de las principales razones para estos beneficios es que las aproximaciones orientadas a objetos controlan la complejidad de los sistemas permitiendo una descomposición jerárquica a través tanto de descomposiciones funcionales como de datos [Booch, 91].

Sin embargo, como Brooks afirma, la parte realmente dura del desarrollo *software* es la manipulación de su esencia debido a la inherente complejidad de los problemas, en lugar de deberse a las dificultades de su traducción a un lenguaje en particular [Brooks, 87]. El proceso de descomposición simplemente ayuda a controlar la complejidad inherente del problema, pero no elimina su complejidad [Tegarden, 95]. Los investigadores que trabajan en orientación a objetos coinciden en que la tarea de identificar los objetos y establecer la jerarquía de clases es complicada y subjetiva [Tervonen, 92].

Los sistemas orientados a objetos se caracterizan por tres propiedades fundamentales que son útiles para controlar la calidad del *software* [Fuertes, 93]:

- Abstracciones de datos y encapsulación: Los conceptos del mundo real se configuran como objetos que son instancias de clases. Éstas se describen como implementaciones de tipos abstractos de datos que permiten almacenar información y disponer de operaciones para su manipulación. Puede definirse, además, que solo ciertos elementos de dichas clases sean visibles desde el exterior, aumentando de esta forma la seguridad.
- Herencia: Es una propiedad basada en la jerarquía por la que una clase deriva de otra u otras clases más simples o generales, de las que hereda sus características. Con ello se permite la extensibilidad y reutilización del *software*.
- Polimorfismo: El polimorfismo (inicialmente descrito por Harland [Harland, 85]) consiste en la capacidad de que un objeto o elemento adopte diferentes formas. Ello posibilita definir distintos métodos con el mismo nombre, de tal manera que puedan realizar tareas diferentes, y que un objeto referencie el método apropiado en tiempo de ejecución y no en tiempo de compilación.

Como se puede ver, los sistemas orientados a objetos se diferencian de los convencionales en aspectos importantes, presentando algunas ventajas significativas. Aunque no se han producido demasiados progresos en el terreno de la calidad para este paradigma hasta la fecha, hay una necesidad creciente de técnicas que lleven a avances inmediatos, al menos para los lenguajes comunes, como C++, Smalltalk y Java.

Sin embargo, la creación de nuevas medidas se tropieza con que se encuentra disponible relativamente poca información sobre la estructura de los sistemas orientados a objetos e, incluso, los modelos cualitativos de calidad son difíciles de encontrar. [Churcher, 95b]

El desarrollo de nuevos modelos de calidad más fiables que incorporar a este paradigma pasa por cambiar las medidas del *software* tradicional, basadas muchas de ellas en el número de líneas de código como medida fundamental [Cámara, 96]. La gestión de un proyecto *software* orientado a objetos necesita apoyarse en un buen plan de medidas que tenga en consideración las diferencias entre un entorno orientado a objetos y un entorno tradicional, como por ejemplo, las diferencias en la arquitectura o el uso de la herencia para inter-relacionar las clases (la presencia de las estructuras de herencia no puede ser descrita por ninguna de las medidas existentes); esto conduce a la necesidad de diseñar un nuevo conjunto de medidas originales especialmente para la orientación a objetos [Henderson-Sellers, 96a] [Jones, 97a].

De esta forma, en la pasada década se han empezado a presentar diferentes medidas pensadas especialmente para la orientación a objetos, puesto que éstas, en general, tienden a ser distintas a las medidas del *software* tradicional [Berard, 98]. No obstante, todavía existen pocos trabajos en esta área y la mayoría de los estudios realizados se basan en el conjunto de seis medidas de Chidamber y Kemerer. Incluso se podrían incluir aquí trabajos previos aplicables a los objetos, como son aquéllos que incorporan medidas para los paquetes de Ada [Gannon, 86] [Rising, 92].

Dentro de los trabajos que incluyen otras medidas para algún aspecto de los desarrollos orientados a objetos o trabajos acerca de las medidas en este paradigma podrían citarse: la estimación del tamaño del sistema [Laranjeira, 90], medidas de la complejidad cognitiva [Cant, 94], las medidas como ayuda para el mantenimiento [Wilde, 92], medidas de la reutilización [Bieman, 95b] [Pant, 96] [Barnard, 98], medidas de la complejidad [Shih, 97]… Por otra parte, también existen algunos trabajos que miden distintos aspectos de la calidad, como medidas del sistema: adaptando las medidas tradicionales [Barnes, 93] o medidas del diseño, tamaño, complejidad, reutilización, productividad y calidad [Abreu, 94]…

Seguidamente se presentan algunas de las medidas más significativas para el paradigma orientado a objetos.

2.3.2.1. Conjunto de Medidas por Chidamber y Kemerer

Shyam R. Chidamber y Chris F. Kemerer presentaron al principio de los noventa un conjunto de medidas para su aplicación en diseños orientados a objetos [Chidamber, 91] [Chidamber, 94]. Estas medidas se resumen a continuación:

- WMC (*Weighted Methods per Class*): Consiste en la suma ponderada de la complejidad de cada uno de los métodos de una clase. Para su cálculo, considérese una clase *C* con sus métodos $M_1, M_2..., M_n$. Sea $c_1, c_2..., c_n$ la complejidad de cada método. Entonces, para dicha clase *C*:

$$WMC = \sum_{i=1}^{n} c_i$$

Los autores dejan libre la elección de una medida para la complejidad de cada método.

- DIT (*Depth of Inheritance Tree*): Mide la profundidad de una clase *C* en el árbol de herencia. En el caso de que exista herencia múltiple, la medida DIT será la máxima longitud desde dicha clase hasta llegar a la raíz. DIT es una medida de cuántas superclases pueden afectar potencialmente a cada clase.

- NOC (*Number Of Children*): Es el número de clases subordinadas inmediatamente a una clase dentro de la jerarquía de herencia de clases. Está relacionada con la noción del alcance de los miembros y es una medida de cuántas subclases heredarán los métodos de la clase padre.

- CBO (*Coupling Between Object classes*): La medida CBO de una clase es la cantidad de clases a la que está acoplada (número de objetos que actúan sobre otro). Está relacionada con la noción de que un objeto está acoplado con otro si uno de ellos actúa sobre el otro, es decir, si los métodos de uno usan métodos o atributos del otro.

- RFC (*Response For a Class*): Es el conjunto de métodos de una clase que potencialmente pueden ejecutarse como respuesta a la llegada de un mensaje recibido en un objeto de esa clase. De esta forma, para una clase *C*:

$$RFC = M \bigcup_{\forall i} R_i$$

siendo *M* el conjunto de todos los métodos de la clase *C* y R_i el conjunto de los métodos llamados por el método *i*.

- LCOM (*Lack of Cohesion Of Methods*): Proporciona una medida de la relativa disparidad natural de los métodos de una clase. Para su cálculo, considérese una clase *C* junto con sus *n* métodos $M_1, M_2..., M_n$. Sea I_i el conjunto de los atributos usados por el método M_i, por lo que se tendrán *n* conjuntos: $I_1, I_2..., I_n$. Sean también los conjuntos:

$$P = \{ (I_i, I_j) \ / \ I_i \cap I_j = \emptyset \}$$
$$Q = \{ (I_i, I_j) \ / \ I_i \cap I_j \neq \emptyset \}$$

Entonces:

$$LCOM = \begin{cases} |P| - |Q| & \text{si } |P| > |Q| \\ 0 & \text{otro caso} \end{cases}$$

Como se acaba de mencionar, existen diversos trabajos que toman como punto de partida estas seis medidas. Algunos de ellos son críticas a algunas de estas medidas o a la forma de calcularlas [Churcher, 95a] [Hitz, 96] [Etzkorn, 99] [Lakshminarayana, 99] [Wilkie, 00], mientras que otros las emplean en sus propias investigaciones [Li, 93a] [Sharble, 93] [Li, 95] [Stiglic, 95] [Basili, 96] [Schach, 96] [Etzkorn, 98] [Briand, 00].

2.3.2.2. Medidas del Diseño por Chen y Lu

Chen y Lu presentaron un conjunto de medidas con el objetivo de evaluar un diseño orientado a objetos [Chen, 93]. Las medidas diseñadas son:

- Complejidad de las operaciones (métodos): mide la complejidad de los métodos de una clase. Sea *O (i)* el valor de la complejidad del método *i*, que se puede evaluar subjetivamente a partir de la Tabla 2.8.

  La complejidad de las operaciones se define entonces como:

$$C = \sum O(i)$$

Clasificación de la complejidad	Valor de complejidad
nula	0
muy baja	1-10
baja	11-20
normal	21-40
alta	41-60
muy alta	61-80
extra alta	81-100

Tabla 2.8: Valores de la complejidad de los métodos

- Complejidad de los argumentos de las operaciones: mide la complejidad de los parámetros de los métodos de una clase. Sea *P (i)* el valor del argumento *i* en cada método. Este valor puede obtenerse a partir de la Tabla 2.9.

Tipo	Valor
lógico o entero	0
carácter	1
real	2
vector o matriz	3-4
puntero	5
registro, estructura u objeto	6-9
fichero	10

Tabla 2.9: Valores de la complejidad de los tipos de datos

  La complejidad de los argumentos se define entonces como:

$$C = \sum P(i)$$

- Complejidad de los atributos: mide la complejidad de los atributos de una clase. Sea *R (i)* el valor del atributo *i* en cada método. Este valor puede obtenerse a partir de la Tabla 2.9. La complejidad de los atributos se define entonces como:

$$C = \sum R(i)$$

- Acoplamiento de las operaciones: mide el acoplamiento existente entre los métodos de una clase y los métodos de otras clases. Se define como la suma del número de

métodos que acceden a otras clases más el número de métodos que son accedidos desde otras clases más el número de métodos que cooperan con otras clases.

- Acoplamiento de las clases: mide el acoplamiento existente entre una clase y las otras clases. Se define como la suma del número de accesos a otras clases más el número de accesos desde otras clases más el número de clases que cooperan.

- Cohesión: mide lo relacionados que se encuentran los argumentos de los métodos de una clase. Suponiendo que una clase tiene *n* métodos, tendrá también *n* conjuntos de argumentos. Sea *m* el número de conjuntos disjuntos formados por la intersección de los *n* conjuntos de argumentos. La medida de la cohesión de la clase vendrá dada por:

$$C = \frac{m}{n} \cdot 100\%$$

- Jerarquía de clases: es una medida de la complejidad de la jerarquía de clases. Para una clase dada, esta medida se calcula sumando la profundidad de la clase en el árbol de herencia, el número de sus subclases, el número de sus superclases directas y el número de métodos locales o heredados disponibles en dicha clase.

- Reutilización: mide si la clase se está reutilizando. Esta medida valdrá 1 si la clase se está reutilizando a partir del proyecto actual o de uno previo. En otro caso, la medida valdrá 0.

Como puede observarse, este conjunto de 8 sencillas medidas tienen algunas dificultades para su utilización en la práctica de manera automática: algunas de ellas tienen un alto grado de subjetividad mientras que otras no están normalizadas, lo que puede hacer difícil su interpretación.

2.3.2.3. Medidas para Pruebas por Binder

El objetivo perseguido por Robert V. Binder para la realización de las medidas era utilizarlas para que sirvieran de apoyo y de guía durante la fase de pruebas de un sistema desarrollado utilizando la tecnología orientada a objetos [Binder, 94b]. Para ello, recopiló una serie de medidas, como las de Chidamber y Kemerer, mientras que otras fueron realizadas por él mismo. A continuación se resumirán aquellas medidas que no hayan sido ya explicadas con anterioridad.

Las medidas utilizadas fueron clasificadas según su objetivo:

- Encapsulación: se emplean tres medidas:

  1. LCOM (*Lack of Cohesion Of Methods*).
  2. PAP (*percent Public And Protected*): mide el porcentaje de atributos públicos o protegidos (en C++) e indica la proporción de información de la clase visible. Un alto valor indica más oportunidades para efectos de lado entre clases.
  3. PAD (*Public Access to Data members*): mide el número de accesos externos a atributos públicos o protegidos (C++), es decir, mide el número de violaciones de la encapsulación.

- Herencia: se incluyen cuatro medidas:

    1. NOR (*Number Of Root classes*): mide el número de jerarquías de herencia empleadas en un programa.
    2. FIN (*Fan In*): indica el número de clases de las que hereda otra clase. Un valor de FIN superior a la unidad solo es posible con herencia múltiple. Un valor elevado de esta medida incrementa la posibilidad de enlaces incorrectos.
    3. NOC (*Number Of Children*).
    4. DIT (*Depth of Inheritance Tree*).

- Polimorfismo: se utilizan cuatro medidas:

    1. OVR (*percent of non-OveRloaded calls*): proporción de llamadas en el sistema que no se realizan a módulos sobrecargados.
    2. DYN (*percent of DYNamic calls*): proporción de mensajes en el sistema cuyo destino se determina en tiempo de ejecución.
    3. Bounce-C: cuenta el número de caminos 'yo-yó' visibles para la clase. Un camino 'yo-yó' es un camino que atraviesa varias jerarquías de clases debido al enlace dinámico.
    4. Bounce-S: cuenta el número de caminos 'yo-yó' en el sistema.

- Complejidad: para este objetivo se usan seis medidas:

    1. WMC (*Weighted Methods per Class*).
    2. RFC (*Response For a Class*).
    3. CBO (*Coupling Between Objects*).
    4. CLC (*CLass Complexity*): representa la complejidad ciclomática del grafo formado por la unión de todos los grafos de control de los métodos que tengan un grafo de transición de estados.
    5. NOM (*Nominal number Of Methods per class*): cuenta el número de funciones y procedimientos definidos en la clase. Se sugiere que el límite superior aceptable para esta medida es de 20.
    6. TOM (*Total number Of Methods per class*): igual que NOM, pero incluyendo también los métodos heredados.

Todas estas medidas se utilizan para estimar la complejidad y la cantidad de casos de prueba necesarios para validar el sistema. El objetivo de estas medidas es su utilización en la fase de pruebas, por lo que están pensadas para ser aplicadas sobre el *código fuente*, aunque muchas de ellas también podrían emplearse en fases anteriores del ciclo de vida. Además, pueden emplearse para obtener una indicación de la calidad del sistema.

Es de destacar que éste es de los pocos trabajos en el que se clasifican las medidas de acuerdo al atributo de calidad que miden, si bien, la cantidad de medidas presentadas y el número de atributos utilizados son bastante escasos.

2.3.2.4. Cohesión y Reutilización por Bieman y Karunanithi

El trabajo de James M. Bieman y Santhi Karunanithi se centró en medir la cohesión y la reutilización en la orientación a objetos, llegando a obtener dos medidas para cada uno de estos conceptos [Bieman, 95a]. Antes de ver las medidas es necesario definir unos conceptos:

Se dice que dos métodos están 'directamente conectados' si existe uno o varios atributos que son usados por ambos métodos. Se llama *NDC (C)* al número de conexiones directas de la clase *C*.

Dos métodos que están conectados a través de otros métodos directamente conectados se dice que están 'indirectamente conectados'; es decir, la relación de conexión indirecta es el cierre transitivo de la relación conexión directa. Se llama *NIC (C)* al número de conexiones indirectas de la clase *C*.

Dada una clase *C*, *NP (C)* será el número total de pares de métodos en la clase o, dicho de otra forma, es el máximo número posible de conexiones directas e indirectas en dicha clase. Si hay *n* métodos en la clase *C*:

$$NP(C) = \frac{n \cdot (n-1)}{2}$$

- TCC (*Tight Class Cohesion*): es una medida de la cohesión fuerte entre clases mediante métodos directamente conectados. Su cálculo está basado en la obtención del número relativo de métodos directamente conectados:

$$TCC(C) = \frac{NDC(C)}{NP(C)}$$

- LCC (*Loose Class Cohesion*): es una medida de la cohesión débil entre clases mediante métodos directamente o indirectamente conectados. Su cálculo se basa en la obtención del número relativo de métodos directamente o indirectamente conectados:

$$LCC(C) = \frac{(NDC(C) + NIC(C))}{NP(C)}$$

- Reutilización por instanciación: mide el hecho de reutilizar una clase instanciando objetos en otras clases. Se calcula contando el número de clases donde la clase dada se instancia.

- Reutilización por herencia: mide el hecho de reutilizar una clase heredando de ella. Se calcula contando el número de clases que heredan de la clase dada, es decir, el número de descendientes, tanto directos como indirectos.

Estas medidas están diseñadas originalmente para su aplicación sobre el *código fuente*, aunque su extensión a la fase de diseño resulta totalmente factible.

2.3.2.5. Complejidad del Código y el Diseño por Etzkorn, Bansiya y Davis

Los autores presentaron una medida para evaluar la complejidad del código, partiendo de la idea de la medida WMC de Chidamber y Kemerer, y otra medida para evaluar el diseño orientado a objetos, basándose en la teoría de la información [Etzkorn, 99]. Estas medidas se definen como sigue:

- AMC (*Average Method Complexity*): esta medida pretende medir la complejidad media de los métodos de una clase. Sea c_1, c_2..., c_n la complejidad (calculada con cualquier medida de complejidad) de los n métodos. Entonces, para dicha clase:

$$AMC = \frac{1}{n}\sum_{i=1}^{n} c_i$$

- CDE (*Class Design Entropy*): es una medida del contenido de información de una clase que evalúa la entropía media del diseño de la clase y da una indicación de su complejidad. Definiendo n como el número de operadores únicos (sin repetición), N como el número total de operadores y f_i como la frecuencia de aparición del operador i, la medida queda:

$$CDE = -\sum_{i=1}^{n} \frac{f_i}{N} \log_2\left(\frac{f_i}{N}\right)$$

Aunque AMC está pensada para medir la complejidad de los métodos sobre el *código fuente*, también podría utilizarse durante el *diseño*, empleando para ello una medida para la complejidad del diseño de los métodos. Análogamente, CDE también podría ser utilizada tras la *implementación*.

2.3.2.6. Ley de Demeter por Lieberherr

Por último, es de destacar una medida ampliamente referenciada denominada la "Ley de Demeter" [Lieberherr, 88] que surgió originalmente como una medida del buen estilo de los programas orientados a objetos. No obstante, después se ha utilizado como medida de la bondad de un programa (en concreto, de la encapsulación y modularidad) contando el número de veces que se viola dicha ley [Lieberherr, 89], llegándose a convertir en una útil medida. La Ley de Demeter tiene el siguiente enunciado:

> "Para todas las clases *C* y todos los métodos *M* de *C*, todos los objetos a los que *M* envía mensajes deben ser:
> 1. Los objetos que son argumentos de *M* (incluyendo el propio objeto de la clase *C*)
> 2. Los objetos que son atributos de *C*."

Como se ha comentado, la medida refleja la cantidad de veces que esta ley no se cumple, obteniéndose el resultado óptimo cuando el valor de la medida sea 0. El interés de esta medida se centra, además, en que es una de las primeras medidas ideadas para la orientación a objetos que se han publicado y utilizado.

2.3.3. CLASIFICACIONES DE LAS MEDIDAS

Debido a la gran cantidad de clases de medidas existentes, en la literatura pueden encontrarse diversos tipos de clasificaciones generales de las medidas. Seguidamente se van a comentar brevemente las principales clasificaciones, tanto para las medidas tradicionales como para las orientadas a objetos

2.3.3.1. Clasificaciones de las Medidas Tradicionales

Entre todas las clasificaciones de las medidas convencionales desarrolladas hasta la fecha, se van a destacar a continuación cuatro de ellas.

1. La primera clasificación de las medidas del *software* está basada en los elementos que se miden y se concentran principalmente en tres grandes grupos [Henry, 81a] [Henry, 81b]:

- **Medidas léxicas**: Muchos estudios han explorado medidas basadas en el contenido léxico de un programa, es decir, basadas en contar diversos *tokens* léxicos sin tener en cuenta las construcciones o estructuras creadas por dichos *tokens*. El primer tipo de trabajo efectuado en este campo está basado en el código del programa y la mayor parte de los trabajos publicados se basan en la investigación acerca de la "Ciencia del *Software*" [Halstead, 77]. La segunda tendencia se centraba en estudiar el flujo de control de un programa en función de un número que, en cierta forma, evalúa su estructura. Su principal exponente fue la complejidad ciclomática de McCabe [McCabe, 76], que supuso un importante paso adelante con respecto a las medidas de código puras, pues las medidas del flujo de control pueden ser (en teoría) obtenidas a partir del diseño detallado de un sistema [Ince, 90]. Este tipo de medidas léxicas simples ha probado ser sorprendentemente robusto a la hora de predecir diversos aspectos de la calidad del *software* [Henry, 81a].

- **Medidas semánticas**: El segundo gran campo de investigación lo constituyen las medidas semánticas que se basan en la aplicación de los conceptos de la teoría de la información (como la entropía) a la formulación de las medidas del *software*. Esta aproximación surgió a partir del trabajo de Alexander [Alexander, 64] en el campo de la arquitectura y el diseño. Channon [Channon, 74] fue el primero que usó esta idea para analizar el diseño de las estructuras *software*. Aunque estas técnicas se apoyan en una base teórica subyacente, no se ha probado que sean tan útiles en la práctica como las medidas léxicas [Henry, 81a].

- **Medidas de conectividad**: El tercer gran tipo de medidas son las de conectividad, que miden el grado de interconectividad entre los componentes del sistema observando el flujo de información entre ellos. El trabajo de Yin y Winchester [Yin, 78] se centraba en la interfaz entre los principales niveles de un gran sistema estructurado jerárquicamente. También entran en esta categoría los trabajos de Henry [Henry, 79] con una aproximación más detallada que el anterior, pues observa todo el flujo de información en lugar de solo el que circula entre los distintos niveles.

Pero probablemente, uno de los campos de investigación en medidas más prometedoras (que Henry no contempló) incluye las medidas del *diseño* del sistema.

Estas medidas se pueden obtener durante el diseño y pueden medir el grado de aislamiento de los módulos de un sistema. La justificación para estas medidas se basa en que un buen sistema es aquél en el que los módulos pueden leerse y probarse aisladamente e integrarse sin problemas [Ince, 90]. Algunos trabajos significativos en este campo son los de Yau y Collofello [Yau, 85] y Kafura y Reddy [Kafura, 87].

2. Una segunda clasificación se limita a dividir las medidas del *producto* en tres tipos que miden distintos aspectos de la calidad [Kafura, 85]:

- **Medidas de la estructura**: Este tipo de medidas se encuentra basado en el análisis de la estructura del diseño. Por ello, pueden calcularse temprano en el ciclo de vida (tras el diseño). Algunas medidas que se encuadrarían dentro de este tipo son la complejidad de invocación [McClure, 78], el flujo de información [Henry, 79] y la estabilidad [Yau, 80].

- **Medidas del código**: Estas medidas están basadas solamente en los detalles del código fuente. Por ello, solo son de aplicación tras la fase de implementación. Algunos ejemplos de medidas de esta clase son las presentadas por [McCabe, 76] [Halstead, 77].

- **Medidas híbridas**: Son aquellas medidas que son una combinación de las dos anteriores, es decir, que se basan tanto en detalles de la implementación como en la estructura del diseño. Por ello, solamente pueden evaluarse una vez finalizada la implementación. Unos ejemplos de medidas híbridas son la complejidad del flujo de información [Henry, 90a] y las definidas en [Kafura, 85], basadas en determinar una medida para un componente ponderando una medida de la estructura de dicho componente por una medida del código.

3. La tercera clasificación está basada en las entidades que se miden con cada medida. Estas entidades pueden ser [Fenton, 91]:

- El **proceso**: Se mide cualquier actividad relacionada con el desarrollo del *software*. Este tipo de medidas es el que más escasea en la literatura.

- El **producto**: Se miden los documentos, programas y cualquier elemento que se entregue como resultado del proceso. Este es el tipo de medidas que más abundan, como las ya citadas [McCabe, 76] [Halstead, 77] [Henry, 81a] [Yau, 80] [Kafura, 85].

- Los **recursos**: Se miden los elementos que constituyen la entrada a los procesos. Este tipo de medida es bastante habitual, aunque las más corrientes suelen centrarse en medir tiempos o costes (por ejemplo, para encontrar y solucionar un error).

En estas tres entidades se puede realizar una subdivisión en dos tipos de atributos [Fenton, 94a]:

- Atributos **internos**: son aquéllos que pueden ser medidos en los propios términos del producto, proceso o recursos.

- Atributos **externos**: son aquéllos que sólo pueden ser medidos con respecto a cómo el producto, el proceso o el recurso se relaciona con otras entidades de su entorno.

La Tabla 2.10 resume estos cinco conceptos, dando un ejemplo para cada uno de los tipos de medidas que se pueden presentar.

	proceso	producto	recursos
atributos internos	nº de fallos descubiertos durante las pruebas	líneas de código (LOC)	experiencia del programador
atributos externos	estabilidad del proceso	fiabilidad	productividad del personal

Tabla 2.10: Clasificación de las medidas por Fenton, con ejemplos

4. La última clasificación se basa en qué parte del desarrollo *software* se quiere medir [Barnes, 93]:

- **Medidas basadas en el objetivo**: Comparan los requerimientos del *software* con su funcionamiento, es decir, determinan lo bien que el *software* cumple con sus requisitos funcionales. La especificación de estas medidas no es trivial y requiere una cooperación entre los clientes, los usuarios y la plantilla que desarrolla el sistema. Como ejemplo, pueden citarse las medidas de Gilb de diseño por objetivos [Gilb, 88] o cualquier medida cuyo fin sea evaluar conceptos como el mantenimiento, la fiabilidad, la utilizabilidad...

- **Medidas basadas en la sintaxis o medidas estáticas**: Estas medidas examinan el código fuente y proporcionan estadísticas de lo bien que está implementado. Son las medidas más comunes y tratan de determinar lo difícil que resulta entender un programa. Se basan en la idea de que ciertas medidas del código pueden constituir una predicción del esfuerzo necesario para el diseño, implementación, modificación y mantenimiento del sistema. Algunos ejemplos lo forman las medidas ya explicadas de [McCabe, 76] [Halstead, 77]. La mayoría de los modelos de estimación de costos se basan, en parte, en la estimación del esfuerzo de desarrollo derivado del tamaño esperado del código (que se obtiene usando este tipo de medidas), como en [Boehm, 81].

- **Medidas basadas en la ejecución o medidas dinámicas**: Miden las características del *software* en ejecución. Pueden medirse los rendimientos y la eficiencia del *software* con respecto a la velocidad y requisitos de memoria, así como los fallos detectados. A menudo suelen usarse las construcciones del lenguaje (aserciones, excepciones...) para obtener valores a utilizar en las medidas (por ejemplo, número de excepciones ocurridas por semana). La medida del tiempo medio hasta el fallo (MTTF) [Hamilton, 78], muy empleado en los componentes físicos, constituye un ejemplo de este tipo de medidas.

2.3.3.2. Clasificaciones de las Medidas Orientadas a Objetos

Aunque para el paradigma orientado a objetos las clasificaciones del apartado anterior también pueden emplearse, se han publicado un par de taxonomías de las medidas del

software orientado a objetos, que incorporan las principales características y propiedades de este tipo de *software*. Seguidamente se resumen estas dos clasificaciones:

1. La primera taxonomía, llevada a cabo por Abreu y Carapuça [Abreu, 94], trata de las medidas del producto y del proceso. Para realizar la clasificación, se basan en el producto cartesiano de dos vectores: (diseño, tamaño, complejidad, reutilización, productividad, calidad) y (método, clase, sistema). Esto produce una matriz con 18 celdas en las que pueden incluirse medidas. Dicha matriz, con algunos ejemplos de medidas, se muestran en la Tabla 2.11.

	Método	Clase	Sistema
Diseño	Porcentaje de atributos usados Densidad de comentarios	LCOM [1] DIT [1] RFC [1]	Tamaño medio de los métodos Nº medio de métodos por clase Nº medio de atributos por clase
Tamaño	Nº de sentencias ejecutables Nº de operadores y operandos [2] Nº de atributos usados	Nº de métodos Nº de atributos Tamaño de la interfaz de la clase	Nº de clases Nº de métodos Nº de atributos
Complejidad	Complejidad ciclomática [3] Volumen [2] Flujo de información [4]	NOC [1] Nº de subclases Nº de superclases	DIT [1] CBO [1]
Reutilización	Nº de veces que se hereda el método Eficiencia de reutilización del método [5]	Porcentaje de métodos heredados sobrecargados Nº de veces que se reutiliza la clase tal cual Nº de veces que se reutiliza la clase adaptándola	Porcentaje de clases reutilizadas tal cual Porcentaje de clases reutilizadas con adaptación Factor de calidad de la librería de clases [6]
Productividad	Esfuerzo para construir un método medio Métodos nuevos desarrollados por unidad de esfuerzo	Esfuerzo para construir una clase media Nuevas clases desarrolladas por unidad de esfuerzo	Esfuerzo medio para construir una clase Clases reutilizadas adaptadas por unidad de esfuerzo
Calidad	Fiabilidad del método Nº de defectos encontrados por unidad de tiempo Eficiencia de mantenimiento	Fiabilidad de la clase Nº medio de defectos por método Nº medio de fallos por método	Efectividad de las pruebas Tiempo medio entre fallos Tiempo medio de aprendizaje

1: [Chidamber, 94]	2: [Halstead, 77]
3: [McCabe, 76]	4: [Henry, 81a]
5: [Abreu, 94]: Sea n_1: número de veces que se hereda un método. Sea n_2: número de veces que se sobrecarga un método. La eficiencia de reutilización del método es: $\frac{n_1 - n_2}{n_1}$	6: [Abreu, 94]: Sea p: número de clases de la librería. Sea n_i: número de veces que se reutilizó la clase i. El factor de calidad de la librería es: $\sum_{i=1}^{p} \frac{n_i}{p}$

Tabla 2.11: Taxonomía de las medidas orientadas a objetos con ejemplos

2. La segunda taxonomía, realizada por Archer y Stinson [Archer, 95a] presenta las propiedades del *software* orientado a objetos jerárquicamente, empezando por las

características de mayor nivel y descendiendo hacia las de menor nivel. Estas características son:

- **Sistema**: Aunque un sistema puede subdividirse en componentes, se ven los componentes actuando como un todo. Además, las características de un buen componente son las de un buen sistema, y viceversa. Ejemplos de medidas del sistema son: la estimación del tamaño [Laranjeira, 90] y el factor de calidad de la librería de clases [Abreu, 94].

- **Acoplamiento y usos**: Las clases a menudo interactúan con otras clases para formar un subsistema. Las características de esta interacción pueden indicar una complejidad resultante proveniente de un acoplamiento excesivo o por usar objetos derivados de otros objetos y así sucesivamente. El acoplamiento y los usos son conceptos relacionados, por lo que deben mostrarse dentro del mismo grupo. Algunos ejemplos de este grupo son: acoplamiento de las operaciones y los métodos [Chen, 93] y las violaciones de la Ley de Demeter [Lieberherr, 89].

- **Herencia**: Las relaciones entre una clase y sus parientes en una jerarquía de herencia pueden indicar al diseñador los lugares donde realizar cambios para mejorar el desarrollo. Medidas de ejemplo, son: DIT y NOC [Chidamber, 94] y la medida de la jerarquía de clases [Chen, 93].

- **Clase**: La clase está compuesta por métodos, y puede incluir algunos innecesarios o demasiado complejos. Puede tener, además, datos extraños que complican el proceso de la implementación. Algunos ejemplos son: WMC, RFC y LCOM [Chidamber, 94], la complejidad de las operaciones, la medida de cohesión y la de reutilización [Chen, 93] y el número de veces que se reutiliza una clase tal cual o adaptándola [Abreu, 94].

- **Método**: Los métodos y atributos aparecen en el último nivel de detalle. Mientras que los atributos son fácilmente comprensibles, los métodos son a menudo desarrollados usando la programación estructurada. Los métodos tienen la complejidad añadida de realizar llamadas a objetos distintos al objeto que los contienen. Algún ejemplo es: LOC, la complejidad ciclomática [McCabe, 76] y las medidas de la "Ciencia del *Software*" [Halstead, 77] aplicadas a los métodos.

2.4. LA MEJORA DEL PROCESO *SOFTWARE*

Uno de los principales objetivos de las investigaciones expuestas en apartados anteriores, conducentes a medir la calidad o sus atributos, consiste en conseguir disminuir las dificultades que surgen en el desarrollo de un sistema *software* y su gestión y seguimiento a lo largo del ciclo de vida. Esto se traduce en mejorar el proceso *software*, aspecto de considerable preocupación en la comunidad de ingenieros del *software*, lo que se demuestra por la gran cantidad de trabajos que se están llevando a cabo en esta área, en paralelo con los trabajos sobre medidas y calidad.

Las mejoras en el proceso *software* se pueden identificar monitorizando, midiendo y revisando el rendimiento del proceso estándar al ser aplicado sobre proyectos individuales. Es importante asegurarse que todo el personal implicado en el proceso

software participe en las actividades de mejora y es esencial mantenerles informados. Resulta primordial fijar y perseguir objetivos cuantitativos y medibles para mejorar el proceso, dirigiendo estos objetivos a perfeccionar la calidad del producto. Además, debería establecerse un programa de mejora del proceso que impulsara a los desarrolladores a mejorar su propio proceso de trabajo y a participar en las mejoras de sus compañeros. [Sanders, 94]

Es fundamental preparar y mantener planes de mejora del proceso para toda la organización. Estos planes deberían definir los recursos necesarios, identificando las áreas de mayor prioridad y especificando claramente las metas perseguidas. Para ello resulta imprescindible tomar mediciones del rendimiento del proceso *software* estándar y analizar estas medidas, pues es esencial usar el análisis para ajustar en proceso con vistas a mejorarlo y estabilizar su rendimiento dentro de unos límites aceptables. [Sanders, 94]

Los principales esfuerzos en la mejora del proceso *software* dieron comienzo en 1987. Aunque el progreso en este campo está siendo lento, los resultados más interesantes se obtuvieron por la Universidad Carnegie Mellon, a través de su Instituto de Ingeniería del *Software* [Humphrey, 95b], que resulta puntera en estos trabajos, habiendo llevado a cabo el *Capability Maturity Model* y el *Personal Software Process*, que se comentan brevemente a continuación, haciendo hincapié en el terreno de las medidas del *software*.

2.4.1. MEDIDAS Y EL CMM

Evidentemente, algunos procesos de desarrollo de *software* se encuentran más evolucionados que otros. Una forma de discriminar entre niveles de madurez del proceso es la habilidad de los desarrolladores y gestores para ver y entender qué es lo que ocurre durante todo el proceso de desarrollo. En el nivel inferior de madurez, no se comprende ni el proceso, pero conforme crece la madurez, se comprende y se puede definir mejor. Tanto las medidas como la habilidad para ver y entender están íntimamente relacionadas, puesto que un desarrollador puede medir sólo lo que resulta visible en un proceso y las medidas ayudan a incrementar esa visibilidad. [Carleton, 92]

El CMM (*Capability Maturity Model*) puede servir como una guía para determinar qué medir primero y cómo planear un plan de medidas comprensivo y adecuado [Humphrey, 89] [Paulk, 91] [Weber, 91]. En [Baumert, 92a] se describe el uso de las medidas del *software* en este contexto.

La Tabla 2.12 [Pfleeger, 90a] resume los tipos de medidas sugeridas por los distintos niveles del CMM. La elección de medidas específicas depende de cada nivel y de la información obtenible. Las medidas del nivel 1 proporcionan las bases para realizar comparaciones. Las medidas en el nivel 2 se centran en la planificación y el seguimiento del proyecto. Las medidas del nivel 3 se dirigen hacia los productos finales e intermedios producidos durante el desarrollo. Las medidas del nivel 4 capturan las características del propio proceso de desarrollo para permitir el control de las actividades individuales del proceso. Y, finalmente, en el nivel 5 los procesos son lo suficientemente maduros y se gestionan con tal cuidado que permiten que las medidas

proporcionen una retroalimentación para modificar el proceso de forma dinámica a lo largo de diferentes proyectos. [Carleton, 92]

Nivel de Madurez	Características	Objetivo de las Medidas
1. Inicial:	Ad hoc, caótico	Establecer las bases para planear y estimar
2. Repetible:	Los procesos dependen de los individuos	Seguimiento y control del proyecto
3. Definido:	Los procesos se definen y se institucionalizan	Definición y cuantificación de los procesos y productos intermedios
4. Gestionable:	Se miden los procesos	Definición, cuantificación y control de los subprocesos y los elementos
5. Optimizado:	Los procesos se retroalimentan con las mejoras	Optimización dinámica y mejora durante el proyecto

Tabla 2.12: Tipos de medidas del CMM

Cada plan de medidas debería comenzar con un examen del modelo de proceso *software* actualmente en uso y una determinación de lo que es visible. Las medidas nunca deberían seleccionarse por el nivel de madurez general. Si una parte de un proceso es más madura que otras, las medidas confeccionadas pueden mejorar la visibilidad de esas partes y ayudar a conseguir los objetivos generales del proyecto mientras que las medidas básicas pueden llevar al resto del proceso hasta un nivel de madurez superior.

La evidencia sugiere que los buenos programas de medidas van creciendo conforme a las necesidades y objetivos de la organización [Rifkin, 91]. Un plan de medidas debería comenzar afrontando los problemas u objetivos críticos de cada proyecto, desde el punto de vista de lo que es significativo al nivel de maduración del proceso de la organización. El marco de madurez actúa entonces como guía para expandir y construir un plan de medidas que no solo tome ventaja de la visibilidad y la madurez sino que también realce las actividades de mejora del proceso. [Carleton, 92]

2.4.2. EL PROCESO *SOFTWARE*

Algunas organizaciones han afrontado el problema de desarrollar grandes sistemas adoptando el concepto de la definición del proceso y realizando una gestión apropiada [Humphrey, 89]. Gestionando adecuadamente el proceso *software*, tanto en pequeños proyectos de laboratorio como en proyectos reales de gran tamaño, estas organizaciones han mejorado satisfactoriamente las capacidades de sus grupos de desarrollo [Dion, 93].

El 'proceso *software*' es la secuencia de pasos necesarios para desarrollar y mantener el *software*. Una 'definición del proceso *software*' es una descripción de este proceso. Cuando se diseña una definición adecuada, ésta guía a los ingenieros en su trabajo. Más específicamente, el proceso *software* establece el marco de trabajo técnico y de gestión para aplicar los métodos, herramientas y personal a la tarea informática, mientras que la definición del proceso identifica los roles y especifica las tareas. La definición del proceso, además, establece medidas y proporciona los criterios de comienzo y finalización para cada paso. También ayuda a asegurarse de que se ha asignado adecuadamente cada trabajo y que se controla su estado. Proporciona, por tanto, un mecanismo apropiado para el aprendizaje: conforme se descubren nuevos métodos, se

incorporan en la definición del proceso de la organización. Una definición del proceso de este tipo permite que cada nuevo proyecto se construya basándose en la experiencia de sus predecesores. [Humphrey, 95a]

El PSP (*Personal Software Process*) es un proceso de perfeccionamiento diseñado para ayudar a controlar, gestionar y mejorar la forma de trabajar. Es un marco de trabajo estructurado con formularios, guías y procedimientos para desarrollar *software*. Adecuadamente utilizado, el PSP proporciona los datos históricos necesarios para mejorar el proceso. Su principal objetivo es conseguir mejores ingenieros del *software*. Permite comprender por qué se han cometido errores y cuál es la mejor forma de encontrarlos. Se puede determinar la calidad de las revisiones, los tipos de errores no detectados y los métodos más efectivos para cada ingeniero. [Humphrey, 95a]

El PSP está dividido en cuatro grandes fases, que se muestran en la Figura 2.18, denominadas PSP0, PSP1, PSP2 y PSP3, aunque las tres primeras van seguidas de unas fases intermedias de mejora, denominadas PSP0.1, PSP1.1 y PSP2.1. Seguidamente, se resumen las distintas fases del PSP.

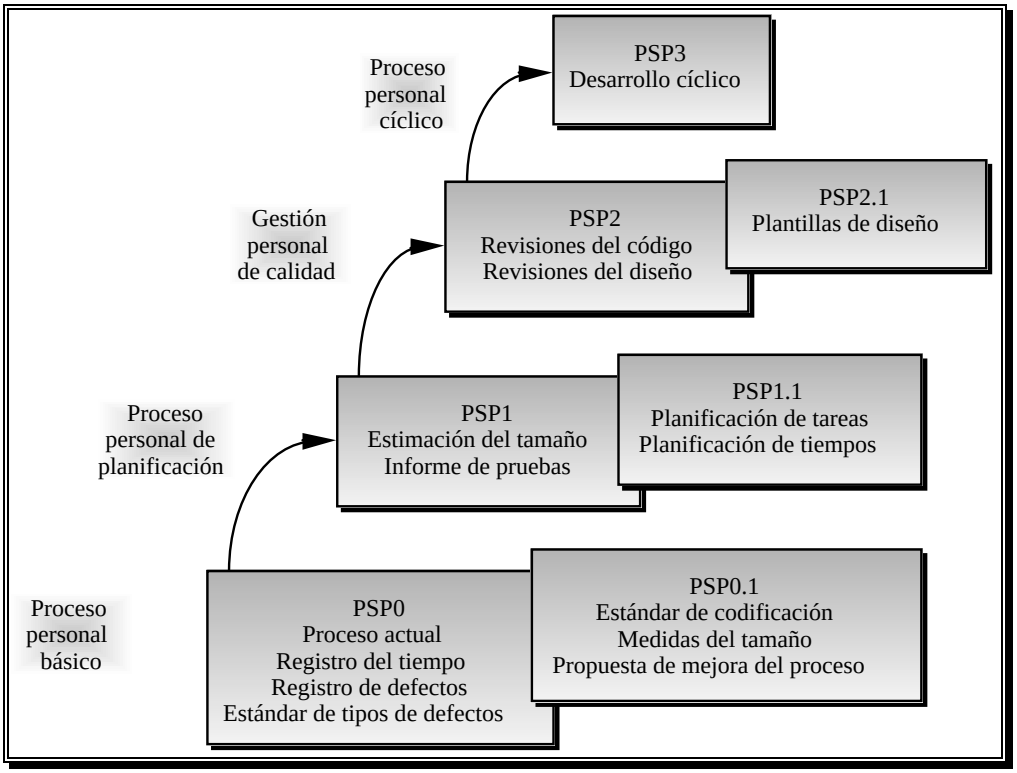


Figura 2.18: Evolución del PSP

- **PSP0: proceso básico**

Los objetivos del primer paso del PSP son incorporar medidas básicas en el proceso de desarrollo *software*, requerir mínimos cambios en las prácticas personales, demostrar el uso de la definición de un proceso y usar el proceso actual como una marco de proceso inicial.

Las medidas utilizadas en esta etapa se limitan a contar el tiempo empleado en cada fase del desarrollo así como el número de defectos encontrados por fase y almacenar los datos para un ulterior estudio.

El PSP0.1 mejora al PSP0 incorporando un estándar de codificación, medidas y estimaciones del tamaño de los programas (usando el número de LOC) y la obtención de la propuesta de mejora del proceso (PIP: *Process Improvement Proposal*). El PIP se usa para almacenar cualquier problema o sugerencias de mejora que ocurren al usar un proceso

- **PSP1: proceso de planificación**

El PSP1 pretende establecer un procedimiento ordenado y repetible para llevar a cabo estimaciones del tamaño del *software* (se continúa utilizando las LOC, tanto para medir como para estimar, pero intentando mejorar la estimación mediante nuevas técnicas), así como estimación de recursos y un informe de pruebas.

El PSP1.1 presenta la realización de planes de recursos y tiempo, el seguimiento del rendimiento en comparación con estos planes y la estimación de la fecha de finalización de un proyecto. Las estimaciones se mejoran y amplían para cubrir también el tiempo de desarrollo y una planificación precisa del proyecto.

- **PSP2: proceso de gestión de calidad**

Los objetivos del PSP2 son introducir revisiones del código y del diseño e introducir métodos para evaluar y mejorar la calidad de estas revisiones. Entre estos métodos, se incluyen la aplicación de medidas para medir el propio proceso de la revisión, e incluyen tiempo de revisión y número de defectos detectados durante la revisión o tras ésta.

El PSP2.1 pretende ayudar a reducir el número de defectos en el diseño, proporcionar criterios para determinar si el diseño está completo y proporcionar un marco de trabajo consistente para verificar la calidad de los diseños. En este estado, las nuevas medidas se limitan a evaluar el coste de esta calidad, viendo el tiempo que se dedica a las nuevas tareas introducidas por el PSP.

- **PSP3: proceso cíclico**

Por último, el principal objetivo del PSP3 consiste en extender, progresivamente, la capacidad personal para desarrollar programas de hasta varios miles de líneas de código. Se hace hincapié en la fase de pruebas. Se propone un desarrollo cíclico, de tal forma que cada ciclo sería esencialmente un proceso PSP2.1 que produce la parte del desarrollo de cada uno de los ciclos.

2.5. CONCLUSIÓN DEL ESTADO DE LA CUESTIÓN

A modo de resumen, las principales conclusiones que pueden extraerse de todo lo expuesto en el presente capítulo, en relación con los objetivos de este trabajo, se exponen a continuación:

1. Existen *algunos* **modelos de calidad**:

   - Todos se basan en dividir la calidad en una serie de atributos, normalmente en varios niveles.

- El que mayor aceptación ha tenido y el más utilizado es el desarrollado por McCall [McCall, 77].

- Se han propuesto diversos cambios sobre este modelo de calidad.

- Todos los modelos de calidad se centran en los paradigmas de programación tradicional, no existiendo ninguno desarrollado específicamente para el paradigma orientado a objetos.

2. Los modelos de calidad existentes son **incompletos** por alguna razón básica:

   - Muchos no incluyen medidas, incluyen un número muy reducido de medidas o bien están incorrectamente definidas.

   - Muchas veces los modelos no son aplicables a todas las fases del ciclo de vida del desarrollo del *software*.

   - La mayoría no incluyen normas de cálculo y evaluación de las medidas ni normas de agregación de los valores por el árbol o grafo de calidad, por lo que pierden utilidad práctica.

   - Habitualmente no incorporan ningún método de aplicación del modelo de calidad.

3. Aunque se ha realizado mucho trabajo acerca de las **medidas** del *software*, se constata que:

   - Un importante número de las medidas desarrolladas se centra en los paradigmas tradicionales, desconociéndose si dichas medidas son de utilidad para otros paradigmas más modernos.

   - La mayoría de las medidas solamente pueden aplicarse sobre el código fuente del programa, es decir, son medidas a posteriori, que no pueden utilizarse a priori, en las fases anteriores del ciclo de vida, para dirigir por el buen camino el desarrollo del sistema.

   - Algunas medidas son demasiado complejas, en contra de la simplicidad de las medidas sugerida por [Rosenberg, 99].

   - En la última década se han llevado a cabo bastantes medidas para el paradigma orientado a objetos, aunque muchas se limitan a "contar" (número de métodos, número de clientes de una clase, tamaño de los métodos... [Bieman, 95b]).

   - Habitualmente las medidas no se encuentran normalizadas, por lo que sus valores pueden ser de lo más variado, dificultándose así su interpretación.

   - En numerosas ocasiones no se relaciona la medida con un atributo de la calidad del *software*, por lo que no queda claro cuál es su objetivo o qué pretenden medir [Alonso, 99] y, por supuesto, no se encuentran ligadas a ningún modelo de calidad.

- Ha quedado puesto de manifiesto la necesidad de crear medidas útiles [Buckley, 89] aplicables a atributos de la calidad. Además, las diferencias de arquitecturas de los programas orientados a objetos frente a los tradicionales conllevan la ineludible necesidad de construir medidas originales especialmente diseñadas para el paradigma orientado a objetos [Henderson-Sellers, 96a].

Todas las cuestiones aquí planteadas son de vital importancia a la hora de definir un nuevo modelo de calidad completo para el paradigma orientado a objetos y deben tenerse en consideración para que pueda tener validez y ser de utilidad práctica.

El 4 de junio de 1996, la lanzadera europea Ariane 5 explotó tras 40 segundos de vuelo, debido a un error de un fragmento de código innecesario. El software *culpable pertenecía al sistema de referencia inercial (SRI): antes del despegue, hay que hacer ciertos cálculos para alinear el SRI, aunque estos cálculos pueden finalizar 9 segundos antes del despegue. Pero como existe la posibilidad de que la cuenta atrás se detenga, los ingenieros decidieron mantenerlo hasta 50 segundos después del despegue, puesto que reiniciar el SRI llevaba varias horas. Una vez en vuelo, los cálculos del SRI son inútiles, pero en el Ariane 5 causaron una excepción que no fue interceptada y... ¡boom! La excepción se debió a una conversión de un valor real de 64 bits (inclinación horizontal) a un entero de 16 bits. El problema es que el valor a convertir no pudo ser almacenado en 16 bits, pero, como no había un manejador de excepciones por omisión, la excepción siguió el destino de todas las excepciones no capturadas, es decir, abortar el* software *y, por tanto, los ordenadores de a bordo y, por tanto, la misión. ¿Por qué no se controló la excepción? Los análisis habían demostrado que el* overflow *no podía ocurrir nunca. Lo cual era completamente cierto para las trayectorias del Ariane 4, para el cual había sido diseñado el SRI, no para las nuevas trayectorias del Ariane 5, para el que se había reutilizado el código. Una reutilización que provocó que un error trivial causara la explosión de un cohete de unos 500 millones de dólares.*

[Jézéquel, 97]

3. HIPÓTESIS DE TRABAJO

3. HIPÓTESIS DE TRABAJO

La orientación a objetos constituye una forma de pensar en los problemas usando modelos organizados en torno a conceptos del mundo real. El desarrollo de aplicaciones utilizando el paradigma de orientación a objetos afecta a todas las fases del ciclo de vida.

La orientación a objetos ofrece una serie de características que la diferencian notablemente de otros paradigmas. Hay que tener en cuenta, además, que se ha asentado como el paradigma de la década de los noventa para el desarrollo de sistemas informáticos y ha ido tomando una creciente importancia al potenciar la producción de programas con una serie de características que posibilitaban mejorar su calidad [Booch, 94] según la opinión de gran número de autores. Esta supuesta mejora de la calidad puede llevar consigo una importante disminución del coste del *software*. Esto se debe a que disminuye el tiempo necesario para desarrollar el producto (debido, por ejemplo, a la reutilización y extensibilidad del *software*), se reducen considerablemente los problemas que se presentaban durante la fase de mantenimiento (debido, entre otras propiedades, a su modularidad con bajo acoplamiento) y se facilita la ampliación o inclusión de mejoras [Schulmeyer, 90].

Pero para poder aprovechar al máximo estas posibilidades es necesario saber aplicar coherentemente esta tecnología. No por el simple hecho de usar las técnicas de la orientación a objetos se garantiza un programa con una buena calidad. Hay que emplearlas correctamente y siguiendo una metodología adecuada basándose en un modelo de calidad orientado a objetos [Alonso, 98b].

Por todo ello, cuando se obtiene un producto *software* con orientación a objetos, interesa saber qué calidad presenta. Deben, además, contestarse, entre otras, a las siguientes preguntas:

- ¿Cuándo se considera que el *software* tiene una buena calidad?
- ¿Qué características influyen al determinar la calidad del *software*?
- ¿Cómo medir la calidad del *software*?

Numerosos autores durante el último cuarto de siglo se han planteado estas o parecidas preguntas y algunos han proporcionado algunas respuestas. Pero debido a que la orientación a objetos es un paradigma relativamente joven, muy pocos autores se han planteado estas cuestiones para aplicarlas directamente en este nuevo campo.

Por otra parte, es fácil comprender que medir la calidad del *software* no es una tarea sencilla. Una razón para esto es que un algoritmo puede ser implementado de diversas maneras y no siempre está claro qué forma es la mejor. Otra razón estriba en que la complejidad y las interacciones presentes en los desarrollos de las aplicaciones de gran tamaño se incrementan de un modo no lineal respecto al tamaño de la aplicación. A pesar de estas y otras dificultades, el progreso realizado en la medida de la calidad del *software* se debe, principalmente, al uso de técnicas propias de la ingeniería del *software*. Mediante la utilización de las metodologías de desarrollo,

ciertos aspectos de la calidad pueden medirse de forma más objetiva, dividiendo la calidad en una serie de factores [Cavano, 78].

Partiendo de todos estos hechos, el presente trabajo tiene como objetivo principal el **definir un modelo de calidad que permita medir la excelencia del *software* construido utilizando las técnicas de la orientación a objetos**, determinando los distintos atributos que influyen en la calidad. Es decir, el modelo construido deberá incorporar una **división de la calidad en los distintos atributos del *software*** que permitan definirla completamente. De esta forma **se obtendrá un nuevo modelo específico y particular adaptado a este paradigma**. Para ello es necesario un estudio previo de los modelos de calidad para los paradigmas tradicionales existentes en la actualidad, con el fin de determinar su adaptación o no al paradigma orientado a objetos.

Sin embargo, una división de la calidad en sus atributos de calidad no será suficiente. Un modelo de calidad no está completo si no se incluyen una serie de medidas que posibiliten evaluar cada uno de dichos atributos con el fin de determinar un valor de calidad global. Hay que recordar que las medidas tradicionales no son suficientes para su aplicación al paradigma orientado a objetos debido a las nuevas abstracciones de datos y de control, como las clases y los objetos, el paso de mensajes, la herencia y el polimorfismo paramétrico, dinámico y de sobrecarga [Chung, 97]. Por ello, al mismo tiempo, se pretende **proporcionar una amplia familia de medidas** que permitan calcular de forma cuasi objetiva dicha calidad para este paradigma. Cada una de estas medidas, que medirá un aspecto particular y concreto del *software* desarrollado, se deberá estudiar minuciosamente, con el fin de determinar si verdaderamente influye en la calidad del *software* y, en caso afirmativo, en qué atributos. El proceso a seguir consistirá en estudiar en primer lugar las principales medidas existentes en la bibliografía, tanto para paradigmas tradicionales como para el paradigma orientado a objetos. Seguidamente habrá que determinar cuáles de dichas medidas pueden aplicarse sin modificación alguna, cuáles necesitarán algún tipo de modificación y cuáles no son válidas para este paradigma. Además, deberán definirse las nuevas medidas que sean necesarias para contemplar el mayor número de situaciones posibles. Por último, será necesario estudiar cada una de las medidas obtenidas con el fin de determinar si realmente miden algún atributo de la calidad del *software*, relacionándolas, en ese caso, con dicho atributo o desechándolas en caso contrario, para terminar realizando una última depuración de las medidas, buscando posibles redundancias o errores.

No obstante, no se pretende proporcionar un modelo de calidad cerrado de utilidad universal. Más bien, lo que se intenta es aportar un marco de trabajo abierto para que cada ingeniero del *software* pueda optimizar y ajustar finamente tanto el modelo como las medidas a su entorno de trabajo, de tal manera que se acomode plenamente a la forma de trabajar de cada equipo de desarrollo. Debe tenerse en cuenta que cada grupo de trabajo va a tener sus propias normas y costumbres, y lo que a unos les puede parecer bueno a otros, quizás, les puede parecer no tan bueno.

Por otra parte, tampoco resulta suficiente disponer de un modelo de calidad junto con sus medidas asociadas. Resulta imprescindible disponer también de un **método que**

permita aplicar el modelo de calidad a los desarrollos *software* orientados a objetos. De esta forma, los ingenieros del *software* que sigan este método podrán estudiar los resultados obtenidos, lo que les posibilitará mejorar tanto el proceso *software* personal como el corporativo.

En el caso de algunas de las medidas del producto que, evidentemente, serán dependientes del código fuente, para facilitar la implementación práctica de las pruebas, se ha decidido seleccionar el lenguaje orientado a objetos C++ [ISO, 98] ideado por Stroustrup [Stroustrup, 86] [Ellis, 91] [Stroustrup, 97], sin perder por ello generalidad (puesto que pueden extrapolarse fácilmente a otros lenguajes de programación orientados a objetos). La elección de este lenguaje ha sido debido a su gran popularidad y a que incorpora todas las características propias y exclusivas de la orientación a objetos.

Para la validación del modelo y de la familia de medidas se aplicará el modelo sobre un conjunto de desarrollos orientados a objetos. Para ello, se utilizará un prototipo informático diseñado a tal efecto que permita su ejecución sobre otros productos *software* con el fin de determinar su calidad, contrastando los resultados obtenidos con los resultados esperados emitidos por algún especialista en orientación a objetos. Estos experimentos permitirán determinar si es necesario algún tipo de ajuste en el modelo construido.

Recapitulando, los principales objetivos a cumplir con el presente trabajo pueden resumirse en los siguientes puntos:

- Definición de un **modelo de calidad** para las distintas fases del ciclo de vida de un producto *software* desarrollado mediante técnicas de orientación a objetos, de tal forma que la calidad quede dividida en diferentes atributos.

- Definición de un **conjunto de medidas** orientadas a objetos para este modelo de calidad, definiendo nuevas medidas y adaptando algunas de las existentes.

- **Relacionar** cada una de las **medidas** obtenidas con el **atributo** de calidad apropiado para completar el modelo.

- Diseño de un **método** que guíe la **aplicación** del modelo de calidad durante el ciclo de vida de un desarrollo *software* orientado a objetos.

Cuenta lo que es contable,
mide lo que es medible,
y, lo que no sea medible,
hazlo medible.

— Galileo Galilei

4. Solución Propuesta

4. SOLUCIÓN PROPUESTA

4.1. INTRODUCCIÓN

Dentro de este capítulo se expondrá detalladamente la solución que se ha construido para la situación planteada en el anterior capítulo.

En primer lugar, se expondrá el **modelo de calidad** especialmente diseñado para el paradigma orientado a objetos, incluyendo la descomposición de la calidad en factores y criterios e insistiendo especialmente en las principales novedades que presenta frente a los modelos tradicionales.

En segundo lugar, se explicarán concisamente las **medidas** diseñadas para el desarrollo orientado a objetos, estableciendo su utilidad dentro del modelo de calidad propuesto.

Por último, se detallará cómo, a partir del modelo de calidad y las medidas explicadas, se puede **mejorar el proceso de desarrollo *software*** tanto en el ámbito personal como corporativo.

4.2. MODELO DE CALIDAD PARA LA ORIENTACIÓN A OBJETOS

Tras el estudio minucioso de los modelos de calidad existentes hasta la actualidad y sus distintos atributos, que se han resumido en el Estado de la Cuestión, se ha podido determinar que el esquema que más se adecua al objetivo perseguido es el sugerido por McCall, que, además, ha sido también adoptado por ISO [ISO, 91] [ISO, 97] para crear su estándar de calidad. La principal razón para esta afirmación estriba en el hecho de que, como ya se ha comentado, este esquema se basa en descomponer la calidad en una serie de factores y, a su vez, estos factores se descomponen en una serie de criterios. En último lugar, a cada criterio se le asocia una o varias medidas (Figura 4.1). Además, los modelos de calidad del *software* aceptados tienden a ser multidimensionales y jerárquicos [Littlefair, 01].

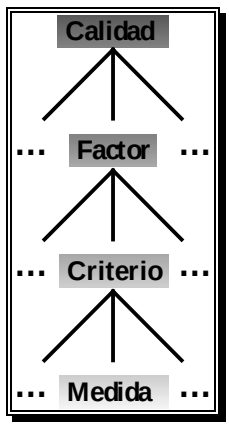


Figura 4.1: Estructura del modelo de calidad

Es evidente que la calidad es un concepto altamente complejo y, por tanto, difícil de evaluar directamente. Con esta forma de dividir la calidad, lo que se está consiguiendo

es simplificar el problema original: el nuevo problema consiste en evaluar los criterios, que al representar conceptos considerablemente más simples, resultan más fáciles de valorar. Este esquema no es más que un reflejo de lo que afirmaba Lao Tzu:

> Afronta las dificultades, mientras todavía es fácil.
> Soluciona los grandes problemas, cuando todavía son pequeños.
> Evita los grandes problemas, dando pequeños pasos; es más fácil que resolverlos.
> Realizando pequeñas acciones, se han alcanzado grandes objetivos.

Los **factores** establecen el conjunto de características *externas* del *software*, es decir, aquellos atributos generales que presenta visto desde el "exterior" por el usuario o el gestor sin entrar en los detalles de su desarrollo. Representan los principales aspectos de la calidad del *software*. Estos factores pueden ser utilizados por los ingenieros del *software* como una ayuda para especificar los objetivos de calidad para sus sistemas, así como para reflejar los requisitos de calidad deseados por el usuario para el producto en desarrollo.

Por el contrario, los criterios establecen el conjunto de características *internas* del *software* que permiten definir los factores, es decir, aquellos atributos que presenta internamente vistos desde el punto de vista del desarrollador. Dicho de otra forma, los criterios permiten representar la serie de características primitivas del *software* que se combinan para constituir conjuntos de condiciones necesarias para definir los diferentes factores.

En el nivel inferior del modelo se encuentran las **medidas**, que representan cálculos cuantitativos de los distintos atributos del *software* definidos en los criterios. Estas medidas se aplicarán sobre el desarrollo y permitirán establecer una valoración objetiva para los criterios y factores de calidad.

De este modo, queda así establecida la estructura que tendrá el nuevo modelo de calidad para el paradigma orientado a objetos que se pretende diseñar. Una vez estipulada la estructura del modelo, el siguiente paso lógico consiste en definir sus elementos. Habiendo analizado los distintos factores y criterios que aparecen en los diferentes modelos de calidad convencionales, se ha decidido tomar como punto de partida los factores y criterios del modelo de McCall por ser los más ampliamente aceptados por la comunidad científica y por ajustarse también más a la realidad del *software* tanto tradicional como orientado a objetos como se verá a continuación. En los siguientes apartados, se explicarán y justificarán los factores y criterios introducidos en este modelo.

4.2.1. FACTORES DE CALIDAD

El primer paso para diseñar el modelo de calidad consiste en identificar los atributos del primer nivel del árbol de calidad. Partiendo de los once factores del modelo de calidad de McCall, se planteó la hipótesis de su validez para el paradigma orientado a objetos. Tras un estudio detallado de dichos factores y de su relación con las características del *software*, los once resultan necesarios y suficientes para establecer un nuevo modelo de calidad orientado a objetos. Todos ellos se encuentran totalmente adaptados a la tecnología orientada a objetos. La razón radica en que los factores originales describían las características del *software* desde un punto de vista externo (como ya se ha comentado), sin tomar en cuenta las propiedades internas (es decir, los criterios) del producto. Y, precisamente, las propiedades externas del *software* son

siempre las mismas, independientemente de la metodología, técnicas o paradigmas utilizados en la creación del *software*. Por tanto, todos estos factores son igualmente adecuados para los paradigmas convencionales y para el orientado a objetos.

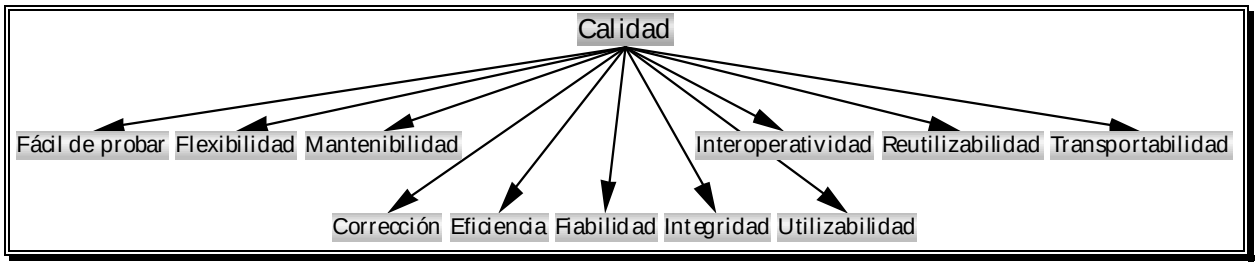


Figura 4.2: Factores de calidad de McCall usados como punto de partida

Los factores de calidad tomados en cuenta, que se muestran en la Figura 4.2, junto con su utilidad en un desarrollo orientado a objetos son:

- **Corrección**: es una característica de los programas que **satisfacen sus objetivos y cumplen adecuadamente todas las especificaciones** proporcionadas por los usuarios. En desarrollos de *software* orientados a objetos resulta esencial que el programa lleve a cabo lo que de él se espera, verificando todos los requisitos de forma apropiada.

- **Eficiencia**: es la cualidad del *software* de ejecutarse **consumiendo la menor cantidad de recursos informáticos** para desarrollar su función. Con el crecimiento de la potencia de los ordenadores y el abaratamiento que está sufriendo continuamente el *hardware*, la tendencia peligrosa es permitir que los programas crezcan innecesariamente. El objetivo, desde el punto de vista de este factor, es que los sistemas orientados a objetos deben seguir persiguiendo esta meta. Esto no debe suponer un detrimento grave en otros factores (habrá que llegar a un compromiso entre los distintos factores, como se verá posteriormente).

- **Fácil de probar**: representa el **esfuerzo requerido en probar un programa** para asegurarse que funciona como debe. La fase de pruebas permite verificar el correcto funcionamiento de los sistemas. Cuanto más sencillo sea de probar, más rápidamente se podrán detectar los defectos y antes se podrán corregir. Debe tenerse en cuenta que mecanismos como la herencia o el polimorfismo pueden dificultar la tarea de probar el sistema.

- **Fiabilidad**: característica presente cuando los programas **llevan a cabo todas sus funciones satisfactoriamente** con la precisión requerida y sin detenerse a causa de posibles errores. Es importante que el usuario pueda confiar en los resultados proporcionados por el *software*. La orientación a objetos favorece este atributo incorporando clases ya desarrolladas y probadas en la construcción de nuevas clases.

- **Flexibilidad**: es el **esfuerzo necesario para modificar un programa operativo**. En todo desarrollo de un sistema informático el usuario siente la necesidad de introducir modificaciones, ampliaciones y mejoras del *software*. El desarrollador debe poder incluir estas modificaciones lo más fácilmente posible y la orientación a objetos presenta como una de sus características la sencillez para realizar cambios.

- **Integridad**: es la capacidad para **controlar y restringir el acceso al *software* o a los datos** por personas no autorizadas. Cada vez resulta más importante poder controlar quién accede tanto a los programas como a los datos manejados por dichos programas, con el fin de evitar su uso fraudulento.

- **Interoperatividad**: es el **esfuerzo necesario para utilizar un programa en conjunción con otro**. Es importante que el programa tenga una organización que le permita comunicarse con otros programas si deben interactuar entre sí. Esta comunicación se realizará, en términos generales, entre componentes de otros programas, con agentes... Hay que tener en cuenta que este factor contempla únicamente la comunicación entre programas, no la comunicación con el usuario.

- **Mantenibilidad**: es el **esfuerzo requerido para localizar y eliminar un error** en un programa operativo. Normalmente, los sistemas informáticos no se encuentran libres de defectos y, por supuesto, tampoco los orientados a objetos, por lo que es necesario corregirlos lo más rápidamente que se pueda para que afecte lo menos posible al funcionamiento normal del programa. Este atributo se ve favorecido en la orientación a objetos al trabajar con unidades autónomas de datos y funciones, como son las clases.

- **Reutilizabilidad**: representa la **posibilidad de utilizar los componentes de un programa en otras aplicaciones** (relacionadas con el alcance de las funciones que efectúa el programa). Una de las ventajas que señala la orientación a objetos es la posibilidad de reutilizar el código. La construcción de nuevas aplicaciones se simplifica bastante si se pueden utilizar fragmentos de sistemas ya existentes.

- **Transportabilidad**: es el **esfuerzo necesario para transferir un programa de un entorno** con una configuración *hardware* o *software* **a otro**. En ocasiones interesa que el programa realizado pueda ser trasladado a otro entorno informático, lo cual puede verse favorecido por las estructuras de objetos.

- **Utilizabilidad**: muestra el **esfuerzo requerido para aprender, operar, preparar las entradas e interpretar la salida** de un programa. La tendencia actual del *software* es que sea lo más sencillo posible de manejar, de rápido aprendizaje y sin dificultades añadidas, pues de lo que se trata es de facilitarle la labor a los usuarios, no de dificultársela.

Debe tenerse en cuenta que éstos son factores generales, es decir, que no necesariamente tienen que estar presentes en todos los sistemas. Puede haber sistemas para los que, por ejemplo, no sean precisos requisitos de seguridad en el acceso o no sea necesario migrar a otro sistema operativo. En esos casos, habrá factores no aplicables. Todos estos factores van a tener distinta importancia dependiendo del dominio de aplicación del sistema en desarrollo, puesto que cada sistema va a tener sus propias características y no tienen por qué coincidir plenamente con las de otros sistemas. Por ello, para cada programa deberá ponderarse la influencia de cada uno de estos factores en la calidad total, de tal forma que aquellos factores que se considere que tienen una mayor relevancia obtengan un peso superior (en el apartado 4.4 se ampliará este aspecto).

También es importante hacer notar que los factores no son completamente independientes entre sí. En la Tabla 4.1 se muestran las relaciones que, en general, existen entre ellos y en qué medida suele existir una influencia de un factor sobre otro. Cuando un alto grado de calidad se encuentra presente en un factor, el signo **+** representa una influencia positiva en el otro factor, mientras que el signo **–** refleja una influencia negativa. Cuando el signo es negro (**+**, **–**), simboliza una influencia fuerte, mientras que un color gris (+, –), significa una influencia débil. Por ello, en ocasiones habrá que llegar a un compromiso; dado que será imposible obtener un máximo valor de calidad en todos los factores, habrá que estudiar cuáles serán los factores más interesantes para cada dominio de problema en particular. La construcción de la tabla se ha realizado teniendo en cuenta la opinión del autor junto con la información al respecto publicada en [Cooper, 79] [Schulmeyer, 87] [Deutsch, 88] [Vincent, 88] [Perry, 91] [Gillies, 92]. De todas formas, hay que tener en cuenta que estas influencias pueden sufrir variaciones entre el desarrollo de un sistema y otro.

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Integridad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad
Corrección											
Eficiencia											
Fácil de probar	+	–									
Fiabilidad	+	–	+								
Flexibilidad	**+**	–	+	+							
Integridad		–		+	–						
Interoperatividad		–				**–**					
Mantenibilidad	**+**	**–**	+	+	**+**						
Reutilizabilidad		**–**	+		+	**–**		**+**			
Transportabilidad		**–**	+				+	+	**+**		
Utilizabilidad	+		+	**+**	+	**+**		**+**			

Tabla 4.1: Influencia de cada factor en otros factores

Como cada sistema está afectado por múltiples atributos, cualquier alteración influirá con distinto grado en todos los atributos. Por ello, la persona responsable de la calidad del proyecto deberá analizar la influencia entre dichos atributos y consensuar un compromiso entre aquéllos que resulten conflictivos entre sí con el fin de intentar satisfacer al máximo los deseos del cliente. Evidentemente, no se debe buscar un compromiso único y universal para resolver todos los problemas, sino que se debe estudiar en cada caso concreto la arquitectura y características del sistema [Barbacci, 95].

4.2.2. CRITERIOS DE CALIDAD

Una vez definidos los factores, deben identificarse los atributos del segundo nivel del árbol. Partiendo de los 23 criterios del modelo de McCall, se planteó la hipótesis de su validez dentro de la orientación a objetos. Después de un análisis cuidadoso de cada

uno, se constató que todos ellos representan una característica que los sistemas orientados a objetos deben poseer (existen relaciones entre ellos y algún atributo de calidad de los sistemas orientados a objetos) para definir adecuadamente los factores. Por esto, se han incorporado dentro del modelo en construcción adaptándolos a este paradigma. No obstante, estos criterios aunque necesarios, no resultan suficientes para la orientación a objetos, por lo que hubo que ampliarlos, como se verá posteriormente. Estos primeros 23 criterios, junto a su utilidad en los desarrollos *software* orientados a objetos, son los que se explican a continuación (entre paréntesis se ha colocado el código de dos letras que, posteriormente, se utilizará para referenciar a los criterios; este código está basado en el nombre en inglés del criterio debido al aspecto internacional de este idioma en este campo):

- **Auditoría de accesos** (AA: *Access Audit*): el sistema puede tener que proporcionar un **registro del acceso al *software* y los datos**. Deben existir métodos para almacenar todos los accesos y para informar inmediatamente de los accesos no autorizados.

- **Auto-descriptivo** (SD: *Self-Descriptiveness*): el programa tiene que aportar una **explicación de la implementación de las funciones**. Un sistema *software* es auto-descriptivo si un lector del código fuente (o de los resultados del análisis y diseño) puede fácilmente comprender cómo se ha implementado una determinada función, esto es, si el programa es legible y tiene una descripción auto-documentada de sí mismo. Dentro de este criterio, se englobarían temas como la claridad de los esquemas y diagramas empleados, la notación para representar las clases y sus relaciones, la sencillez del pseudo-código de los métodos, la legibilidad de los lenguajes empleados, los comentarios del código, los nombres de los identificadores, el sangrado, etc.

- **Completitud** (CM: *CoMpleteness*): el *software* ha de **incluir la implementación completa de todas las funciones requeridas**. Un sistema *software* está completo cuando hay suficientes clases, métodos, atributos, variables, datos, sentencias, etc. para cumplir todas las capacidades requeridas, además de encontrarse plenamente definidas. Además, todos los requerimientos deben encontrarse plasmados en los elementos presentes en el sistema.

- **Comunicaciones estándar** (CC: *Communications Commonality*): la aplicación tiene que proporcionar la **utilización de protocolos y rutinas de interfaz estándar**. El *software* incorpora esta característica cuando emplea normas establecidas para comunicarse en una red de comunicaciones. Para ello, el diseñador deberá obtener los modelos uniformes definidos para las interfaces de comunicaciones de datos y diseñar las clases de comunicaciones adecuadas teniendo estos estándares como guía.

- **Comunicatividad** (CO: *COmmunicativeness*): los sistemas deben **proporcionar entradas y salidas útiles** que puedan ser asimiladas por los usuarios. Este criterio se refiere tanto a las entradas del usuario y las salidas que proporciona el *software* (y su utilidad, facilidad de comprensión, control de errores…), como a los dispositivos válidos para realizar la entrada y la salida. Con ello se pretende facilitar la

introducción de datos y la interpretación de los resultados. Por ello, las clases de entrada/salida deben incluir la posibilidad de utilizar distintos tipos de dispositivos. Este criterio tiene que ver con la utilidad de la interacción hombre-máquina. Hay que tener en cuenta que en este criterio intervendrán tanto aspectos objetivos como subjetivos.

- **Concisión** (CN: *CoNciseness*): el *software* ha de proporcionar la **implementación de una función con la mínima cantidad de código**. El número total de operadores y el número total de operandos debe ser el mínimo posible para representar en el lenguaje de implementación elegido la solución al problema. Cada clase, método, atributo, variable, dato, sentencia, etc. ha de resultar necesario para cumplir con las capacidades requeridas. No deben incluirse elementos innecesarios o superfluos para desarrollar las operaciones precisas. Este criterio constituye un buen ejemplo del compromiso que debe existir entre los distintos criterios (posteriormente será comentado), en este caso, con la legibilidad del código.

- **Consistencia** (CS: *ConSistency*): el sistema debe proporcionar un **diseño, una notación y unas técnicas de implementación uniformes**. El *software* es consistente cuando durante todo el ciclo de vida del producto se han utilizado los mismos estándares. Para conseguir la consistencia, es necesario, por tanto, establecer un conjunto de estándares antes de comenzar a desarrollar el sistema, asegurándose de que dichos estándares se cumplen de manera uniforme. Dentro de estas normas, deben verificarse también ciertas restricciones de elegancia o buen estilo recomendadas por los especialistas en el lenguaje de programación empleado que, por supuesto, debe ser un lenguaje orientado a objetos.

- **Control de acceso** (AC: *Access Control*): en ocasiones, el *software* ha de permitir **controlar el acceso al programa y a los datos**. El sistema debe poder tener un control completo de quién accede, cuándo y cómo a él. Es decir, este criterio tiene relación con la seguridad, incluyendo alertas y respuestas ante violaciones en los accesos. La principal diferencia con el criterio auditoría de accesos estriba en que éste registra los accesos mientras que el presente criterio se limita a controlarlos.

- **Datos estándar** (DC: *Data Commonality*): el programa tiene que ofrecer la utilización de **representaciones de datos estándar**. En la orientación a objetos esto significa también que deben utilizarse clases para representar las estructuras de datos, proporcionando como interfaz una serie de métodos sencillos y operadores iguales a los empleados en los datos simples.

- **Eficiencia de almacenamiento** (SE: *Storage Efficiency*): el sistema debe **minimizar las necesidades de almacenamiento** durante su funcionamiento. El *software* almacena datos eficientemente si no utiliza más memoria o espacio en disco que el necesario para realizar las operaciones requeridas, es decir, si se minimiza el número de *bytes* utilizados. Para ello, las clases no deben tener atributos ni variables no usadas ni se deben emplear datos o ficheros con información inútil o redundante.

- **Eficiencia de ejecución** (EE: *Execution Efficiency*): el sistema ha de **minimizar el tiempo de proceso** para llevar a cabo cada una de sus funcionalidades. El *software* se ejecuta eficientemente si minimiza el número de instrucciones ejecutadas por el microprocesador para desarrollar la operación requerida.

- **Entrenamiento** (TA: *TrAining*): la aplicación tiene que contemplar la **transición desde el sistema anterior** o la **familiarización inicial con el nuevo sistema**. El *software* es capaz de adiestrar a los usuarios si incluye previsiones para ayudarles a aprender cómo utilizar el *software* para desarrollar sus trabajos. Estas previsiones incluyen clases de entrenamiento, capacidades de ayuda en línea y niveles configurables de la sofisticación de la interfaz de usuario con el fin de facilitar el aprendizaje de la aplicación.

- **Expansibilidad** (EX: *EXpandability*): los sistemas facilitarán la **expansión de las funcionalidades o de las necesidades de almacenamiento de los datos**. El *software* es expansible si resulta fácil incorporarle nuevas capacidades funcionales o nuevos datos, esto es, si es sencillo aumentar sus posibilidades sin necesidad de un gran trabajo de rediseño o modificación de las clases.

- **Generalidad** (GE: *GEnerality*): los sistemas deben contener **funciones cuya utilidad sea lo más amplia posible**. Los métodos de una clase son generales si proporcionan servicio a más de una clase, es decir, si son llamados desde distintas clases. Normalmente, puede decirse que cuanto más sea utilizado un elemento desde diferentes lugares, mayor será su generalidad. Dentro de este criterio también se incluye el concepto de genericidad [Alonso, 95] que es la capacidad para formar funciones o clases paramétricas que describen estructuras de datos genéricas, y que en la orientación a objetos se incorpora como polimorfismo paramétrico.

- **Independencia de la máquina** (MI: *Machine Independence*): determina su **independencia del *hardware***. Un *software* es dependiente de la máquina si necesita conocer información exacta de su arquitectura, de los dispositivos existentes y de su modo de funcionamiento, de tal forma que con otra arquitectura o sin la presencia de dichos dispositivos el *software* no pueda realizar su labor.

- **Independencia del sistema *software*** (SS: *Software System independence*): determina su **independencia del entorno *software*** (sistema operativo, utilidades, rutinas de entrada/salida, etc.). Una aplicación que es independiente del sistema *software* no contiene ninguna referencia a su entorno, esto es, el *software* puede ser trasladado y utilizado en otros entornos con ninguna o escasas modificaciones. Evidentemente, puede resultar imposible construir un sistema completo totalmente independiente del entorno *software*, pero se deben intentar minimizar dichas dependencias.

- **Instrumentación** (IN: *INstrumentation*): el sistema ha de permitir **realizar medidas del uso del *software* o de identificación de errores**. El *software* debe incorporar la posibilidad de generar un diario o *log* de las operaciones realizadas. Debe, además, incluir algún mecanismo que permita identificar los errores que se puedan producir, con el fin de informar claramente al usuario de lo que ha hecho mal y de

las razones, incorporando toda la información necesaria para que no vuelva a ocurrir.

- **Modularidad** (MO: *MOdularity*): los sistemas deben disponer de una **estructura de módulos y clases altamente independientes**. El *software* es modular si está diseñado e implementado mediante una estructura de componentes ordenados, cohesivos y con un acoplamiento óptimo. La modularidad es un conglomerado de diversas técnicas, metas y características que se ha comprobado que conducen a un *software* fácilmente mantenible. Una de las claves para alcanzar la modularidad es el uso de un diseño orientado a objetos adecuado que permitirá obtener como resultado un diagrama de clases bien organizadas y definidas.

- **Operatividad** (OP: *OPerability*): representa aquellos atributos del *software* que **facilitan su uso y su funcionamiento**. El *software* es operativo si es fácil de utilizar y el usuario se puede comunicar con él de forma sencilla, es decir, si incluye, entre otros elementos, una interfaz de usuario comprensible, opciones seleccionables para adaptar el *software* a cada usuario específico, capacidad para mantener al usuario informado de las condiciones de funcionamiento, etc. La operatividad incluye también el enmascaramiento de detalles de implementación de tal forma que realizar cualquier trabajo sea algo natural desde la perspectiva del usuario. Este criterio constituye una parte de los objetivos perseguidos por el factor utilizable.

- **Precisión** (AU: *AccUracy*): el sistema debe ofrecer la **exactitud** necesaria **en los cálculos y en los resultados** del programa. Es decir, el *software* ha de producir los resultados dentro de los límites de tolerancia establecidos. Para ser preciso, deben emplearse bibliotecas de clases matemáticas, métodos numéricos y conversiones de tipos de datos que preserven el número de bits de precisión durante los cálculos.

- **Seguimiento** (TR: *TRaceability*): durante el desarrollo del sistema es conveniente establecer una **conexión entre los requisitos, el análisis, el diseño y la implementación** para poder realizar un seguimiento del desarrollo. Se puede realizar un seguimiento del *software* cuando es fácil encontrar todo el código, el diseño y el análisis que implementa un requisito particular y, análogamente, dado un fragmento concreto de código es fácil localizar su diseño, su análisis y sus requisitos correspondientes. Este criterio se puede conseguir asociando cuidadosamente los requisitos a los componentes del *software* durante las fases de análisis, diseño e implementación y a lo largo de la documentación técnica.

- **Simplicidad** (SI: *SImplicity*): un sistema es simple si proporciona una implementación de las funciones, un análisis y un diseño de la forma más **clara, sencilla, simple y comprensible** posible, evitando análisis, diseño y código enrevesado, complejo, confuso, indescifrable, desordenado y, en definitiva, poco legible. Evidentemente, para cumplir con el objetivo de este criterio habrá que llegar a un compromiso con otros, como, por ejemplo, la concisión.

- **Tolerancia a errores** (ET: *Error Tolerance*): los sistemas han de proporcionar **continuidad en el funcionamiento ante circunstancias anómalas**. El *software* es capaz de manejar las situaciones anómalas adecuadamente si las puede detectar y recuperarse de dichas situaciones en lugar de proseguir erróneamente o detenerse. En esencia, el *software* debe estar diseñado para sobrevivir ante fallos del *software* o del *hardware*. La tolerancia a errores, o robustez, incluye la detección y recuperación ante datos de entrada inadecuados, fallos computacionales, defectos en el *hardware*, fallos de los dispositivos y errores en las comunicaciones.

Pero estos 23 criterios no son suficientes para representar todos los atributos de calidad del *software* construido con el paradigma orientado a objetos, como se ha comentado. Hay algunos aspectos que no pueden ser tratados perfectamente con ninguno de dichos criterios, con lo que quedan sin analizar algunas características importantes del *software*. De esta manera, para que el modelo fuera lo más completo y general posible, se han identificado tres nuevos criterios para su inclusión entre el resto de los criterios del modelo descrito. Estos tres criterios son los que se explican a continuación:

- **Documentación** (DO: *DOcumentation*): un sistema *software* está documentado, si **su propósito, estrategia, intención y propiedades están precisa y explícitamente definidos** en el contexto del programa [Dromey, 95]. El *software* es una combinación de los programas, los datos y la documentación, por lo que es fundamental que todo esté bien documentado. Se puede decir que el *software* está bien documentado, si una persona puede acceder fácilmente a la información sobre su diseño y sus funcionalidades y, además, si dicha información es comprensible y completa [Deutsch, 88]. El principal objetivo de este criterio es lograr una suficiencia y adecuación de la documentación de los productos *software* que es necesaria en los entornos de soporte del *software* en la fase de funcionamiento posterior al desarrollo [Schulmeyer, 90]. Dentro del paradigma orientado a objetos, este criterio tiene una especial importancia para favorecer a factores como mantenibilidad, reutilizabilidad y la flexibilidad de las clases debido a las especiales características de este tipo de desarrollos: encapsulación, herencia y polimorfismo. Debe tenerse en cuenta que en la época de McCall, los desarrollos *software* tenían, por lo general, un tamaño considerablemente inferior a los actuales, por lo que la documentación no resultaba tan importante como hoy en día, donde los grandes sistemas pueden contener varios millones de líneas de código, junto con gran cantidad de documentación surgida de las distintas fases del ciclo de vida.

- **Estabilidad** (ST: *STability*): la estabilidad de un programa se puede definir como la capacidad para **mantener un funcionamiento adecuado ante la ampliación o incorporación de cambios** en dicho programa [Yau, 80]. Por tanto, este atributo trata de la invulnerabilidad de todo el programa ante el impacto de una modificación teniendo en cuenta consideraciones lógicas y de rendimiento. Evidentemente, propiedades de la orientación a objetos como la encapsulación y la herencia deben facilitar la adecuación de este criterio. Una de las ventajas que propugna la orientación a objetos (frente a otros paradigmas) consiste en facilitar la incorporación de nuevas funcionalidades o la modificación de las actuales. Por ello, este criterio toma su verdadero sentido en el paradigma orientado a objetos, reflejando que cualquier cambio no debe afectar al funcionamiento del código antiguo. Aunque a

primera vista pueda parecer que este criterio es igual a la expansibilidad, tiene un matiz que los diferencia: mientras que la expansibilidad mide la facilidad para introducir nuevas funcionalidades, la estabilidad mide el comportamiento de las funcionalidades originales ante la introducción de las nuevas.

- **Estructuración** (SR: *StRucturedness*): un programa está estructurado si mantiene un **patrón definido de la organización de sus partes interdependientes** [Boehm, 78]. Es decir, este criterio hace referencia a la estructura interna de cada clase. Ésta es una de las propiedades clave del desarrollo orientado a objetos pues representa la característica de la encapsulación [Rumbaugh, 91] que todo programa orientado a objetos debe utilizar (aunque no es una característica única de la orientación a objetos, toma su mayor sentido en este paradigma). Este criterio parece tener un cierto parecido con la modularidad, pero en realidad se complementan mútuamente: la modularidad se encarga de la organización y relación entre las clases, mientras que la estructuración se encarga de la organización interna de cada clase.

Al igual que ocurre con los factores, hay que considerar que los criterios descritos son generales, esto es, no deben encontrarse presentes obligatoriamente en todos los sistemas orientados a objetos. Pueden existir sistemas para los que, por ejemplo, no sea preciso establecer un control de acceso o que sea un desarrollo para un *hardware* en concreto y no se pueda establecer una independencia de la máquina. En estos casos, habrá criterios no aplicables. Todos estos criterios tendrán una importancia diferente según el dominio de aplicación del sistema en desarrollo, puesto que cada sistema tendrá sus propias peculiaridades que no deberán de coincidir completamente con las de otros sistemas. Por ello, al igual que ocurría con los factores, será necesario establecer una ponderación de cada uno de los criterios con el fin de reflejar la distinta relevancia que presentan debido al tipo de aplicación. Esta ponderación podría ser realizada por un experto, por el ingeniero del *software* o por el encargado de calidad de la empresa.

Igualmente es de destacar que los criterios no son independientes entre sí. En la Tabla 4.2 se muestran las relaciones generales que pueden existir entre ellos y en qué medida se produce una influencia de un criterio sobre otro. Los símbolos situados en las casillas de la tabla indican el tipo de relación existente entre los criterios. Así, existe una relación directa (se representa con el signo **+**), cuando al aumentar el valor de calidad de uno de los criterios, es probable que el otro criterio también aumente el suyo. Por el contrario, existe una relación inversa (se representa con el signo **—**), cuando al incrementarse el valor de calidad de uno de los criterios, es probable que el otro criterio disminuya el suyo. Y existirá una relación neutral (se representa por la casilla en blanco), cuando es probable que una modificación del valor de calidad de uno de los criterios no afecte al otro, o bien, cuando no resulte posible generalizar la existencia de esta influencia. Cuando el signo es de color negro (**+**, **—**), simboliza una fuerte influencia, mientras que un color gris (+, —), significa una influencia débil. Por ello, se tendrá que establecer un compromiso entre los criterios que intervienen al estudiar un problema; dado que será imposible obtener un máximo valor de calidad en todos ellos, habrá que estudiar cuáles serán los más interesantes para cada dominio de problema en particular. La construcción de la tabla se ha realizado teniendo en cuenta la opinión del autor junto con la de varios ingenieros del *software* consultados, así como del estudio comparativo de las medidas del modelo de calidad presentado.

	AA	SD	CM	CC	CO	CN	CS	AC	DC	DO	SE	EE	TA	ST	SR	EX	GE	MI	SS	IN	MO	OP	AU	TR	SI	ET
AA																										
SD																										
CM		+																								
CC																										
CO		+																								
CN		−																								
CS			+		+	+																				
AC	+																									
DC							+																			
DO		+	+		+		+																			
SE	−	−			−			−		−																
EE	−	−			−		−	−		−	−															
TA					+					+																
ST		+				−	+			+	−		−													
SR	−	+	+		+		+	−	+	+	−	−		+												
EX		+				+	+		+	+		−		+	+											
GE	−	+	+			+	+	−	+	+	−	−			+	+										
MI		+		+						+	−	−			+		+									
SS		+								+	−	−			+		+	+								
IN		+			+		+			+	−	−			+		+									
MO	−	+	+		+		+	−		+	−	−		+	+	+	+	+	+	+						
OP					+					+	−	−								+						
AU																										
TR		+			+		+			+	−	−		+	+	+	+			+	+					
SI		+	+		+	−	+			+	−		+	+	+	+		+	+	+	+			+		
ET			+		+		+			+	−	−	+	+						+	+	+				

Tabla 4.2: Influencia de cada criterio en otros criterios

4.2.3. RELACIÓN ENTRE LOS FACTORES Y LOS CRITERIOS DE CALIDAD

Una vez definidos los factores y los criterios de calidad que conforman el nuevo modelo, la tarea a realizar consiste en establecer y evaluar las relaciones existentes entre los factores y cada uno de los criterios con el fin de que el cometido de los factores quede reflejado por sus criterios, teniendo siempre en cuenta el paradigma orientado a objetos. Seguidamente, se resumen estas relaciones, presentando los factores junto con cada uno de los criterios con los que se ha establecido alguna relación, que se muestra en las figuras correspondientes.

Para el factor "corrección" (Figura 4.3) se han encontrado las cuatro relaciones relativas a los criterios que se especifican en la Tabla 4.3, incluido el nuevo criterio de documentación.

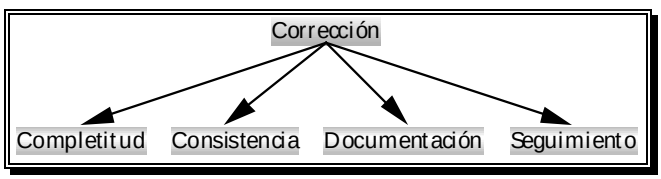


Figura 4.3: Criterios de corrección

Corrección	
Completitud	Para conseguir la corrección de un producto, entendida como el cumplimiento de todas sus especificaciones y sus objetivos, es necesario que todas las funciones requeridas se encuentren reflejadas en el código desarrollado.

Consistencia	El seguimiento de una serie de estándares, metodologías y normas de desarrollo facilita la tarea de conseguir un *software* que verifique adecuadamente todos los requisitos del usuario debido a que la metodología dirige el desarrollo evitando la aparición de defectos en el producto.
Documentación	Entre las necesidades de todo producto se debe encontrar una adecuada documentación, pero no sólo la documentación del usuario, sino también la documentación técnica que permite a los desarrolladores comprobar y verificar la corrección del *software* en construcción. Por ello, dentro del paradigma orientado a objetos, la documentación resulta imprescindible para comprobar la corrección debido al gran volumen habitual del *software* actual.
Seguimiento	El establecimiento de un proceso de seguimiento entre los requisitos y la implementación, pasando por el análisis y el diseño, garantiza la realización de todas las funciones necesarias además de ofrecer un mecanismo para verificar que se han desarrollado completamente y de forma adecuada.

Tabla 4.3: Criterios de corrección

Para la "eficiencia" (Figura 4.4) se han determinado las tres relaciones con los criterios reseñados en la Tabla 4.4, habiéndose introducido el criterio de concisión con respecto al modelo de McCall.

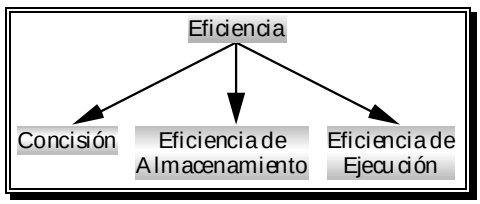


Figura 4.4: Criterios de eficiencia

Eficiencia	
Concisión	Un sistema que implemente un algoritmo de forma concisa favorecerá la eficiencia, en general, tanto de almacenamiento (especialmente el espacio de almacenamiento necesario para almacenar el código objeto) como de ejecución. La orientación a objetos generalmente no promueve la concisión, por lo que este criterio adquiere su importancia con el fin de que la eficiencia no se vea afectada en gran medida. La concisión fue relacionada con la eficiencia en [Arthur, 85].
Eficiencia de almacenamiento	Una de las áreas donde se puede lograr la eficiencia del producto la constituye la cantidad de espacio de almacenamiento (tanto en memoria principal como secundaria), la cual debe intentar mantenerse lo menor posible, procurando aprovecharla al máximo sin ningún desperdicio inútil.
Eficiencia de ejecución	Análogamente, la otra área donde la eficiencia tiene un peso importante la constituye la ejecución. El tiempo de proceso debe intentar también reducirse al máximo para cada una de las operaciones necesarias.

Tabla 4.4: Criterios de eficiencia

Para el factor "fácil de probar" (Figura 4.5) se han descubierto las siete relaciones con los criterios que se describen en la Tabla 4.5, siendo los criterios comunicactividad, documentación y estructuración los que establecen las nuevas relaciones.

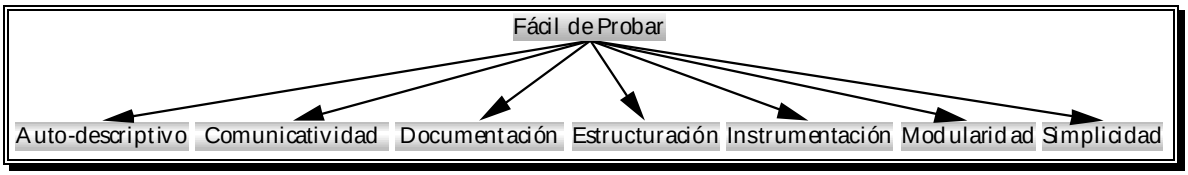


Figura 4.5: Criterios del factor fácil de probar

Fácil de probar	
Auto-descriptivo	Si el código del programa se encuentra adecuadamente comentado, la tarea de comprobar si su funcionamiento es el apropiado se ve intensamente facilitado pues se pueden probar los elementos del programa por separado, por ejemplo, un método o una clase, verificando que su funcionamiento es el descrito.
Comunicatividad	Cuanto más sencillas sean las entradas a un sistema y más simples sean las salidas proporcionadas, más sencilla será la tarea de verificar que los resultados obtenidos se corresponden con los datos de entrada introducidos. Este criterio resulta importante para este factor dentro de la orientación a objetos debido a que en este paradigma, el diseño está básicamente dirigido por los eventos de entrada/salida en lugar de por el control, como ocurre en el paradigma procedimental. Este criterio fue relacionado con el factor en [Boehm, 76].
Documentación	Una buena y completa documentación permite comprobar que los resultados generados por el sistema *software* se corresponden con el funcionamiento descrito. Si la documentación no es clara o contiene, por ejemplo, ambigüedades o inconsistencias, se verá dificultada la tarea de la verificación del correcto funcionamiento del *software*. Este factor fue relacionado con el criterio en [Arthur, 85].
Estructuración	Si el *software* posee una estructura clara y definida, siguiendo una organización de sus partes según un determinado patrón, será más fácil realizar la comprobación del correspondiente funcionamiento de cada una de estas partes por separado. Una de las ventajas de la orientación a objetos consiste en disponer todos los componentes del *software* de tal forma que puedan desarrollarse y probarse aisladamente.
Instrumentación	A la hora de probar un sistema es útil que el propio programa pueda proporcionar una serie de medidas acerca de su propio funcionamiento o, incluso, un *log* de las operaciones realizadas, con el fin de facilitar su seguimiento y la detección, en su caso, de los posibles errores. Además, dentro del conjunto de pruebas al que se somete al sistema se encontrarán aquellas situaciones anómalas que el *software* deberá detectar e informar de manera adecuada.
Modularidad	Si el sistema posee una estructura de módulos y clases altamente independientes, las pruebas podrán centrarse, en un primer lugar, en la verificación del funcionamiento de estos módulos o clases de forma independiente para después comprobar que su interconexión se ha realizado de forma correcta.

Simplicidad	Cuanto más simple sea el código que implementa una determinada funcionalidad, la tarea de probarla también resultará más sencilla.

Tabla 4.5: Criterios del factor fácil de probar

Para la "fiabilidad" (Figura 4.6) se han encontrado las cinco relaciones con los criterios referidos en la Tabla 4.6, donde la completitud y simplicidad determinan las nuevas relaciones.

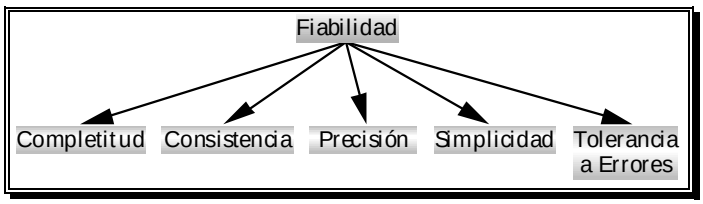


Figura 4.6: Criterios de fiabilidad

Fiabilidad	
Completitud	Se espera que el programa lleve a cabo los objetivos previstos de forma adecuada con la precisión requerida si todas las funciones requeridas se encuentran implementadas minuciosamente. Esta relación fue definida en [Boehm, 76] y [Dromey, 95].
Consistencia	La uniformidad en la notación y en la implementación facilita la ejecución satisfactoria de todas las operaciones necesarias.
Precisión	Si se cuida la precisión con que se realizan los diferentes cálculos dentro del sistema, éste podrá ofrecer al usuario un funcionamiento más fiable.
Simplicidad	Si los algoritmos necesarios se han implementado de una forma sencilla, los resultados no se verán afectados por pérdidas de exactitud debido a la complejidad de las operaciones. Fue relacionada con la fiabilidad en [Arthur, 85] y [Deutsch, 88].
Tolerancia a errores	Si el sistema ofrece al usuario un mecanismo que sea capaz de interceptar todos los posibles errores que puedan producirse, dando los mensajes adecuados y tratando de recuperarse de la situación anómala, el usuario podrá confiar en los resultados que le proporcione, pues estará seguro de que no han sido influidos por ningún tipo de error.

Tabla 4.6: Criterios de fiabilidad

Para el factor "flexibilidad" (Figura 4.7) se han determinado las diez relaciones con los criterios presentes en la Tabla 4.7, siendo nuevas las relaciones establecidas con comunicaciones estándar, datos estándar, documentación, estabilidad, estructuración y simplicidad.

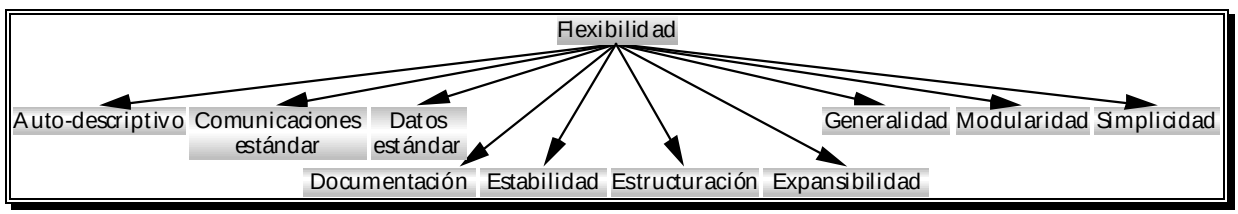


Figura 4.7: Criterios de flexibilidad

Flexibilidad	
Auto-descriptivo	Si el código se encuentra adecuadamente comentado, de tal forma que se describa con claridad el objetivo de sus elementos, el esfuerzo para realizar una modificación en un programa en funcionamiento con el fin de incluir ampliaciones y mejoras, no será demasiado elevado, dependiendo casi exclusivamente de la complejidad de los cambios en la funcionalidad.
Comunicaciones estándar	La utilización de protocolos estándar de comunicación facilitará la realización de ampliaciones en el sistema relacionadas con las comunicaciones, que no requerirán un gran esfuerzo.
Datos estándar	El uso de datos y estructuras estándar simplifican la implantación de posibles modificaciones en el sistema, puesto que las posibles ampliaciones se verán beneficiadas por la reutilización de dichas estructuras, aspecto fundamental de la orientación a objetos.
Documentación	Para modificar un programa que se encuentra operativo, normalmente es necesario localizar cierta información (acerca del funcionamiento, objetivos, funcionalidades, etc. de las clases y sus métodos) a lo largo de la documentación del *software*, teniendo en cuenta que toda esta información debe ser comprensible. Sin una documentación adecuada, no resultaría viable la modificación del *software* debido al elevado tamaño que suelen tener las aplicaciones orientadas a objetos. La relación con este factor fue ya establecida por [Arthur, 85].
Estabilidad	Por la propia definición de estabilidad [Yau, 80], si se realiza una modificación en un programa, el funcionamiento del resto del programa no debe verse afectado por tales cambios.
Estructuración	Si las clases del programa tienen una estructura interna adecuadamente construida, cualquier modificación será rápida y no necesitará un esfuerzo desmesurado debido a que, normalmente, se verá restringida a cambiar o incorporar algún elemento nuevo sobre la estructura previa.
Expansibilidad	Si al *software* se le pueden incorporar con facilidad nuevas funcionalidades, entonces el esfuerzo para modificar su funcionamiento será pequeño.
Generalidad	Si las funciones desarrolladas han sido pensadas desde un punto de vista general, previendo que puedan ser utilizadas desde distintos lugares, todas las modificaciones a realizar se podrán apoyar en dichas funciones disminuyendo de esta forma el trabajo necesario.
Modularidad	Si el programa está organizado en una serie de módulos independientes, los cambios o mejoras necesarias se podrán limitar a la modificación de alguno de estos módulos, de tal forma que no afecte al exterior, o a la inclusión de algún nuevo módulo que incorpore las nuevas funcionalidades.
Simplicidad	Evidentemente, cuanto más simple resulte el código desarrollado, más sencillas resultarán de realizar las modificaciones. En la orientación a objetos, la simplicidad de los métodos es importante por su gran proliferación, generados por esta filosofía de diseño. Esta relación fue definida en [Arthur, 85] y [Deutsch, 88].

Tabla 4.7: Criterios de flexibilidad

Para el factor "integridad" (Figura 4.8) se han encontrado las dos relaciones con los criterios que se cuentan en la Tabla 4.8.

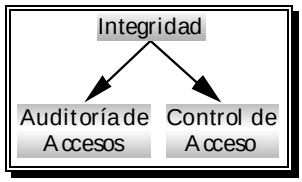


Figura 4.8: Criterios de integridad

Integridad	
Auditoría de accesos	Para poder controlar el acceso al *software*, es necesario que éste incorpore un mecanismo que registre quién, cuándo, cómo y desde dónde se accede a dicho *software*, informando de todos los intentos fallidos.
Control de acceso	Mediante el control de acceso se permite la utilización del *software* solamente por las personas autorizadas, permitiendo de esta forma conservar la integridad del *software*.

Tabla 4.8: Criterios de integridad

Para el factor "interoperatividad" (Figura 4.9) se han hallado las cinco relaciones con los criterios detallados en la Tabla 4.9, definiendo los nuevos criterios de documentación y estructuración las dos nuevas relaciones de este factor.

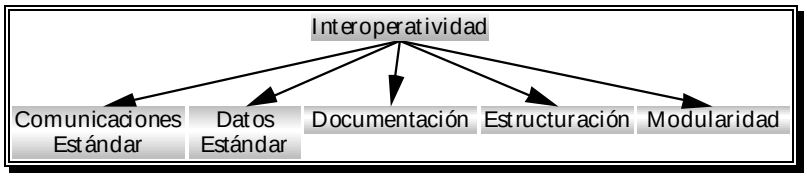


Figura 4.9: Criterios del factor interoperatividad

Interoperatividad	
Comunicaciones estándar	La utilización de protocolos de comunicaciones y rutinas de interfaz estándar facilita la comunicación con otros sistemas, tanto situados dentro del mismo equipo informático como en otros equipos, siendo entonces la comunicación a través de una red de comunicaciones.
Datos estándar	El empleo de representaciones de datos estándar facilita el intercambio de estos datos con otros sistemas.
Documentación	La conexión de un sistema con otro será más simple si todos los datos y toda la información necesaria acerca de las clases que intervienen, puede encontrarse fácilmente en la documentación que acompaña al *software*.
Estructuración	Si el programa tiene una buena estructura, las clases que intervengan en el enlace con otros sistemas estarán claramente definidas, evitando de esta forma su interactuación con el resto de los elementos del programa.
Modularidad	Todas las rutinas necesarias para la comunicación entre programas deberán estar situadas en los módulos de comunicación, de tal forma que las rutinas que se relacionen con otros sistemas se encuentren solamente en dichos módulos.

Tabla 4.9: Criterios del factor interoperatividad

Para la "mantenibilidad" (Figura 4.10) se han identificado las nueve relaciones con los criterios explicadas en la Tabla 4.10. La documentación, estabilidad, estructuración y seguimiento definen cuatro relaciones nuevas no presentes en el modelo original.

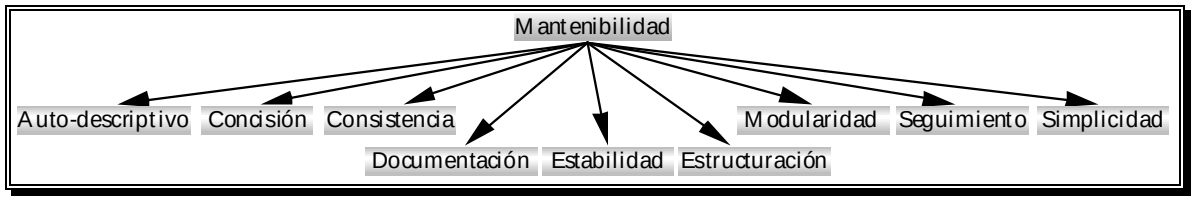


Figura 4.10: Criterios del factor mantenibilidad

Mantenibilidad	
Auto-descriptivo	Las tareas de mantenimiento resultan simplificadas si el propio código es claro y se encuentra adecuadamente comentado y descrito, además de disponer de una documentación de apoyo con diagramas, esquemas y pseudo-código claramente explicados.
Concisión	Al estar cada función implementada con la menor cantidad de código posible la tarea de detectar y corregir errores del *software* se puede realizar más rápida y sencillamente.
Consistencia	El mantenimiento se facilita si el *software* se ha desarrollado siguiendo una serie de normas y técnicas estándar y uniformes a lo largo de todo el desarrollo.
Documentación	Para realizar el mantenimiento es fundamental disponer de una documentación buena y completa, que pueda ser consultada rápidamente en caso de necesidad. En orientación a objetos resulta esencial con el fin de facilitar la localización de las clases que necesitan un mantenimiento. Este criterio influye en la mantenibilidad según [Arthur, 85].
Estabilidad	Cuando se corrige un error en el *software*, el resto del sistema no debe verse afectado en su correcto comportamiento [Yau, 80].
Estructuración	Una buena estructura interna ayuda a encontrar fácilmente cualquier elemento dentro de una clase, guía la incorporación del nuevo código y clarifica el impacto de los cambios [Boreham, 97].
Modularidad	La detección de un error resulta mucho más simple si se puede aislar la presencia de dicho error en uno de los módulos que componen el programa, lo cual puede conseguirse si se ha realizado una adecuada descomposición modular.
Seguimiento	Esta característica es importante al realizar el mantenimiento porque hace que resulte sencillo encontrar todas las clases y métodos que implementan un requerimiento particular y, similarmente, hace que sea sencillo encontrar todos los requerimientos que se reflejan en una determinada porción de código [Deutsch, 88]. En la orientación a objetos es particularmente deseable porque esta filosofía de diseño está dirigida principalmente al desarrollo de grandes programas. Esta relación también está presente en [Boehm, 76].
Simplicidad	Cuanto más clara, sencilla, simple y comprensible se haya realizado la implementación de una función, más rápida y fácilmente se podrá detectar la existencia de un error y su corrección resultará más asequible.

Tabla 4.10: Criterios del factor mantenibilidad

Para el factor "reutilizabilidad" (Figura 4.11) se han establecido las doce relaciones con los criterios especificados en la Tabla 4.11. Las nuevas relaciones vienen determinadas por los criterios comunicaciones estándar, datos estándar, documentación, estabilidad, estructuración, seguimiento y simplicidad.

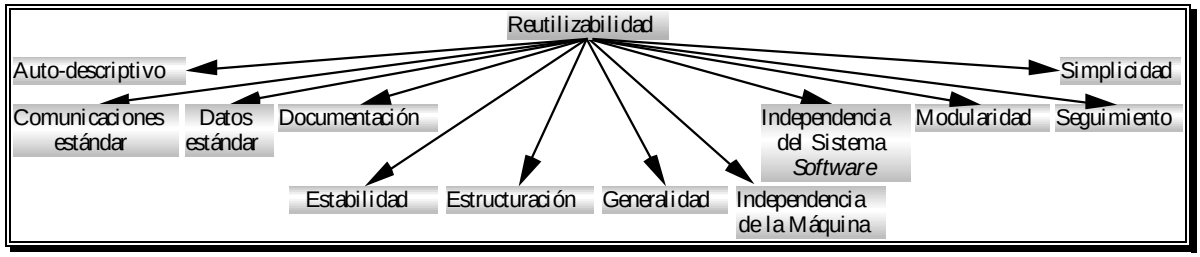


Figura 4.11: Criterios del factor reutilizabilidad

Reutilizabilidad	
Auto-descriptivo	Cuanto mejor comentado esté el código y los esquemas que le acompañen, más sencillo resultará extraer un fragmento de dicho código para utilizarlo en otra parte del sistema o, incluso, en otros sistemas.
Comunicaciones estándar	La utilización de protocolos y rutinas de interfaz estándar de comunicaciones facilitará la reutilización de las clases que implementen o usen dichos elementos de comunicación en otros sistemas que sigan los mismos estándares.
Datos estándar	Si se emplean representaciones de datos estándar, usando los mecanismos que facilita la orientación a objetos, se facilitará su reutilización o el uso de las clases que empleen dichas representaciones.
Documentación	Para reutilizar una clase en otras aplicaciones, se necesita una documentación completa sobre cada una de las clases. De esta forma, la reutilización (que es una de las características de la orientación a objetos) podrá alcanzarse de una forma más favorable. [Arthur, 85] ya definió esta relación.
Estabilidad	Si una clase se reutiliza en otro sistema, debe mantenerse el funcionamiento adecuado de dicha clase. Este comportamiento debe preservarse gracias a la característica de la encapsulación que proporciona la orientación a objetos.
Estructuración	La utilización de algunas de las clases en otros sistemas puede realizarse con poco esfuerzo, si el sistema posee una estructura clara y fácilmente comprensible.
Generalidad	Cuanto más genéricas sean las funciones desarrolladas, más probable será que puedan utilizarse en otros sistemas.
Independencia de la máquina	Si el *software* no realiza ninguna operación dependiente del *hardware*, sus funciones podrán ser reutilizadas en cualquier otro sistema que funcione sobre cualquier ordenador.
Independencia del sistema *software*	Si el funcionamiento del sistema no depende de ningún componente *software* externo (como, por ejemplo, el sistema operativo u otros sistemas), se podrá reutilizar el código en cualquier entorno *software*.

Modularidad	La división de la solución del problema en distintos módulos o clases independientes, lo cual se ve favorecido por la característica de encapsulación que proporciona la orientación a objetos, facilita la reutilización puesto que solamente será necesario exportar el módulo o clase necesario que realiza la tarea requerida.
Seguimiento	Si existe una clara conexión entre las distintas fases del ciclo de vida, la tarea de reutilizar alguno de los componentes se simplifica, al tener toda la información de dichos componentes al alcance. En la orientación a objetos, la reutilización es una de sus principales características y se ve favorecida si se puede reproducir el proceso de construcción de cada clase de principio a fin. Aunque se ha dicho poco acerca del seguimiento en la orientación a objetos [Antoniol, 00], éste permite controlar sus características a lo largo de las distintas fases del ciclo de vida y, por tanto, conocer plenamente sus funcionalidades y objetivos, lo cual facilita la utilización de una clase en otro proyecto.
Simplicidad	Cuanto más sencillo y claro se haya realizado el diseño e implementación de las funciones necesarias, más fácil resultará su reutilización en otro lugar. Este criterio se incorporó, como parte del factor reutilizabilidad, en el modelo de Deutsch [Deutsch, 88].

Tabla 4.11: Criterios del factor reutilizabilidad

Para el factor "transportabilidad" (Figura 4.12) se han descubierto las siete relaciones con los criterios que se reseñan en la Tabla 4.12. Los criterios comunicaciones estándar, documentación y generalidad definen las relaciones nuevas.

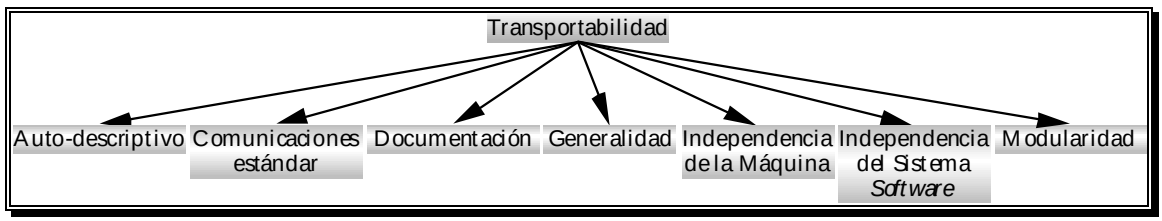


Figura 4.12: Criterios del factor transportabilidad

Transportabilidad	
Auto-descriptivo	Si el programador es capaz de comprender rápidamente cómo se ha implementado una funcionalidad, será sencillo estudiar si su funcionamiento es compatible con el entorno a donde se desea trasladar dicha funcionalidad.
Comunicaciones estándar	Los protocolos estándar de comunicación facilitarán la transportabilidad del sistema hacia otros entornos al poder reutilizar las clases que gestionan las comunicaciones.
Documentación	El esfuerzo requerido para transferir un programa a otro entorno o configuración se verá considerablemente reducido si la documentación disponible acerca de cada una de las jerarquías de clases es válida y adecuada. Esta relación fue definida en [Deutsch, 88].
Generalidad	Cuanto más genéricas sean las funciones desarrolladas, menos trabajo será requerido para cambiar el sistema a otra arquitectura. Este criterio fue considerado como parte del factor transportabilidad en [Arthur, 85] y [Dromey, 95].
Independencia de	Evidentemente, si el sistema se ha desarrollado teniendo en cuenta su

la máquina	independencia del *hardware*, no será requerida modificación alguna para cambiarlo de máquina, salvo las relativas al *software*.
Independencia del sistema *software*	Si el sistema es independiente al entorno *software*, las únicas transformaciones necesarias para cambiarlo de máquina serán aquéllas que tengan que ver directamente con el *hardware*.
Modularidad	Un sistema modular agrupará en pocos módulos o clases todos los elementos del programa que tengan que ser alterados a la hora de cambiarlo de entorno de ejecución. Al desarrollar un sistema se pueden prever los cambios necesarios para trasladar dicho sistema a otra configuración y situar todos aquellos elementos que puedan requerir cambios en unas pocas e independientes clases o módulos del código bien localizados.

Tabla 4.12: Criterios del factor transportabilidad

Para el factor "utilizabilidad" (Figura 4.13) se han incluido las seis relaciones con los criterios descritos en la Tabla 4.13, de las cuales, son nuevas las establecidas con documentación, instrumentación y tolerancia a errores.

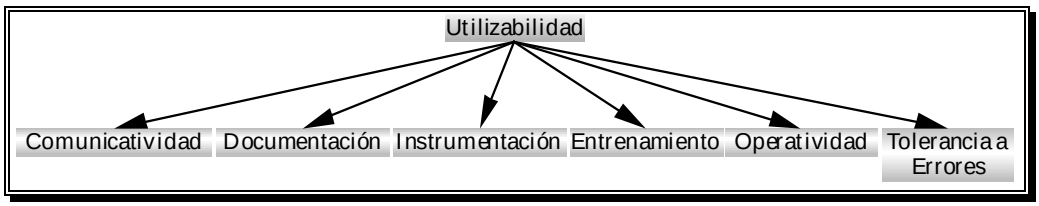


Figura 4.13: Criterios del factor utilizabilidad

Utilizabilidad	
Comunicatividad	Si tanto las entradas de información requeridas como las salidas proporcionadas son adecuadas y sencillas, la utilización del producto se verá grandemente favorecida.
Documentación	Una de las propiedades fundamentales del éxito de un sistema es que éste sea utilizado. Los usuarios utilizarán un sistema si entre sus características se encuentra un manual de usuario satisfactorio y adecuado, con explicaciones detalladas, descripciones y ejemplos de uso.
Instrumentación	La notificación adecuada al usuario de los errores producidos, así como el mantenimiento de un registro de las incidencias ocurridas, facilitará la utilización del *software*, puesto que los usuarios sabrán exactamente lo que ha ocurrido en cada momento. En el paradigma orientado a objetos, cada clase puede responsabilizarse de sus propias incidencias.
Entrenamiento	La inclusión de tutoriales, ayudas en línea y clases de entrenamiento facilitarán el acercamiento del usuario al sistema y, por tanto, su utilización.
Operatividad	La sencillez de uso de un sistema es directamente proporcional a la comodidad que siente el usuario en su utilización y, por tanto, en su deseo de emplear dicho sistema.

Tolerancia a errores	Conforme más robusto sea el *software* ante fallos o errores de cualquier tipo, más cómodo y confiado se sentirá el usuario utilizando el sistema. Los lenguajes orientados a objetos suelen incorporar mecanismos para gestionar los errores en cada clase, lo cual facilita su utilización puesto que cada clase sería capaz de manejar los errores que pudieran producirse en su seno.

Tabla 4.13: Criterios del factor utilizabilidad

4.3. MEDIDAS DE CALIDAD

Todo modelo de calidad del *software*, como el recientemente expuesto en el apartado anterior, no estará completo si no dispone de un mecanismo que posibilite realizar su evaluación de forma más o menos objetiva. Además, hay que tener en cuenta que una única medida no puede presentarse como una forma exclusiva de evaluar la calidad del *software* [Littlefair, 01]. En este apartado se va a definir un conjunto de medidas aplicables durante el desarrollo de sistemas orientados a objetos que permitirán calcular el nivel de calidad del sistema. Estas medidas evaluarán aspectos particulares de cada uno de los criterios de calidad enunciados. Para cada criterio, se obtendrá un único valor, fruto de sus diferentes medidas, que se propagarán por el árbol de calidad. Cada factor estará entonces compuesto por los distintos valores de sus criterios, con los que se obtendrá de nuevo un valor único que, ponderado entre todos los valores producidos para cada factor, proporcionará el nivel de calidad global del sistema. No obstante, como se indicará más adelante, también podrán utilizarse los vectores de valores obtenidos para cada factor, con el fin de poder estudiar más detenidamente algún aspecto especialmente relevante en el dominio del problema que se está desarrollando.

La mayor parte de las medidas (y prácticamente la totalidad de las fórmulas que las acompañan) constituyen una aportación inédita en este trabajo, a lo que contribuye también el hecho de que cada medida haya sido asociada a un criterio de calidad. Puede afirmarse que nunca nadie hasta este momento había desarrollado un conjunto de medidas de calidad tan amplio y detallado como el que aquí se presenta [Fuertes, 99]. El trabajo que más se aproxima es [Abreu, 94] que reúne 50 medidas (la mayoría no originales y ninguna de ellas incluye una fórmula para su aplicación directa), para el producto *software* orientado a objetos. Para obtener estas nuevas medidas, se han estudiado detenidamente los criterios establecidos y los atributos del *software* que podían contribuir a la calidad de dichos criterios.

El resto de las medidas están basadas o, simplemente, inspiradas en la amplia bibliografía sobre el tema. Para la incorporación de estas medidas, se ha estudiado su utilidad con relación a alguno de los criterios de calidad y su adecuación a la orientación a objetos, y se presentan adaptadas a cada situación específica o transformadas para permitir manejar los aspectos particulares de la orientación a objetos. Se ha decidido no incluir las referencias (salvo en algún caso muy concreto, en el que la medida se presenta casi sin alteraciones) debido a la gran diversidad de fuentes consultadas, lo cual hace que en muchas ocasiones no se tenga la certeza de cuál ha sido exactamente la fuente real de inspiración o, por el contrario, éstas han sido muy diversas. Tan solo sería de destacar, que hay un buen número de medidas inspiradas en

las ideas de muchas de las cerca de 170 medidas comentadas en el trabajo [McCall, 77] (aunque el 43% carecían de fórmulas).

En lo referente a algunas de las medidas relacionadas directamente con el lenguaje de implementación y que calculan aspectos particulares proporcionados por el lenguaje, éstas se han desarrollado teniendo en cuenta que se ha elegido el lenguaje de programación C++, según viene descrito en [Ellis, 91] [Stroustrup, 97] [ISO, 98], por ser éste uno de los de más amplia difusión en la programación orientada a objetos. Si se deseara utilizar otro de los lenguajes orientados a objetos existentes, no habría más que modificar o eliminar aquellas medidas relacionadas directamente con el lenguaje C++ e incorporar una serie de medidas propias del nuevo lenguaje escogido. También se utiliza este lenguaje en los ejemplos.

Hay que hacer notar que en las medidas específicas del lenguaje, en general, no se tienen en cuenta los defectos o errores que los compiladores pueden normalmente detectar. Tampoco se toman en consideración ciertas comprobaciones (como `x = x;`), puesto que existen gran número de herramientas [Meyers, 97] que advierten de estas situaciones. Por último, es necesario indicar que sólo se han considerado medidas estáticas, es decir, no se han estudiado las medidas dinámicas, que se evalúan durante la ejecución del sistema desarrollado.

Para nombrar a las medidas, con el fin de facilitar su consulta, se ha decidido utilizar la siguiente notación: el nombre estará compuesto por dos letras mayúsculas, que indicarán el criterio donde se encuentra definida, seguido por un subíndice que indica el número de orden que ocupa dentro de ese criterio.

Para cada medida se define la fórmula de cálculo, teniendo en cuenta que sus valores han sido normalizados[1] [Barba-Romero, 87] en el rango $[0, 1]$, indicando el cero un valor bajo del criterio y el uno un valor alto. Los diez tipos de normalizaciones utilizados más habitualmente en las fórmulas se muestran en la Figura 4.14, junto con las respectivas gráficas que las representan (X y T muestran los datos para construir la medida). Se han seleccionado estas normalizaciones debido a su sencillez y a que resultan suficientes para representar los distintos casos. Así, las fórmulas de la primera columna muestran curvas descendentes, mientras que las de la segunda, representan curvas ascendentes. Además, estudiando la primera columna (la segunda es análoga), las distintas fórmulas reflejan distinta pendiente, de tal forma que: (a) desciende bruscamente al principio y después lentamente a medida que se avanza, acercándose asintóticamente al valor cero; (c)[2] desciende linealmente; (e) desciende bastante al principio y después más suavemente hasta llegar al cero cuando se alcanza el valor T; (g) desciende lentamente al principio para apresurarse después; e (i) comienza su bajada al llegar a T para después proseguir un lento y asintótico descenso.

[1] El término normalización aquí utilizado simplemente hace referencia a una transformación que permite que los valores de las evaluaciones de cada medida sean comparables entre sí [Barba-Romero, 87].

[2] En el caso de las normalizaciones (c) y (d) se ha eliminado la función *máx* o *mín* de la fórmula de aquellas medidas en las que no es necesario por la naturaleza de los datos.

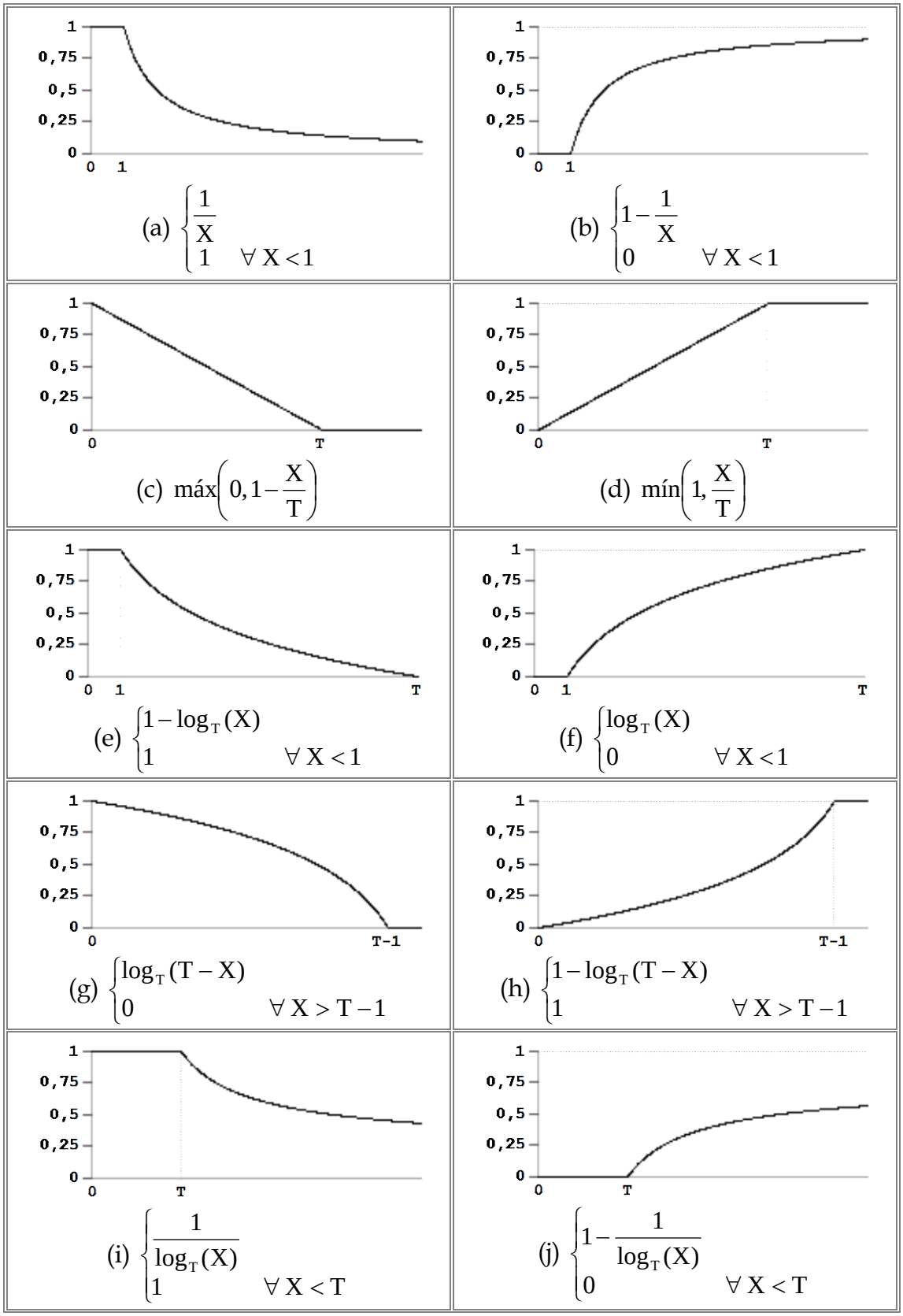


Figura 4.14: Tipos de normalizaciones

Con relación a las fórmulas, se entiende que en el caso de que la base del logaritmo o un denominador tome el valor cero o se intente calcular el logaritmo de un valor negativo significa la falta de aplicabilidad de la medida en ese caso concreto. También es

necesario mencionar que no todas las medidas son de aplicación en todos los sistemas. En tal situación, dichas medidas no deberán calcularse y no tomarán parte, por tanto, en la evaluación de la calidad del sistema.

Las fórmulas se han asignado a cada medida tras un estudio detallado del concepto que mide cada una de las medidas, apoyándose en una encuesta realizada a nueve ingenieros del *software* con amplia experiencia en orientación a objetos. En la Tabla 4.14 se muestra, a modo de ejemplo, una selección aleatoria de algunas de las medidas y respuestas proporcionadas por estos ingenieros del *software*. El procedimiento realizado para la realización de las encuestas fue proporcionarles el concepto que pretende evaluar cada medida de un criterio para que estimaran qué tipo de curva (de las indicadas en la columna de la izquierda de la Figura 4.14 más una curva discreta 0-1) refleja mejor el comportamiento que debe seguir. Con todas las respuestas proporcionadas por cada medida, se obtuvo el tipo de curva a incluir en la medida, teniendo en cuenta las respuestas mayoritarias y las características de las curvas indicadas.

	IS1	IS2	IS3	IS4	IS5	IS6	IS7	IS8	IS9	**Resultado**
SD_{22}	a-e	a	a-e	a	NC	c	g	a-e	a	**a**
CM_2	c	e	0-1	e	0-1	e	e	c	a	**e**
CM_{16}	g	g	a-e	g	c	e	c	g	g	**g**
CM_{18}	c	c	0-1	e	0-1	g	g	c	a	**c**
CM_{19}	c	g	0-1	g	0-1	e	g	c-g	a	**g**
CC_6	c	c	a-e	c	e	c	e	c	g	**c**
CO_2	c	c	i	a	a	c	e	c	a	**c**
CO_{15}	c	c	a-e	c	a	e	c	c-g	c	**c**
CN_2	i	e	e	e	e	i	e	e-i	NC	**e**
CN_3	e	e	NC	e	0-1	i	i	e-i	NC	**e**
CN_5	i	e	NC	e	0-1	e	i	e	NC	**e**
CN_7	i	e	e	e	0-1	g	i	e	NC	**e**
CS_2	e	e	e	e	a	c	e	a	e	**e**
CS_6	c	g	a-e	g	0-1	g	c	g	e	**g**
DC_3	e	a	a-i	e	0-1	0-1	c	e	e	**e**
DO_{10}	c	c	e	c	a	c	g	c	e	**c**
EE_6	0-1	a	i	a	a	a	i	a	g	**a**
ST_6	e	e	NC	e	0-1	a	a	e	NC	**e**
ST_7	e	e	a	e	0-1	e	c	e	NC	**e**
ST_{36}	a	a	NC	a	0-1	a	a	a	NC	**a**
EX_5	0-1	0-1	NC	a	NC	0-1	0-1	0-1	a	**0-1**
EX_8	c	c	i	c	NC	c	g	c	0-1	**c**
GE_{13}	e	e	c	e	e	e	i	e	g	**e**
MI_{11}	e	e	c	e	e	e	g	e	i	**e**
MO_{17}	i	i	a-e	i	0-1	i	a	i	g	**i**
AU_6	e-i	e	e-i	e	c	e	g	e	a	**e**
SI_9	g	g	c	g	a	g	a	g	e	**g**
SI_{17}	g	c	c	c	a	c	g	c	e	**c**
SI_{66}	i	i	g	i	a	i	c	i	NC	**i**
ET_{12}	g	g	a-e	g	0-1	g	e	g	e	**g**

Tabla 4.14: Resultados parciales de la encuesta sobre el comportamiento de las medidas (NC: no sabe/no contesta)

Evidentemente, se puede comprobar que no todas las medidas presentadas pueden automatizarse mediante un programa, en el estado actual de la tecnología. En este caso, el ingeniero del *software* deberá proporcionar el valor de la medida tras calcularla manualmente. Este cálculo lo podrá realizar evaluando la medida con exactitud, evaluando la medida sobre una muestra del programa o realizando una estimación subjetiva. En cualquier caso, se ha desarrollado una herramienta que permite evaluar las medidas automatizables sobre el código y que proporciona una interfaz amigable para que el usuario introduzca el resto de valores de las medidas (véase el apartado III.2).

En lo referente a las medidas cuyo valor se obtiene seleccionando una de las opciones de la lista dada, no se deben entender siempre en sentido estricto. Muchas de estas medidas tienen un cierto componente de subjetividad, por lo que el ingeniero del *software* podrá responder con valores intermedios.

Ejemplo: Sea la medida: "El código fuente tiene que estar adecuadamente comentado." Los valores posibles son:

1: el código está bien comentado
0: el código no está comentado
(0, 1): otro caso

Debe entenderse que se podrá asignar cualquier valor en el rango [0, 1] a juicio del ingeniero del *software*, pudiendo obtener, por ejemplo, un 0,75 si considera que está bastante bien comentado, un 0,10 si tuviera algún comentario esporádico o un 0,00 si carece de comentarios.

Debido al gran número de medidas que se presentan y a la dificultad para detallarlas concisamente, se ha optado, para su descripción, por agrupar cada medida dentro del criterio donde se ha englobado el atributo de la calidad del *software* que mide específicamente (no hay que olvidar que para que las medidas sean útiles es necesario determinar los atributos de calidad usados para identificarlas [Littlefair, 01]). En el caso de que una medida pueda atribuirse a más de un criterio, se definirá en el primero de ellos, según un orden alfabético, apareciendo únicamente una referencia en el resto de los criterios. Sobre cada una de ellas se proporciona una breve descripción o explicación de la medida. Además, se adjunta una indicación de las fases del ciclo de vida en las que pueden ser aplicadas y a qué elementos del sistema en concreto afecta. Esta información se presenta en dos grupos de palabras encerradas entre llaves con el siguiente significado:

- El primer grupo representa los elementos del sistema sobre los que es aplicable la medida. Las palabras utilizadas son:

Atributo	los atributos de una clase
Método	los métodos de una clase
Clase	las clases del sistema
Jerarquía	las jerarquías[3] de clases
Sistema	el sistema completo

[3] Por jerarquía de clases se entiende cualquier conjunto de clases (no vacío) relacionadas entre sí por una relación de generalización (herencia).

- El segundo grupo representa las fases del ciclo de vida a partir de las cuales se puede evaluar la medida. Las palabras utilizadas son:

Requisitos	fase de requisitos *software*
Análisis	fase de análisis orientado a objetos
Diseño	fase de diseño orientado a objetos
Implementación	fase de implementación orientada a objetos

Esto quiere decir que las medidas se calcularán sobre uno de los elementos del sistema, obteniéndose un vector de valores. Después, se propagará este vector hacia el siguiente elemento y así sucesivamente hasta llegar al sistema. Una explicación más detallada se dará posteriormente (véase el apartado 4.4.1).

Ejemplo: Sea la medida: "Debe evitarse la presencia de saltos puesto que atenta contra las normas elementales de la programación estructurada."

N: número de saltos

$$SI_{20} = \begin{cases} \frac{1}{N} & \\ 1 & \forall N < 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

El primer grupo tiene el siguiente significado: La palabra "Método" refleja que deberá evaluarse la medida para cada uno de los métodos de la clase. La palabra "Clase" indica que habrá que obtenerse un valor de la medida para cada clase a partir de los resultados obtenidos para los métodos. Análogamente, deberá obtenerse un valor para cada "Jerarquía" a partir de las clases y un único valor final para el "Sistema" a partir de las jerarquías.

Es decir, se obtendrá un valor de la medida SI_{20} para cada método. Posteriormente, los valores de los métodos de una clase se agregarán juntos para obtener el valor de SI_{20} de su clase. De esta forma, se obtendrá una serie de valores para las clases. Estos valores se agregarán de nuevo para cada jerarquía, obteniendo un conjunto de valores para las jerarquías, que, agregados por última vez, proporcionarán el valor de la medida SI_{20} para el sistema.

El segundo grupo refleja que esta medida solamente deberá calcularse durante las fases de "Diseño" e "Implementación".

Asimismo, para facilitar la comprensión de tal cantidad de medidas, con cada criterio se presenta un gráfico en el que aparecen reflejados los factores a los que afecta dicho criterio, junto con la lista de medidas incluidas. Cuando una de estas medidas sea una referencia a otra medida descrita con anterioridad, se incluirá en su lugar la medida referenciada, distinguiendo, en distintos tonos, las relaciones directas de las inversas[4] (véase el apartado 4.2.2. Criterios de Calidad). También se distinguirán aquellas medidas que serán referenciadas, tanto directa como inversamente, por alguna medida de otro criterio. Un resumen de la notación de este tipo de gráfico se muestra en la Figura 4.15, donde los conceptos que aparecen tienen el siguiente significado:

[4] El concepto de "inverso" utilizado aquí no se corresponde con el concepto matemático de función inversa, sino más bien con el de complementario.

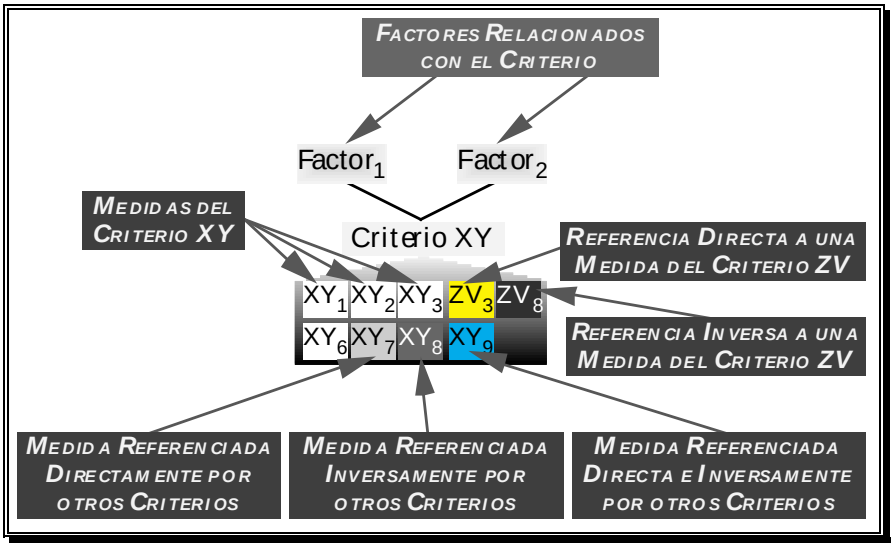


Figura 4.15: Notación de los gráficos de las medidas de los criterios

- Medidas del criterio XY: son las medidas definidas para el criterio actual (XY) y que no son utilizadas en ningún otro criterio. Es decir, estas medidas solo aparecerán una vez en el árbol de calidad, a diferencia de las otras, que podrán encontrarse varias veces.
- Medida referenciada directamente por otros criterios: son las medidas definidas para el criterio actual (XY) y que son utilizadas, tal cual, por otros criterios. Es decir, en otro criterio (por ejemplo, el VW) aparecerá la fórmula $VW_i = XY_7$.
- Medida referenciada inversamente por otros criterios: son las medidas definidas para el criterio actual (XY) y que son utilizadas, en sentido contrario, por otros criterios. Es decir, en otro criterio (por ejemplo, el VW) aparecerá una fórmula del estilo $VW_i = 1 - XY_8$.
- Medida referenciada directa e inversamente por otros criterios: son las medidas definidas para el criterio actual (XY) y que son utilizadas por otros criterios, en ambos sentidos. Es decir, es la unión de los dos casos anteriores, por lo que en otros criterios (por ejemplo, el VW y el TU) aparecerán fórmulas del estilo $VW_i = XY_9$ y $TU_j = 1 - XY_9$.
- Referencia directa a una medida del criterio ZV: son las medidas que ya han sido definidas en un criterio anterior, por lo que en el criterio actual solamente se pone su referencia tal cual, como por ejemplo, $XY_4 = ZV_3$.
- Referencia inversa a una medida del criterio ZV: son las medidas que ya han sido definidas en un criterio anterior aunque en sentido inverso, por lo que en el criterio actual se pone una referencia invirtiendo el significado de la medida, como por ejemplo, $XY_5 = 1 - ZV_8$.

La notación empleada para referirse en castellano a los distintos elementos de los sistemas se basa en la utilizada habitualmente en la orientación a objetos (clases, objetos, métodos, mensajes, atributos...), mientras que la empleada para hacer referencia a los elementos particulares y propios del lenguaje C++ está basada en la notación utilizada en [Fuertes, 93].

Por último, es necesario mencionar que se ha desarrollado una sencilla base de datos para almacenar y consultar las medidas del modelo, ofreciendo distintas posibilidades de búsqueda según distintos criterios (véase el apartado III.4).

A continuación se presentan, agrupadas por criterios (véase la Tabla 4.15), las 545 medidas obtenidas como resultado de este trabajo, de las cuales, 350 poseen una fórmula propia única. El resto constituyen referencias a fórmulas ya utilizadas en una medida de otro criterio descrito previamente, por lo que no se repite la fórmula.

Auditoría de accesos	3	Estabilidad	37
Auto-descriptivo	23	Estructuración	13
Completitud	29	Expansibilidad	25
Comunicaciones estándar	6	Generalidad	36
Comunicatividad	15	Independencia de la máquina	14
Concisión	16	Independencia del sistema *software*	9
Consistencia	35	Instrumentación	13
Control de acceso	4	Modularidad	30
Datos estándar	16	Operatividad	18
Documentación	19	Precisión	6
Eficiencia de almacenamiento	18	Seguimiento	3
Eficiencia de ejecución	23	Simplicidad[5]	102
Entrenamiento	10	Tolerancia a errores	22

Tabla 4.15: Distribución de medidas por criterios

4.3.1. AUDITORÍA DE ACCESOS (AA)

Hay tres medidas correspondientes a la auditoría de accesos (Figura 4.16), relacionada únicamente con el factor integridad, y las tres se pueden obtener mediante una lista de preguntas. Debe tenerse en cuenta que este criterio (y sus medidas) no siempre es necesario, pues en algunos sistemas no será preciso considerar esta posibilidad. Puede verse que las medidas de este criterio son independientes del paradigma utilizado en el desarrollo.

Figura 4.16: Medidas del criterio auditoría de accesos

[5] Podría argumentarse que la cantidad de medidas de la simplicidad es excesiva. No obstante, hay que tener en cuenta que este criterio resulta el más sencillo de evaluar (por la cantidad de atributos de calidad que influyen) y es en el que más trabajos se han publicado (de las referencias bibliográficas que incluyen medidas, aproximadamente el 60% presentan medidas de simplicidad).

AA_1: Deben realizarse previsiones para **almacenar los accesos al sistema**: el *software* debe disponer de mecanismos que permitan conocer quién, cuándo, dónde y cómo se accede al sistema.

1: se almacenan codificados o protegidos los accesos al sistema
0: no se almacenan los accesos al sistema
(0, 1): se almacenan sin codificar o proteger los accesos al sistema, o no se almacena toda la información necesaria

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

AA_2: Deben realizarse previsiones para **informar de los accesos al sistema**: el *software* debe disponer de mecanismos que permitan informar quién, cuándo, dónde y cómo se accede al sistema.

1: se informa de los accesos al sistema
0: no se informa de los accesos al sistema
(0, 1): no se informa completamente de los accesos al sistema

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

AA_3: Deben realizarse previsiones para **indicar inmediatamente los accesos no autorizados**: el sistema debe disponer de mecanismos que permitan informar quién, cuándo, dónde y cómo se accede al sistema sin disponer de autorización.

1: se informa inmediatamente de los accesos no autorizados
0: no se informa de los accesos no autorizados
(0, 1): se informa de los accesos no autorizados, aunque no inmediatamente o con ausencia de información

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

4.3.2. AUTO-DESCRIPTIVO (SD)

Hay un total de veintitrés medidas correspondientes al criterio auto-descriptivo (Figura 4.17), que está relacionado con cinco factores.

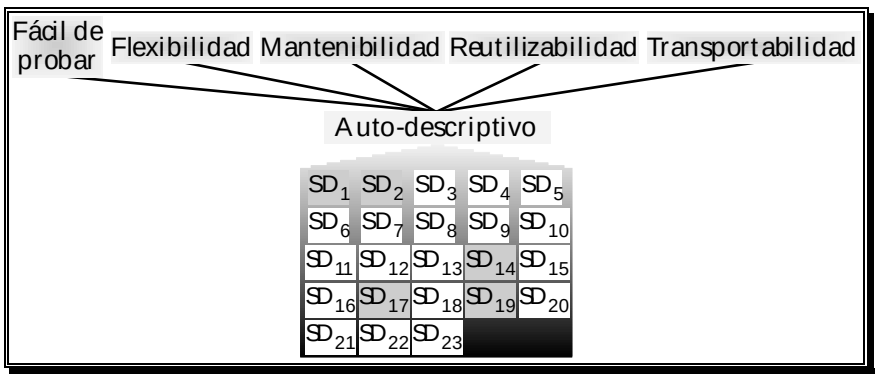


Figura 4.17: Medidas del criterio auto-descriptivo

SD_1: El código debe estar **adecuadamente comentado** puesto que esto ayuda a su estudio. Las líneas en blanco y, principalmente, las líneas con comentarios deben repartirse por todo el código. En la medida se ha premiado esencialmente los comentarios, aunque sin olvidar las líneas en blanco puesto que facilitan la legibilidad.

C: número de líneas del programa fuente con comentarios
B: número de líneas del programa fuente en blanco
N: número total de líneas de código fuente

$$SD_1 = \begin{cases} 1 - \log_{N-B}(N - B - C) \\ 1 \qquad\qquad \forall\, C > N - B - 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: consistencia, expansibilidad y simplicidad.

SD_2: Todas las **clases** deben disponer de una serie de **comentarios con una estructura estándar**, lo cual permite una fácil identificación de cada parte del comentario. Este estándar deberá ser definido por la organización al principio del proyecto, basándose en su experiencia o en alguna estructura de comentarios ya definida (como la de [Molloy, 95]).

N: número de clases con estructura de comentarios estándar
T: total de clases

$$SD_2 = \frac{N}{T}$$

{Jerarquía, Sistema} {Implementación}

Favorece a: modularidad.

SD_3: Los **métodos** deben disponer de una serie de **comentarios descriptivos con una estructura estándar**. Este tipo de información es especialmente valorada por el personal nuevo que tiene que trabajar con el *software* tras el desarrollo, llevando a cabo el mantenimiento, pruebas, cambios...

N: número de métodos con estructura de comentarios estándar
T: total de métodos

$$SD_3 = \frac{N}{T}$$

{Clase, Jerarquía, Sistema} {Implementación}

SD_4: Los **comentarios deben diferenciarse bien del código** de forma clara y uniforme (existen multitud de técnicas, como líneas en blanco, líneas de asteriscos, estilo de letra diferente, palabras específicas en mayúsculas...)

1: los comentarios se diferencian clara y uniformemente del código
0: los comentarios no se diferencian del código
(0, 1): los comentarios no siempre se diferencian bien del código

{Método, Clase, Jerarquía, Sistema} {Implementación}

SD_5: Todos los **saltos** (es decir, sentencias `goto`) y los **destinos de los saltos** deben ir **acompañados de comentarios**. Estos comentarios ayudan en la comprensión y en la habilidad necesaria para seguir la lógica de la función.

N: número de saltos y destinos con comentarios
T: total de saltos y destinos

$$SD_5 = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Implementación}

SD_6: Todas las **sentencias que dependan del *hardware*** (por ejemplo, el manejo de interrupciones) **o del entorno *software*** (por ejemplo, las llamadas al sistema) **han de llevar comentarios**. Estos comentarios no sólo son importantes para explicar qué se está haciendo, sino también para identificar claramente las partes del programa dependientes de la máquina.

N: número de sentencias dependientes del *hardware* o del *software* con comentarios
T: total de sentencias dependientes del *hardware* o del *software*

$$SD_6 = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Implementación}

SD_7: Todas las rutinas e instrucciones escritas en **código máquina o ensamblador deben llevar comentarios** para facilitar su legibilidad sin necesidad de estudiar dichas instrucciones.

N: número de rutinas en ensamblador comentadas
T: total de rutinas en ensamblador

$$SD_7 = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Implementación}

SD_8: Las **sentencias o construcciones no estándar** del lenguaje deben estar bien **comentadas**. Se entiende por construcciones no estándar aquéllas que son propias del compilador y no han sido incorporadas dentro del estándar del lenguaje. Por ejemplo, la declaración `char huge *s;` es una declaración propia del compilador de C++ de Borland, ya que la palabra `huge` no forma parte del estándar de C++ [ISO, 98].

N: número de sentencias no estándar comentadas
T: total de sentencias no estándar

$$SD_8 = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Implementación}

SD_9: Las **declaraciones** de todas las **variables, tipos de datos, etiquetas, identificadores constantes, nombres simbólicos, macros y directivas de compilación deben estar comentadas**, explicando su utilización, propiedades, misión, unidades...

N: declaraciones de variables, tipos de datos, etiquetas, identificadores constantes, nombres simbólicos, macros y directivas de compilación con comentarios
T: total de declaraciones de variables, tipos de datos, etiquetas, identificadores constantes, nombres simbólicos, macros y directivas de compilación

$$SD_9 = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SD_{10}: Las **declaraciones de los atributos deben estar comentadas**, explicando su utilización, propiedades, unidades...

N: número de declaraciones de atributos con comentarios
T: total de declaraciones de atributos

$$SD_{10} = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SD_{11}: Las **definiciones de los métodos deben estar comentadas**, explicando su utilización, propiedades, funcionamiento...

N: número de métodos con comentarios
T: total de métodos

$$SD_{11} = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SD_{12}: Los **parámetros formales** de los métodos **deben ir comentados**, explicando su utilización, propiedades, unidades...

N: número de parámetros formales con comentarios
T: total de parámetros formales

$$SD_{12} = \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SD_{13}: Los **comentarios** deben estar escritos de tal forma que **amplíen el significado del código**, aportando alguna información adicional útil. Por ejemplo, el comentario que acompaña a la sentencia `a = a + 1; /* se incrementa el valor de a en una unidad */` tiene un contenido nulo de información útil. Los comentarios deben decir el porqué se hace algo o qué hace un fragmento de código, pero no limitarse a traducir el código fuente a lenguaje natural. El ingeniero del *software* deberá estimar el valor de esta medida según su criterio.

1: los comentarios amplían el significado del código
0: los comentarios no aportan información
(0, 1): no todos los comentarios amplían el significado del código

{Atributo, Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SD_{14}: El **número ciclomático** de McCabe [McCabe, 76] mide el grado de complejidad de los distintos módulos del programa y, por tanto, puede entenderse como una medida de lo fácil o difícil de entender el código del programa.

- π: número de predicados (en sentencias condicionales o repetitivas) de la función (los predicados compuestos aportan una unidad al valor de π por cada condición que tengan)
- v: complejidad ciclomática de cada función
- k: el valor ideal máximo debería ser 4 según [Schneidewind, 79] ó 7 según [Bowen, 78], y nunca debería superar 10 según McCabe, por lo que se sugiere un valor de *k* entre 4 y 7, prefiriéndose el primero (o, incluso, inferior) para favorecer la sencillez de los métodos de las clases

$$v = \pi + 1$$

$$SD_{14} = \begin{cases} \dfrac{1}{\log_k(v)} \\ 1 \end{cases} \quad \forall\, v < k$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: simplicidad.

SD_{15}: La escritura del **código debe seguir una organización estándar y uniforme** (estructura de los comentarios, declaraciones, sentencias...), para facilitar su lectura e interpretación. Se pueden utilizar estándares de codificación como [Molloy, 95].

- N: número de clases que siguen una organización estándar del código
- T: total de clases

$$SD_{15} = \frac{N}{T}$$

{Jerarquía, Sistema} {Diseño, Implementación}

SD_{16}: Los **identificadores deben tener nombres claros y representativos**, que indiquen la propiedad física o funcional representada.

- N: número de identificadores con nombres realmente representativos
- T: total de identificadores

$$SD_{16} = \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SD_{17}: Los **módulos** (que incluyen las clases, métodos y demás elementos del lenguaje) **deben tener un sangrado del código de forma consistente**, siguiendo una serie de reglas uniformes predefinidas, lo cual favorece la lectura del código.

- N: número de módulos con sangrado uniforme del código
- T: total de módulos

$$SD_{17} = \frac{N}{T}$$

{Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia.

SD_{18}: El código fuente debe estar escrito de tal forma que **las sentencias no ocupen más de una línea y que cada línea contenga como máximo una sentencia** (se entiende que una línea finaliza con un salto de línea).

S: número de sentencias que ocupan más de una línea
L: número de líneas con más de una sentencia
T: total de líneas de código

$$SD_{18} = 1 - \frac{S + L}{T}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

SD_{19}: Los **nombres** que se dan a todos los **identificadores deben tener una estructura y un formato uniforme**, siguiendo alguna convención estándar dependiendo de su tipo, como por ejemplo, la notación húngara.

1: los nombres tienen una estructura y formato uniformes
0: los nombres no tienen una estructura y formato uniformes
(0, 1): no todos los nombres tienen una estructura y formato uniformes

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia y simplicidad.

SD_{20}: Los **nombres de los identificadores deben ser** lo suficientemente **descriptivos**, para lo cual se necesita que, por término medio, tengan una determinada longitud en cuanto al número de caracteres que los conforman.

N: número medio de caracteres del nombre de los identificadores
k: El ingeniero del *software* deberá determinar cuál es el valor más apropiado para *k*, aunque se recomienda un valor bajo (por ejemplo, entre 2 y 3). Un valor alto solo proporcionaría buenos resultados a la medida si los identificadores tuvieran un número de caracteres excesivamente elevado (si *k* fuera 10, para que la medida llegara a 0,5, los identificadores deberían tener 100 caracteres).

$$SD_{20} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \quad \forall\, N < k$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SD_{21}: Las **clases** deben ir acompañadas de **comentarios que clarifiquen su utilización y su objetivo**.

N: número de clases con comentarios
T: total de clases

$$SD_{21} = \frac{N}{T}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SD_{22}: Debe **evitarse** al máximo el uso de **valores constantes si éstos pueden ser sustituidos por identificadores constantes o nombres simbólicos**.

N: número de constantes susceptibles de cambio a identificador constante o nombre simbólico

$$SD_{22} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Implementación}

SD_{23}: La **comprensión del código no tiene que requerir de un gran esfuerzo mental** [Halstead, 77]. Esta medida, como otras de la "Ciencia del *Software*", se basa en la entropía.

N_1: número total de operadores
N_2: número total de operandos
N: longitud del programa o número total de operadores y operandos
η_1: número de operadores distintos
η_2: número de operandos distintos
η: número de operandos y operadores únicos
η_2^*: número de operandos potenciales
η^*: vocabulario potencial
V: volumen del programa
V^*: volumen potencial del programa
E: esfuerzo mental

$$N = N_1 + N_2 \qquad \eta = \eta_1 + \eta_2 \qquad \eta^* = 2 + \eta_2^*$$

$$V = N \cdot \log_2 \eta \qquad V^* = \eta^* \cdot \log_2 \eta^* \qquad E = \frac{V^2}{V^*}$$

$$SD_{23} = \frac{1}{\log E}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

4.3.3. COMPLETITUD (CM)

Hay un total de veintinueve medidas correspondientes a la completitud (Figura 4.18), y tiene influencia en dos de los factores del modelo de calidad: la corrección y la fiabilidad.

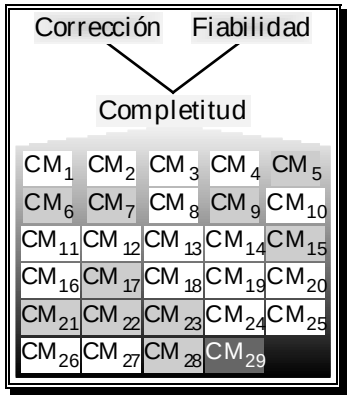


Figura 4.18: Medidas del criterio completitud

CM_1: Todos los **atributos deben estar definidos satisfactoriamente** (es decir, con su nombre, su tipo, su visibilidad o alcance...).

N: número de atributos no definidos, mal definidos o mal especificados
T: total de atributos

$$CM_1 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall\, N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_2: Todos los **métodos deben estar definidos satisfactoriamente** (es decir, con su nombre, su tipo, el número y tipo de sus argumentos, su visibilidad o alcance...).

N: número de métodos no definidos, mal definidos o mal especificados
T: total de métodos

$$CM_2 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall\, N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_3: Todas las **clases deben estar definidas satisfactoriamente** (es decir, con su nombre, sus miembros, su visibilidad o alcance...).

N: número de clases no definidas, mal definidas o mal especificadas
T: total de clases

$$CM_3 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall\, N < 1$$

{Jerarquía, Sistema} {Diseño, Implementación}

CM_4: Todas las **variables deben estar definidas satisfactoriamente** (es decir, con su nombre, su tipo, su visibilidad o alcance...).

N: número de variables no definidas, mal definidas o mal especificadas
T: total de variables

$$CM_4 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall\, N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_5: Cuando se manejan variables o atributos de tipo **puntero**, es **necesario obtener memoria** para almacenar sus correspondientes valores, bien solicitándola al sistema operativo, bien inicializando el puntero con otro puntero.

N: número de punteros para los que se obtiene memoria adecuadamente
T: total de declaraciones de punteros

$$CM_5 = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \quad N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia

CM_6: **Todos los atributos** que hayan sido definidos **deben utilizarse**. Todo atributo definido pero no utilizado puede reflejar una referencia que ha sido omitida.

N: número de atributos definidos en la clase que no son usados
T: total de atributos definidos

$$CM_6 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \quad N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: concisión, consistencia, eficiencia de almacenamiento y simplicidad.

CM_7: **Todos los métodos** que hayan sido definidos **deben utilizarse**. Todo método definido pero no utilizado puede reflejar una referencia que ha sido omitida.

N: número de métodos no usados
T: total de métodos

$$CM_7 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \quad N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: concisión, generalidad y simplicidad.

CM_8: **Todas las clases** que hayan sido definidas **deben utilizarse**. Toda clase definida pero no utilizada puede reflejar una referencia que ha sido omitida o bien ha sido definida innecesariamente.

N: número de clases no usadas
T: total de clases

$$CM_8 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \quad N < 1$$

{Jerarquía, Sistema} {Diseño, Implementación}

CM_9: **Todas las variables** que hayan sido definidas **deben utilizarse**. Toda variable definida pero no utilizada puede reflejar una referencia omitida.

N: número de variables no usadas
T: total de variables

$$CM_9 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: concisión.

CM_{10}: **Los atributos se deben utilizar tanto para modificar su valor como para consultarlo**. No realizar alguna de las dos operaciones implica algún tipo de omisión o defecto en la utilidad de dicho atributo. Como circunstancia particular, está el caso en el que solamente se modifica una vez su valor o solo se consulta una vez; esto puede representar la ausencia de usos del atributo.

L: número de veces que se usa el atributo para consultar su valor
E: número de veces que se usa el atributo para modificar su valor

$$CM_{10} = \begin{cases} 0 & \text{si } L = 0 \text{ ó } E = 0 \\ 0.75 & \text{si } L = 1 \text{ ó } E = 1 \\ 1 & \text{otro caso} \end{cases}$$

{Atributo, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{11}: **Las referencias a los atributos deben quedar resueltas**, es decir, no deben ser ambiguas (el término referencia no alude a las referencias de C++ con el símbolo &).

N: número de referencias ambiguas a atributos
k: se recomienda un valor bajo (entre 2 ó 3) para que cualquier anomalía pueda ser detectada rápidamente

$$CM_{11} = \begin{cases} 1 - \log_k(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{12}: **Las referencias a los métodos deben poderse resolver**, es decir, no deben dar lugar a ambigüedad.

N: número de referencias ambiguas a métodos
k: se recomienda un valor bajo (entre 2 ó 3) para que cualquier anomalía pueda ser detectada rápidamente

$$CM_{12} = \begin{cases} 1 - \log_k(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{13}: **Las referencias a las clases no deben ser ambiguas**.

N: número de referencias ambiguas a clases
k: se recomienda un valor bajo (entre 2 ó 3) para que cualquier anomalía pueda ser detectada rápidamente

$$CM_{13} = \begin{cases} 1 - \log_k(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{14}: **Las referencias a las variables no deben ser ambiguas**.

N: número de referencias ambiguas a variables
k: se recomienda un valor bajo (entre 2 ó 3) para que cualquier anomalía pueda ser detectada rápidamente

$$CM_{14} = \begin{cases} 1 - \log_k(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{15}: Si alguno de los **atributos** de una clase necesita una **gestión dinámica de memoria** o interviene en una **jerarquía**, entonces dicha clase debe contener un **constructor por omisión** (método `Create` en Eiffel), un **constructor de copia** (método `Clone` en Eiffel), un **destructor** y el **operador de asignación definidos explícitamente**, aunque el lenguaje o la clase no lo exija. De esta forma se puede definir explícitamente su comportamiento sin dejar que el compilador les asigne un funcionamiento por omisión. Si el lenguaje carece de alguno de estos elementos, debe omitirse la parte correspondiente de la fórmula.

O = 1: si se ha definido el constructor por omisión
C = 1: si se ha definido el constructor de copia
D = 1: si se ha definido el destructor
A = 1: si se ha definido el operador de asignación

$$CM_{15} = \frac{O + C + D + A}{4}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia y estructuración.

CM_{16}: Los **parámetros actuales** de los mensajes enviados a los objetos **deben coincidir en cuanto al número y al tipo de los posibles parámetros formales** del método que recibirá el mensaje. No resulta aconsejable que el sistema realice conversiones de tipos automáticas (*casting*).

N: número de mensajes enviados con distinto número de argumentos a los previstos o con argumentos con tipos diferentes (aunque sean compatibles)
T: total de mensajes enviados

$$CM_{16} = \begin{cases} \log_T(T - N) \\ 0 \end{cases} \quad \forall \; N > T - 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{17}: **Las clases deben contener métodos**, puesto que en caso contrario pueden reflejar la ausencia de servicios proporcionados por la clase.

N: número de clases sin métodos

$$CM_{17} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia

CM_{18}: **Todos los elementos referenciados** (atributos, métodos, variables, tipos...) **deben haber sido declarados previamente**. Un sistema no está completo si, en cualquiera de sus fases de desarrollo, se pretenden utilizar elementos no declarados.

N: número de elementos referenciados declarados
T: número de elementos referenciados

$$CM_{18} = \begin{cases} 1 - \log_T (T - N) \\ 1 \end{cases} \quad \forall \quad N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{19}: Todas **las consultas del valor de un identificador deben realizarse tras haber sido definido** previamente dicho identificador. Esta definición puede realizarse con datos en una inicialización, calculándolos en una expresión u obteniéndolos en una llamada a una función (que podría conseguirlos del exterior). Es decir, cada dato debe tener un origen específico. En estos identificadores también se incluyen los punteros.

N: número de referencias a datos definidos, calculados u obtenidos de una función
T: total de referencias a datos

$$CM_{19} = \begin{cases} 1 - \log_T (T - N) \\ 1 \end{cases} \quad \forall \quad N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{20}: **Los puntos de decisión deben tener sus condiciones definidas así como disponer de código alternativo** (por ejemplo, podría ser recomendable poner la componente `else` en la sentencia condicional o `default` en la sentencia de selección múltiple [Borrajo, 90]). Todas las alternativas deben ser previstas en tiempo de diseño para no dejar cabos sueltos en la implementación.

N: número de condiciones definidas y con código alternativo en los puntos de decisión
T: total de puntos de decisión

$$CM_{20} = \begin{cases} \log_T (N) \\ 0 \end{cases} \quad \forall \quad N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

CM_{21}: **Los tipos abstractos de datos** (TAD) creados **deberán estar dotados del conjunto de operadores necesarios** según el TAD, redefiniendo y adaptando (por medio de la sobrecarga de operadores) el comportamiento habitual de dichos operadores.

N: número de operadores redefinidos para el tipo abstracto de datos
T: total de operadores que tienen significado para el tipo abstracto de datos

$$CM_{21} = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: datos estándar

CM_{22}: Las **clases y métodos que puedan ser aplicadas sobre distintos tipos de datos deben estar parametrizadas** (ser de tipo `template` en C++ [Stroustrup, 88b]) o bien sobrecargadas.

N: número de clases y métodos parametrizados o sobrecargados
T: total de clases y métodos que se aplican a más de un tipo de datos

$$CM_{22} = \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

CM_{23}: Las **clases deben contener atributos** (o relaciones estructurales con otras clases), puesto que una ausencia de atributos (o relaciones) puede reflejar una incompletitud de la clase.

N: número de clases sin atributos (o relaciones)

$$CM_{23} = \frac{1}{N+1}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, datos estándar y estructuración.

CM_{24}: Las **clases concretas que hereden de una clase abstracta deben tener redefinidos los métodos virtuales puros heredados**.

N: número de clases hijas de una clase abstracta sin redefinición de un método virtual puro heredado
T: total de clases hijas de una clase abstracta

$$CM_{24} = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$ {Jerarquía, Sistema} {Diseño, Implementación}

CM_{25}: **El análisis tiene que satisfacer todos los requerimientos descritos en la especificación de requisitos *software***. Hay que tener en consideración las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual implica modificaciones a realizar en el análisis.

1: el análisis satisface los requisitos
0: el análisis no satisface los requisitos
(0, 1): el análisis satisface parcialmente los requisitos

{Sistema} {Análisis}

CM_{26}: **El diseño tiene que satisfacer todos los aspectos contemplados en el análisis**. Hay que tener en cuenta las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual hace cambiar el análisis y, por tanto, supone cambios en el diseño.

1: el diseño satisface el análisis
0: el diseño no satisface el análisis
(0, 1): el diseño satisface parcialmente el análisis

{Sistema} {Diseño}

CM_{27}: **La implementación tiene que satisfacer todos los aspectos contemplados en el diseño**. Hay que considerar las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual hace cambiar el análisis y el diseño, por lo que hay que mantener actualizada la implementación.

1: la implementación satisface el diseño
0: la implementación no satisface el diseño
(0, 1): la implementación satisface parcialmente el diseño

{Sistema} {Implementación}

CM_{28}: Tienen que **utilizarse todos los parámetros formales de los métodos**.

N: número de parámetros no usados
T: total de parámetros

$$CM_{28} = \begin{cases} \log_T(T-N) \\ 0 \end{cases} \quad \forall \; N > T-1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

CM_{29}: Conviene que **los métodos sean utilizados varias veces**.

N: número de veces que se usa cada método
k: el ingeniero del *software* deberá definir el valor más apropiado de esta constante, intentando que no tenga un valor demasiado elevado; este valor representa el mínimo número de veces que debe usarse cada método

$$CM_{29} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \quad \forall\, N < k$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: estabilidad.

4.3.4. COMUNICACIONES ESTÁNDAR (CC)

Se han definido seis medidas correspondientes al criterio comunicaciones estándar (Figura 4.18), que está relacionado con cuatro de los factores presentes en el modelo de calidad.

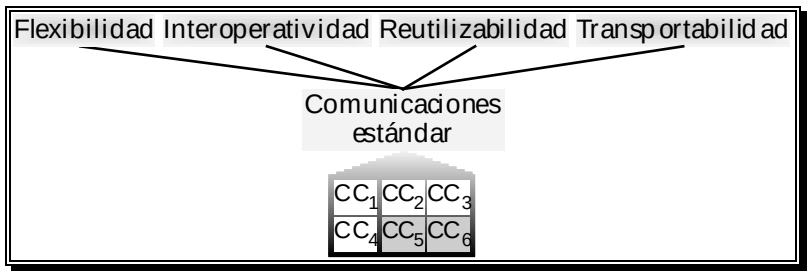


Figura 4.19: Medidas del criterio comunicaciones estándar

CC_1: Deben estar definidos los **requisitos de comunicación con otros sistemas o máquinas**.

1: están definidos los requisitos de comunicación con otros sistemas o máquinas
0: no lo están
(0, 1): no están adecuadamente definidos

{Sistema} {Requisitos}

CC_2: Deben establecerse **protocolos estándar de comunicaciones con otros sistemas** (por ejemplo, mediante RPC [Moulet, 92]).

1: se han establecido protocolos estándar de comunicaciones
0: no se han establecido
(0, 1): no se han establecido correctamente

{Sistema} {Diseño, Implementación}

CC_3: Deben establecerse **interfaces para facilitar la entrada desde otros sistemas** (por ejemplo, si se está desarrollando una aplicación en MS-Windows [Petzold, 96], a través de OLE, DDE, Active-X...).

N: número de interfaces definidas para la entrada desde otro sistema
T: total de interfaces de entrada distintas requeridas

$$CC_3 = \frac{N}{T}$$ {Sistema} {Análisis, Diseño, Implementación}

CC_4: Deben establecerse **interfaces para facilitar la comunicación de salida hacia otros sistemas**.

N: número de interfaces definidas para la salida a otro sistema
T: total de interfaces de salida distintas requeridas

$$CC_4 = \frac{N}{T}$$ {Sistema} {Análisis, Diseño, Implementación}

CC_5: La **entrada de información** al sistema debe estar **localizada en pocos módulos o clases**, puesto que cuanto mayor sea el número de módulos que actúan como interfaz de entrada con otro sistema, más complicado resulta esta comunicación, así como la implementación de los protocolos estándar.

N: número de clases en las que se realiza la entrada de datos desde otros sistemas
T: total de clases

$$CC_5 = 1 - \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: independencia de la máquina.

CC_6: La **salida de la información** proporcionada por el sistema debe estar **localizada en pocos módulos o clases**, puesto que cuanto mayor sea el número de módulos que actúan como interfaz de salida con otro sistema, más complicado resulta esta comunicación, así como la implementación de los protocolos estándar.

N: número de clases en las que se realiza la salida de datos hacia otros sistemas
T: total de clases

$$CC_6 = 1 - \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: independencia de la máquina.

4.3.5. COMUNICATIVIDAD (CO)

Hay quince medidas correspondientes a la comunicatividad (Figura 4.20), que está relacionada con dos de los factores.

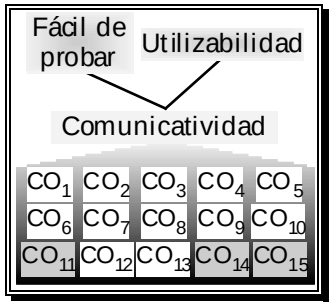


Figura 4.20: Medidas del criterio comunicatividad

CO_1: Deben **definirse valores por omisión en los datos de entrada del usuario**, puesto que es una forma de minimizar la cantidad de información requerida para su introducción. Evidentemente, en ciertos entornos, como es el de la Inteligencia Artificial, puede ser necesario desactivar esta medida puesto que el desconocimiento de un dato es un tipo de información que se puede manejar.

N: número de valores por omisión en los parámetros de entrada del usuario
T: total de parámetros de entrada

$$CO_1 = \begin{cases} \log_T(N) \\ 0 \qquad \forall\ N < 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

CO_2: Deben **definirse formatos uniformes para la introducción de los mismos tipos de datos**, puesto que cuanto mayor sea el número de formatos diferentes, será más difícil de utilizar el sistema.

N: número de tipos de datos para los que se realiza entrada de información
T: total de formatos diferentes de entrada

$$CO_2 = \text{máx}\left(0, 1 - \frac{N}{T}\right)$$

{Sistema} {Análisis, Diseño, Implementación}

CO_3: **Los registros de entrada deben estar claramente identificados**.

N: número de registros de entrada sin identificar claramente
T: total de registros de entrada

$$CO_3 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$

{Sistema} {Implementación}

CO_4: **Las entradas de datos tienen que poder ser verificadas por el usuario antes de ser procesadas** (por ejemplo, mostrando un eco de la entrada o presentándolas en pantalla bajo petición).

N: número de entradas que el usuario no puede validar o verificar
T: total de entradas

$$CO_4 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

CO_5: **Los registros de entrada deben finalizar con una instrucción de fin de entrada explícita**.

N: número de entradas sin fin de entrada explícita
T: total de entradas

$$CO_5 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$ {Sistema} {Diseño, Implementación}

CO_6: Debe existir una previsión para realizar las **entradas de información al sistema desde diferentes dispositivos**, lo cual proporciona una gran flexibilidad en la entrada de datos.

N: número de dispositivos desde los que se puede realizar la entrada
T: total de dispositivos desde los que se debe poder realizar la entrada según la especificación de requisitos

$$CO_6 = \begin{cases} 1 - \log_T(T - N) \\ 1 \qquad \forall\ N > T - 1 \end{cases}$$ {Sistema} {Diseño, Implementación}

CO_7: Deben **proporcionarse controles selectivos** (salidas específicas, formatos, precisión...) **para la salida de información**.

N: número de salidas sin controles
T: total de salidas

$$CO_7 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$ {Sistema} {Diseño, Implementación}

CO_8: **Deben poderse identificar perfectamente cada una de las salidas o resultados** proporcionados por el sistema.

N: número de salidas sin identificar
T: total de salidas

$$CO_8 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$ {Sistema} {Implementación}

CO_9: Deben definirse **formatos uniformes para las salidas de datos del mismo tipo**.

N: número de tipos de datos para los que se realiza salida de información
T: total de formatos diferentes de salida

$$CO_9 = \text{máx}\left(0, 1 - \frac{N}{T}\right)$$ {Sistema} {Análisis, Diseño, Implementación}

CO_{10}: Las **salidas** deben estar **agrupadas por temáticas y separadas del resto** (por líneas en blanco, líneas, asteriscos, distintos colores...).

N: número de salidas sin agrupar
T: total de salidas

$$CO_{10} = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall\ N < 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

CO_{11}: Los **mensajes de error y avisos al usuario deben ser claros y no ambiguos**. Su redacción debe cumplir una serie de normas generales para facilitar la interacción usuario-máquina: relacionados con el error detectado, expresados en términos que el usuario pueda comprender (no en terminología del programador), específicos, localizados, completos, legibles, amables...

N: número de mensajes y avisos claros sin ambigüedad
T: total de mensajes y avisos

$$CO_{11} = \frac{N}{T}$$

{Sistema} {Implementación}

Favorece a: instrumentación y operatividad.

CO_{12}: Debe existir una **previsión para poder enviar las salidas del sistema a diferentes dispositivos**.

N: número de dispositivos a los que se puede enviar la salida
T: total de dispositivos para la salida previstos en los requisitos

$$CO_{12} = \begin{cases} 1 - \log_T(T - N) \\ 1 \qquad \forall\ N > T - 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

CO_{13}: La **interfaz de entrada/salida debe seguir una serie de normas de utilización estándar**, con un aspecto y forma de manejo uniforme (por ejemplo, las aplicaciones para ordenadores Apple Macintosh deben seguir el estilo y el *look and feel* de la interfaz uniforme que ofrece este ordenador).

1: interfaz de usuario estándar
0: interfaz de usuario no estándar
(0, 1): interfaz de usuario no completamente estándar

{Sistema} {Diseño, Implementación}

CO_{14}: La interfaz de usuario debe permitir a la persona **operar con el sistema de forma ergonómica y confortablemente**. Debe ser sencilla, atractiva y fácil de utilizar.

1: interfaz de usuario amigable
0: interfaz de usuario no amigable

{Sistema} {Diseño, Implementación}

Favorece a: entrenamiento y operatividad.

CO_{15}: Todos los **errores** que se puedan producir deberán ir **asociados con un mensaje de error**, así como disponer de **una explicación adicional** para aumentar la información y aclarar al máximo la situación al usuario.

N: número de mensajes de error con explicación adicional
T: total de mensajes de error

$$CO_{15} = \frac{N}{T}$$

{Sistema} {Implementación}

Favorece a: instrumentación.

4.3.6. CONCISIÓN (CN)

Hay un total de dieciséis medidas correspondientes a la concisión (Figura 4.21), que está relacionada con dos de los factores. Tres de ellas son referencias a medidas ya presentadas en otro de los criterios, en este caso, en la completitud.

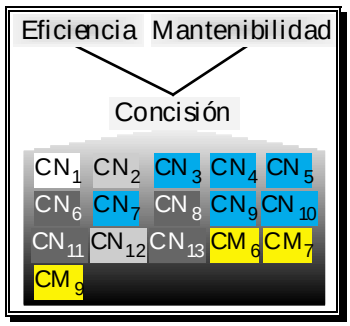


Figura 4.21: Medidas del criterio concisión

CN_1: Comparación entre la longitud real del código y la estimada, basándose en el concepto de longitud de [Halstead, 77].

N_1: número total de operadores
N_2: número total de operandos
N: longitud total
η_1: número de operadores distintos
η_2: número de operandos distintos
$\hat{N}$: longitud total estimada

$$N = N_1 + N_2 \qquad \hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

$$CN_1 = \begin{cases} 1 - \left| \frac{\hat{N} - N}{N} \right| \\ 0 \quad si < 0 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

CN_2: La presencia de **métodos operador permite que el código que los utilice sea más conciso**.

N: número de métodos operador
T: total de métodos

$$CN_2 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall\ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, expansibilidad, generalidad y modularidad.

CN_3: La presencia de **métodos virtuales** (`deferred` en Eiffel o métodos genéricos en CLOS) **permite que el código fuente sea más conciso**. Esto se debe a que no hay que generar un código específico para seleccionar el método apropiado durante el desarrollo, ya que se enlaza dinámicamente durante la ejecución.

N: número de métodos virtuales
T: total de métodos

$$CN_3 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall\ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: expansibilidad y simplicidad.
Perjudica a: eficiencia de ejecución.

CN_4: La **reutilización de métodos mediante la herencia hace que el código de las clases disminuya** al no tener que redefinir los métodos que se heredan. Cuantas más clases hereden un mismo método, mayor será el beneficio obtenido de dicho método.

N: número de clases que heredan cada uno de los métodos

$$CN_4 = \begin{cases} 1 - \dfrac{1}{N} \\ 0 \end{cases} \quad \forall\ N < 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.
Perjudica a: estabilidad y simplicidad.

CN_5: La definición de **métodos genéricos** (`template` en C++) **evita tener que definir varios métodos con el mismo comportamiento para distintos tipos de datos.**

N: número de métodos genéricos
T: total de métodos

$$CN_5 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall\ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, datos estándar, expansibilidad y generalidad.
Perjudica a: simplicidad.

CN_6: **Si un método es capaz de admitir un número variable de parámetros** (en C++ se implementa utilizando la elipsis), **se podrá utilizar en distintas situaciones evitando tener que definir varios métodos similares**, cuya única diferencia sea el número de parámetros formales.

N: número de métodos con número variable de argumentos
T: total de métodos

$$CN_6 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: simplicidad.

CN_7: La definición de **clases genéricas** (`generic` en Eiffel) **evita tener que definir varias clases con las mismas funcionalidades y estructura para ser utilizadas con distintos tipos de datos**.

N: número de clases genéricas
T: total de clases

$$CN_7 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall \ N < 1$$

{Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, datos estándar, estructuración, expansibilidad y generalidad.
Perjudica a: simplicidad.

CN_8: **Una clase que pueda heredar de muchas otras clases evita tener que implementar de nuevo la mayoría de sus funciones**. Es decir, para este criterio, se considera beneficiosa la herencia múltiple, si bien, para otros criterios será perjudicial.

N: número de padres directos de una clase

$$CN_8 = \begin{cases} 1 - \frac{1}{N} \\ 0 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Perjudica a: estabilidad y simplicidad.

CN_9: **La reutilización de las clases sin modificación por medio del uso o la herencia evita tener que definir nuevas clases iguales para realizar las mismas tareas**.

N: número de veces que se usa una clase, tal cual, por relaciones de uso o herencia

k: se recomienda un valor muy bajo (por ejemplo, 2) para que los valores de la medida crezcan rápidamente, puesto que las clases no se suelen reutilizar muchas veces dentro de un sistema.

$$CN_9 = \begin{cases} 1 - \dfrac{1}{\log_k(N)} & \\ 0 & \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, estructuración, generalidad y modularidad.
Perjudica a: estabilidad y simplicidad.

CN_{10}: **La reutilización de las clases, modificando su comportamiento, por medio de la herencia evita tener que definir completamente nuevas clases semejantes para realizar tareas similares**.

N: número de veces que se usa una clase, adaptándola, por herencia

k: se recomienda un valor muy bajo (por ejemplo, 2) para que los valores de la medida crezcan rápidamente, puesto que las clases no se suelen reutilizar muchas veces dentro de un sistema.

$$CN_{10} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} & \\ 0 & \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: consistencia, estructuración, expansibilidad, generalidad y modularidad.
Perjudica a: estabilidad y simplicidad.

CN_{11}: Dentro de una jerarquía de clases, **cuanto mayor sea el número de herencias, más concisas serán las clases de dicha jerarquía**, debido a la reutilización.

N: número de herencias en la jerarquía

k: se recomienda un valor muy bajo (por ejemplo, 2) para que los valores de la medida crezcan rápidamente, puesto que el número de herencias en una jerarquía no suele ser muy elevado.

$$CN_{11} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} & \\ 0 & \forall\, N < k \end{cases}$$

{Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: estabilidad, modularidad y simplicidad.

CN_{12}: La utilización de **librerías estándar de clases**, proporcionadas por muchos de los compiladores de lenguajes orientados a objetos, **favorece la concisión**, puesto que muchos componentes no tendrán que ser implementados por el programador.

1: se usan las librerías de clases del compilador
0: no se usan o no están disponibles
(0, 1): otro caso

{Sistema} {Implementación}

Favorece a: datos estándar y generalidad.

CN_{13}: **El enlace dinámico permite la escritura de un código más conciso**, debido a que la llamada al método adecuado se realiza automáticamente según el objeto utilizado, no teniéndose que preocupar de ello el programador.

N: número de mensajes que se resuelven en tiempo de ejecución
T: total de mensajes

$$CN_{13} = \frac{N}{T}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: eficiencia de ejecución y simplicidad.

$CN_{14} = CM_6$

Todos los atributos que hayan sido definidos deben utilizarse. Todo atributo definido pero no utilizado puede reflejar una definición innecesaria, disminuyendo la concisión.

$CN_{15} = CM_7$

Todos los métodos que hayan sido definidos deben utilizarse. Todo método definido pero no utilizado puede reflejar una definición innecesaria, aumentando inútilmente el tamaño del código.

$CN_{16} = CM_9$

Todas las variables que hayan sido definidas deben utilizarse. Toda variable definida pero no utilizada puede reflejar una definición no necesaria, disminuyendo la concisión del código.

4.3.7. CONSISTENCIA (CS)

Hay un total de treinta y cinco medidas correspondientes a la consistencia (Figura 4.22), que está relacionada con tres de los factores. De ellas, trece son referencia directa a medidas previas de otros criterios.

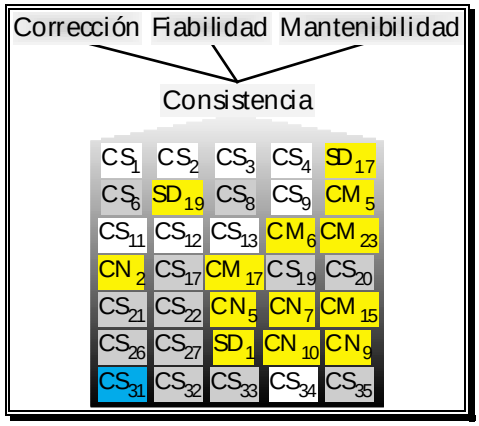


Figura 4.22: Medidas del criterio consistencia

CS_1: Debe realizarse una **representación estándar del análisis y del diseño** utilizando, además, alguna metodología orientada a objetos. Debe definirse y seguirse un estándar de representación de los elementos del sistema.

1: se utiliza una representación estándar
0: no se utiliza una representación estándar
(0, 1): otro caso

{Sistema} {Análisis, Diseño}

CS_2: Deben establecerse una serie de **convenciones uniformes para el manejo consistente de las operaciones de entrada/salida**.

N: número de clases que violan las convenciones para la entrada/salida
T: total de clases que realizan entrada/salida

$$CS_2 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Jerarquía, Sistema} {Diseño, Implementación}

CS_3: Deben establecerse una serie de **convenciones uniformes para el manejo consistente de los errores** que pudieran producirse.

N: número de clases que no cumplen las convenciones para el manejo de errores
T: total de clases

$$CS_3 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Jerarquía, Sistema} {Diseño, Implementación}

CS_4: **Ley de Demeter**: Para todas las clases *C*, y para todos sus métodos *M*, todos los objetos a los que *M* envía un mensaje deben ser: (1) un objeto enviado como parámetro a *M*, incluido el objeto indicado por `this` (en C++), `Current` (en Eiffel) o `self` (en Smalltalk o Flavors); o (2) un objeto que sea atributo de la clase *C* [Lieberherr, 88].

N: número de veces que se viola la ley de Demeter

$$CS_4 = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\ N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$CS_5 = SD_{17}$

Los módulos del programa deben tener un **sangrado del código de forma uniforme** en todo el programa, siguiendo una serie de reglas estándar predefinidas.

CS_6: Se debe utilizar una **representación estándar para los tipos abstractos de datos**.

N: número de tipos abstractos de datos que siguen la representación estándar
T: total de tipos abstractos de datos

$$CS_6 = \begin{cases} 1 - \log_T(T - N) \\ 1 \qquad\qquad \forall\ N > T - 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

Favorece a: datos estándar.

$CS_7 = SD_{19}$

Debe seguirse una **convención uniforme en los nombres** que se dan a todos los identificadores.

CS_8: Debe seguirse una **estructura uniforme en la forma y en el orden de definir todos los miembros de las clases**.

N: número de clases cuya definición no sigue las convenciones
T: total de clases

$$CS_8 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad\qquad \forall\ N < 1 \end{cases}$$

{Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: datos estándar.

CS_9: Cuando en una aplicación se usa memoria dinámica, debe **emplearse un único gestor de memoria**, puesto que la utilización de más de uno puede llevar a un mal funcionamiento y a inconsistencias de la información almacenada (por ejemplo, en C++ están disponibles dos gestores: el proporcionado por el operador `new` de C++ y el proporcionado por las funciones de la librería `alloc.h` de C).

1: se utiliza un único gestor de memoria dinámica (en C++, `new`)
0.5: se utiliza un gestor de memoria no propio del lenguaje (en C++, los de `alloc.h`)
0: se utilizan simultáneamente varios gestores de memoria dinámica

{Sistema} {Implementación}

$CS_{10} = CM_5$

Al utilizar **punteros**, **debe obtenerse memoria** para almacenar sus valores, de forma que la información siempre sea consistente.

CS_{11}: La **redefinición de los operadores debe hacerse de manera uniforme, sin modificar su semántica**.

N: número de definiciones de operadores que no conservan la semántica
T: total de definiciones de operadores

$$CS_{11} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

CS_{12}: El código debe estar comentado adecuadamente, de tal forma que los **comentarios tengan una estructura uniforme y predefinida** para las clases, métodos, tipos de datos, ficheros...

N: número de comentarios con estructura no uniforme
T: total de comentarios

$$CS_{12} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Atributo, Método, Clase, Jerarquía, Sistema} {Implementación}

CS_{13}: Todo desarrollo de *software* debe ir acompañado de una serie de documentación y manuales (de usuario, técnico...). Todos los **manuales deben seguir un esquema uniforme**.

N: número de manuales con estructura no uniforme
T: total de manuales

$$CS_{13} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

$CS_{14} = CM_6$

Los atributos definidos deben utilizarse, puesto que, si no lo son, puede ser un síntoma de algún tipo de inconsistencia.

CS_{15}= CM_{23}

Las clases deben contener atributos definidos, para ser consistentes con la idea de la orientación a objetos [Rumbaugh, 91].

CS_{16}= CN_2

Los **métodos operador permiten una mayor consistencia** en el código generado, al poder realizar la llamada de igual forma que cuando se utiliza un operador normal.

CS_{17}: Aquellos **métodos que no modifican el valor de los atributos del objeto** desde el cual ha sido llamado **deben ser de tipo constante** para conservar su consistencia.

N: número de métodos declarados como constantes
T: total de métodos constantes

$$CS_{17} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Clase, Jerarquía, Sistema} {Implementación}

Favorece a: estructuración.

CS_{18}= CM_{17}

Las **clases deben ofrecer una serie de servicios por sí mismas**. Si no ofrecen ningún servicio, entonces es que se está utilizando (incorrectamente) una clase como una estructura (registro) de datos. Si solo ofrece servicios heredados, lo más probable es que la clase no aporte nada nuevo, por lo que es cuestionable su existencia. En estos casos, las clases no son consistentes con la idea de clase.

CS_{19}: El **uso de atributos por otras clases no resulta consistente con la idea de la ocultación de la información**, uno de los aspectos claves de la orientación a objetos.

N: número de atributos usados desde el exterior de la clase
T: total de atributos definidos en la clase

$$CS_{19} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad, estructuración, expansibilidad, modularidad y simplicidad.

CS_{20}: Un **método no debe utilizar atributos externos** de otras clases.

N: número de atributos externos utilizados por cada método

$$CS_{20} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad, estructuración, expansibilidad, modularidad y simplicidad.

CS_{21}: **Cada método solamente debe estar definido con el mismo nombre en una clase**. Hay que exceptuar a los métodos operador y aquellos métodos que son redefiniciones o sobrecargas de un método heredado.

N: número de clases que tienen un método definido con el mismo nombre

$$CS_{21} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: modularidad y simplicidad.

CS_{22}: **Las funciones no miembro no son consistentes con la idea de una programación orientada a objetos**, basada en unas clases que proporcionan una serie de servicios. Hay que exceptuar las funciones obligatorias de ciertos lenguajes (como la función `main` de C++).

N: número de funciones no miembro

$$CS_{22} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estructuración, modularidad y simplicidad

$CS_{23} = CN_5$

Los **métodos genéricos favorecen la consistencia** del programa al proporcionar un mismo servicio (con el mismo nombre e interfaz) para distintos tipos de datos.

$CS_{24} = CN_7$

Las **clases genéricas favorecen la consistencia** del programa al disponer de una misma organización de datos y servicios aplicable a distintos tipos de datos.

$CS_{25} = CM_{15}$

Para lograr mejorar la consistencia en cuanto a los elementos que deben contener las **clases con gestión dinámica de memoria o que participen en una jerarquía**, es necesario definir explícitamente un **constructor por omisión, un constructor de copia, un destructor y el operador de asignación**.

CS_{26}: Resulta aconsejable repartir **cada clase** de un programa **en un fichero** distinto [Molloy, 95].

F: número de ficheros (en C++ se debe considerar el fichero CPP junto con su cabecera como uno solo)
C: número de clases

k: se recomienda un valor bajo (por ejemplo, 2) para comenzar a penalizar rápidamente la violación de la norma

$$CS_{26} = \begin{cases} \dfrac{1}{\log_k(|F-C|)} \\ 1 \qquad \forall |F-C| < k \end{cases}$$ {Sistema} {Implementación}

Favorece a: expansibilidad, modularidad y simplicidad.

CS_{27}: Las **clases sólo deben permitir el acceso a sus atributos a través de sus métodos**. Así, debe evitarse el uso de la amistad de C++ puesto que no es consistente con la idea de la ocultación de la información de la orientación a objetos.

N: número de funciones o clases amigas

$$CS_{27} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: datos estándar, estabilidad, estructuración, modularidad y simplicidad.

$CS_{28} = SD_1$

El código debe disponer de **comentarios y líneas en blanco de separación** puesto que esto facilita su consistencia.

$CS_{29} = CN_{10}$

La **reutilización de las clases, modificando su comportamiento, por medio de la herencia** es el mecanismo que ofrece la orientación a objetos que evita tener que definir completamente nuevas clases semejantes para realizar tareas similares.

$CS_{30} = CN_9$

La **reutilización de las clases sin modificación por medio del uso o la herencia** evita tener que definir nuevas clases iguales para realizar las mismas tareas, lo cual resulta consistente con la idea de la orientación a objetos del aprovechamiento de las clases existentes.

CS_{31}: Las **clases se deben reutilizar**, bien por medio de la herencia, bien por una relación de uso.

N: número de clases que se reutilizan (por uso o herencia)
T: número de clases

$$CS_{31} = \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: expansibilidad y generalidad.
Perjudica a: estabilidad.

CS_{32}: Para evitar confusiones o ambigüedades es conveniente **no repetir los nombres** en identificadores diferentes dentro de un mismo ámbito.

N: número de nombres repetidos
T: total de identificadores distintos declarados

$$CS_{32} = \begin{cases} \log_T(T-N) \\ 0 \end{cases} \quad \forall \; N > T-1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estructuración, expansibilidad y simplicidad.

CS_{33}: Si se está realizando un desarrollo orientado a objetos, para mantener la consistencia entre el diseño y la implementación, deberá **utilizarse un lenguaje orientado a objetos**. Los lenguajes basados en objetos poseen muchas de las características de los orientados a objetos, excepto la herencia [Alonso, 95], por lo que, si es posible, también habrá que evitarlos.

1: se utiliza un lenguaje orientado a objetos
0.5: se utiliza un lenguaje basado en objetos
0: se utiliza otro tipo de lenguaje

{Sistema} {Implementación}

Favorece a: simplicidad.

CS_{34}: En un desarrollo orientado a objetos, para mantener la consistencia con el diseño, deberá utilizarse un **lenguaje** que, además de ser orientado a objetos, **soporte la herencia múltiple** (por ejemplo, C++, CLOS o Eiffel). Si el lenguaje elegido no lo soporta (como ocurre con las implementaciones estándar de Smalltalk o Java), habrá que realizar algún tipo de traducción del diseño para su implementación. Evidentemente, si el diseño no ha previsto la utilización de herencia múltiple, esta medida podría no tenerse en cuenta.

1: se utiliza un lenguaje que soporta herencia múltiple
0: se utiliza un lenguaje que no soporta herencia múltiple

{Sistema} {Implementación}

CS_{35}: Es conveniente que los **destructores de las clases raíz de una jerarquía** (con clases derivadas) **sean virtuales** (en C++) [Ellis, 91].

N: número de destructores virtuales en clases raíz
T: total de destructores en clases raíz

$$CS_{35} = \begin{cases} 1 - \log_T(T-N) \\ 1 \end{cases} \quad \forall \; N > T-1$$

{Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad y expansibilidad.

4.3.8. CONTROL DE ACCESO (AC)

Hay cuatro medidas correspondientes al criterio control de acceso (Figura 4.23), que está relacionado con un factor. Este criterio (y sus medidas) no resulta necesario para todos los sistemas desarrollados.

Figura 4.23: Medidas del criterio control de acceso

AC_1: Debe realizarse un **control de acceso a los usuarios para evitar accesos no autorizados al sistema**.

1: control estricto de acceso al sistema por usuarios
0: sin control
(0, 1): existen algunos controles de acceso al sistema

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

AC_2: Debe realizarse un **control de acceso a los usuarios para evitar accesos no autorizados a las bases de datos**.

1: control estricto de acceso a los datos por usuarios
0: sin control
(0, 1): existen algunos controles de acceso a los datos

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

AC_3: **Ante accesos no autorizados** (o intentos reiterados) **el sistema debe responder con acciones** como alertas visuales o sonoras, bloqueos del terminal, avisos a los administradores...

1: se inician acciones de alerta ante violaciones del sistema o intentos
0: no se toman acciones de alerta
(0, 1): otro caso

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

AC_4: **Dependiendo de los privilegios de acceso del usuario el sistema limita su actividad**, restringiendo su acceso sólo a los lugares o a las acciones autorizadas.

1: se limita el uso del *software* dependiendo de los privilegios del usuario
0: no se limita el uso del *software*
(0, 1): otro caso

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

4.3.9. DATOS ESTÁNDAR (DC)

Hay dieciséis medidas correspondientes al criterio datos estándar (Figura 4.24), que está relacionado con tres de los factores. Entre las medidas, hay ocho referencias directas a medidas correspondientes a criterios anteriores.

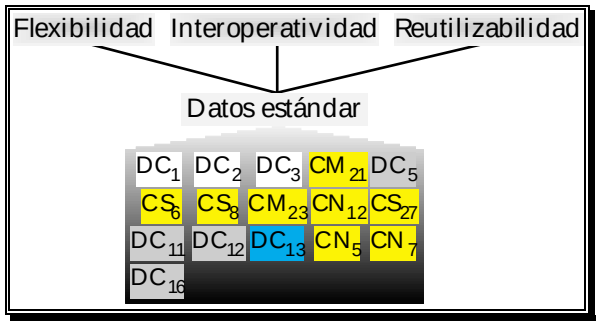


Figura 4.24: Medidas del criterio datos estándar

DC_1: Debe definirse una **representación de datos estándar para la comunicación con otros sistemas**.

1: se define la representación de datos estándar
0: no se define
(0, 1): otro caso

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

DC_2: Deben establecerse **estándares de traducción entre distintas representaciones de información**.

1: establecidos estándares de traducción entre representaciones
0: no se establecen
(0, 1): otro caso

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

DC_3: Las traducciones o **transformaciones entre tipos abstractos de datos** deben realizarse **dentro de cada tipo abstracto de datos**.

N: número de transformaciones realizadas fuera del tipo abstracto de datos
T: total de transformaciones entre tipos abstractos de datos

$$DC_3 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Sistema} {Diseño, Implementación}

$DC_4 = CM_{21}$

Los **tipos abstractos de datos** creados deberán estar **dotados de un completo conjunto de operadores** redefiniendo el comportamiento habitual de dichos operadores, lo que constituye la forma estándar de implementar este modo de funcionamiento en los lenguajes que permiten sobrecarga de operadores.

DC_5: **Las clases deben permitir el acceso a sus atributos únicamente a través de sus métodos, ocultando sus atributos** (Smalltalk lo hace así siempre). La presencia de atributos públicos y, en menor medida, de atributos protegidos va en contra de la idea estándar de las clases como estructuras de datos dotadas de servicios, ya que se podría acceder a su información sin utilizar la interfaz prevista. Hay que evitar principalmente el uso de atributos públicos para evitar accesos peligrosos. Así mismo, hay que reducir al máximo los atributos protegidos, ya que puede llegarse a permitir el acceso a su información tras heredar de su clase. La constante "2" en la fórmula se justifica por la doble vía de acceso a los atributos públicos: mediante la herencia y mediante su uso.

N: número de atributos públicos
P: número de atributos protegidos
T: total de atributos de la clase (definidos o heredados)

$$DC_5 = \begin{cases} 1 - \log_{2\cdot T}(2\cdot N + P) \\ 1 \qquad\qquad \forall\ (2\cdot N + P) < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad y modularidad.

$DC_6 = CS_6$

Los **tipos abstractos de datos** deben **seguir una representación estándar**.

$DC_7 = CS_8$

Debe seguirse un **estándar en la forma y en el orden de definir los miembros de las clases**.

$DC_8 = CM_{23}$

Las **clases** diseñadas de forma estándar **deben contener atributos definidos**.

$DC_9 = CN_{12}$

La **utilización de librerías genéricas estándar de clases**, proporcionadas por muchos de los lenguajes orientados a objetos, favorece la utilización de representaciones de datos estándar.

$DC_{10} = CS_{27}$

Las clases sólo deben de permitir el acceso a sus atributos a través de sus métodos, **evitando el uso de la amistad**.

DC_{11}: **Cuando todos los objetos de una misma clase necesitan compartir una misma información, se debe utilizar un atributo de clase** (un atributo estático en C++), que es el mecanismo que proporciona la orientación a objetos.

N: número de atributos de clase cuando es necesario que los objetos compartan información

$$DC_{11} = \begin{cases} 1 - \frac{1}{N} & \\ 0 & \forall\ N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: eficiencia de almacenamiento.

DC_{12}: Deben **sobrecargarse los métodos para permitir manejar distintos tipos de datos**.

N: número de métodos sobrecargados
T: total de métodos de la clase (definidos y heredados)

$$DC_{12} = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

DC_{13}: Conviene **sobrecargar los métodos varias veces** para permitir manejar la mayor cantidad de tipos de datos distintos.

N: número de veces que se sobrecarga cada método

$$DC_{13} = \begin{cases} 1 - \frac{1}{N} & \\ 0 & \forall\ N < 1 \end{cases}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.
Perjudica a: estabilidad y simplicidad.

$DC_{14} = CN_5$

Los **métodos genéricos** proporcionan una vía estándar para **definir varios métodos con el mismo comportamiento** para distintos tipos de datos.

$DC_{15} = CN_7$

Las **clases genéricas** son un mecanismo estándar para **definir varias clases con la misma estructura y funcionalidades** para ser utilizadas con distintos tipos de datos.

DC_{16}: Las **clases genéricas** (`template` en C++) permiten trabajar sobre distintos tipos de datos. Sin embargo, **puede ser necesario sobrecargarlas** para manejar ciertos tipos de datos que necesitan un manejo diferente.

N: número de veces que se sobrecarga una clase genérica

$$DC_{16} = \begin{cases} 1 - \frac{1}{N} \\ 0 \qquad \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estructuración, expansibilidad y generalidad.

4.3.10. DOCUMENTACIÓN (DO)

Hay diecinueve medidas correspondientes a la documentación (Figura 4.25), que influye en ocho de los factores de calidad.

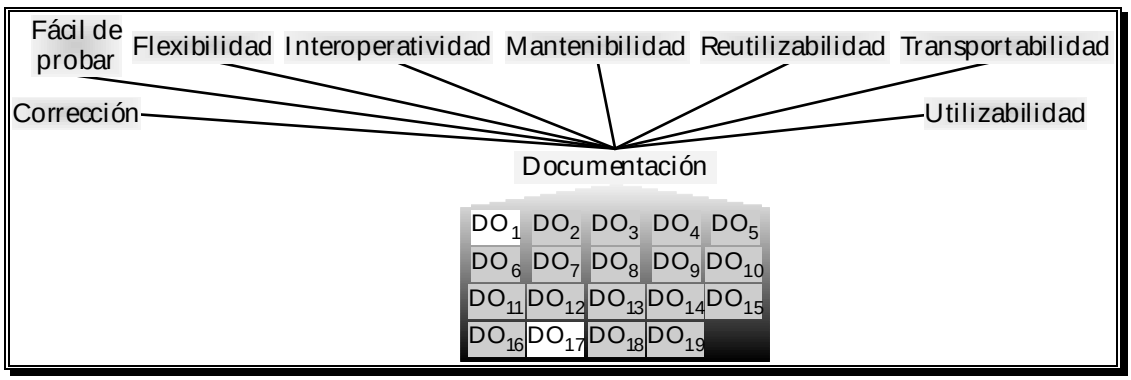


Figura 4.25: Medidas del criterio documentación

DO_1: La cantidad de **documentación técnica** debe ir **en proporción a** la cantidad de **código** generado.

N: número de páginas de documentación
T: tamaño del código (líneas de código ejecutables)
k: relación teórica entre el número de páginas de documentación y el tamaño del código (debería establecerse a priori). La fórmula concreta deberá ser definida por el ingeniero del *software* (aquí se propone que al menos debe haber una página de documentación por cada cien líneas de código ejecutables).

$$k = \frac{T}{N/100} > 1$$

$$DO_1 = \begin{cases} 1 - \frac{1}{\log_k\left(\frac{T}{N/100}\right)} \\ 0 \qquad \forall\, \frac{T}{N/100} < k \end{cases}$$

{Jerarquía, Sistema} {Implementación}

DO_2: **Cada atributo debe ir acompañado de documentación técnica** completa que justifique su uso y su utilidad, explicando bien su empleo.

N: número de atributos con documentación técnica
T: total de atributos

$$DO_2 = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad.

DO_3: **Cada método debe ir acompañado de documentación técnica** completa que justifique su uso y su utilidad, explicando bien su funcionamiento.

N: número de métodos con documentación técnica
T: total de métodos

$$DO_3 = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad.

DO_4: **Cada clase debe ir acompañada de documentación técnica** completa que justifique su estructura, su uso y su utilidad, explicando bien su modo de empleo y funcionamiento.

N: número de clases con documentación técnica
T: total de clases

$$DO_4 = \frac{N}{T}$$ {Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad y modularidad.

DO_5: **Cada jerarquía de clases debe ir acompañada de documentación técnica** completa que justifique su estructura, su uso y su utilidad, explicando bien su funcionamiento.

N: número de jerarquías con documentación técnica
T: total de jerarquías

$$DO_5 = \frac{N}{T}$$ {Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad.

DO_6: **Cada tipo de datos y variable debe ir acompañado de documentación técnica** completa que justifique su uso y su utilidad, explicando bien su funcionamiento.

N: número de tipos y variables con documentación técnica
T: total de tipos y variables

$$DO_6 = \frac{N}{T}$$ {Sistema} {Diseño, Implementación}

Favorece a: generalidad.

DO_7: **Índice de la legibilidad de la documentación** [Kincaid, 81]. Este índice es una fórmula de legibilidad del inglés basado en las sílabas, cuyo objetivo es controlar el nivel de dificultad del texto. Se ha usado como estándar en el Departamento de Defensa de los Estados Unidos para la escritura de manuales de usuario y material de enseñanza.

F: longitud media en palabras de las frases de la documentación
S: número medio de sílabas por palabra
N: índice de legibilidad de la documentación
k: se recomienda un valor de 10, puesto que para valores superiores del índice se dificulta la legibilidad de la documentación

$$N = 0.39 \cdot F + 11.8 \cdot S - 15.59$$

$$DO_7 = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: entrenamiento y operatividad.

DO_8: **Cada atributo no privado debe ir acompañado de documentación** para los usuarios de la clase.

N: número de atributos no privados con documentación de usuario
T: total de atributos no privados

$$DO_8 = \frac{N}{T}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

DO_9: **Cada método no privado debe ir acompañado de documentación** para los usuarios de la clase.

N: número de métodos no privados con documentación de usuario
T: total de métodos no privados

$$DO_9 = \frac{N}{T}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

DO_{10}: **Cada clase debe ir acompañada de documentación** para sus usuarios.

N: número de clases con documentación de usuario
T: total de clases

$$DO_{10} = \frac{N}{T}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad.

DO_{11}: **Cada jerarquía de clases debe ir acompañada de documentación** para sus usuarios.

N: número de jerarquías de clases con documentación de usuario
T: total de jerarquías de clases

$$DO_{11} = \frac{N}{T}$$ {Sistema} {Análisis, Diseño, Implementación}

Favorece a: generalidad.

DO_{12}: **Cada tipo de datos y variable debe ir acompañada de documentación** para sus usuarios.

N: número de tipos de datos y variables con documentación de usuario
T: total de tipos de datos y variables

$$DO_{12} = \frac{N}{T}$$ {Sistema} {Diseño, Implementación}

Favorece a: generalidad.

DO_{13}: En la **documentación de usuario** debe incluirse una **lista de todos los mensajes y errores** con sus causas y sus posibles soluciones debidamente expuestas.

N: número de errores y mensajes explicados en el manual con causas y soluciones
T: total de errores y mensajes que se pueden producir

$$DO_{13} = \frac{N}{T}$$ {Sistema} {Implementación}

Favorece a: entrenamiento.

DO_{14}: Uno de los componentes de la documentación debe ser un **completo manual de usuario**.

1: se incluye un manual de usuario completo
0: no hay manual de usuario
(0, 1): se incluye un manual de usuario

{Sistema} {Implementación}

Favorece a: entrenamiento y operatividad.

DO_{15}: El *software* debe ir acompañado de un **sistema de ayuda en línea interactivo**.

1: se incluye ayuda interactiva completa
0: no hay ayuda
(0, 1): se incluye ayuda interactiva

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: entrenamiento y operatividad.

DO_{16}: Es conveniente que el *software* vaya acompañado de un **tutorial** de manejo.

1: se incluye un tutorial completo
0: no hay tutorial
(0, 1): se incluye un tutorial

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: operatividad.

DO_{17}: El **código fuente debe estar** claramente **comentado**, de forma uniforme y completa, para facilitar la comprensión y la creación de la documentación.

1: el código está bien comentado
0: el código no está comentado
(0, 1): el código está comentado

{Sistema} {Implementación}

DO_{18}: **Nivel de *niebla* de la documentación** [Gunning, 68]. Este índice es una medida para el nivel de dificultad de lectura de documentos escritos en inglés. Mide la niebla o la oscuridad del texto y, según el autor, es proporcional con el tiempo que una persona tardaría en leer y comprender un pasaje.

F: longitud media en palabras de las frases de la documentación
S: porcentaje de palabras con tres ó más sílabas
N: nivel de niebla de la documentación
k: se recomienda un valor de 10, puesto que para valores superiores del índice se dificulta la lectura de la documentación

$$N = 0.4 \cdot (F + S)$$

$$DO_{18} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: operatividad.

DO_{19}: El *software* debe ir acompañado de un sistema de **ayudas sensibles al contexto**.

1: se incluye ayuda sensible al contexto completa
0: no hay ayudas
(0, 1): se incluye ayuda sensible al contexto

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: entrenamiento y operatividad.

4.3.11. EFICIENCIA DE ALMACENAMIENTO (SE)

Hay dieciocho medidas correspondientes al criterio eficiencia de almacenamiento (Figura 4.1), que está relacionado con un factor. Entre las medidas, hay dos referencias a medidas anteriores correspondientes a otros criterios.

Figura 4.1: Medidas del criterio eficiencia de almacenamiento

SE_1: **El análisis y el diseño deben reflejar los requerimientos de almacenamiento**.

1: el análisis y el diseño reflejan los requerimientos de almacenamiento
0: no lo reflejan

{Sistema} {Análisis, Diseño}

SE_2: El uso de **las uniones** de C++ **disminuye las necesidades de almacenamiento** (aunque deben manejarse con cuidado).

N: número de uniones

$$SE_2 = \begin{cases} 1 - \frac{1}{N} \\ 0 \end{cases} \quad \forall\, N < 1$$

{Sistema} {Diseño, Implementación}

SE_3: Debe intentarse **segmentar el código** para que utilice la menor cantidad de memoria posible, mediante mecanismos como *overlays* o librerías de enlace dinámico.

N: estimación del tamaño máximo de código simultáneo en memoria
T: tamaño total del código

$$SE_3 = 1 - \frac{N}{T}$$

{Sistema} {Implementación}

SE_4: Todos **los datos que se almacenen deben ser utilizados**.

N: tamaño de los datos no usados
T: tamaño de todos los datos

$$SE_4 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall\ N < 1$$

{Sistema} {Implementación}

SE_5: **Debe utilizarse memoria dinámica** en lugar de la memoria estática, puesto que se minimiza la cantidad de memoria requerida durante toda la ejecución.

1: se usa memoria dinámica
0: no se utiliza memoria dinámica

(0, 1): se usa memoria dinámica, aunque no siempre que sería recomendable
{Jerarquía, Sistema} {Diseño, Implementación}

SE_6: El sistema debe estar **libre de código inútil**, que consume recursos (espacio de memoria) sin mejorar el proceso.

N: número de sentencias de código inútil
T: total de sentencias

$$SE_6 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Implementación}

Favorece a: eficiencia de ejecución y simplicidad.

SE_7: No debe existir **código duplicado**, puesto que ocupa espacio sin aportar nada al funcionamiento del sistema.

N: número de sentencias duplicadas
T: total de sentencias

$$SE_7 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Implementación}

Favorece a: eficiencia de ejecución.

SE_8: Deben **evitarse los datos u objetos globales**, puesto que consumen memoria durante toda la ejecución del programa, sin que pueda ser utilizada para almacenar otra información.

N: número de variables globales

$$SE_8 = \begin{cases} \dfrac{1}{N} \\ 1 \end{cases} \quad \forall \ N < 1$$

{Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

SE_9: No debe existir **información redundante**.

N: número de datos redundantes
T: total de datos

$$SE_9 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Sistema} {Diseño, Implementación}

SE_{10}: Para disminuir el espacio de almacenamiento necesario deben emplearse, siempre que sea posible, **estructuras de bits**.

N: número de datos almacenados en campos de bits.
T: total de datos

$$SE_{10} = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall \ N < 1$$ {Sistema} {Implementación}

SE_{11}: La utilización de **operaciones con bits** reduce el espacio de almacenamiento necesario para albergar los operandos y el resultado.

N: número de operaciones con bits

$$SE_{11} = \begin{cases} 1 - \frac{1}{N} \\ 0 \end{cases} \quad \forall N < 1$$ {Sistema} {Implementación}

Perjudica a: simplicidad.

SE_{12}: Las **funciones que se expanden en la línea** de la llamada hacen que el tamaño final del código objeto resultante pueda crecer desmesuradamente, por lo que no debe abusarse de esta posibilidad.

N: número de métodos `inline`
T: total de métodos candidatos (véase [Fuertes, 93]) a `inline`

$$SE_{12} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$ {Clase, Jerarquía, Sistema} {Implementación}

Perjudica a: eficiencia de ejecución.

SE_{13}: Deben utilizarse las **opciones de compilación adecuadas** para optimizar el tamaño de almacenamiento.

1: se usan las opciones de compilación para optimizar el almacenamiento
0: no se utilizan
(0, 1): se usan algunas opciones de compilación para optimizar el almacenamiento
{Sistema} {Implementación}

SE_{14}: La existencia de **métodos virtuales** (`virtual` en C++, `deferred` en Eiffel o métodos genéricos en CLOS) necesita internamente de una estructura de datos para almacenar diferente información durante la ejecución [Rumbaugh, 91], con lo que se **utiliza un espacio de memoria extra** para cada método, que no sería necesario si no fueran virtuales.

N: número de métodos virtuales
T: total de métodos

$$SE_{14} = 1 - \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SE_{15} = CM_6$

Si se define una serie de atributos en una clase es para utilizarlos. Si no se van a usar, están ocupando un espacio en memoria que se desperdicia.

$SE_{16} = DC_{11}$

Los atributos de clase se utilizan cuando los objetos de una misma clase necesitan compartir una información común, lo cual posibilita economizar espacio de almacenamiento.

SE_{17}: Cuanto mayor sea el **tamaño de las variables locales**, mayor espacio de memoria (pila) será necesario para almacenarlas.

N: tamaño (en palabras) de las variables de cada método
k: se sugiere un valor de 4, para que la medida empiece a penalizar a partir de 5 palabras (si se tiene en cuenta que lo métodos deben mantenerse pequeños, este valor resulta suficiente para la mayoría de los métodos)

$$SE_{17} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: simplicidad.

SE_{18}: El paso de argumentos a los métodos puede consumir una cantidad excesiva de espacio de memoria en la pila. Por ello, debe intentarse **reducir el tamaño de los parámetros formales**.

N: tamaño (en palabras) de los parámetros formales de un método
k: se sugiere un valor de 4, para que la medida empiece a penalizar a partir de 5 palabras (lo cual es bastante para la mayoría de los métodos)

$$SE_{18} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: eficiencia de ejecución y simplicidad.

4.3.12. EFICIENCIA DE EJECUCIÓN (EE)

Hay veintitrés medidas correspondientes al criterio eficiencia de ejecución (Figura 4.26), que está relacionado con el factor eficiencia. Entre las medidas, hay seis referencias a medidas anteriores correspondientes a otros criterios, de las cuales, tres son referencias inversas.

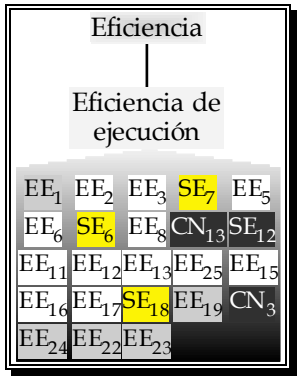


Figura 4.26: Medidas del criterio eficiencia de ejecución

EE_1: **Las operaciones no deben producir una excesiva cantidad de procesamiento**. Una gran parte del tiempo de procesamiento necesario para ejecutar una función viene dado por la cantidad de mensajes que envía.

N: número de mensajes distintos enviados desde cada método
k: el valor de esta constante depende mucho del volumen del sistema, pero para mantener una buena eficiencia no se recomienda que supere el valor de 10

$$EE_1 = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad y simplicidad.

EE_2: Los **requisitos de ejecución deben reflejarse en las siguientes fases** del ciclo de vida del desarrollo del *software*.

1: se han contemplado los requisitos de ejecución
0: no se han tenido en cuenta
(0, 1): se han contemplado parcialmente los requisitos de ejecución

{Sistema} {Análisis, Diseño, Implementación}

EE_3: Se deben **excluir de los bucles las operaciones independientes al bucle** (como por ejemplo, la evaluación de expresiones constantes).

N: número de sentencias no dependientes del bucle que se encuentran en su interior
T: total de sentencias del bucle

$$EE_3 = 1 - \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$EE_4 = SE_7$

La existencia de **código duplicado inútilmente**, **disminuye la velocidad** de ejecución del sistema.

EE_5: Desde el punto de vista de la eficiencia, **no debe repetirse innecesariamente la evaluación de las expresiones compuestas**.

N: número de expresiones compuestas calculadas en más de un lugar
T: total de expresiones compuestas

$$EE_5 = 1 - \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

EE_6: El **uso de *overlays* o de librerías dinámicas causa una disminución de la eficiencia** al tener que cargarlas y descargarlas de memoria durante la ejecución.

N: número de *overlays* o librerías dinámicas

$$EE_6 = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$ {Sistema} {Implementación}

$EE_7 = SE_6$

El sistema debe estar **libre de código inútil** (segmentos de código que no efectúan una función relevante), puesto que consume recursos (tiempo de proceso) sin suponer una mejora en el funcionamiento.

EE_8: Un aspecto de los lenguajes orientados a objetos que puede afectar a la eficiencia de ejecución es la **resolución de métodos en tiempo de ejecución** (enlace dinámico) para implementar las operaciones polimórficas, que consiste en enlazar una operación sobre un objeto con un método específico. Esto podría requerir una búsqueda, en tiempo de ejecución, por el árbol de herencia para encontrar la clase que implementa la operación para dicho objeto. Sin embargo, la mayoría de los lenguajes (como ocurre en C++ [Ellis, 91]) optimizan este mecanismo de búsqueda para mejorar su eficiencia. Como la estructura de las clases permanece invariable durante la ejecución, el método correcto para cada operación puede almacenarse localmente en una subclase. Con esta técnica, conocida como *method caching*, el enlace dinámico se reduce a una búsqueda en una tabla *hash*, que se ejecuta en un tiempo siempre fijo, independientemente de la profundidad del árbol de herencia o del número de métodos de la clase.

1: el compilador implementa *method caching*
0: el compilador no implementa *method caching*

{Sistema} {Implementación}

$EE_9 = 1 - CN_{13}$

Las llamadas a métodos que se resuelven en tiempo de ejecución (enlace dinámico) consumen una mayor cantidad de proceso en su invocación. La **llamada a un método virtual emplea más tiempo que la llamada a un método no virtual** porque en tiempo de ejecución debe determinarse (consultando una tabla) a qué función hay que invocar dependiendo del tipo del objeto desde el que se realiza la llamada.

$EE_{10} = 1 - SE_{12}$

Las **funciones que se expanden en la línea de la llamada son considerablemente más eficientes** debido a que no tienen que ejecutar ni el protocolo de llamada ni el protocolo de retorno que tiene una función normal.

EE_{11}: **Las expresiones deben implementarse de forma eficiente**, empleando para ello adecuadamente los operadores que ofrezca cada lenguaje. Así, por ejemplo, en C++ resulta más óptimo usar `a++` que `a = a + 1`; `a += b` que `a = a + b` o `a = !b` que `a = b == 0`.

N: número de expresiones con uso ineficiente de operadores
T: total de expresiones

$$EE_{11} = 1 - \frac{N}{T}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

EE_{12}: Deben **utilizarse las opciones de compilación** adecuadas para optimizar la eficiencia de ejecución del programa.

1: se usan las opciones de compilación para optimizar la eficiencia de ejecución
0: no se usan
(0, 1): se usan algunas opciones de compilación para optimizar la ejecución

{Sistema} {Implementación}

EE_{13}: Deben **agruparse los datos relacionados** para que su procesamiento sea más eficiente, utilizando, por ejemplo, vectores o registros.

1: los datos se agrupan para un procesamiento eficiente
0: no se agrupan
(0, 1): se agrupan algunos datos

{Sistema} {Implementación}

EE_{14}: El sistema deberá ser desarrollado sobre un **lenguaje para el que exista compilador**. La utilización de un intérprete afectará a su eficiencia. Los lenguajes orientados a objetos modernos disponen de compilador, aunque originalmente muchos de los lenguajes empleaban un intérprete (Smalltalk, CLOS, Java...).

1: se dispone de un compilador para el lenguaje usado
0: no se dispone de compilador

{Sistema} {Implementación}

EE_{15}: Cuando hay que **enviar como argumentos una gran cantidad de datos, resulta más eficiente pasarlos por referencia** que por valor, ya que de esta forma se evita tener que copiar toda la información.

N: número de veces que se pasan parámetros de gran tamaño por referencia
T: total de veces que se pasan parámetros de gran tamaño

$$EE_{15} = \begin{cases} 1 - \log_T (T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$ {Método, Clase, Jerarquía, Sistema} {Implementación}

EE_{16}: Debe **evitarse el uso abusivo de objetos temporales o copias intermedias de objetos**.

N: número de variables temporales, o copias intermedias, de objetos
T: total de objetos

$$EE_{16} = \begin{cases} 1 - \dfrac{1}{\log_T(N)} \\ 0 \end{cases} \quad \forall\, N < T$$

{Clase, Jerarquía, Sistema} {Implementación}

EE_{17}: Debe **evitarse el uso de operaciones innecesarias con números reales**. Por ejemplo, `a = b + 2.0`, siendo las variables `a` y `b` enteras.

N: número de operaciones innecesarias con reales
T: total de operaciones con reales

$$EE_{17} = 1 - \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$EE_{18} = SE_{18}$

El **tamaño de los parámetros formales de los métodos no debe ser demasiado elevado**, para no consumir un tiempo considerable debido a que es necesario copiar la información desde los parámetros actuales a los parámetros formales.

EE_{19}: El paso de argumentos a los métodos consume un tiempo considerable debido a que es necesario copiar toda la información de los parámetros actuales a los parámetros formales. Debe intentarse que **el tamaño de los parámetros actuales sea lo menor posible**.

N: tamaño (en palabras) de los parámetros actuales en la llamada a un método
k: se sugiere un valor de 4, para que la medida empiece a penalizar a partir de 5 palabras (lo cual es bastante para la mayoría de los métodos)

$$EE_{19} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \end{cases} \quad \forall\, N < k$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: simplicidad.

$EE_{20} = 1 - CN_3$

Los **métodos virtuales penalizan el tiempo de proceso**, puesto que es necesario buscar la función apropiada durante la ejecución.

EE_{21}: Los **métodos de un tamaño excesivamente pequeño penalizan la eficiencia** de la ejecución debido a que una parte importante del tiempo de proceso se gasta durante la secuencia de llamada y la secuencia de retorno. Se entiende por tamaño excesivamente pequeño aquél que es inferior a la quinta parte de los tamaños reducidos estimados por [Lorenz, 93], es decir, cinco líneas de código C++ (a menos que sea `inline`) o dos de Smalltalk.

N: número de métodos excesivamente pequeños
T: total de métodos

$$EE_{21} = \begin{cases} \log_T(T - N) \\ 0 \qquad \forall\, N > T - 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Implementación}

Favorece a: modularidad.

EE_{22}: Si los **métodos devuelven un objeto** de una clase como valor de retorno, habrá que copiar toda la información de dicho objeto, lo que **puede consumir bastante tiempo**.

N: número de objetos devueltos por los métodos de cada clase

$$EE_{22} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad y simplicidad.

EE_{23}: La **cantidad de métodos invocados** al activarse cada método **debe mantenerse baja** para conservar la eficiencia del procesamiento de los mensajes.

N: número de métodos que se llaman (directa o indirectamente) para procesar cada mensaje
k: aunque el valor de esta constante depende de la dificultad del problema, se recomienda un valor inferior a 5, para mantener la eficiencia de los distintos métodos

$$EE_{23} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estabilidad y simplicidad.

4.3.13. ENTRENAMIENTO (TA)

Hay diez medidas correspondientes al criterio entrenamiento (Figura 4.27), que está relacionado con un factor. Entre las medidas, hay siete referencias a medidas anteriores correspondientes a otros dos criterios.

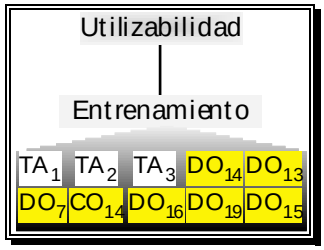


Figura 4.27: Medidas del criterio entrenamiento

TA_1: Debe incorporarse una serie de **lecciones y material de enseñanza** para los usuarios.

1: hay lecciones y material de enseñanza
0: no hay material didáctico
(0, 1): hay algún material didáctico, pero insuficiente

{Sistema} {Requisitos, Implementación}

TA_2: Deben proporcionarse **ejercicios y ejemplos reales** de funcionamiento.

1: hay ejercicios y ejemplos reales
0: los ejercicios y ejemplos no son realistas o no hay
(0, 1): hay algún ejercicio o ejemplo, pero no suficientes

{Sistema} {Implementación}

TA_3: Deben existir **manuales en soporte magnético**.

1: se incluyen los manuales completos en soporte magnético
0: no hay manuales en línea
(0, 1): se incluyen los manuales incompletos

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

$TA_4 = DO_{14}$

Si uno de los componentes de la documentación es un **completo manual de usuario**, el aprendizaje de los usuarios será más sencillo.

$TA_5 = DO_{13}$

La documentación debe incluir una **lista de todos los mensajes y errores** con sus causas y sus soluciones debidamente expuestas para facilitar el aprendizaje del sistema.

$TA_6 = DO_7$

La **documentación debe tener una legibilidad aceptable**, con el fin de facilitar su lectura por los usuarios.

$TA_7 = CO_{14}$

La **interfaz de usuario** debe permitirle operar con el sistema de manera ergonómica y confortable. Debe ser sencilla, atractiva y fácil de utilizar. Éste es un factor decisivo a la hora de determinar la aceptación del sistema para su uso.

$TA_8 = DO_{16}$

La presencia de un **tutorial** de manejo del *software* facilita el aprendizaje.

$TA_9 = DO_{19}$

Debe ofrecerse un completo sistema de **ayuda sensible al contexto**.

$TA_{10} = DO_{15}$

Debe ofrecerse un completo sistema de **ayuda en línea interactivo**.

4.3.14. ESTABILIDAD (ST)

Hay un total de treinta y siete medidas correspondientes a la estabilidad (Figura 4.1), que influye en tres factores. Entre las medidas, hay dieciséis referencias a medidas anteriores correspondientes a otros criterios, ocho de las cuales son relaciones inversas.

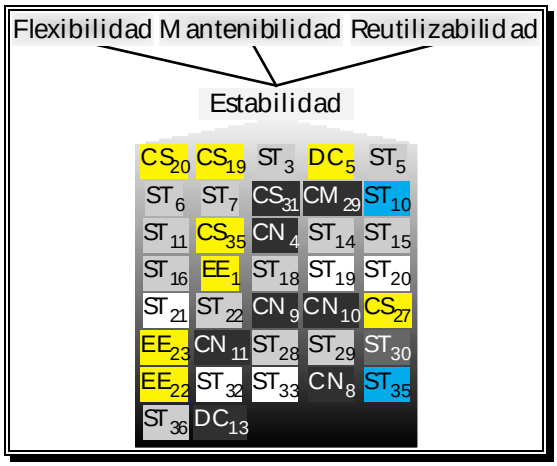


Figura 4.1: Medidas del criterio estabilidad

$ST_1 = CS_{20}$

Si un método utiliza atributos externos de otras clases, **un cambio** en cualquiera de estas clases **afectará** a su funcionamiento.

$ST_2 = CS_{19}$

Si otras clases utilizan atributos de una clase, los cambios en esta clase afectará a las otras.

ST_3: La **utilización de un atributo desde fuera de la propia clase** puede causar que un cambio en la clase provoque que sus clientes dejen de funcionar.

N: número de veces que se usa el atributo desde otra clase

$$ST_3 = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Atributo, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: estructuración, expansibilidad, modularidad y simplicidad.

$ST_4 = DC_5$

Las **clases deben evitar el acceso a sus atributos desde el exterior** puesto que modificaciones internas afectarán a los clientes. Las clases deben ocultar todos sus atributos, permitiendo el acceso a su información a través de la interfaz prevista.

ST_5: Los **métodos deben ser independientes** del sistema desarrollado, puesto que un cambio en éste podría afectar al funcionamiento del método.

N: número de métodos independientes del sistema
T: total de métodos

$$ST_5 = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Implementación}

Favorece a: generalidad.

ST_6: **Cuantas más clases utilicen un atributo, mayor será la influencia de los cambios** en el sistema.

N: número de clases que usan cada atributo
T: total de clases

$$ST_6 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Atributo, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

ST_7: **Cuantos más métodos utilicen un atributo, mayor** será la **posibilidad** de un **funcionamiento incorrecto debido a una modificación** en la clase donde está definido el atributo.

N: número de métodos que usan un mismo atributo
T: total de métodos

$$ST_7 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Atributo, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

$ST_8 = 1 - CS_{31}$

La **reutilización de las clases**, mediante **herencia** o mediante una **relación de uso** puede influir en el funcionamiento del sistema si éstas sufren alguna alteración.

$ST_9 = 1 - CM_{29}$

Si **los métodos se utilizan varias veces**, las **modificaciones** que puedan sufrir **afectarán** en mayor medida al funcionamiento del sistema.

ST_{10}: Si se usan **métodos sobrecargados, una modificación** en el modo de funcionamiento del servicio implantado en estos métodos **podrá afectar a sus usuarios**. Si algún método sobrecargado sufre alguna modificación, habrá que tenerlo en cuenta cada vez que sea llamado.

N: número de veces que se llama a un método sobrecargado

$$ST_{10} = \frac{1}{N}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.
Perjudica a: generalidad.

ST_{11}: Si algún **método sobre-escrito sufre alguna modificación**, habrá que tenerlo en cuenta cada vez que sea llamado, para verificar que la modificación no ha alterado su funcionamiento.

N: número de veces que se llama a un método sobre-escrito

$$ST_{11} = \frac{1}{N}$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

$ST_{12} = CS_{35}$

Si por una modificación, un puntero a un objeto necesita utilizar el mecanismo de enlace dinámico, **puede causar problemas si los destructores de las clases raíz de una jerarquía no han sido declarados virtuales**.

$ST_{13} = 1 - CN_4$

Cuantas **más clases hereden un mismo método**, **mayor** será el **número** de **clases que habrá que modificar si se realiza algún cambio** significativo sobre el comportamiento del método.

ST_{14}: Cuantas **más clases utilicen un método**, crecerá la influencia de los cambios sobre el sistema.

N: número de clases desde las que se llama a cada método
T: total de clases

$$ST_{14} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

ST_{15}: Cuanto mayor sea el **número de métodos que envían un mismo mensaje**, más resistencia a las modificaciones tendrá el programa.

N: número de métodos que llaman a cada método
T: total de métodos

$$ST_{15} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

ST_{16}: Una **clase englobada en una jerarquía** de herencia tendrá menos posibilidades de verse afectada por una modificación cuanto más cercana se encuentre de la raíz de dicha jerarquía.

N: número máximo de arcos de herencia desde la clase hasta una raíz en el grafo de herencia

$$ST_{16} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: simplicidad.

$ST_{17} = EE_1$

Cuanto mayor sea el **número de métodos llamados por cada método**, más posibilidades habrá de que cualquier modificación en estos métodos deje al otro con un funcionamiento incorrecto.

ST_{18}: **Las clases no deben utilizar información de otras clases**, puesto que éstas pueden ser cambiadas en cualquier momento. Por ello, será más sensible a los cambios sufridos en otras partes del programa, además de dificultar el mantenimiento.

N: número de clases utilizadas por un método
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$ST_{18} = \begin{cases} \frac{1}{\log_k(N)} \\ 1 \end{cases} \quad \forall N < k$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: modularidad y simplicidad.

ST_{19}: El **uso de las clases** hace que crezca la posibilidad de un impacto en su modo de funcionamiento por una modificación.

N: número de veces que se usa una clase
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$ST_{19} = \begin{cases} \frac{1}{\log_k(N)} \\ 1 \end{cases} \quad \forall N < k$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

ST_{20}: El **uso de muchos objetos de una clase** hace que una modificación pueda afectar a su modo de funcionamiento con mayor facilidad.

N: número de veces que se usa un objeto de una clase
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$ST_{20} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

ST_{21}: El **uso de las clases genéricas** (`template` en C++ o `generic` en Eiffel) puede causar que una modificación afecte al modo de funcionamiento de una buena parte del programa.

N: número de veces que se usa una clase genérica
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$ST_{21} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

ST_{22}: Cuantos **más objetos se definan en el seno de cada clase**, mayor será la influencia de un posible cambio en la definición de dichos objetos.

N: número de objetos definidos en cada clase
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$ST_{22} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

$ST_{23} = 1 - CN_9$

La **reutilización de las clases sin modificación por medio del uso o la herencia** puede no resultar adecuada si se cambia la definición de alguna de estas clases.

$ST_{24} = 1 - CN_{10}$

La **reutilización de las clases, con modificación por medio de la herencia**, puede no resultar adecuada si se cambia la definición de alguna de estas clases.

$ST_{25} = CS_{27}$

Los **amigos de una clase** puede que no se "enteren" de que dicha clase ha sufrido algún tipo de modificación.

$ST_{26} = EE_{23}$

Si el **número de los métodos invocados al activarse cada método es elevado**, suben las posibilidades que cualquier cambio afecte a su funcionamiento.

$ST_{27} = 1 - CN_{11}$

Cuanto mayor sea el **número de herencias en una jerarquía**, mayor será la posibilidad de que un cambio en una de las clases afecte al funcionamiento de la jerarquía [Harrison, 00].

ST_{28}: **No debe abusarse de la posibilidad de herencia múltiple**, puesto que un cambio en cualquier padre puede afectar al futuro funcionamiento de sus hijos.

N: número de relaciones de herencia múltiple
T: total de relaciones de herencia

$$ST_{28} = \begin{cases} \log_T(T-N) \\ 0 \end{cases} \quad \forall \; N > T-1$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: simplicidad.

ST_{29}: Conforme se tengan más **clases raíz en la jerarquía**, más fácil será que un cambio en una de las clases afecte a su funcionamiento.

N: número de clases raíz en la jerarquía

$$ST_{29} = \frac{1}{N}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: simplicidad.

ST_{30}: Cuantos más **mensajes se reciban del exterior a la clase**, indicará que dicha clase está siendo muy usada, teniendo por ello una mayor posibilidad de que una modificación impacte en su modo de funcionamiento y afecte a sus usuarios.

N: número de mensajes recibidos del exterior

$$ST_{30} = \frac{1}{N}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: generalidad.

$ST_{31} = EE_{22}$

Si los **métodos devuelven un objeto** de una clase como valor de retorno, una modificación de la clase o del método puede afectar a su funcionamiento.

ST_{32}: Los **lenguajes orientados a objetos fuertemente tipados** ayudan al programador a detectar posibles inconsistencias de tipos en las expresiones y sentencias y, por tanto, a evitar posibles errores.

1: el lenguaje usado es fuertemente tipado (como Eiffel o Java)
0.5: el lenguaje usado tiene un tipado medio (como C++) o bien es fuertemente tipado, aunque no sea obligatoria su utilización (como CLOS u Objective-C)
0: el lenguaje usado es débilmente tipado (como Smalltalk)

{Sistema} {Implementación}

ST_{33}: Las modificaciones podrán afectar más a las clases cuantas **más veces se comuniquen con otra clase**.

N: número de clases que envían mensajes a otra clase o se utilizan los atributos de esa otra clase
T: total de clases

$$ST_{33} = 1 - \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

$ST_{34} = 1 - CN_8$

Una **clase que pueda heredar de muchas otras clases** puede verse afectada por un cambio en cualquiera de estas clases.

ST_{35}: Cuanto mayor sea el **número de descendientes de una clase**, mayor será su impacto en ellos si dicha clase sufre una alteración.

N: número de descendientes (hijos, nietos...) de cada clase

$$ST_{35} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: simplicidad.
Perjudica a: generalidad.

ST_{36}: Cuanto mayor sea el **número de hijos directos** de una clase, a más clases afectará un posible cambio en su padre.

N: número de hijos directos de una clase

$$ST_{36} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Favorece a: expansibilidad y simplicidad.

$ST_{37} = 1 - DC_{13}$

Los **métodos sobrecargados** hacen que una modificación en el modo de funcionamiento del servicio implantado en estos métodos suponga (generalmente) una modificación en cada uno de los métodos.

4.3.15. ESTRUCTURACIÓN (SR)

Hay trece medidas correspondientes al criterio estructuración (Figura 4.2), que está relacionado con cinco factores. Todas las medidas son referencias directas a medidas anteriores correspondientes a otros criterios.

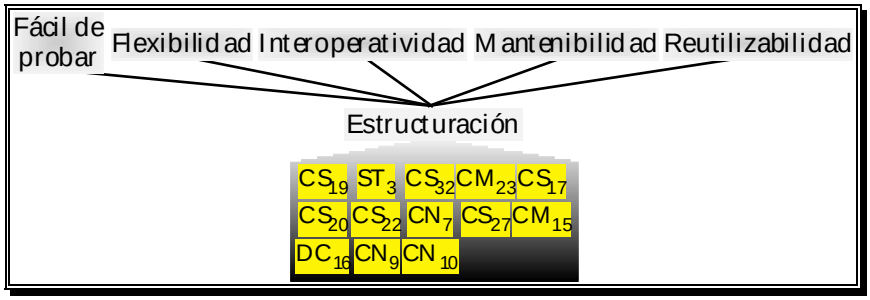


Figura 4.2: Medidas del criterio estructuración

$SR_1 = CS_{19}$

La **utilización de atributos desde fuera** de la clase disminuye la estructuración de la clase.

$SR_2 = ST_3$

Si se **utiliza muchas veces un atributo desde el exterior** a la clase, causa una disminución en la estructuración de la clase.

$SR_3 = CS_{32}$

La utilización del **mismo nombre para un identificador** puede impedir una adecuada estructura del programa.

$SR_4 = CM_{23}$

Para que las clases tengan una estructura uniforme, **deben contener atributos declarados**.

$SR_5 = CS_{17}$

Los **métodos que no modifican el valor de los atributos** del objeto desde el que ha sido llamado **deben ser de tipo constante** para conservar una correcta estructura de los métodos.

$SR_6 = CS_{20}$

Para mantener una adecuada estructura de las clases, sus **métodos no deben utilizar atributos externos** de otras clases.

$SR_7 = CS_{22}$

Las **funciones no miembro** no permiten conservar una adecuada estructura de acuerdo a la ocultación de información propia de la programación orientada a objetos.

$SR_8 = CN_7$

La inclusión de clases genéricas permite conservar la estructura de las clases, de forma que puedan ser utilizadas con distintos tipos de datos sin modificaciones.

$SR_9 = CS_{27}$

Las clases sólo deben de permitir el **acceso a sus atributos a través de los mecanismos proporcionados por la estructura de la clase**. El uso de la amistad permite acceder a las clases saltándose la estructura impuesta por el mecanismo de la ocultación de la información que ofrece la orientación a objetos.

$SR_{10} = CM_{15}$

Las clases deben tener una estructura uniforme, con un **constructor por omisión**, un **constructor de copia**, un **destructor** y el **operador de asignación** definidos explícitamente, cuando las características de las clases lo exigan.

$SR_{11} = DC_{16}$

Las **clases genéricas sobrecargadas** son un mecanismo proporcionado por el lenguaje que permite conservar la estructura de la clase para manejar ciertos tipos de datos particulares.

$SR_{12} = CN_9$

La **reutilización de las clases sin modificación por una relación de uso o de herencia** facilita la construcción de jerarquías con una estructura uniforme.

$SR_{13} = CN_{10}$

La **reutilización de las clases, modificando su comportamiento, por medio de la herencia** facilita la construcción de jerarquías con una estructura uniforme.

4.3.16. EXPANSIBILIDAD (EX)

Hay veinticinco medidas correspondientes a la expansibilidad (Figura 4.3), que está relacionada con un factor. Entre las medidas, hay quince referencias directas a medidas anteriores correspondientes a otros criterios.

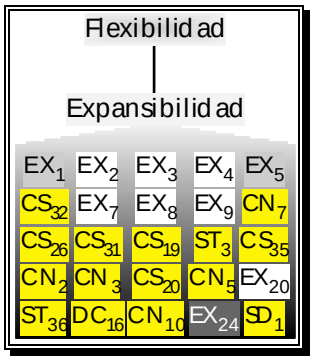


Figura 4.3: Medidas del criterio expansibilidad

EX_1: El **proceso lógico de los métodos de una clase debe ser independiente de constantes** (valores límite, tamaños de vectores o de *buffers*...) o debe estar parametrizado.

N: número de métodos cuyo proceso es independiente de constantes o están parametrizadas
T: total de métodos

$$EX_1 = \begin{cases} 1 - \log_T (T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

EX_2: Las **clases deben facilitar las especificaciones**, permitiendo que se herede de ella todo lo necesario para construir clases más específicas.

1: las clases permiten heredar lo necesario
0: no permiten herencia
(0, 1): algunas clases permiten heredar lo necesario

{Jerarquía, Sistema} {Diseño, Implementación}

EX_3: Todas las **posibles ampliaciones o modificaciones han sido comentadas y documentadas**.

1: se documentan todas las ampliaciones posibles
0: no se documentan
(0, 1): se documentan algunas ampliaciones

{Clase, Jerarquía, Sistema} {Requisitos, Análisis, Diseño, Implementación}

EX_4: **Se deben prever ampliaciones** incluyendo código, que debe estar oculto y bien documentado.

1: hay código oculto para acometer ampliaciones
0: no se ha previsto código para ampliaciones
(0, 1): se ha previsto parcialmente código para ampliaciones

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

EX_5: **No deben existir efectos colaterales** o, caso de existir, deben estar muy bien documentados.

1: no hay efectos laterales
0: hay efectos laterales sin documentar
(0, 1): hay efectos laterales documentados

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: generalidad.

$EX_6 = CS_{32}$

La **repetición de los nombres de los identificadores** puede causar problemas al ampliar las funcionalidades del sistema.

EX_7: Las **interfaces de las clases** deben estar **en concordancia durante todas las fases** del desarrollo.

1: las interfaces de las clases están perfectamente definidas
0: las interfaces de las clases están mal definidas
(0, 1): las interfaces de las clases están definidas irregularmente

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

EX_8: **Deben quedar recursos disponibles para facilitar las ampliaciones**. Por ejemplo, en los programas en DOS, existe la limitación de los segmentos de memoria de 64Kb de tamaño máximo, límite que no debe ser alcanzado.

N: cantidad de recursos disponibles sin usar completamente
T: cantidad total de recursos utilizables

$$EX_8 = \frac{N}{T}$$

{Sistema} {Implementación}

EX_9: **Las clases no deben disponer de métodos en los que el tiempo de ejecución sea crítico**, es decir, las clases que pertenecen a un sistema de tiempo real ven dificultada su ampliación.

N: número de métodos en tiempo real
T: total de métodos

$$EX_9 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$EX_{10} = CN_7$

La **definición de clases genéricas facilita la ampliación** del sistema, puesto que con muchas probabilidades, dichas clases genéricas se podrán seguir utilizando sin modificación.

$EX_{11} = CS_{26}$

La **distribución de clases en ficheros favorece la expansión** del código al poderse localizar fácilmente cada uno de los módulos del sistema.

$EX_{12} = CS_{31}$

Las **reutilizaciones de clases por medio de la herencia o por una relación de uso** facilitan su reutilización posterior debida a una ampliación del sistema.

$EX_{13} = CS_{19}$

Utilizar atributos desde el exterior de la clase dificulta la expansión de la clase.

$EX_{14} = ST_3$

La cantidad de **veces que se utiliza cada atributo desde fuera** de la propia clase penaliza la expansibilidad de la clase.

$EX_{15} = CS_{35}$

La utilización de destructores virtuales en las clases base de una jerarquía facilitará la expansión del código, al no ser necesario modificar dicha clase si se ha de emplear el enlace dinámico al destruir un objeto.

$EX_{16} = CN_2$

Los **métodos operador** permiten una ampliación del código de forma sencilla e intuitiva.

$EX_{17} = CN_3$

Los **métodos virtuales** permiten una fácil ampliación del sistema al poder utilizarlos tal cual o con modificación. Incluso puede ocurrir que no sea necesario modificar las invocaciones a dichos métodos, puesto que el enlace con el método adecuado se realiza en tiempo de ejecución.

$EX_{18} = CS_{20}$

Para facilitar futuras extensiones del sistema, **un método no debe utilizar atributos externos de otras clases**.

$EX_{19} = CN_5$

La **definición de métodos genéricos** facilita la ampliación del sistema, pues, en términos generales, evitarán tener que modificarlos ya que están preparados para manejar distintos tipos de datos.

EX_{20}: **La utilización de herramientas** para la edición o lectura del código fuente, la compilación, la depuración, la integración del sistema y las pruebas **facilitan** la labor de realizar **ampliaciones al sistema**. Estas herramientas son especialmente importantes para los lenguajes orientados a objetos por la dificultad de manejar la herencia y el enlace dinámico en grandes sistemas.

1: se utilizan herramientas
0: no se usan herramientas
(0, 1): se utiliza alguna herramienta

{Sistema} {Implementación}

$EX_{21} = ST_{36}$

Cuanto mayor sea el número de hijos directos de una clase, más complicado será expandir dicha clase debido a que las modificaciones pueden afectar a sus hijos [Harrison, 00].

EX_{22}= DC_{16}

Las **clases genéricas sobrecargadas** (de tipo específico) **facilitan** aún más **la ampliación del sistema** pues permiten trabajar sobre distintos tipos de datos, aunque no sean los habituales.

EX_{23}= CN_{10}

La **reutilización de las clases**, modificando su comportamiento, **por medio de la herencia facilita la realización de futuras ampliaciones**.

EX_{24}: En un sistema, **cuantas más jerarquías de clases independientes existan, más sencilla será la ampliación del sistema**, debido a su menor interrelación.

N: número de jerarquías independientes
k: valor a definir por el ingeniero del *software* dependiendo del tamaño del problema; para un problema pequeño, podría utilizarse un valor de 2 ó 3

$$EX_{24} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \qquad \forall\, N < k$$

{Sistema} {Análisis, Diseño, Implementación}

Perjudica a: simplicidad.

EX_{25}= SD_1

El código debe estar adecuadamente comentado puesto que esto ayuda a su ampliación. Son importantes tanto las líneas con comentarios como las líneas en blanco que actúan como separadores.

4.3.17. GENERALIDAD (GE)

Hay treinta y seis medidas correspondientes a la generalidad (Figura 4.4), que está relacionada con tres de los factores. De las medidas, veinte son referencias a medidas anteriores correspondientes a otros criterios, de las cuales cuatro son inversas. También hay una medida relacionada con diez medidas de la documentación (cuya información no ha sido incorporada en la figura).

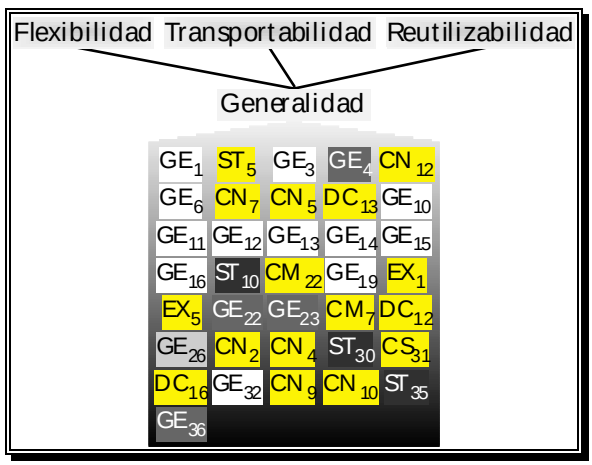


Figura 4.4: Medidas del criterio generalidad

GE_1: Los **atributos** deben ser **independientes de la aplicación**. Es decir, no deben ser resultado de la presencia de otros componentes.

N: número de atributos dependientes de la aplicación
T: total de atributos

$$GE_1 = 1 - \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$GE_2 = ST_5$

Los **métodos** deben ser **independientes de la aplicación**. Es decir, no deben ser resultado de la presencia de otros componentes.

GE_3: Las **clases** deben ser **independientes de la aplicación**. Es decir, no deben ser resultado de la presencia de otros componentes.

N: número de clases dependientes de la aplicación
T: total de clases

$$GE_3 = 1 - \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

GE_4: Una **clase será más general si su interfaz proporciona un amplio número de elementos** accesibles.

N: número de miembros no privados de una clase
k: el umbral dependerá del tamaño medio de las clases, aunque con valores entre 3 y 5 se obtienen unos resultados aceptables, según la recomendación de [Lorenz, 93]

$$GE_{37} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \qquad \forall\, N < k \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: simplicidad.

$GE_5 = CN_{12}$

El uso de **librerías genéricas estándar de clases** beneficia la generalidad del código.

GE_6: **Conviene utilizar métodos sobrecargados**.

N: número de métodos sobrecargados

$$GE_6 = \begin{cases} 1 - \dfrac{1}{N} \\ 0 \qquad \forall\, N < 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$GE_7 = CN_7$

Como su propio nombre indica, la **definición de clases genéricas** proporciona generalidad al sistema, al poder manejar distintos tipos de datos.

$GE_8 = CN_5$

La **definición de métodos genéricos** proporciona, evidentemente, generalidad al sistema, al poder operar sobre distintos tipos de datos.

$GE_9 = DC_{13}$

Conviene **sobrecargar los métodos varias veces** con el fin de dotarles de una mayor generalidad para manejar la mayor cantidad de tipos de datos distintos.

GE_{10}: De existir algún **método no público** (privado o protegido en C++ o no exportado en Eiffel) en una clase, **debe ser utilizado por el mayor número posible de métodos** de dicha clase.

N: número de métodos de la clase que llaman a cada método no público
T: total de métodos de la clase

$$GE_{10} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

GE_{11}: Las **clases deben ser utilizadas por el mayor número posible de clases**.

N: número de clases que usan cada clase
T: total de clases

$$GE_{11} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

GE_{12}: **La entrada, el proceso y la salida no deben mezclarse en un mismo método**. Cada método se debe encargar de una única tarea. Un método que realiza la entrada/salida y el proceso no es tan general como, por ejemplo, un método que solo lleva a cabo el proceso.

N: número de métodos que realizan la entrada, el proceso y la salida
T: total de métodos

$$GE_{12} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

GE_{13}: No deben existir métodos con **referencias a funciones dependientes de la máquina** (tanto *software* como *hardware*), puesto que disminuyen la generalidad.

N: número de métodos con funciones dependientes de la máquina
T: total de métodos

$$GE_{13} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Implementación}

GE_{14}: **El volumen de datos que pueda procesar un método no debe estar limitado**. Un método que ha sido diseñado e implementado para no aceptar más de 100 entradas para su procesamiento es, ciertamente, no tan general como un método que acepta cualquier volumen de datos de entrada.

N: número de métodos que solo pueden procesar un número limitado de datos
T: total de métodos

$$GE_{14} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

GE_{15}: **Los valores de los datos a procesar por los métodos no deben estar limitados**. Cuanto más pequeño sea el subconjunto de todas las posibles entradas válidas, menos general es.

N: número de métodos que procesan datos con valores limitados
T: total de métodos

$$GE_{15} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

GE_{16}: **Cada constante y nombre simbólico debe estar definido una y solo una vez**.

N: número de veces que se define cada constante o nombre simbólico

$$GE_{16} = \frac{1}{N}$$

{Sistema} {Implementación}

$GE_{17} = 1 - ST_{10}$

Cuanto **más se utiliza un método sobrecargado**, proporciona una indicación de generalidad y utilidad de la sobrecarga de dicho método.

$GE_{18} = CM_{22}$

Al **sobrecargar o estar parametrizadas las clases y métodos** para que se apliquen sobre distintos tipos de datos se está dotando de generalidad a dichos elementos.

GE_{19}: Debe disponerse de **documentación de cada uno de los componentes** del sistema.

$$GE_{19} = \frac{DO_2 + DO_3 + DO_4 + DO_5 + DO_6 + DO_8 + DO_9 + DO_{10} + DO_{11} + DO_{12}}{10}$$

{Sistema} {Análisis, Diseño, Implementación}

$GE_{20} = EX_1$

Para obtener una mayor generalidad, **el proceso lógico de los métodos de una clase ha de ser independiente de constantes o ha de estar parametrizado**.

$GE_{21} = EX_5$

En un sistema general, **no deben existir efectos colaterales** o, caso de existir, deben estar muy bien documentados.

GE_{22}: La **herencia de un atributo por** parte de **diversas clases** indica que dicho atributo es general.

N: número de clases que heredan cada atributo

$$GE_{22} = \begin{cases} 1 - \frac{1}{N} & \\ 0 & \forall\, N < 1 \end{cases}$$

{Atributo, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: simplicidad.

GE_{23}: La **cantidad de atributos que se heredan** proporciona una idea de lo general que es la clase.

N: número de atributos que se heredan

$$GE_{23} = \begin{cases} 1 - \frac{1}{N} & \\ 0 & \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Perjudica a: simplicidad.

$GE_{24} = CM_7$

Para no perder generalidad, **deben utilizarse todos los métodos definidos**.

$GE_{25} = DC_{12}$

Deben sobrecargarse los métodos para permitir manejar un conjunto general de tipos de datos.

GE_{26}: **Un método es menos general si hay que sobre-escribirlo**, puesto que esto indica que tal como estaba diseñado no cumplía plenamente la función para la que se pensó.

N: número de veces que se sobre-escribe un método
k: se recomienda un valor de 2 ó 3: a partir de 2 ó 3 sobre-escrituras de un método empieza a ser sospechoso de poca generalidad

$$GE_{26} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

$GE_{27} = CN_2$

La **definición de métodos operador** permite que el código generado sea más general en cuanto a que su aplicación se realiza igual que para tipos de datos normales.

$GE_{28} = CN_4$

La **reutilización de métodos mediante la herencia** da una indicación de lo generales que son dichos métodos.

$GE_{29} = 1 - ST_{30}$

Cuantos **más mensajes se reciban del exterior a la clase**, querrá indicar que dicha clase está siendo muy utilizada, reflejando, por tanto, una gran generalidad.

$GE_{30} = CS_{31}$

Las clases se deben reutilizar, bien por medio de la herencia, bien por una relación de uso, puesto que cuantas más se reutilicen, serán más generales.

$GE_{31} = DC_{16}$

La **sobrecarga de las clases genéricas** dota de mayor generalidad a este tipo de clases, ya que permiten trabajar sobre nuevos tipos de datos que necesitan de un manejo diferente.

GE_{32}: Deben **definirse distintos tipos de constructores para cada clase**, para que ésta sea lo más general posible, es decir, pueda inicializarse del mayor número de formas posibles.

N: número de constructores

$$SR_9 = \begin{cases} 1 - \dfrac{1}{N} & \\ 0 & \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$GE_{33} = CN_9$

Si **las clases se reutilizan sin modificación** por medio del uso o la herencia, quiere decir que son clases generales que pueden utilizarse en diversos lugares.

$GE_{34} = CN_{10}$

Si **las clases se reutilizan, modificando su comportamiento**, por medio de la herencia, implica que son clases más generales y que pueden emplearse en distintos lugares con una pequeña adaptación.

$GE_{35} = 1 - ST_{35}$

Cuanto mayor sea el **número de descendientes de una clase**, mayor será su generalidad.

GE_{36}: **Cuantas más clases tenga una jerarquía**, más sencillo será que alguna clase de dicha jerarquía pueda ser utilizada para obtener un mayor provecho.

N: número de clases por jerarquía
k: el ingeniero del *software* definirá un valor apropiado dependiendo el volumen de la jerarquía, aunque se recomiendan valores por debajo de 5 con el fin de no obligar a jerarquías excesivamente grandes

$$GE_{36} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \quad \forall\, N < k$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

Perjudica a: simplicidad.

4.3.18. INDEPENDENCIA DE LA MÁQUINA (MI)

Hay catorce medidas correspondientes al criterio independencia de la máquina (Figura 4.5), que está relacionada con dos factores. Entre las medidas, hay dos referencias directas a medidas anteriores pertenecientes al criterio comunicaciones estándar.

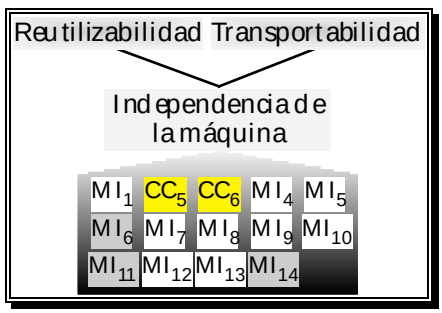


Figura 4.5: Medidas del criterio independencia de la máquina

MI_1: Debe utilizarse para la implementación del sistema un **lenguaje** de programación que se encuentre **disponible en otras máquinas** (la misma versión y dialecto del lenguaje).

1: el lenguaje utilizado está disponible en otras máquinas
0: el lenguaje no está disponible en otras máquinas

{Sistema} {Implementación}

$MI_2 = CC_5$

La **entrada de información** al sistema es uno de los elementos próximos a la máquina y, por tanto, **debe estar localizada en pocos módulos** o clases, para evitar que el resto tengan esta dependencia de la máquina.

$MI_3 = CC_6$

La **salida de información** proporcionada por el sistema es uno de los elementos cercanos a la máquina y, por tanto, **debe estar localizada en pocos módulos** o clases, para evitar que el resto tengan también esta dependencia de la máquina.

MI_4: El **código** debe ser **independiente de los tamaños de las palabras**, caracteres, etc. de la máquina.

N: número de clases cuyo código depende de los tamaños de las palabras
T: total de clases

$$MI_4 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Jerarquía, Sistema} {Implementación}

MI_5: Los **datos** deben ser **independientes de los tamaños de las palabras, caracteres**, etc. de la máquina.

N: número de clases con datos que dependen de los tamaños de las palabras
T: total de clases

$$MI_5 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Jerarquía, Sistema} {Implementación}

MI_6: **No se debe utilizar código máquina** o ensamblador.

N: número de clases con código máquina o ensamblador
T: total de clases

$$MI_6 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Jerarquía, Sistema} {Implementación}

Favorece a: simplicidad.

MI$_7$: Se deben **utilizar librerías de entrada/salida estándar**.

1: se usan funciones de librería estándar de entrada/salida
0: no se usan librerías estándar de entrada/salida
(0, 1): se usa alguna función de librería estándar de entrada/salida

{Jerarquía, Sistema} {Implementación}

MI$_8$: La **salida gráfica** es **independiente del tipo de monitor y de la resolución y número de colores de la tarjeta de vídeo**.

1: la salida no depende de la tarjeta de vídeo
0: la salida depende de la tarjeta de vídeo
(0, 1): la salida tiene alguna dependencia de la tarjeta de vídeo

{Jerarquía, Sistema} {Implementación}

MI$_9$: La **salida impresa** es **independiente del tipo de impresora, trazador...**

1: la salida no depende del dispositivo de impresión
0: la salida depende del dispositivo de impresión
(0, 1): la salida tiene alguna dependencia del dispositivo de impresión

{Jerarquía, Sistema} {Implementación}

MI$_{10}$: Las **comunicaciones por los puertos** han de ser **independientes de su tipo y de su configuración**.

1: comunicaciones independientes de los puertos
0: comunicaciones dependen de los puertos
(0, 1): las comunicaciones tienen alguna dependencia de los puertos

{Jerarquía, Sistema} {Implementación}

MI$_{11}$: **No deben incluirse llamadas al sistema operativo relacionadas con un *hardware*** determinado.

N: número de clases con llamadas al sistema operativo
T: total de clases

$$MI_{11} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Jerarquía, Sistema} {Implementación}

Favorece a: independencia del sistema *software*.

MI$_{12}$: **Los atributos no deben depender de la máquina**.

N: número de atributos independientes de la máquina
T: total de atributos

$$MI_{12} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \ N > T - 1$$

{Clase, Jerarquía, Sistema} {Implementación}

MI_{13}: Debe **evitarse el uso de métodos dependientes de la máquina**.

N: número de métodos independientes de la máquina
T: total de métodos

$$MI_{13} = \begin{cases} 1 - \log_T(T-N) \\ 1 \end{cases} \quad \forall \; N > T-1$$

{Clase, Jerarquía, Sistema} {Implementación}

MI_{14}: En el programa **no deben existir partes del código que tengan algún tipo de dependencia de la máquina**.

N: número de sentencias independientes de la máquina
T: total de sentencias

$$MI_{14} = \begin{cases} 1 - \log_T(T-N) \\ 1 \end{cases} \quad \forall \; N > T-1$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

Favorece a: simplicidad.

4.3.19. INDEPENDENCIA DEL SISTEMA *SOFTWARE* (SS)

Hay nueve medidas correspondientes al criterio independencia del sistema *software* (Figura 4.6), que influye en dos factores. Una de las medidas constituye una referencia directa a una medida del criterio anterior.

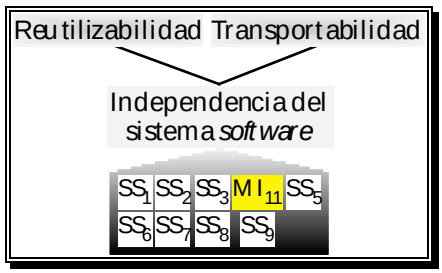


Figura 4.6: Medidas del criterio independencia del sistema *software*

SS_1: El sistema desarrollado n**o debe tener dependencia de utilidades *software* externas**. Cuantas más utilidades externas se usen, el programa será más dependiente del sistema *software*.

1: no tiene dependencia de utilidades externas
0: tiene dependencia de utilidades externas
(0, 1): tiene dependencia de alguna utilidad externa

{Sistema} {Implementación}

SS_2: Debe **reducirse la utilización de bibliotecas de rutinas externas no estándar**, puesto que una función proporcionada por el compilador o por el sistema operativo puede no ser exactamente la misma que en otro.

N: número de bibliotecas de rutinas externas no estándar usadas
T: total de bibliotecas de rutinas usadas

$$SS_2 = 1 - \frac{N}{T}$$ {Jerarquía, Sistema} {Implementación}

SS_3: Debe **evitarse la utilización de construcciones no estándar** del lenguaje. El uso de ciertas construcciones de un lenguaje permitidas por ciertos compiladores puede causar problemas de conversión cuando el *software* se migra a un nuevo entorno.

N: número de clases con construcciones no estándar del lenguaje
T: total de clases

$$SS_3 = \begin{cases} 1 - \log_T(N) \\ 1 \qquad \forall \ N < 1 \end{cases}$$ {Jerarquía, Sistema} {Implementación}

$SS_4 = MI_{11}$

No deben incluirse llamadas al sistema operativo, pues se estaría ligando el programa a dicho sistema operativo.

SS_5: Debe **evitarse el uso de atributos dependientes del sistema *software***.

N: número de atributos independientes del *software*
T: total de atributos

$$SS_5 = \begin{cases} 1 - \log_T(T - N) \\ 1 \qquad \forall \ N > T - 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Implementación}

SS_6: Debe **evitarse el uso de métodos dependientes del sistema *software*** (referencias a la FAT o los nodos-i, por ejemplo).

N: número de métodos independientes del *software*
T: total de métodos

$$SS_6 = \begin{cases} 1 - \log_T(T - N) \\ 1 \qquad \forall \ N > T - 1 \end{cases}$$ {Clase, Jerarquía, Sistema} {Implementación}

SS_7: Debe **evitarse el uso de directivas de compilación propias del compilador** (como ejemplo, la directiva `pragma` del preprocesador de C++).

N: número de directivas de compilación no estándar

$$SS_7 = \begin{cases} \frac{1}{N} \\ 1 \quad \forall N < 1 \end{cases}$$ {Sistema} {Implementación}

SS_8: **No debe utilizarse código dependiente del compilador**.

1: se usa código independiente del compilador
0: se usa código dependiente del compilador
(0, 1): ocasionalmente se usa código dependiente del compilador

{Jerarquía, Sistema} {Implementación}

SS_9: El sistema debe resultar **independiente del sistema de gestión de bases de datos** empleado.

1: es independiente respecto al sistema de gestión de bases de datos
0: depende del sistema de gestión de bases de datos
(0, 1): tiene alguna dependencia del sistema de gestión de bases de datos

{Sistema} {Implementación}

4.3.20. INSTRUMENTACIÓN (IN)

Hay trece medidas correspondientes a la instrumentación (Figura 4.7), que está relacionada con dos factores. Entre las medidas, hay dos referencias directas a medidas anteriores correspondientes al criterio comunicatividad.

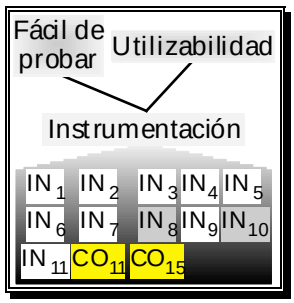


Figura 4.7: Medidas del criterio instrumentación

IN_1: Deben **cubrirse el máximo número de posibilidades de ejecución** durante las pruebas modulares.

N: número de caminos probados
T: total de caminos posibles

$$IN_1 = \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

IN_2: Deben **probarse los valores límites de los parámetros** de entrada.

N: número de parámetros de entrada cuyos valores límite se prueban
T: total de parámetros de entrada con valores límite

$$IN_2 = \frac{N}{T}$$

{Jerarquía, Sistema} {Diseño, Implementación}

IN_3: Debe **incorporarse código específico para** ayudar a la realización de **pruebas**.

1: se ha incluido código específico de ayuda a las pruebas
0: no se ha incluido
(0, 1): hay algo de código específico para ayudar en las pruebas

{Clase, Jerarquía, Sistema} {Implementación}

IN_4: Una de las pruebas de integración consiste en **probar todas las interfaces de las clases**.

N: número de interfaces probadas
T: total de interfaces

$$IN_4 = \frac{N}{T}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

IN_5: Uno de los aspectos de las pruebas de integración comprende **verificar que se han cumplido los requisitos de ejecución**.

N: número de requisitos de ejecución comprobados
T: total de requisitos de ejecución

$$IN_5 = \frac{N}{T}$$

{Sistema} {Implementación}

IN_6: Deben utilizarse **escenarios** (o un mecanismo equivalente) **para probar** los distintos módulos del sistema.

N: número de clases que cubre cada escenario
T: total de clases

$$IN_6 = \frac{N}{T}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

IN_7: Debe incorporarse la posibilidad de poder **resumir las entradas y las salidas proporcionadas al realizar las pruebas**. Los resultados de las pruebas y la manera en que se presentan estos resultados es muy importante para la efectividad de las pruebas. Esto es especialmente útil durante las pruebas del sistema por el potencialmente gran volumen de datos de entrada y salida que se producen.

1: se pueden imprimir resúmenes de entradas y salidas de las pruebas
0: no se puede
(0, 1): se puede imprimir parte de las entradas y salidas de las pruebas

{Jerarquía, Sistema} {Implementación}

IN_8: Para poder afrontar los **errores** que puedan producirse, deben **interceptarse** para poder manejarlos de la forma apropiada.

N: número de excepciones consideradas
k: umbral a definir por el ingeniero del *software*, aunque se recomienda un valor bajo (por ejemplo, 2 ó 3). Usar menos excepciones probablemente significaría que no se está aprovechando esta posibilidad.

$$IN_8 = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \quad \forall\ N < k$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: tolerancia a errores.

IN_9: El sistema ha de incluir la posibilidad de realizar un **registro de las operaciones realizadas por el usuario**.

1: se puede realizar un registro de lo realizado por el usuario
0: no se puede
(0, 1): se registran algunas operaciones

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

IN_{10}: El sistema ha de incluir la posibilidad de **medir y resumir las operaciones realizadas por el usuario**, almacenándolas en un fichero o en un listado, con el fin de analizar las más frecuentes para intentar optimizarlas o mejorarlas en la medida de lo posible.

1: se puede medir lo realizado por el usuario
0: no se puede
(0, 1): se miden algunas operaciones

{Sistema} {Requisitos, Análisis, Diseño, Implementación}

Favorece a: operatividad.

IN_{11}: El sistema ha de incluir la posibilidad de **medir y resumir los errores generados** por el sistema, con el fin de poder estudiar los que se comenten con mayor frecuencia para intentar subsanarlos de la mejor manera posible.

1: se pueden medir los errores generados
0: no se puede
(0, 1): se miden algunos errores

{Sistema} {Implementación}

$IN_{12} = CO_{11}$

Los **mensajes** de error y avisos al usuario deben ser **claros y no ambiguos**, aportando toda la información necesaria para su corrección o evitar su repetición.

$IN_{13}= CO_{15}$

Cada error generado deberá ir **acompañado de un mensaje de error y una explicación adicional** para aumentar la información y aclarar al usuario la situación producida.

4.3.21. MODULARIDAD (MO)

Hay treinta medidas correspondientes a la modularidad (Figura 4.8), que influye en seis de los factores. Entre las medidas, dieciséis son referencias a medidas anteriores correspondientes a otros criterios, una de ellas inversa.

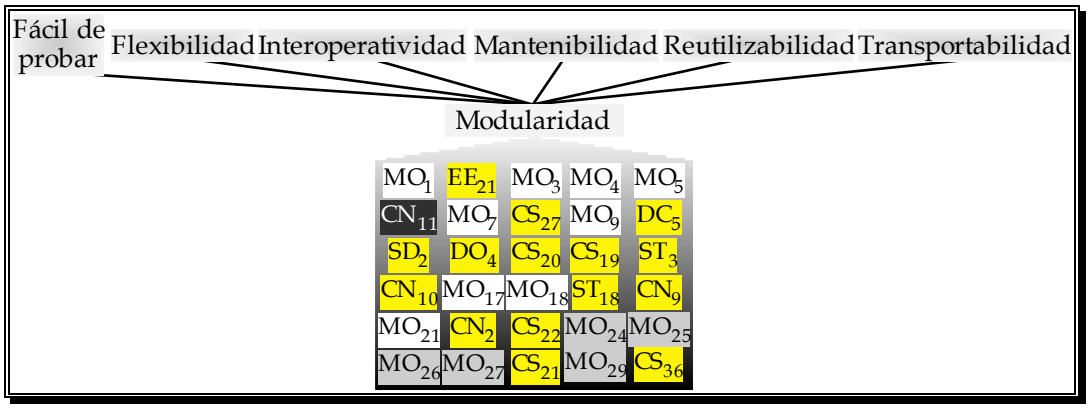


Figura 4.8: Medidas del criterio modularidad

MO_1: Los **métodos deben tener un tamaño reducido**. [Lorenz, 93] estima que se entiende por tamaño reducido aquel método que no supera las 25 líneas de código C++ (9 en Smalltalk) con sentencias ejecutables o declaraciones.

N: número de métodos con un tamaño reducido
T: total de métodos

$$MO_1 = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Clase, Jerarquía, Sistema} {Implementación}

$MO_2= EE_{21}$

Los **métodos no deben ser excesivamente pequeños**, puesto que, en ese caso, se estaría alcanzando una modularidad exagerada, es decir, se podría llegar a una atomización de los componentes.

MO_3: Las clases deben tener el **número apropiado de métodos**. El número medio adecuado según [Lorenz, 93] se estima que debería ser 20. Una media superior indica demasiadas responsabilidades en demasiadas pocas clases. Una media inferior refleja demasiadas clases para pocos servicios, esto es, una modularización excesiva.

N: número de métodos por clase

$$MO_3 = \begin{cases} 1 - \dfrac{|20 - N|}{20} \\ 0 \quad si N > 40 \end{cases}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

MO_4: Las clases deben tener el **número apropiado de atributos**. El número medio máximo según [Lorenz, 93] se estima que debería ser 6, puesto que una media superior podría indicar que la clase hace más de lo que debe, mientras que una media inferior podría reflejar una clase demasiado elemental.

N: número de atributos por clase

$$MO_4 = \begin{cases} 1 - \dfrac{|6 - N|}{6} \\ 0 \quad si N > 12 \end{cases}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

MO_5: Las **clases deben tener un único objetivo**. El concepto de modularidad se basa en que cada módulo represente una única funcionalidad.

1: las clases tienen solo un objetivo
0: las clases tienen varios objetivos

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$MO_6 = 1 - CN_{11}$

Dentro de una jerarquía de clases, **cuanto mayor sea el número de relaciones de herencia, más disminuirá la modularidad** debido a las interrelaciones que se crean.

MO_7: Las jerarquías de clases deben tener el **número apropiado de clases**, atendiendo a la finalidad de la jerarquía.

1: las jerarquías de clases tienen el número apropiado de clases
0: las jerarquías de clases tienen pocas o demasiadas clases
(0, 1): las jerarquías no tienen el número apropiado de clases

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$MO_8 = CS_{27}$

Debe **evitarse el uso de la amistad** puesto que atenta contra la modularidad. Las clases sólo deben permitir el acceso a sus atributos a través de sus métodos.

MO_9: **Las clases deben ofrecer constructores, destructor y funciones operador**.

N: número de clases con constructores, destructor y funciones operador
T: total de clases

$$MO_9 = \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

$MO_{10} = DC_5$

Las clases sólo deben permitir el acceso a sus atributos a través de sus métodos. **Las clases deben ocultar todos sus atributos**. La presencia de atributos protegidos y, en mayor medida, de atributos públicos disminuye la modularidad de la clase ya que se podría acceder a esta información sin utilizar la interfaz prevista.

$MO_{11} = SD_2$

Las clases deben disponer de **comentarios que sigan una estructura estándar** (por ejemplo, la definida en [Molloy, 95]).

$MO_{12} = DO_4$

Las clases deben llevar una **documentación técnica completa**, que describa su estructura, uso y utilidad, y explicando su modo de empleo y su funcionamiento.

$MO_{13} = CS_{20}$

Un programa modular no permitiría a un método **utilizar atributos externos de otras clases**, dado que a esta información se debe acceder a través de los servicios proporcionados por la clase.

$MO_{14} = CS_{19}$

Cuantos más atributos sean usados por otras clases, menor será la modularidad, puesto que no se estarán usando apropiadamente los servicios de estas clases.

$MO_{15} = ST_3$

Permitir **la utilización de un atributo muchas veces desde fuera de la propia clase** atenta contra la idea de una clase modular.

$MO_{16} = CN_{10}$

La **reutilización de las clases, modificando su comportamiento**, por medio de la herencia contribuye a la estructura modular del sistema.

MO_{17}: Una clase debe **ofrecer un buen número de métodos** para dar un amplio servicio cumpliendo las normas de la modularidad.

N: número de métodos definidos en la clase
k: con valores entre 2 y 5 se empiezan a obtener valores aceptables para la recomendación de [Lorenz, 93]

$$MO_{17} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \qquad \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

MO_{18}: Las **clases** deben ser autosuficientes y **necesitar el menor número posible de servicios de otras clases**.

N: número de métodos externos llamados por un método

$$MO_{18} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$MO_{19} = ST_{18}$

Las **clases** deben ser autosuficientes y **necesitar, lo menos posible, información de otras clases**.

$MO_{20} = CN_9$

La **reutilización de las clases sin modificación por medio del uso o la herencia** contribuye a aumentar la estructura modular del sistema.

MO_{21}: Las **clases abstractas** favorecen la modularidad debido a que permiten definir esquemas de comportamiento comunes a otras clases.

N: número de clases abstractas

$$MO_{21} = \begin{cases} 1 - \dfrac{1}{N} \\ 0 \qquad \forall\, N < 1 \end{cases}$$

{Sistema} {Diseño, Implementación}

$MO_{22} = CN_2$

La presencia de **métodos operador** aumenta la modularidad debido a que es un servicio (estándar) más que ofrecen las clases.

$MO_{23} = CS_{22}$

La presencia de **funciones no miembro** atenta contra los principios de la modularidad.

MO_{24}: **Cohesión entre los atributos** usados por los métodos [Chidamber, 94]. La cohesión de los métodos de una clase es deseable, puesto que favorece la encapsulación. La falta de cohesión puede indicar que la clase debería ser dividida en dos o más subclases

I_i: conjunto de atributos utilizados por el método *i*
P: número de pares de conjuntos (I_i, I_j) con intersección nula
Q: número de pares de conjuntos (I_i, I_j) con intersección no nula

$$MO_{24} = \begin{cases} \dfrac{1}{P-Q} \\ 1 \end{cases} \quad \forall\, P \le Q$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

MO_{25}: **Cohesión fuerte en las clases** [Bieman, 95a] (véase el apartado 2.3.2.4).

D: número de conexiones directas entre métodos
N: número de métodos
NP: número máximo posible de conexiones

$$NP = \frac{N \cdot (N-1)}{2}$$

$$MO_{25} = \frac{D}{NP}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

MO_{26}: **Cohesión débil en las clases** [Bieman, 95a] (véase el apartado 2.3.2.4).

D: número de conexiones directas entre métodos
N: número de métodos
I: número de conexiones indirectas entre métodos
NP: número máximo posible de conexiones

$$NP = \frac{N \cdot (N-1)}{2}$$

$$MO_{26} = \frac{D+I}{NP}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

MO_{27}: Medida de la **cohesión entre los argumentos** de los métodos de las clases [Chen, 93].

N: número de métodos de la clase, que definen *N* conjuntos con los parámetros que utiliza cada uno de ellos
M: número de conjuntos disjuntos formados por la unión de los conjuntos con intersección no vacía

$$MO_{27} = \text{máx}\left(0, 1 - \frac{M}{N}\right)$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

Favorece a: simplicidad.

$MO_{28} = CS_{21}$

Para conseguir una buena modularidad, **cada método solamente debe estar definido en una clase**.

MO_{29}: **Cohesión entre los métodos** de las clases [Bansiya, 99a].

M_i: conjunto de los tipos de los parámetros del método *i*
T: conjunto de los tipos de los parámetros de los *n* métodos de una clase

$$T = \bigcup_{i=1}^{n} M_i$$

$$P_i = M_i \cap T$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$$MO_{29} = \frac{\sum_{i=1}^{n} |P_i|}{|T| \cdot n}$$

Favorece a: simplicidad.

$MO_{30} = CS_{26}$

La **distribución de clases en ficheros** favorece la modularidad del sistema.

4.3.22. OPERATIVIDAD (OP)

Hay dieciocho medidas correspondientes al criterio operatividad (Figura 4.9), que está relacionado con un factor. Entre las medidas, hay nueve referencias directas a medidas anteriores correspondientes a otros criterios.

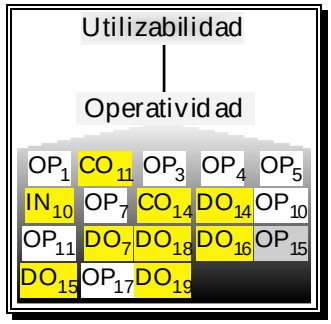


Figura 4.9: Medidas del criterio operatividad

OP_1: Deben estar descritos detalladamente todos los **pasos necesarios para poner en marcha** y hacer funcionar adecuadamente **al sistema**.

1: están descritos los pasos para el funcionamiento
0: no están descritos
(0, 1): no están descritos adecuadamente todos los pasos

{Sistema} {Implementación}

$OP_2 = CO_{11}$

Todos los **errores y mensajes al usuario deben estar descritos** apropiadamente.

OP_3: Debe existir la posibilidad de que **el usuario pueda interrumpir el proceso y poder continuarlo más tarde**.

1: puede interrumpirse y continuarse el proceso
0: no se puede interrumpir y continuar
(0, 1): no siempre se puede interrumpir y continuar

{Sistema} {Diseño, Implementación}

OP_4: El **número de intervenciones requeridas del usuario** debe estar **limitado**.

1: el usuario tiene que intervenir poco
0: el usuario ha de tener un gran número de intervenciones
(0, 1): caso intermedio

{Sistema} {Diseño, Implementación}

OP_5: Debe existir una **descripción precisa de cómo empezar y terminar cada una de las operaciones**.

1: está descrito cómo realizar cada operación
0: no está descrito
(0, 1): no están descritas adecuadamente todas las operaciones

{Sistema} {Implementación}

$OP_6 = IN_{10}$

Se puede almacenar un fichero o un listado con el **registro de todas las acciones del usuario**.

OP_7: El **diálogo con el usuario** debe realizarse de manera **estándar y uniforme**.

1: los mensajes y respuestas del usuario se realizan de forma estándar
0: no son estándar
(0, 1): algunos mensajes y respuestas no son estándar

{Sistema} {Implementación}

$OP_8 = CO_{14}$

La **interfaz de usuario** debe ser sencilla, atractiva y fácil de utilizar, en una palabra, **amigable**.

$OP_9 = DO_{14}$

Para facilitar el manejo del sistema, uno de los componentes de la documentación debe ser un **completo manual de usuario**.

OP_{10}: El **esfuerzo** requerido **para aprender a manejar el sistema** y a interpretar o analizar los resultados no debe ser muy elevado.

1: se requiere poco esfuerzo para el aprendizaje
0: se necesita un gran esfuerzo de aprendizaje
(0, 1): se requiere un esfuerzo medio

{Sistema} {Implementación}

OP_{11}: La **entrada y salida de datos** deben ser **sencillas, generales, independientes del dispositivo y estar bien documentadas**.

1: la entrada y salida de datos es sencilla, general, independiente del dispositivo y está bien documentada
0: nunca lo es
(0, 1): no siempre lo es

{Sistema} {Implementación}

$OP_{12} = DO_7$

Para facilitar el manejo del sistema es importante que la **documentación tenga una buena legibilidad**.

$OP_{13} = DO_{18}$

Para facilitar el manejo del sistema es importante que la **documentación sea fácilmente comprensible**.

$OP_{14} = DO_{16}$

La inclusión de un **tutorial de manejo** del sistema, permite ayudar a utilizar el *software*.

OP_{15}: **Influencia de los errores en el entorno**. Si un error causa una caída del sistema, siendo necesario un reinicio, se dificultará el uso del *software*.

N: número de errores que provocan una caída del sistema
T: número total de errores

$$OP_{15} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$

{Sistema} {Implementación}

Favorece a: tolerancia a errores.

$OP_{16} = DO_{15}$

El programa debe ir acompañado de un **sistema de ayuda en línea interactivo**.

OP_{17}: El sistema permite que **el usuario pueda configurar distintas opciones** de funcionamiento y de interfaz según sus gustos y necesidades.

1: el sistema es plenamente configurable
0: el sistema no es configurable
(0, 1): el sistema es configurable en parte

{Sistema} {Diseño, Implementación}

$OP_{18} = DO_{19}$

El programa debe ir acompañado de un **sistema de ayudas sensibles al contexto**.

4.3.23. PRECISIÓN (AU)

Hay seis medidas correspondientes a la precisión (Figura 4.10), que está relacionada con un factor.

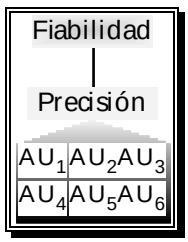


Figura 4.10: Medidas del criterio precisión

AU_1: Debe realizarse, y presupuestarse, la realización de un **análisis de errores**.

1: realizado análisis de errores durante el análisis de requisitos
0: no se ha realizado
(0, 1): no se ha realizado completamente

{Sistema} {Requisitos}

AU_2: La **librería matemática** utilizada debe proporcionar la **precisión suficiente**, para lo que debe ser chequeada teniendo en cuenta los objetivos generales de precisión.

1: la librería matemática tiene precisión suficiente
0: no tiene suficiente precisión
(0, 1): algunas funciones no tienen la precisión suficiente

{Sistema} {Implementación}

AU_3: Deben existir **requisitos de precisión** en los datos que se manejan.

E = 1: si existen requisitos de precisión en las entradas
S = 1: si existen requisitos de precisión en las salidas
C = 1: si existen requisitos de precisión en las constantes
P = 1: si existen requisitos de precisión en el proceso

$$AU_3 = \frac{E + S + C + P}{4}$$

{Sistema} {Requisitos}

AU_4: Los **algoritmos numéricos** utilizados deben proporcionar la **precisión requerida**.

1: los algoritmos numéricos tienen precisión suficiente
0: no tienen suficiente precisión
(0, 1): no todos los algoritmos numéricos tienen la precisión requerida

{Sistema} {Implementación}

AU_5: Los **resultados** alcanzados deben quedar **dentro de los límites de tolerancia**.

N: número de resultados (salidas) dentro de los límites de tolerancia
T: total de resultados (salidas)

$$AU_5 = \begin{cases} \log_T(N) \\ 0 \end{cases} \quad \forall \; N < 1$$

{Sistema} {Implementación}

AU_6: No debe haber **pérdida de precisión** en las asignaciones o expresiones.

N: número de asignaciones y expresiones con pérdida de precisión
T: total de asignaciones y expresiones numéricas

$$AU_6 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

4.3.24. SEGUIMIENTO (TR)

Hay tres medidas correspondientes al criterio seguimiento (Figura 4.11), que está relacionado con tres factores.

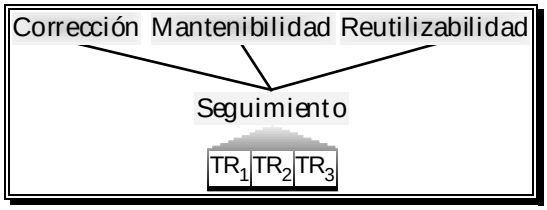


Figura 4.11: Medidas del criterio seguimiento

TR_1: Los **requerimientos** de la especificación de requisitos *software* deben estar **reflejados** e identificados en la documentación del **análisis**.

N: número de requisitos referenciados en el análisis
T: total de requisitos

$$TR_1 = \frac{N}{T}$$ {Sistema} {Análisis}

TR_2: Los **requerimientos** de la especificación de requisitos *software* deben estar **reflejados** e identificados en los productos obtenidos durante el **diseño**.

N: número de requisitos referenciados en el diseño
T: total de requisitos

$$TR_2 = \frac{N}{T}$$ {Sistema} {Diseño}

TR_3: Los **requerimientos** de la especificación de requisitos *software* deben estar **reflejados** e identificados en la **implementación**. Debe usarse alguna notación uniforme, prólogos de comentarios o comentarios embebidos para proporcionar esta referencia cruzada.

N: número de requisitos referenciados en la implementación
T: total de requisitos

$$TR_3 = \frac{N}{T}$$ {Sistema} {Implementación}

4.3.25. SIMPLICIDAD

Hay ciento dos medidas correspondientes a la simplicidad (Figura 4.12), que está relacionada con cinco factores. Entre las medidas, hay sesenta referencias a medidas anteriores correspondientes a otros criterios, de las cuales, quince son referencias inversas.

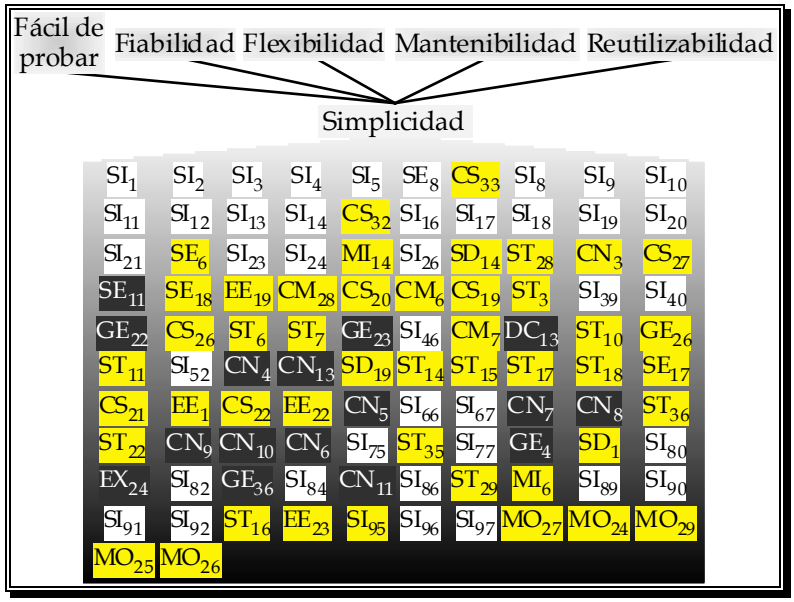


Figura 4.12: Medidas del criterio simplicidad

SI_1: La realización del sistema resulta más sencilla si se utiliza un **diseño *top-down*,** puesto que permite seguir una línea de razonamiento más natural.

1: diseño *top-down*
0: diseño *bottom-up*

{Sistema} {Diseño}

SI_2: **No** deben existir **métodos ni clases duplicadas**.

1: no hay métodos ni clases duplicadas
0: hay métodos o clases duplicadas

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SI_3: El **procesamiento de la información no debe depender del origen de la entrada de los datos ni del destino de la salida**.

N: número de métodos cuyo proceso depende del origen de la entrada o del destino de la salida
T: total de métodos

$$SI_3 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_4: El procesamiento no debe tener **memoria de las ejecuciones anteriores**, es decir, no debe depender del conocimiento o resultados de procesamientos previos.

N: número de funciones que pueden mantener información entre activaciones sucesivas (en C++, con variables locales estáticas)

$$SI_4 = \begin{cases} \frac{1}{N} \\ 1 \end{cases} \quad \forall \; N < 1$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_5: Cada clase debe incluir una **descripción de las entradas, las salidas, el proceso y sus limitaciones**.

N: número de métodos con documentación de las entradas, salidas, proceso y limitaciones
T: total de métodos

$$SI_5 = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_6 = SE_8$

Debe **evitarse** al máximo **la presencia de datos u objetos globales**, pues su manejo puede resultar más complicado ya que se puede acceder a ellos desde cualquier parte del programa y, por tanto, ser modificados sin desearlo.

$SI_7 = CS_{33}$

La **utilización de un lenguaje orientado a objetos** favorece la sencillez del código.

SI_8: La **ejecución de cada método** debe producirse de forma **secuencial**, es decir, no deben producirse saltos en su interior, debiendo comenzar siempre por el principio (sin puntos de entrada alternativos, como en FORTRAN) y sin poder regresar a distintos lugares tras su ejecución.

N: número de métodos que no se ejecutan secuencialmente
T: total de métodos

$$SI_8 = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \ N < 1$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_9: Deben **evitarse las expresiones lógicas** excesivamente **complejas o negativas**, puesto que estos tipos de expresiones incrementan la dificultad de comprensión. Las expresiones compuestas con negaciones o dos o más operadores lógicos a menudo pueden ser eludidas.

N: número de expresiones lógicas complejas o negativas
T: total de expresiones lógicas

$$SI_9 = \begin{cases} \log_T(T - N) \\ 0 \end{cases} \quad \forall \ N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{10}: Deben **evitarse los saltos hacia el interior de los bucles y los saltos desde el interior de un bucle hacia el exterior**. Los bucles deben tener un único punto de entrada y un único punto de salida.

N: número de saltos hacia el interior de un bucle y desde un bucle al exterior (debe tenerse en cuenta las sentencias `break` y `continue` de C++ dentro de los bucles)
T: total de bucles

$$SI_{10} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{11}: Debe **evitarse** la **modificación de un índice de un bucle** tipo `for` dentro de su cuerpo porque no solo complica la lógica del bucle sino que puede causar problemas a la hora de realizar la depuración.

N: número de bucles `for` en los que sus índices son modificados en el cuerpo del bucle
T: total de bucles `for`

$$SI_{11} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{12}: El **código** de las clases **no debe ser modificado desde la ejecución** del propio programa. Si un módulo tiene la capacidad de modificar su lógica de proceso, sería muy difícil reconocer su estado si ocurre un error. Además, el estudio estático de la lógica del módulo se complica notablemente.

N: número de módulos cuyo código se automodifica
T: total de módulos

$$SI_{12} = \begin{cases} 1 - \log_T(N) \\ 1 \end{cases} \quad \forall \; N < 1$$ {Jerarquía, Sistema} {Implementación}

SI_{13}: **Todos los datos que necesite un método tienen que ser pasados como parámetros o bien ser atributos** de la clase a la que pertenece, evitándose los problemas potenciales que pueden aparecer si se usan variables globales, constantes...

N: número de datos de entrada a un método que no son parámetros ni atributos de su clase

$$SI_{13} = \begin{cases} \dfrac{1}{N} \\ 1 \end{cases} \quad \forall \; N < 1$$ {Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{14}: Las **etiquetas** (destino de un salto) **no son necesarias** (no se incluye aquí el concepto de etiqueta de las sentencias `switch` del lenguaje C++), a menos que se realice algún salto, por lo que su presencia, en términos generales, complicará el código haciéndolo más difícil de comprender.

N: número de etiquetas del método

$$SI_{14} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Implementación}

$SI_{15} = CS_{32}$

La **repetición de los nombres de los identificadores** complica en exceso la comprensión del código y su modificación.

SI_{16}: **Cada variable debe utilizarse para una única finalidad**.

1: cada variable se usa para una única finalidad
0: hay variables con diversas finalidades

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{17}: La presencia de **diferentes tipos de datos en una misma expresión** complica el código.

N: número de expresiones con más de un tipo
T: total de expresiones

$$SI_{17} = 1 - \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{18}: La existencia de demasiados **bloques anidados en un módulo**, aumenta la complejidad de la lógica del programa.

N: máximo nivel de anidamiento de bloques en un método
k: se recomienda un valor bajo (por ejemplo, 2) para intentar que el número de bloques anidados se mantenga también bajo

$$SI_{18} = \begin{cases} \frac{1}{\log_k(N)} \\ 1 \quad \forall\, N < k \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{19}: La cantidad de posibles ramas o **caminos de ejecución** debe mantenerse lo suficientemente bajo para aumentar la simplicidad.

N: número de sentencias condicionales o repetitivas
T: total de sentencias ejecutables

$$SI_{19} = \begin{cases} \log_T(T - N) \\ 0 \quad \forall\ N > T - 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{20}: Debe **evitarse la presencia de saltos** (sentencias `goto`) dentro del programa, puesto que atenta contra las normas elementales de la programación estructurada.

N: número de saltos

$$SI_{20} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{21}: Debe intentarse **reducir el uso de punteros**, puesto que complican considerablemente el código.

N: número de declaraciones de atributos, variables o argumentos de tipo puntero
T: total de declaraciones de atributos, variables o argumentos

$$SI_{21} = \begin{cases} \log_T (T - N) \\ 0 \qquad\quad \forall\ N > T - 1 \end{cases}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{22} = SE_6$

El **código inútil** (que no se puede ejecutar nunca) no hace más que complicar el código del programa sin aportar ningún beneficio.

SI_{23}: La presencia de **datos con distintos alcances** (locales, parámetros, globales y atributos) en los métodos complica el código.

N: número de alcances distintos de identificadores a los que accede un método (incluye los locales, los parámetros, los globales y los atributos)

$$SI_{23} = 1 - \frac{N}{4}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{24}: La **densidad de variables usadas en los métodos no debe ser excesiva**, puesto que cuantas más variables (locales, parámetros, globales y atributos) se usen, mayor complejidad tendrá dicho método.

N: número de sentencias que usan más de tres variables
T: total de sentencias ejecutables

$$SI_{24} = 1 - \frac{N}{T}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{25} = MI_{14}$

El **código dependiente de la máquina** complica su comprensión.

SI_{26}: La **sobre-escritura de atributos y métodos** puede complicar el código del sistema debido a la aparición de identificadores con el mismo nombre y significado distinto según su contexto.

N: número de atributos y métodos sobre-escritos
T: total de atributos y métodos

$$SI_{26} = 1 - \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{27} = SD_{14}$

La **complejidad ciclomática** de McCabe da una idea de lo complicado que resulta cada método. (El valor de esta medida por cada clase corresponde a la media de la complejidad de los métodos publicada en [Etzkorn, 99].)

$SI_{28} = ST_{28}$

No debe abusarse de la posibilidad de **herencia múltiple**, puesto que complica la comprensión de la jerarquía.

$SI_{29} = CN_3$

Los **métodos virtuales** simplifican el código, puesto que no hay que distinguir con qué clase concreta se está trabajando en cada instante.

$SI_{30} = CS_{27}$

Las clases sólo deben permitir el acceso a sus atributos a través de sus métodos. Debe **evitarse el uso de la amistad** puesto que provoca un código más complejo al tenerse una puerta abierta para violar la privacidad de los miembros de las clases.

$SI_{31} = 1 - SE_{11}$

El uso de **operaciones con bits** debe ser reducido, puesto que complica el código.

$SI_{32} = SE_{18}$

El **tamaño de los parámetros formales** no debe ser demasiado elevado, puesto que a menor tamaño, mayor simplicidad del código de la función.

$SI_{33} = EE_{19}$

El **tamaño de los parámetros actuales** no debe ser demasiado elevado, puesto que a menor tamaño, mayor simplicidad del código de la llamada a la función.

$SI_{34} = CM_{28}$

Todos **los parámetros formales** de los métodos **tienen que ser utilizados** en su cuerpo.

$SI_{35}= CS_{20}$

Un **método no debe utilizar atributos externos** de otras clases puesto que este hecho aumenta la complejidad del sistema.

$SI_{36}= CM_{6}$

Todos **los atributos definidos** en una clase **deben utilizarse**.

$SI_{37}= CS_{19}$

No deben usarse atributos por otras clases, puesto que aumenta la dificultad para seguir el código.

$SI_{38}= ST_{3}$

La **utilización de un atributo desde fuera de la propia clase** causa un incremento de su complejidad, dado que cualquiera puede acceder a él.

SI_{39}: **Cuantos menos atributos se tengan en una clase, más sencilla será**. Debe tenerse en cuenta que las clases grandes pueden resultar más difíciles de entender y, por tanto, de reutilizar (principalmente a través de la herencia, requiriendo modificaciones en la clase).

N: número de atributos (definidos y heredados) de cada clase
k: deberá intentarse fijar un valor lo más bajo posible (2 ó 3, por ejemplo)

$$SI_{39} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SI_{40}: **Sobre-escribir los atributos** muchas veces puede complicar el programa en exceso.

N: número de veces que se sobre-escribe un atributo
k: se recomienda un valor bajo (2, por ejemplo)

$$SI_{40} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{41}= 1 - GE_{22}$

Cuantas más **clases hereden un atributo**, mayor complejidad tendrá.

$SI_{42}= CS_{26}$

La implementación será más clara y comprensible si **cada clase** se sitúa **en un fichero diferente**.

$SI_{43} = ST_6$

Cuantas más **clases usen un atributo** (por herencia o composición), mayor será la complejidad del sistema.

$SI_{44} = ST_7$

Cuantos más **métodos utilicen un atributo**, mayor será la complejidad del sistema.

$SI_{45} = 1 - GE_{23}$

Cuanto mayor sea la **cantidad de atributos que se heredan**, mayor será la complejidad.

SI_{46}: Cuantos **más métodos tenga una clase** (tanto definidos como heredados), más compleja será su utilización debido a que su interfaz será más complicada. Hay que tener en cuenta, además, que a mayor número de métodos en una clase, mayor será el posible impacto en sus hijos, debido a que éstos heredarán dichos métodos [Rosenberg, 99].

N: número de métodos (definidos o heredados) de una clase
k: es recomendable un valor bajo (por ejemplo, 2 ó 3)

$$SI_{46} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \quad \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$SI_{47} = CM_7$

Todos **los métodos** definidos en una clase **deben ser utilizados**.

$SI_{48} = 1 - DC_{13}$

Cuantas más veces se tenga que **sobrecargar un método**, más complicada resultará la construcción de la clase.

$SI_{49} = ST_{10}$

La **utilización de métodos sobrecargados** puede complicar la comprensión del código porque la decisión de a qué método hay que llamar exactamente depende no solo del nombre del método sino también del tipo de los parámetros actuales y de su número.

$SI_{50} = GE_{26}$

El **número de veces que se sobre-escriba un método**, va en proporción a lo complejas que quedarán las clases al alterarse su comportamiento.

$SI_{51} = ST_{11}$

La **utilización de métodos sobre-escritos** complica el código porque el método llamado depende del ámbito desde el que se realice la invocación.

SI_{52}: La **utilización de métodos operador** simplifica el código generado, debido a que no es necesario emplear nombres de métodos para realizar ciertas operaciones habituales sobre tipos definidos por el usuario.

N: número de veces que se utiliza un método operador
k: es recomendable un valor bajo (por ejemplo, 2)

$$SI_{52} = \begin{cases} 1 - \dfrac{1}{\log_k(N)} \\ 0 \end{cases} \quad \forall\, N < k$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{53} = 1 - CN_4$

Cuantas más **clases hereden un mismo método**, mayor será la complejidad del sistema.

$SI_{54} = 1 - CN_{13}$

Aunque el **enlace dinámico** permite la escritura de un código más conciso, resulta más complicado comprender su funcionamiento en tiempo de desarrollo.

$SI_{55} = SD_{19}$

Si no se sigue una **convención uniforme en los nombres** que se dan a todos los identificadores, la comprensión y desarrollo del sistema se puede complicar considerablemente.

$SI_{56} = ST_{14}$

Cuanto mayor sea el **número de clases que llaman a un método**, mayor complejidad tendrá el programa.

$SI_{57} = ST_{15}$

Cuanto mayor sea el **número de métodos que llaman a un método**, más complejo será el programa.

$SI_{58} = ST_{17}$

Cuanto mayor sea el **número de métodos llamados por cada método**, mayor será la complejidad de dicho método.

$SI_{59} = ST_{18}$

Cuantas más **clases utilicen información de otras clases**, mayor será su complejidad.

$SI_{60} = SE_{17}$

La simplicidad del cuerpo de los métodos se verá afectada conforme crezca el **tamaño de memoria necesario para almacenar sus variables locales**.

$SI_{61} = CS_{21}$

Para conservar la simplicidad, solo debe **definirse cada método en una clase**.

$SI_{62} = EE_{1}$

La **cantidad de mensajes enviados desde una clase** influye negativamente en su complejidad.

$SI_{63} = CS_{22}$

La **utilización de funciones no miembro** (en aquellos lenguajes que se permita) causa un aumento de la complejidad de los programas, debido a que no es un mecanismo estándar de la orientación a objetos.

$SI_{64} = EE_{22}$

El manejo de los **métodos** resulta más complicado si como **valor de retorno**, éstos devuelven **un objeto** de una clase.

$SI_{65} = 1 - CN_{5}$

Cuanto mayor sea el **número de definiciones de métodos genéricos**, más complicado será el desarrollo.

SI_{66}: Cuanto menor sea el **número de clases** existentes en un sistema, será más fácil de controlar y, por tanto, será más simple.

N: número de clases
k: según el volumen del sistema, será recomendable un valor entre 2 y 5

$$SI_{66} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\ N < k \end{cases}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

SI_{67}: Cuantos **más objetos de cada clase se definan**, mayor será la complejidad del sistema.

N: número de objetos definidos de cada clase
k: dependiendo de la magnitud del problema, se recomiendan valores entre 2 y 5

$$SI_{67} = \begin{cases} \dfrac{1}{\log_k(N)} & \\ 1 & \forall\ N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{68} = 1 - CN_7$

Cuantas **más definiciones de clases genéricas** se realicen, mayor complejidad se alcanzará en dichas clases.

$SI_{69} = 1 - CN_8$

La complejidad de cada jerarquía se verá incrementada con el **número de padres** de cada clase.

$SI_{70} = ST_{36}$

La complejidad de cada jerarquía se verá incrementada con el **número de hijos** que tenga cada clase [Harrison, 00]. Cuantos más hijos tenga, más posibilidades habrá para un uso inadecuado de la clase padre, además de requerir un mayor trabajo en las pruebas.

$SI_{71} = ST_{22}$

Cuantos **más objetos se definan** en el seno de cada clase, mayor será la complejidad del sistema.

$SI_{72} = 1 - CN_9$

La complejidad se incrementa con cada **reutilización de las clases sin modificación** por medio del uso o la herencia.

$SI_{73} = 1 - CN_{10}$

La complejidad se ve incrementada con cada **reutilización de las clases, modificando su comportamiento**, por medio de la herencia.

$SI_{74} = 1 - CN_6$

La posibilidad de admitir un **número variable de parámetros** (elipsis en C++), complica los métodos al tener que incorporar código especial para obtener los valores de los parámetros no declarados.

SI_{75}: Una clase será más simple cuando el **número de clases de las que hereda** (directa o indirectamente) no sea elevado.

N: número de ascendientes (padres, abuelos...) de cada clase

$$SI_{75} = \begin{cases} \frac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$SI_{76} = ST_{35}$

Cuanto mayor sea el **número de descendientes de una clase**, mayor será la complejidad.

SI_{77}: Una gran cantidad de **clases aisladas** (sin ninguna relación de herencia con el resto) implica una creciente complejidad.

N: número de clases aisladas
k: es recomendable un valor bajo (por ejemplo, 2 ó 3)

$$SI_{77} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Sistema} {Análisis, Diseño, Implementación}

$SI_{78} = 1 - GE_4$

La interfaz de una clase será más complicada de manejar conforme tenga un mayor **número de elementos accesibles**.

$SI_{79} = SD_1$

El **código debe estar adecuadamente comentado** puesto que esto disminuye la complejidad de comprensión del sistema.

SI_{80}: El **número de relaciones de herencia entre una clase raíz y una clase hoja** debe mantenerse lo más limitado posible para no complicar en exceso la comprensión de la jerarquía. Cuanto más profundo es un grafo de herencia, mayor es la complejidad para su diseño e implementación, puesto que se verán involucrados un mayor número de métodos y clases.

N: número máximo de arcos desde una raíz hasta una hoja en el grafo de relaciones de herencia

$$SI_{80} = \frac{1}{N}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$SI_{81} = 1 - EX_{24}$

Conforme mayor sea el **número de jerarquías de clases independientes**, mayor será la complejidad del sistema.

SI_{82}: Las **clases no deben aparecer más de una vez en cada jerarquía de herencia**. Además, esto puede reflejar un error en la jerarquía.

N: número de veces que aparece una clase en la jerarquía

$$SI_{82} = \frac{1}{N}$$

{Clase, Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$SI_{83} = 1 - GE_{36}$

Cuantas **más clases tenga una jerarquía**, más complicada será la estructura de dicha jerarquía.

SI_{84}: Las **jerarquías de herencia serán más simples si utilizan pocos objetos**.

N: número de objetos usados en la jerarquía
k: es recomendable un valor bajo del umbral (entre 2 y 3)

$$SI_{84} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Jerarquía, Sistema} {Diseño, Implementación}

$SI_{85} = 1 - CN_{11}$

Cuanto mayor sea el **número de herencias de una jerarquía**, más compleja será dicha jerarquía.

SI_{86}: Las jerarquías de herencia serán menos simples conforme tengan más **clases hoja de la jerarquía**.

N: número de clases hoja en la jerarquía
k: es recomendable un valor bajo (por ejemplo, 2)

$$SI_{86} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \qquad \forall\, N < k \end{cases}$$

{Jerarquía, Sistema} {Análisis, Diseño, Implementación}

$SI_{87} = ST_{29}$

Las jerarquías de herencia serán menos simples conforme tengan más **clases raíz de la jerarquía**.

$SI_{88} = MI_6$

La utilización de **código máquina o ensamblador** complica el código.

SI_{89}: La complejidad se ve aumentada por la presencia de **variables volátiles** (propias de lenguajes como C++).

N: número de variables volátiles

$$SI_{89} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Jerarquía, Sistema} {Implementación}

SI_{90}: Debe evitarse el **uso de alias** o referencias entre nombres de variables.

N: número de variables alias de otras (en C++, uso de declaraciones con &)

$$SI_{90} = \begin{cases} \dfrac{1}{N} \\ 1 \quad \forall\, N < 1 \end{cases}$$

{Jerarquía, Sistema} {Implementación}

SI_{91}: La **cantidad de mensajes enviados al exterior** de una clase va en oposición a la sencillez del código.

N: número de mensajes enviados al exterior de la clase
k: se recomienda un valor bajo (por ejemplo, 2) para no elevar la complejidad de la clase

$$SI_{91} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \quad \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{92}: La **cantidad de mensajes enviados al interior** de una clase va en oposición a la sencillez del código.

N: número de mensajes enviados al interior de la clase
k: se recomienda un valor bajo (por ejemplo, 2 ó 3), aunque depende del volumen medio de las clases del sistema

$$SI_{92} = \begin{cases} \dfrac{1}{\log_k(N)} \\ 1 \quad \forall\, N < k \end{cases}$$

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{93} = ST_{16}$

Una **clase** englobada **en una jerarquía** de herencia llevará asociada menos complejidad cuanto **más cercana se encuentre de la raíz** de dicha jerarquía. Según un estudio en [Lake, 92a], los programadores depuran y modifican las clases más eficientemente si están cerca de la raíz, que si están en las profundidades de la jerarquía.

$SI_{94} = EE_{23}$

La complejidad de cada método dependerá de la **cantidad de métodos que se invoquen** al activarse dicho método. La dificultad de depuración y pruebas de la clase depende del número de llamadas realizadas, dado que requerirá una mayor comprensión de dicha clase.

SI_{95}: **Contenido de inteligencia** [Halstead, 77]. Cuanto mayor sea el contenido de inteligencia que posee el programa, mayor será su complejidad.

N_1: número total de operadores
N_2: número total de operandos
N: longitud del programa o número total de operadores y operandos
η_1: número de operadores distintos
η_2: número de operandos distintos
η: número de operandos y operadores distintos
V: volumen del programa
$\hat{L}$: nivel estimado del programa
I: contenido de inteligencia del programa

$$N = N_1 + N_2 \quad \eta = \eta_1 + \eta_2$$
$$V = N \cdot \log_2 \eta \quad \hat{L} = \frac{2 \cdot \eta_2}{\eta_1 \cdot N_2} \quad I = \hat{L} \cdot V$$
$$SI_{95} = \frac{1}{\log I}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{96}: **Volumen del programa** [Halstead, 77]. A mayor volumen, mayor complejidad.

N_1: número total de operadores
N_2: número total de operandos
N: longitud del programa o número total de operadores y operandos
η_1: número de operadores distintos
η_2: número de operandos distintos
η: número de operandos y operadores distintos
V: volumen del programa

$$N = N_1 + N_2 \quad \eta = \eta_1 + \eta_2 \quad V = N \cdot \log_2 \eta$$
$$SI_{96} = \frac{2}{\log V}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

SI_{97}: **Dificultad del programa** [Halstead, 77]. Cuanto menor sea la dificultad, mayor simplicidad tendrá el programa.

N_1: número total de operadores
N_2: número total de operandos
N: longitud del programa o número total de operadores y operandos
η_1: número de operadores distintos
η_2: número de operandos distintos
η: número de operandos y operadores distintos
η_2^*: número de operandos potenciales
η^*: vocabulario potencial
V: volumen del programa
V^*: volumen potencial del programa
D: dificultad del programa

$$N = N_1 + N_2 \quad \eta = \eta_1 + \eta_2 \quad \eta^* = 2 + \eta_2^*$$
$$V = N \cdot \log_2 \eta \quad V^* = \eta^* \cdot \log_2 \eta^* \quad D = \frac{V}{V^*}$$
$$SI_{97} = \frac{1}{D}$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$SI_{98} = MO_{27}$

Una **baja cohesión entre los argumentos** de los métodos de las clases aumenta la complejidad.

$SI_{99} = MO_{24}$

Una **baja cohesión entre los atributos** de una clase aumenta la complejidad, incrementándose también, por tanto, la posibilidad de cometer errores durante el desarrollo.

$SI_{100} = MO_{29}$

Una **baja cohesión entre los métodos** de las clases aumenta la complejidad.

$SI_{101} = MO_{25}$

Una **baja cohesión fuerte de las clases** aumenta la complejidad.

$SI_{102} = MO_{26}$

Una **baja cohesión débil de las clases** aumenta la complejidad.

4.3.26. TOLERANCIA A ERRORES (ET)

Hay veintidós medidas correspondientes al criterio tolerancia a errores (Figura 4.13), que está relacionado con dos factores. Entre las medidas, hay dos referencias directas a medidas de criterios anteriores.

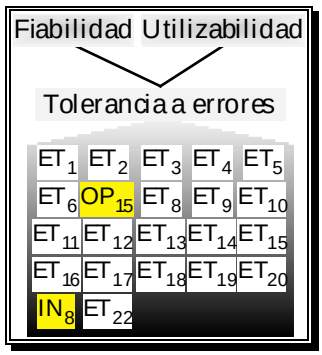


Figura 4.13: Medidas del criterio tolerancia a errores

ET_1: La capacidad de disponer de un **proceso paralelo** debe estar controlada. Las funciones que puedan ser utilizadas concurrentemente deben ser controladas de forma centralizada para proporcionar el control de la concurrencia, bloqueos de entrada/salida, etc.

1: se incluyen controles para un proceso paralelo
0: no se incluyen

{Sistema} {Diseño, Implementación}

ET_2: Debe incluirse la posibilidad de que **al producirse un error**, **el usuario pueda solucionar** sobre la marcha **el error** producido y, a continuación, **proseguir** con la ejecución del proceso, si lo desea.

1: cuando se detecta un error, se da al usuario la posibilidad de corregirlo y continuar
0: no se tiene esta posibilidad
(0, 1): otro caso intermedio

{Jerarquía, Sistema} {Implementación}

ET_3: **Cuando se detecte un error**, debe **informarse al módulo que ha realizado la llamada**, para que éste pueda tomar la decisión de cómo procesar el error.

1: al detectar un error, se informa al llamador
0: no se informa al llamador
(0, 1): no siempre se informa al llamador

{Jerarquía, Sistema} {Implementación}

ET_4: En la especificación de requisitos deben indicarse los **requerimientos de recuperación de errores en los datos de entrada**.

1: los requisitos identifican las capacidades de recuperación de errores en las entradas
0: no se identifican
(0, 1): no las identifcan completamente

{Sistema} {Requisitos}

ET_5: La **entrada de información debe ser chequeada** en cuanto al tipo de datos, tamaño o longitud de la entrada, rango de valores válidos... antes de iniciar su procesamiento, con el fin de identificar todos los posibles datos erróneos. El procesamiento no debe comenzar hasta que los errores hayan sido notificados y las correcciones hayan sido realizadas.

N: número de datos de entrada que se chequean
T: total de datos de entrada

$$ET_5 = \frac{N}{T}$$

{Sistema} {Diseño, Implementación}

ET_6: Debe **comprobarse** la existencia de **combinaciones ilegales o contradictorias en los datos de entrada**.

1: se chequea si existen datos contradictorios o combinaciones ilegales en las entradas
0: no se chequea
(0, 1): no se comprueban siempre

{Sistema} {Diseño, Implementación}

$ET_7 = OP_{15}$

Si se produce algún **error**, **no debe afectar al sistema**, de tal forma que se pueda seguir trabajando normalmente.

ET_8: Se debe **comprobar** que toda **la información de entrada necesaria para el proceso se encuentra disponible**, para evitar avanzar a lo largo de varios pasos del proceso antes de descubrir que los datos son incompletos.

1: se determina que todos los datos para el proceso están presentes
0: no se comprueba
(0, 1): no se comprueban siempre

{Sistema} {Diseño, Implementación}

ET_9: Deben definirse requisitos para la **recuperación ante fallos computacionales**.

1: hay requisitos para la recuperación de errores computacionales
0: no hay requisitos
(0, 1): no están completamente definidos

{Sistema} {Requisitos}

ET_{10}: Deben **comprobarse los valores de los índices en los bucles y transferencias de múltiples datos** antes de su utilización.

N: número de índices de bucles y transferencias múltiples que se comprueban antes de su uso
T: total de índices

$$ET_{10} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{11}: Debe **verificarse** que los **índices de los vectores** (o matrices) se encuentran dentro del rango permitido antes de su utilización.

N: número de índices de vectores que se comprueban antes de usarlos
T: total de accesos a vectores

$$ET_{11} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{12}: Debe **verificarse** la **cantidad de memoria dinámica proporcionada** por el sistema operativo tras una petición de memoria.

N: número de veces que se comprueba la memoria dinámica obtenida
T: total de solicitudes de memoria dinámica

$$ET_{12} = \begin{cases} 1 - \log_T(T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{13}: Debe **liberarse toda la memoria dinámica** proporcionada por el sistema operativo tras una petición de memoria.

1: se libera toda la memoria dinámica solicitada
0: no se libera la memoria dinámica
(0, 1): no se libera toda la memoria dinámica

{Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{14}: Debe evitarse el **acceso a punteros con un valor nulo**.

N: número de veces que se comprueba que un puntero no tiene un valor nulo antes de su utilización
T: total de accesos a través de punteros

$$ET_{14} = \begin{cases} 1 - \log_T (T - N) \\ 1 \end{cases} \quad \forall \; N > T - 1$$

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{15}: Deben **verificarse los resultados de los procesos**.

1: se chequean los resultados de los procesos
0: no se chequean
(0, 1): se chequean algunos resultados

{Sistema} {Diseño, Implementación}

ET_{16}: Deben definirse **requisitos** para la **recuperación ante fallos del *hardware***.

1: hay requisitos para la recuperación de errores *hardware*
0: no hay requisitos
(0, 1): hay requisitos, pero definidos de forma incompleta

{Sistema} {Requisitos}

ET_{17}: Deben establecerse **mecanismos** para la **recuperación ante fallos del *hardware***.

1: se ha previsto la recuperación de errores *hardware*
0: no hay recuperación de errores *hardware*
(0, 1): hay algún mecanismo

{Sistema} {Análisis, Diseño, Implementación}

ET_{18}: Deben definirse **requisitos** para la **recuperación ante fallos de los dispositivos** de entrada/salida, como un fin de fichero inesperado, errores de lectura o escritura...

1: hay requisitos para la recuperación de errores de entrada/salida
0: no hay requisitos
(0, 1): hay requisitos, pero definidos de forma incompleta

{Sistema} {Requisitos}

ET_{19}: Se deben **prever los errores en el acceso a dispositivos** de entrada/salida y tratarlos adecuadamente.

N: número de accesos a dispositivos con tratamiento de errores
T: total de accesos a dispositivos

$$ET_{19} = \frac{N}{T}$$ {Clase, Jerarquía, Sistema} {Diseño, Implementación}

ET_{20}: Debe incluirse la **gestión de excepciones**.

1: se ha contemplado el manejo de excepciones
0: no se ha contemplado

{Método, Clase, Jerarquía, Sistema} {Diseño, Implementación}

$ET_{21} = IN_8$

Para poder afrontar posibles **errores, deben interceptarse**, mediante la gestión de excepciones, cuando éstos se produzcan, y manejarlos adecuadamente.

ET_{22}: Las clases deben haber previsto la **posibilidad de que ocurran errores y deben ser capaces de manejarlos** adecuadamente.

N: número de clases con gestión de excepciones
T: total de clases

$$ET_{22} = \frac{N}{T}$$ {Jerarquía, Sistema} {Diseño, Implementación}

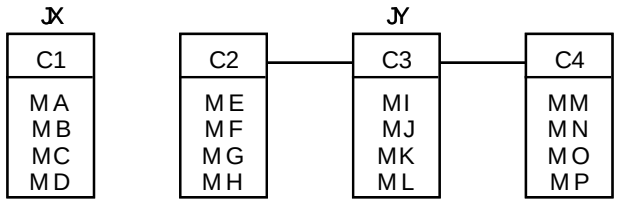
4.4. UTILIZACIÓN DEL MODELO DE CALIDAD

En este apartado se van a comentar tres aspectos relacionados con la utilización del modelo de calidad que acaba de ser descrito. El primero de ellos tiene que ver con la forma de obtener y calcular los valores de los criterios, factores y la calidad total a lo largo del árbol. El segundo aspecto se refiere a las formas de utilizar la información obtenida con el árbol de calidad. Finalmente, se resume el método de aplicación del modelo de calidad durante el desarrollo de un sistema.

4.4.1. AGREGACIÓN DE LOS VALORES EN EL ÁRBOL DE CALIDAD

Según se acaba de ver, para una única medida se pueden obtener distintos valores dependiendo de si son de aplicación sobre atributos, métodos, clases, jerarquías o el sistema. Para obtener un único valor de cada medida con el fin de agregar su valor con el resto de las medidas y propagarlo por el árbol, se deberán acumular los distintos valores obtenidos mediante una media aritmética. De esta forma, los valores para atributos y métodos se acumularán sobre la clase; éstas sobre la jerarquía y éstas sobre el sistema. Siguiendo este proceso, se conseguirá un único valor de calidad de la medida acumulado sobre el sistema, que será la que se utilizará para la propagación por el árbol.

Ejemplo: Supóngase que un sistema lo forman dos jerarquías, compuestas por un total de cuatro clases que albergan 16 métodos, tal como indica el siguiente diagrama:



Para calcular, por ejemplo, el valor de la medida 19 de la simplicidad, habría que evaluarla primero para cada uno de los métodos que forman el sistema. Una vez finalizados estos cálculos, habrá que acumularlos para obtener los valores de las clases, mediante la media aritmética de los valores de los métodos de cada clase. Después se calculará el valor de cada jerarquía a partir de los valores de sus clases para, finalmente, obtener el valor de la medida para el sistema con los valores de las jerarquías.

La siguiente tabla refleja los cálculos realizados:

método:	MA	MB	MC	MD	ME	MF	MG	MH	MI	MJ	MK	ML	MM	MN	MO	MP
N:	3	6	7	0	7	6	0	1	0	5	0	8	3	0	2	6
T:	6	10	9	2	16	42	9	6	5	13	9	15	4	1	9	14
SI_{19}:	0,61	0,60	0,32	1,00	0,79	0,96	1,00	0,90	1,00	0,81	1,00	0,72	0,00	1,00	0,89	0,79
clase:	C1=0,63				C2=0,91				C3=0,88				C4=0,67			
jerarquía:	JX=0,63				JY=0,82											
sistema:	0,73															

Como ya se ha comentado, todas estas medidas que se acaban de explicar miden algún atributo del *software* que puede clasificarse en uno o más de los criterios de calidad obtenidos. Para obtener un valor único para cada uno de estos criterios, deberá tomarse el valor de cada medida, exceptuando aquéllas que resultan no aplicables, y calcular una ponderación de la siguiente forma:

$$XY = \sum_i \omega_i^{XY} \cdot XY_i$$

donde *XY* representa el nombre del criterio; XY_i representa el valor de la medida número *i* para dicho criterio y ω_i^{XY} representa el peso (entre cero y uno) de la medida *i* dentro del criterio *XY*. En dicho peso se debe tener en cuenta el número de medidas aplicables, de tal forma que se cumpla:

$$\sum_i \omega_i^{XY} = 1 \quad \forall i \,/\, XY_i \in [0,1]$$

Análogamente, una vez obtenidos los valores de calidad para todos los criterios, hay que obtener un valor único para cada uno de los factores. El mecanismo es similar: se pondera el valor de cada criterio hasta obtener el valor para el factor:

$$F = \sum_k \omega_k^F \cdot XY^k$$

donde *F* será el factor cuyo valor se está calculando; XY^k representa el valor de cada uno de los criterios que intervienen en el factor *F* y ω_k^F representa el peso del criterio *k* dentro del factor *F* (debe tenerse en cuenta también la posibilidad de que el criterio no tenga ningún valor válido).

Por último, el valor global de la calidad del sistema en una determinada fase, vendrá dado por una ponderación de todos los valores válidos de los factores obtenidos:

$$Q = \sum_j \omega_j \cdot F^j$$

donde *Q* representa el valor final proporcionado por el modelo de calidad; F^j representa el valor de cada uno de los 11 factores que influyen en la calidad (si son aplicables) y ω_j representa el peso de cada uno de los factores en la calidad total.

Con el fin de clarificar esta explicación, la Figura 4.1 muestra, a modo de ejemplo, un árbol de calidad ficticio con dos factores, cinco criterios y doce medidas. Se han etiquetado las ramas con los pesos, mostrándose junto a cada nodo el valor de calidad, obtenido de la forma indicada en la tabla. De haber realizado los cálculos sin tener en cuenta los pesos, el valor de calidad global habría sido de 0,57.

C1 = 0,4·0,5 + 0,1·0,4 + 0,5·0,6 = 0,54 C2 = 0,6·0,9 + 0,4·0,8 = 0,86	C3 = 0,3·0,2 + 0,2·0,1 + 0,5·0,8 = 0,48 C4 = 0,7·1,0 + 0,3·0,7 = 0,91 C5 = 0,5·0,3 + 0,5·0,0 = 0,15
F1 = 0,6·0,54 + 0,4·0,86 = 0,67	F2 = 0,3·0,48 + 0,4·0,91 + 0,3·0,15 = 0,55
Calidad = 0,7·0,67 + 0,3·0,55 = 0,63	

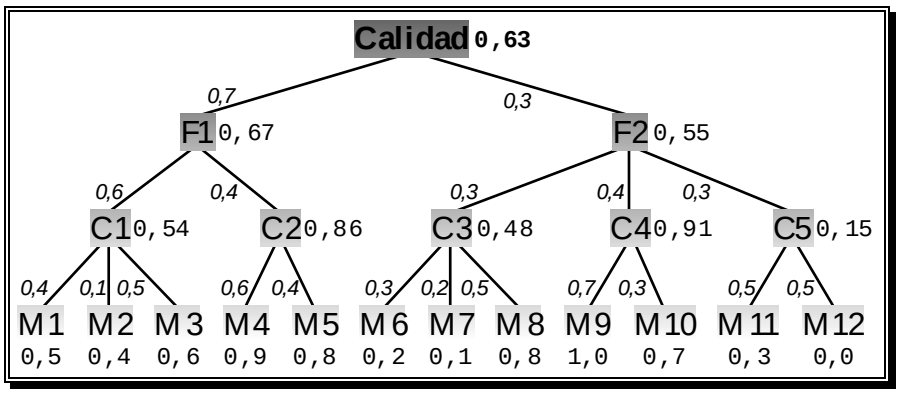


Figura 4.1: Árbol de ejemplo con los valores de calidad agregados

Cálculo de los pesos:

El valor de los distintos pesos que intervienen en los diferentes cálculos deberá ser definido por el ingeniero del *software* de acuerdo a su experiencia en el entorno de trabajo, siguiendo algunas de las técnicas existentes para realizar este proceso. Los procedimientos más utilizados para la estimación de los pesos suelen ser [Barba-Romero, 87]:

- Estimación directa: se basa en asignar los pesos uno a uno, directamente basándose en opiniones subjetivas. Después, simplemente, hay que normalizarlos.
- Método de las utilidades relativas [Churchman, 57]: consiste básicamente en ir afinando unas estimaciones provisionales, en el sentido de mejorar su consistencia interna, por medio de comparaciones binarias de subgrupos de criterios.
- Procedimiento del valor medio: está en la misma línea de estimación directa de los pesos y se ha empleado ampliamente en el contexto de los modelos de utilidad multiatributo.
- Métodos basados en la entropía [Zeleny, 74]: la idea esencial es que la importancia relativa de un criterio está directamente relacionada con la información intrínseca promedio generada por el conjunto de alternativas respecto a dicho criterio y por la asignación subjetiva de la importancia que el decisor le otorgue.
- Procedimientos por aproximación: el decisor estima una matriz de medidas subjetivas de los criterios; esta matriz tiene unas características particulares que permiten demostrar que su autovector dominante normalizado es el vector de los pesos de los criterios.
- Procedimientos de aproximación: en una línea similar al anterior, además de los datos de la matriz se utilizan las comparaciones binarias de las alternativas que se demandan al decisor; todo ello conduce, en último término, a problemas de programación matemática.

Una de las técnicas que, por su elegancia, ha sido aplicada abundantemente en multitud de situaciones, demostrando que su validez y utilidad están fuera de toda duda es el Método de Análisis Jerárquico [Saaty, 90] (del tipo de procedimientos por aproximación), una técnica para la toma de decisiones multicriterio en la que los atributos se organizan en una estructura jerárquica, partiendo desde la calidad total a los factores, criterios y medidas. Posteriormente, estos pesos podrán ser refinados basándose, por ejemplo, en la aparición de errores o en datos del mantenimiento del *software*.

Este método permite establecer la importancia relativa de los factores dentro de la calidad, de los criterios dentro de los factores e, incluso, de las medidas dentro de los criterios. Para cuantificar esta importancia relativa, se pueden construir unas **matrices de relevancia**. Para calcular una completa evaluación de la calidad, se lleva a cabo un proceso de composición desde los factores a las medidas. Los resultados se emplean nivel a nivel para obtener la calidad global del sistema.

La ventaja de definir la importancia relativa entre los elementos del modelo estriba en que se obtiene una evaluación de la calidad más objetiva. La importancia relativa de cada elemento es una cuestión dependiente de la organización; cada organización puede establecer sus propios parámetros (incluso, dependiendo del sistema). Esta importancia entre los elementos del modelo debe obtenerse a partir de la opinión de expertos, para lo cual pueden realizarse entrevistas a los gestores, los productores, los ingenieros del *software*, los usuarios...

Para determinar el grado de importancia de los elementos del modelo, hay que realizar una comparación dos a dos de los elementos del mismo nivel en el árbol de calidad. Se utiliza una escala de 1 a 9 para determinar la importancia relativa de cada par de

elementos *A* y *B*, cuya interpretación se resume en la Tabla 4.16[6]. Los valores pares, que no aparecen detallados en la tabla, también pueden utilizarse para indicar una valoración intermedia.

Valor	Interpretación
1	*A* y *B* tienen la misma importancia
3	*A* es ligeramente más importante que *B*
5	*A* es bastante más importante que *B*
7	*A* es mucho más importante que *B*
9	*A* es totalmente más importante que *B*

Tabla 4.16: Interpretación de los valores de importancia

Para clarificar el mecanismo de obtención de los pesos, la matriz de relevancia de la Tabla 4.17 muestra una posible asignación de importancia a los criterios de la corrección para un determinado sistema en un entorno concreto. Los valores de dicha Tabla 4.17 se han obtenido comparando dos a dos los criterios, teniendo en cuenta la corrección como pauta para la comparación. Así, la completitud se considera que es el criterio más importante, comparándolo con la consistencia, la documentación y el seguimiento: la completitud es mucho más importante que la consistencia, bastante más que la documentación y ligeramente más que el seguimiento. Análogamente, la importancia asignada a la consistencia puede considerarse inferior a la del seguimiento y bastante inferior a la de la documentación. Por último, se considera que el seguimiento y la documentación tienen una importancia similar desde el punto de vista de la corrección. Debe notarse que los valores inversos se utilizan para definir el resto de los valores de la matriz; aunque no es obligatorio utilizar un inverso, generalmente resulta lo más lógico. Además, los valores de la diagonal principal de la matriz están predeterminados a la unidad puesto que cada criterio tiene, evidentemente, la misma importancia que él mismo.

	Completitud	Consistencia	Documentación	Seguimiento
Completitud	1	7	5	3
Consistencia	1/7	1	1/5	1/3
Documentación	1/5	5	1	1
Seguimiento	1/3	3	1	1

Tabla 4.17: Matriz de relevancia de la corrección

Cada matriz genera un autovector, calculado normalizando los valores por columnas y obteniendo los valores medios por filas, lo que permite obtener un ordenamiento de la importancia relativa de los elementos en cada nivel del modelo.

[6] Esta escala se basa en los estudios experimentales realizados durante los años 50 por el psicólogo George Miller, que encontró que, en general, la gente solo es capaz de manejar información que incluya simultáneamente 7 ± 2 conceptos [Miller, 56].

	Completitud	Consistencia	Documentación	Seguimiento	**Autovector**
Completitud	1	7	5	3	**0,5728**
Consistencia	1/7	1	1/5	1/3	**0,0595**
Documentación	1/5	5	1	1	**0,1895**
Seguimiento	1/3	3	1	1	**0,1782**
Normalizado por:	176/105	16	36/5	16/3	

Tabla 4.18: Autovector para la matriz de corrección

La Tabla 4.18 muestra los resultados de calcular el autovector correspondiente a la matriz de la Tabla 4.17. Las clasificaciones para la última columna son 3, 1/3, 1 y 1. Sumando estos valores, se obtiene una total de 16/3. Cada valor es entonces normalizado por este total produciendo 9/16, 1/16, 3/16 y 3/16. Finalmente, una vez que cada columna ha sido normalizada, se calcula la media por filas ponderada según el valor de normalización. Así, para la primera fila, se obtiene el primer elemento del autovector:

$$\frac{\frac{105}{176}+\frac{7}{16}+\frac{25}{36}+\frac{9}{16}}{4}=0,5728$$

Los otros valores del autovector se calculan siguiendo el mismo procedimiento para cada fila de la matriz.

Una vez determinados todos los autovectores, cada elemento del autovector será uno de los pesos mencionados anteriormente, con lo que la evaluación de todos los valores de la calidad podrá ser realizada completamente.

La Tabla 4.19 muestra una matriz de relevancia general para los factores de la calidad obtenida a partir de entrevistas con *diez especialistas* en desarrollo de *software* en diversas áreas más *dos usuarios*. Esta matriz puede utilizarse como punto de partida o, simplemente, como un ejemplo referencial.

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Integridad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	6	5	2	5	5	5	4	5	5	4	**0,2648**
Eficiencia	1/6	1	1	1/4	1	1	2	1/2	1	2	1/3	**0,0490**
Fácil de probar	1/5	1	1	1/4	1	2	3	1/2	2	2	1/2	**0,0620**
Fiabilidad	1/2	4	4	1	5	5	5	3	4	5	4	**0,2088**
Flexibilidad	1/5	1	1	1/5	1	2	2	1/2	2	2	1/3	**0,0567**
Integridad	1/5	1	1/2	1/5	1/2	1	2	1/2	1	1	1/4	**0,0405**
Interoperatividad	1/5	1/2	1/3	1/5	1/2	1/2	1	1/3	1/2	1	1/3	**0,0300**
Mantenibilidad	1/4	2	2	1/3	2	2	3	1	3	4	1/2	**0,0924**
Reutilizabilidad	1/5	1	1/2	1/4	1/2	1	2	1/3	1	3	1/3	**0,0469**
Transportabilidad	1/5	1/2	1/2	1/5	1/2	1	1	1/4	1/3	1	1/4	**0,0308**
Utilizabilidad	1/4	3	2	1/4	3	4	3	2	3	4	1	**0,1181**

Tabla 4.19: Ejemplo de matriz de relevancia de los factores dentro de la calidad y su autovector

No obstante, si se desean obtener pesos para las medidas de un criterio, este método puede resultar demasiado tedioso[7], por lo que se pueden utilizar otras técnicas más simples, como puede ser una estimación directa de los pesos de las medidas uno a uno. Una vez asignados todos los pesos (por una o varias personas), no habría más que normalizarlos para que su suma fuera la unidad.

4.4.2. EVALUACIÓN DE LOS VALORES DE CALIDAD

Una vez que se ha obtenido un árbol de calidad, tras haber calculado todas las medidas pertinentes y haber agregado los valores a lo largo del árbol, se pueden establecer dos mecanismos para estudiar los resultados.

Factores	Mínimo	Máximo
Corrección	0,000	1,000
Eficiencia	0,038	0,962
Fácil de probar	0,031	0,969
Fiabilidad	0,037	0,963
Flexibilidad	0,053	0,947
Integridad	0,000	1,000
Interoperatividad	0,007	0,993
Mantenibilidad	0,050	0,950
Reutilizabilidad	0,044	0,956
Transportabilidad	0,016	0,984
Utilizabilidad	0,001	1,000
CALIDAD	0,025	0,975

Tabla 4.20: Valores mínimos y máximos teóricos para cada factor de calidad

El primero, consiste en obtener una visión global de la calidad del sistema estudiado, observando meramente el **valor de calidad total** obtenido en la raíz del árbol (el valor *Q* mencionado anteriormente). Este valor podrá dar una idea general de la calidad del sistema, pero hay que tener en cuenta que, dada la naturaleza de muchas de las medidas y las relaciones inversas existentes entre estas, entre los criterios y entre los factores, será imposible obtener para un sistema un valor de *Q* igual a la unidad. En concreto, los valores de calidad obtenidos (sin emplear pesos) se encontrarán, aproximadamente, en el rango [0,025, 0,975], debido a estas relaciones (la Tabla 4.20 muestra los mínimos y máximos teóricos para cada criterio obtenidos al minimizar y maximizar, respectivamente, las medidas[8]). Empíricamente, hay que determinar qué valores son adecuados para un sistema. Según las pruebas realizadas (véase el Anexo I: Experimentos), podría considerarse que los valores cercanos a 0,5 se corresponden a sistemas con una calidad aceptable, mientras que aquellos valores inferiores a un tercio corresponderían a sistemas con una calidad cuestionable; aquellos sistemas que logren

[7] Por ejemplo, para obtener los pesos de las medidas de simplicidad (102 medidas) habría que construir una matriz de relevancia de 10.404 elementos y responder a 5.151 comparaciones.

[8] Las medidas se han minimizado o maximizado según el orden en que aparecen en este trabajo, es decir, en el criterio donde se define su fórmula. Esto implica que las relaciones directas e inversas se basan en el valor obtenido. Si se escogiera otro orden, los valores mínimos y máximos teóricos de los factores podrían variar ligeramente.

pasar de 0,75 pueden considerarse sistemas de alta calidad. Estos umbrales también pueden ser aplicados a los factores o a los criterios, tomando igualmente en consideración los valores mínimos y máximos alcanzables, debido a las relaciones inversas antedichas.

Este único valor de calidad total (Q), que representa la calidad relativa de un sistema, encaja con el concepto de *figure of merit*, propuesto por Ramamoorthy [Ramamoorthy, 75] y basado en la media ponderada de varias medidas, que proporciona una indicación objetiva de la presencia o ausencia de las características deseadas teniendo en cuenta su importancia relativa. Este número puede utilizarse para estudiar el impacto de modificaciones o correcciones sobre la calidad del sistema, sirviendo probablemente para prevenir la tendencia habitual hacia la entropía, o el desorden, causada por los cambios durante la vida útil del producto [Brown, 87].

El segundo mecanismo para estudiar los resultados del modelo de calidad, que puede utilizarse como base o complemento del anterior, resulta más interesante en cuanto a que se extrae una mayor cantidad de información de los resultados. Consiste en estudiar los valores, no como un único valor de calidad, sino como un **vector de valores de calidad**. Este vector representa los valores de calidad obtenidos para cada uno de los factores del modelo. De esta manera, se podrá estudiar detenidamente la información proporcionada acerca de cada uno de los factores. Así se podrán identificar aquellos factores más problemáticos e intentar solucionarlos o mejorarlos. Seguidamente, puede repetirse el análisis dentro de cada factor, estudiando un vector con los valores obtenidos en los criterios, con el fin de identificar los criterios que más influyen en la baja calidad del factor. Tras haberlos detectado, se podrán, ya por último, examinar las medidas con valores más bajos. Estas medidas, junto con la breve explicación que las acompaña, permitirán establecer las verdaderas causas de los problemas, puesto que las medidas se centran, como ya ha sido mencionado anteriormente, en medir aspectos muy concretos de los atributos del *software*. Esta información deberá ser suficiente para que el ingeniero del *software* la transmita al equipo de desarrollo para que pueda proceder a la mejora del sistema.

Evidentemente, se pueden emplear ambos modos de estudiar los resultados (mediante el valor de calidad global y el vector) conjuntamente, por ejemplo, para hacer destacar aquellos módulos que deberían ser escudriñados con un especial cuidado o para comparar diferentes soluciones (o sus partes) a un mismo problema.

Una vez modificado el sistema, para solucionar los problemas encontrados, deberá recalcularse de nuevo los valores de calidad de todo el modelo, puesto que un cambio puede afectar a cualquiera de las medidas de cualquiera de los criterios, por lo que habrá que asegurarse que no han disminuido otros valores de calidad. En cualquier caso, hay que tener siempre en cuenta la influencia existente entre las diferentes medidas, criterios y factores, que aparecen reflejadas en las tablas y figuras de los anteriores apartados.

Por último, también debe estudiarse detenidamente la acción a realizar cuando se obtenga un valor bajo para, por ejemplo, uno de los criterios. Una solución consiste, como se ha visto, en intentar mejorarlo. Pero otra posible solución que debe tenerse en

cuenta estriba en aceptar esa baja calidad a fin de no empeorar al resto de los criterios. A modo de ejemplo, considérese el caso de un sistema de alarmas de una central nuclear. Evidentemente, el ingeniero del *software* que esté evaluando este sistema dará una importancia especial a criterios como control de acceso, eficiencia de ejecución y tolerancia a errores (e intentará mejorarlos) pero, quizás, le importe menos obtener valores no tan buenos en criterios como eficiencia de almacenamiento, generalidad e independencia de la máquina.

4.4.3. MÉTODO DE UTILIZACIÓN DEL MODELO DE CALIDAD

A modo de resumen, se enumeran a continuación todos los pasos a realizar para utilizar el modelo de calidad a un proyecto en un instante dado del ciclo de vida.

- Obtener los pesos de los distintos factores, criterios y medidas. Para ello, se puede emplear el Método de Jerarquía Analítica que permite obtener las diferentes matrices de relevancia adaptadas al entorno del problema.
  - Calcular los pesos de los factores, obteniendo la matriz de relevancia mediante encuestas al cliente, usuarios e ingenieros del *software*.
  - Calcular los pesos de los criterios dentro de cada factor, obteniendo las matrices de relevancia mediante encuestas a ingenieros del *software*.
  - Calcular los pesos de las medidas dentro de cada criterio. Si se desean asignar pesos diferentes a las medidas, lo más sencillo resulta la asignación directa y posterior normalización.

- Evaluar las medidas correspondientes a la fase actual. Para ello deberá calcularse la fórmula de cada medida dependiendo del ámbito de aplicación:
  - Medida aplicable a atributos. Una vez evaluada sobre cada atributo, deberá calcularse la media de la clase. A continuación, deberá evaluarse la media de la jerarquía y, por último, la media para el sistema a partir de las jerarquías.
  - Medida aplicable a métodos. Una vez evaluada sobre cada método, deberá calcularse la media de la clase. A continuación, deberá evaluarse la media de la jerarquía y, por último, la media para el sistema a partir de las jerarquías.
  - Medida aplicable a clases. Una vez evaluada sobre cada clase, deberá calcularse la media de la jerarquía. Finalmente, deberá evaluarse la media del sistema a partir de las jerarquías.
  - Medida aplicable a jerarquías. Una vez evaluada sobre cada jerarquía, deberá evaluarse la media del sistema.
  - Medida aplicable al sistema. Una vez evaluada, no precisa ningún cálculo adicional.

- Agregar las medidas en el árbol de calidad. A partir de los pesos y los valores de las medidas, se obtendrán:
  - Vectores de valores de calidad de los criterios de cada factor.
  - Vector de valores de calidad para los factores.
  - Valor de calidad total.

- Analizar los resultados.
  - Estudiar si el valor de calidad total resulta adecuado para el proyecto.
  - Estudiar el vector de valores de calidad de cada uno de los factores, con el fin de determinar cuáles son los factores susceptibles de mejora.
  - Estudiar el vector de valores de calidad de los criterios de los factores determinados en el punto anterior. De esta forma, se obtendrán los criterios susceptibles de mejora.
  - De los criterios obtenidos en el punto anterior, determinar cuáles son las medidas susceptibles de mejora.
  - De las medidas determinadas en el punto anterior, determinar qué jerarquías, clases, métodos o atributos deben ser estudiadas con el fin de solucionar el problema puesto de manifiesto por las medidas afectadas.

4.5. MEJORA DEL PROCESO *SOFTWARE* ORIENTADO A OBJETOS

Disponer de un completo modelo de calidad orientado a objetos supone una novedad científica, pero para que el éxito sea absoluto es necesario que dicho modelo pueda ser aplicado en la práctica, es decir, durante las fases del desarrollo de un sistema. En este apartado se va a exponer la definición de un proceso *software iterativo e incremental*, basándose en el modelo de calidad definido con anterioridad, de tal forma que pueda guiar a los ingenieros del *software* en su trabajo. Si se utiliza adecuadamente, será capaz también de proporcionar una base de datos histórica que puede emplearse para estudiar los desarrollos del pasado con el fin de descubrir los defectos del proceso y mejorarlos para evitar que sucedan en proyectos futuros.

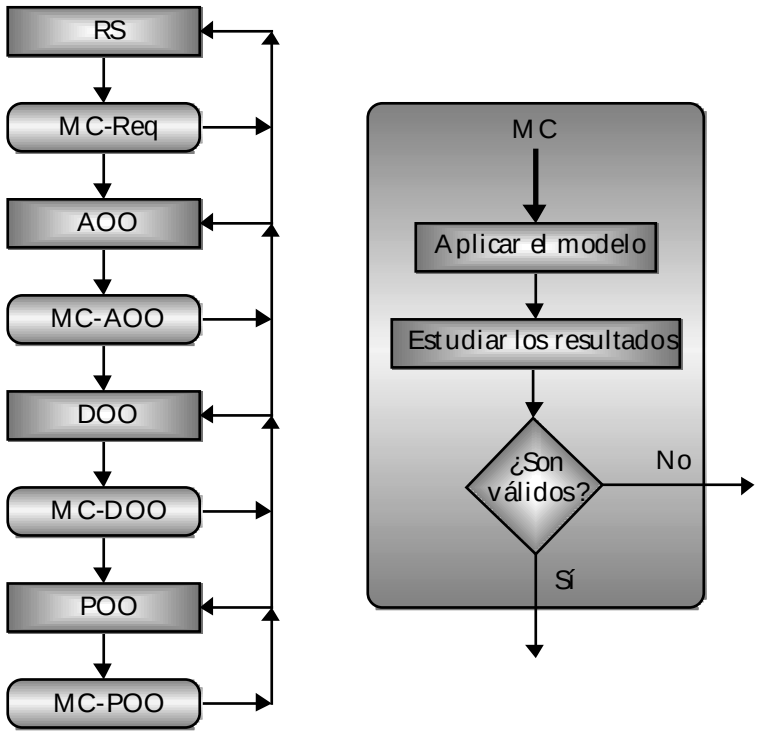


Figura 4.2: Hitos de un desarrollo orientado a objetos y la aplicación del modelo de calidad

En la Figura 4.2 se resumen los hitos principales de un desarrollo *software* orientado a objetos, independientemente de la metodología de desarrollo empleada (la mayoría

de las metodologías orientadas a objetos poseen las fases de requisitos, análisis, diseño e implementación, proporcionando, además, similares características [Embley, 95]). Sobre estos hitos se han incluido las comprobaciones a realizar empleando el modelo de calidad definido. Este modelo de calidad, evaluando las medidas en cada uno de los niveles, puede informar si se ha obtenido un valor adecuado para cada criterio, para cada factor y, en resumen, si se ha alcanzado un nivel de calidad global adecuado.

Para utilizar apropiadamente el modelo de calidad presentado en los anteriores apartados, a continuación se van a definir las fases del proceso necesario para conseguir mejorar el proceso *software*. La descripción de cada una de las fases del proceso se encuentra basada en la definición de [Humphrey, 95a]:

- Fase: Nombre de la fase y acrónimo
- Propósito: ¿Por qué se lleva a cabo esta fase?
- Criterios de comienzo: ¿Cuáles son las entradas necesarias y qué condiciones deben satisfacerse antes de comenzar la fase? ¿Qué elementos del proceso se necesitan y de dónde se obtienen?
- Tareas: ¿Qué tareas se llevan a cabo en este proceso?
- Criterios de terminación: ¿Qué condiciones deben satisfacerse al término de la fase y qué resultados se producen?
- Siguiente fase: ¿Cuáles son los pasos que vienen a continuación y cuáles son las condiciones que deben cumplirse para su ejecución o para ser seleccionados?

Por todo ello, formalmente hablando, la definición completa de las fases del proceso (únicamente se definen las fases en las que se emplea el modelo de calidad), tal como han sido explicadas, se exponen brevemente desde la Tabla 4.21 hasta la Tabla 4.24.

Fase	Calidad de los requisitos (MC-Req).
Propósito	Identificar posibles problemas sobre los requerimientos.
Criterios de comienzo	Requisitos *software*. Resultados de calidad obtenidos en esta fase previamente (en este o en otros proyectos).
Tareas	Evaluar las medidas para los requisitos, obteniendo los valores de calidad para cada criterio y cada factor. Estudiar todos los resultados: la medida de cada atributo, el valor de cada criterio, el valor de cada factor y el valor de calidad global. Comparar los resultados actuales con los previos (si existen). Analizar los problemas encontrados y extraer las conclusiones pertinentes. Almacenar los resultados (para que puedan ser utilizados posteriormente o en otros proyectos).
Criterios de terminación	Haber calculado todas las medidas correspondientes a esta fase. Haber estudiado todos los resultados y haber tomado una decisión. Haber registrado todos los datos.
Siguiente fase	Si los resultados son aceptables, comenzar la siguiente fase (AOO). En caso contrario, repetir la fase de especificación de requisitos corrigiendo los defectos detectados.

Tabla 4.21: Definición de la fase de calidad de requisitos

Fase	Calidad del análisis orientado a objetos (MC-AOO).
Propósito	Determinar la calidad del análisis, detectando las posibles anomalías o defectos que podrían conducir a un diseño de baja calidad.
Criterios de comienzo	Análisis orientado a objetos completo: deberá incluir, al menos, la lista de clases, las relaciones entre ellas, sus atributos y métodos, etc. Resultados de calidad obtenidos en esta fase previamente (en este o en otros proyectos) Resultados de calidad de la anterior fase (MC-Req).
Tareas	Evaluar las medidas para el análisis, obteniendo los valores de calidad para cada criterio y cada factor. Estudiar todos los resultados: la medida de cada atributo, el valor de cada criterio, el valor de cada factor y el valor de calidad global. Comparar los resultados actuales con los resultados previos obtenidos en la fase MC-Req y en anteriores MC-AOO (si existen). Analizar los problemas encontrados y extraer las conclusiones pertinentes. Almacenar los resultados (para que puedan ser utilizados posteriormente o en otros proyectos).
Criterios de terminación	Haber calculado todas las medidas correspondientes a esta fase. Haber estudiado todos los resultados y haber tomado una decisión. Haber registrado todos los datos.
Siguiente fase	Si los resultados son aceptables, comenzar la siguiente fase (DOO). En caso contrario, si los problemas se deben al análisis, repetir la fase de AOO corrigiendo los defectos detectados en esta fase. En caso contrario, repetir la fase de especificación de requisitos corrigiendo los defectos detectados.

Tabla 4.22: Definición de la fase de calidad del análisis

Fase	Calidad del diseño orientado a objetos (MC-DOO).
Propósito	Determinar la calidad del diseño, detectando las posibles anomalías o defectos que podrían conducir a una implementación de baja calidad.
Criterios de comienzo	Diseño orientado a objetos completo: deberá incluir, al menos, la lista de clases completa (incluyendo TAD y clases internas), las relaciones entre ellas, sus atributos y los prototipos de los métodos, visibilidad, algoritmos, etc. Resultados de calidad obtenidos en esta fase previamente (en este o en otros proyectos). Resultados de calidad de las fases anteriores (MC-Req y MC-AOO).
Tareas	Evaluar las medidas para el diseño, obteniendo los valores de calidad para cada criterio y cada factor. Estudiar todos los resultados: la medida de cada atributo, el valor de cada criterio, el valor de cada factor y el valor de calidad global. Comparar los resultados actuales con los resultados previos obtenidos en las fases MC-Req y MC-AOO y en anteriores MC-DOO (si existen). Analizar los problemas encontrados y extraer las conclusiones pertinentes. Almacenar los resultados (para que puedan ser utilizados posteriormente o en otros proyectos).
Criterios de terminación	Haber calculado todas las medidas correspondientes a esta fase. Haber estudiado todos los resultados y haber tomado una decisión. Haber registrado todos los datos.
Siguiente fase	Si los resultados son aceptables, comenzar la siguiente fase (POO). En caso contrario, si los problemas se deben al diseño, repetir la fase de DOO corrigiendo los defectos detectados en esta fase. En caso contrario, si los problemas se deben al análisis, repetir la fase de AOO corrigiendo los defectos detectados en esa fase. En caso contrario, repetir la fase de especificación de requisitos corrigiendo los defectos detectados.

Tabla 4.23: Definición de la fase de calidad del diseño

Fase	Calidad de la implementación orientada a objetos (MC-POO).
Propósito	Determinar la calidad del producto, identificando el código sospechoso que pueda causar un funcionamiento no adecuado, dificultad de mantenimiento o ampliación, reutilización compleja, etc.
Criterios de comienzo	*Software* completado, incluyendo, por tanto, todo el código fuente y la documentación. Resultados de calidad obtenidos en esta fase previamente (en este o en otros proyectos). Resultados de calidad de las fases anteriores (MC-Req, MC-AOO y MC-DOO).
Tareas	Evaluar las medidas para la implementación, obteniendo los valores de calidad para cada criterio y cada factor. Estudiar todos los resultados: la medida de cada atributo, el valor de cada criterio, el valor de cada factor y el valor de calidad global. Comparar los resultados actuales con los resultados previos obtenidos en las fases MC-Req, MC-AOO y MC-DOO y en anteriores MC-POO (si existen). Analizar los problemas encontrados y extraer las conclusiones pertinentes. Almacenar los resultados (para que puedan ser utilizados posteriormente o en otros proyectos).
Criterios de terminación	Haber calculado todas las medidas correspondientes a esta fase. Haber estudiado todos los resultados y haber tomado una decisión. Haber registrado todos los datos.
Siguiente fase	Si los resultados son aceptables, iniciar la transferencia del producto. En caso contrario, si los problemas se deben a la implementación, repetir la fase de POO corrigiendo los defectos detectados en esa fase. En caso contrario, si los problemas se deben al diseño, repetir la fase de DOO corrigiendo los defectos detectados en esta fase. En caso contrario, si los problemas se deben al análisis, repetir la fase de AOO corrigiendo los defectos detectados en esa fase. En caso contrario, repetir la fase de especificación de requisitos corrigiendo los defectos detectados.

Tabla 4.24: Definición de la fase de calidad de la implementación

Todo este mecanismo permitirá avanzar hasta la siguiente fase del desarrollo del sistema solamente en el caso de que se observe que el trabajo realizado hasta entonces ha alcanzado un nivel de calidad satisfactorio. Evidentemente, es responsabilidad del ingeniero del *software* determinar cuándo pasar a la siguiente fase o no; el procedimiento descrito proporciona únicamente un medio para evaluar la calidad.

A modo de aclaración, seguidamente se va a explicar con un ejemplo cuál sería el procedimiento que debería seguir el ingeniero del *software* durante una de las fases del desarrollo de un sistema.

Ejemplo: En un determinado momento, el desarrollo se encuentra en la fase de diseño orientado a objetos (DOO). Cuando se considera que esta fase ha finalizado y, por tanto, se ha alcanzado el hito correspondiente, se comienza con la fase denominada calidad del diseño (MC-DOO). Para llevar a cabo esta fase, es necesario disponer de toda la documentación y elementos generados durante las anteriores fases del desarrollo y, en particular, los resultados de calidad obtenidos en otros proyectos (similares) en esta misma fase y los resultados de las evaluaciones anteriores.

Las tareas a realizar comienzan con la aplicación del modelo de calidad, consistente en la evaluación de todas las medidas pertinentes sobre el diseño producido. Esto conduce a la obtención de un vector de valores asociados a cada uno de los criterios y a cada uno de los factores, juntamente con un valor de calidad global. Todos estos valores deben ser cuidadosamente estudiados por el ingeniero del *software* con el fin de detectar cualquier

posible problema y buscar la solución. El orden sugerido para el estudio podría ser el siguiente: en primer lugar, comprobar el valor de calidad global; seguidamente, ver si alguno de los factores no alcanza un umbral adecuado; a continuación, investigar los criterios con valores inadecuados para terminar analizando las medidas que no sean lo suficientemente buenas.

En este caso, el factor corrección ha obtenido un valor de 0,49. Esto hace sospechar de sus criterios, sospecha que se cumple al comprobar que la documentación tiene un 0,16, mientras que la completitud tiene un 0,63, la consistencia tiene un 0,65 y el seguimiento un 0,52.

En aquellos elementos (como la documentación) en que no se ha alcanzado un nivel de calidad apropiado deberán buscarse sus causas y qué representan. Entonces, deberán examinarse sus medidas para llegar a la raíz del problema. El modelo ayudará en cada instante puesto que cada medida irá acompañada de una breve explicación de su razón de ser. Mirando las medidas de la documentación, se obtendrán las causas del bajo valor obtenido.

Si se determina que los resultados obtenidos son válidos, entonces se puede proseguir el desarrollo con la siguiente fase, en este caso, la implementación orientada a objetos (POO). Pero si se piensa que la calidad del diseño no es lo suficientemente buena, deberá regresarse a la fase de DOO con el fin de corregir los problemas detectados. Incluso, si el ingeniero del *software* así lo considera tras la inspección detenida de los datos, deberá retornarse a fases anteriores, como a la fase de análisis orientado a objetos (AOO) o antes aún para arreglar todos los problemas detectados gracias al modelo de calidad.

Un aspecto a tener en cuenta durante la búsqueda de los problemas en los resultados consiste en que si, por ejemplo, un factor proporciona un valor de calidad bajo, habrá que buscar qué criterios causan esta anomalía y, una vez encontrados, habrá que encontrar qué atributos de calidad proporcionan unas medidas con valores bajos. Estos atributos estarán asociados a una explicación de sus medidas, con lo que la vía de solución aparecerá clara. No obstante, un valor adecuado en uno de los factores no implicará necesariamente que todos sus criterios tengan valores adecuados, puesto que uno de ellos podría tener un valor bajo que podría quedar oculto por valores altos del resto de los criterios que conforman el factor. Análogamente ocurriría entre los criterios y las medidas de los atributos. Por todo ello, la búsqueda tendrá que realizarse de forma exhaustiva a lo largo y ancho de todo el árbol de calidad. La utilización de una herramienta informática (como la que aparece descrita en el apartado 2 del Anexo III) que implemente el presente modelo de calidad y que informe de aquellos elementos con una baja calidad ayudará enormemente en este arduo proceso.

La identificación de cada uno de los problemas de cada fase del desarrollo del sistema podrá servir, si se almacenan adecuadamente en un histórico, para aprender de los errores cometidos y, por tanto, para intentar mejorar el proceso *software* tanto en el ámbito personal como corporativo. De esta forma, se obtendrán entonces productos con una calidad superior a un coste inferior, pues serán necesarios menos recursos tanto económicos como de tiempo (al cometerse menos errores, el esfuerzo necesario para las fases de validación y mantenimiento se verán reducidos).

4.6. COLOFÓN

A modo de resumen del presente capítulo, se presentan a continuación los principales aspectos cubiertos en el trabajo realizado:

Se ha definido un **modelo** que permite evaluar la **calidad del *software*** construido mediante las técnicas de la **orientación a objetos**. Este modelo se basa en una división arborescente de la **calidad** en diversos **factores**, que, a su vez, se dividen en **criterios** que pueden ser evaluados con medidas.

Para evaluar los criterios de calidad, se ha definido un amplio **conjunto de medidas**, utilizables en un desarrollo orientado a objetos, que permiten estimar el valor de calidad de cada uno de los criterios. Las medidas aportan, además, información acerca de distintos atributos de calidad que pueden estar presentes en el *software*.

Los cálculos necesarios desde que se evalúan las medidas hasta que se obtiene un valor de calidad del sistema también han sido detallados, con la descripción de la **agregación de los valores** a lo largo del árbol de calidad. Para ello, se ha propuesto también una técnica que permite proporcionar **pesos** a los distintos nodos del árbol.

Por último, se ha proporcionado un **método** que permite aplicar el modelo de calidad obtenido a los desarrollos de *software* orientados a objetos durante todo el ciclo de vida del sistema.

Elimina la causa y el efecto cesará.

— Miguel de Cervantes y Saavedra

5. RESULTADOS

5. RESULTADOS

Se ha presentado el diseño de un modelo de calidad de aplicación en el paradigma orientado a objetos, en el que se ha dividido la calidad de un sistema en 11 factores y éstos a su vez en 26 criterios. Se han proporcionado medidas que permiten evaluar estos criterios y se ha indicado un método de utilización del modelo de calidad para guiar la construcción de *software*. No obstante, el trabajo realizado no finaliza aquí, puesto que toda investigación necesita una validación. Muchos estudios sobre la calidad del *software* han examinado un único desarrollo, lo cual resulta claramente insuficiente [Khoshgoftaar, 00]. Por todo ello, en este apartado se va a mostrar el diseño experimental realizado, junto con los principales resultados obtenidos. El Anexo I completará esta validación incluyendo datos y detalles de los distintos casos experimentales.

5.1. DISEÑO EXPERIMENTAL

En la Tabla 5.1 se resumen los principales objetivos perseguidos al realizar el diseño experimental.

Objetivo	Experimentos	Experto
Validar medidas de la fase de análisis	1.1, 2.1	√
Validar medidas de la fase de diseño	1.2, 2.2	√
Validar medidas de la fase de implementación	1.3, 2.3	√
Validar criterios de la fase de análisis	3.1, 4.1	√
Validar criterios de la fase de diseño	3.2, 4.2	√
Validar criterios de la fase de implementación	3.3, 4.3	√
Validar factores de la fase de análisis	5.1, 6.1	√
Validar factores de la fase de diseño	5.2, 6.2	√
Validar factores de la fase de implementación	5.3, 6.3	√
Validar la calidad en la fase de análisis	7.1, 8.1	√
Validar la calidad en la fase de diseño	7.2, 8.2	√
Validar la calidad en la fase de implementación	7.3, 8.3	√
Validar los pesos en las medidas	9	√
Validar los pesos en los criterios	5, 6	√
Validar los pesos en los factores	7, 8, 10, 11, 14	√
Contrastar la calidad en varios desarrollos	10, 26	√
Contrastar la calidad en paradigma imperativo y orientado a objetos	11	
Comprobar el funcionamiento del modelo con otros lenguajes	11.1, 12, 13	
Estudiar el comportamiento al eliminar criterios innecesarios	14, 15	
Estudiar el comportamiento al mejorar un criterio	16	
Interpretar los valores de calidad	17	√
Comprobar el impacto de los resultados en el programador	18	
Verificar el funcionamiento del método de aplicación	19	
Aplicar el modelo a un proyecto de tamaño grande	20, 21	
Comprobar la mejora del proceso *software*	16, 19, 20	
Interpretar detalladamente las medidas de un criterio	22	
Analizar los resultados del modelo de calidad	23, 24, 25	

Tabla 5.1: Diseño experimental

El significado de las distintas columnas es el siguiente:

- Objetivo: indica qué se quiere probar en concreto, cuál es el objetivo principal.
- Experimento: enumera los códigos de los experimentos en los que se prueba el objetivo; estos experimentos se resumirán en los siguientes subapartados.
- Experto: indica si en los experimentos se ha contado con la opinión de expertos[1] en orientación a objetos.

5.2. EXPERIMENTOS

Seguidamente se enumeran los distintos experimentos realizados. Para cada uno de ellos, se comentará su planteamiento (objetivos), su diseño (descripción del procedimiento seguido) y la interpretación (análisis de resultados). Se dejará para el Anexo I (en aras de la brevedad del presente capítulo) la descripción del sistema empleado, sus características, los datos concretos y los resultados particulares obtenidos en cada experimento. Pero antes se expondrá un breve resumen de los sistemas empleados para la realización de los experimentos:

- "Agenda Personal". Es una agenda personal sencilla con datos acerca de personas y de citas. Se permiten distintas operaciones de mantenimiento y consulta. Está desarrollada en C++.
- "Caballo de Ajedrez". Este sistema resuelve el problema del salto del caballo de ajedrez. Se ha desarrollado en C y en C++.
- "Cajero Automático". Implementa en C++ las funciones típicas de un cajero automático de un banco.
- "Creador de Cursos". Este sistema constituye un módulo de una herramienta de autor que permite recoger la información necesaria para elaborar un curso. Está desarrollado en Object Pascal.
- "Curso de Idiomas". Este sistema consiste en el desarrollo de un curso de idiomas para ciegos. Está realizado en C++.
- "Desarrollo Gráfico". Es una aplicación para realizar el desarrollo de sistemas orientados a objetos de forma gráfica, generando código Java. Está desarrollada en Java.
- "Juego de Dados". Es el juego de dados *Greed* desarrollado en C++.
- "Máquina de Café". Es un sistema que simula el comportamiento del *software* de una máquina automática de café, que proporciona distintos tipos de café y cobra el importe adecuado. Está desarrollado en C++.
- "Productor-Consumidor". Este sistema resuelve el problema del productor-consumidor utilizando una cola de prioridades. Está implementado en C++.
- "Simulador de Gases". Este programa en C++ simula el movimiento de un número reducido de partículas de un gas dentro de un recinto cerrado.
- "Sistema Retributivo". Consiste en un sistema que prepara las retribuciones de los distintos tipos de empleados de una empresa. Se ha desarrollado en C++.

[1] Los expertos que han colaborado en los experimentos y demás encuestas mencionadas en este trabajo provienen de distintas universidades, empresas y centros de investigación, como: Aena, CETTICO, Colt Telecom, Dinamic Multimedia, Lucent Technologies, ONCE, Prisa, Telefónica I+D, Universidad Alfonso X el Sabio, Universidad Carlos III de Madrid, Universidad Politécnica de Madrid, etc.

5.2.1. EXPERIMENTO 1: VALIDACIÓN DE LAS MEDIDAS

El principal objetivo del presente experimento consiste en realizar una validación de las medidas del modelo de calidad. Para ello, se ha dividido en tres experimentos, para las fases de análisis, diseño e implementación.

Para la realización del experimento se escogió un programa de tamaño relativamente pequeño (la "Agenda Personal"), para que fuera fácilmente manejable por personas.

5.2.1.1. Experimento 1.1

• Planteamiento: El principal objetivo del presente experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de análisis.

• Diseño: El procedimiento seguido fue solicitar a dos expertos en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas del análisis (pero sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el análisis del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: La aplicación del modelo sobre el análisis del sistema proporcionó una calidad de 0,59. Para el criterio auto-descriptivo se estudiaron las tres medidas pertenecientes a la fase de análisis, comparándolas con la opinión de los dos expertos por medio del test de Kolmogorov-Smirnov [Chakravart, 67]. Para ello, se agregó, utilizando una media aritmética, la información facilitada por los expertos, con el fin de construir la figura de evaluador tipo. El resultado del test permite admitir la hipótesis de que ambas muestras provienen de la misma distribución, si bien, el nivel de significación obtenido (0,10) indica cierto componente subjetivo en los datos.

5.2.1.2. Experimento 1.2

• Planteamiento: El objetivo de este experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de diseño.

• Diseño: El procedimiento fue solicitar a dos expertos en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas del diseño (sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el diseño del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: La aplicación del modelo sobre el diseño del sistema reflejó un valor de calidad de 0,67. Para el criterio auto-descriptivo se estudiaron las medidas de diseño, comparándolas con la opinión de los expertos con el test de Kolmogorov-Smirnov. Agregando con una media aritmética los datos de los expertos, para construir el evaluador tipo, el test permite afirmar que las dos muestras siguen la misma distribución.

5.2.1.3. Experimento 1.3

• Planteamiento: El principal objetivo del experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de implementación.

• Diseño: El procedimiento seguido consistió en solicitar a dos expertos en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas de la implementación (pero sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el código fuente del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: El modelo puso de manifiesto una calidad de 0,73. Seguidamente, se estudiaron las medidas aplicables del criterio auto-descriptivo, realizando una comparación con la opinión de los dos expertos mediante el test de Kolmogorov-Smirnov. Con este fin, se agregó, mediante la media aritmética, la información facilitada por los expertos para lograr un evaluador tipo. El resultado del test permite no rechazar la hipótesis de que ambas muestras provienen de la misma distribución, si bien, el relativamente bajo nivel de significación obtenido (0,07) indica un importante componente subjetivo en los datos.

5.2.2. EXPERIMENTO 2: VALIDACIÓN DE LAS MEDIDAS

El principal objetivo del presente experimento consiste en realizar una validación de las medidas del modelo de calidad. Para ello, se ha dividido en tres experimentos, uno para el análisis, otro para el diseño y, el tercero, para la implementación.

Este experimento se ha realizado sobre el sistema del "Cajero Automático" realizado en C++. Al utilizarse un sistema con necesidades diferentes al del experimento anterior, se emplearán ciertas medidas que en el otro problema no habían sido consideradas.

5.2.2.1. Experimento 2.1

• Planteamiento: El principal objetivo del presente experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de análisis.

• Diseño: El procedimiento seguido fue solicitar a dos expertos en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas del análisis (sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el análisis del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: El test de Kolmogorov-Smirnov [Chadravart, 67] permite comparar poblaciones sin especificar la distribución y se ha usado para realizar la comparación entre los resultados de las medidas y la opinión de los expertos. En términos generales, el nivel de significación ha arrojado buenos valores para las medidas del análisis. De esto se puede deducir que las medidas pueden predecir la opinión de los expertos.

5.2.2.2. Experimento 2.2

• Planteamiento: El objetivo de este experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de diseño.

• Diseño: El procedimiento fue solicitar a dos especialistas en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas del diseño (sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el diseño del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: El test de Kolmogorov-Smirnov, utilizado para comparar los resultados de las medidas y la opinión de los expertos, ha proporcionado buenos valores para las medidas del diseño, de lo que se deduce que las medidas del modelo se pueden considerar como un buen predictor de la opinión de los expertos.

5.2.2.3. Experimento 2.3

• Planteamiento: El principal objetivo del experimento consiste en validar cada una de las medidas del modelo de calidad para la fase de implementación.

• Diseño: El procedimiento seguido consistió en solicitar a dos expertos en orientación a objetos su valoración (de 0 a 10) sobre cada una de las medidas de la implementación (pero sin proporcionarles las fórmulas). Seguidamente se aplicó el modelo de calidad sobre el código fuente del sistema obteniendo los valores de las medidas. Finalmente, se compararon los dos tipos de valores.

• Interpretación: El test de Kolmogorov-Smirnov, al comparar las medidas y la opinión de los expertos, permite afirmar que la valoración por ambos métodos produce resultados similares que, por tanto, pueden utilizarse indistintamente para evaluar una implementación.

5.2.3. EXPERIMENTO 3: VALIDACIÓN DE LOS CRITERIOS

El objetivo primordial de este experimento es validar los distintos criterios del modelo de calidad. Para facilitar la labor, se ha dividido en tres experimentos, correspondientes a las fases de análisis, diseño e implementación.

El sistema empleado en este tercer experimento fue el problema del "Cajero Automático".

5.2.3.1. Experimento 3.1

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los distintos criterios durante la fase de análisis.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para el análisis del "Cajero Automático". Seguidamente se aplicó el modelo de calidad sobre el análisis y se compararon los datos obtenidos.

• Interpretación: La aplicación del modelo de calidad proporcionó el vector de valores correspondientes a los distintos criterios. Analizando dicho vector, se observa que el análisis no es perfecto, al no haber alcanzado valores suficientes en los criterios concisión, entrenamiento, generalidad y operatividad.

El resultado de la comparación entre los datos del modelo y de los expertos, con la utilización del test de Kolmogorov-Smirnov, ha permitido comprobar que ambos datos proceden de la misma distribución, lo que indica que no hay grandes diferencias entre la opinión de los expertos y la información proporcionada por el modelo.

5.2.3.2. Experimento 3.2

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los distintos criterios durante la fase de diseño.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para el diseño del "Cajero Automático". Seguidamente, se aplicó el modelo de calidad sobre el diseño obtenido y se compararon los datos.

• Interpretación: Analizando el vector de valores de los criterios, se descubre que el diseño tiene defectos, al no haber alcanzado valores suficientes en los criterios concisión, documentación, entrenamiento y operatividad, aunque han mejorado algo desde el análisis.

La comparación entre los datos del modelo y la de los expertos, con el test de Kolmogorov-Smirnov, permite afirmar que ambos datos pueden proceder de la misma distribución, lo que indica que solo hay ligeras diferencias entre la opinión de los expertos y la información del modelo.

5.2.3.3. Experimento 3.3

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los distintos criterios tras haber finalizado la implementación.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para la implementación del "Cajero Automático". Seguidamente se aplicó el modelo de calidad sobre el código fuente y se compararon los datos obtenidos.

• Interpretación: Tras aplicar el modelo de calidad, donde fueron aplicables el 92% de las medidas, y observar los criterios, se ve cómo la mayoría tienen valores adecuados. Los más bajos son la concisión y el entrenamiento, habiendo aumentado los de la documentación y la operatividad gracias a la información proporcionada en la fase previa por el modelo de calidad.

El test de Kolmogorov-Smirnov entre el modelo y los expertos indica que las muestras proceden, indubitablemente, de la misma distribución.

Por tanto, estos resultados permiten afirmar que el cálculo de los valores de los criterios se asemeja al cálculo que, subjetivamente, realiza una persona, demostrando, de esta manera, la utilidad del modelo de calidad.

5.2.4. EXPERIMENTO 4: VALIDACIÓN DE LOS CRITERIOS

El principal objetivo del experimento consiste en validar los criterios del modelo de calidad. Para facilitar el estudio, se ha dividido en tres experimentos, para las fases de análisis, diseño e implementación.

El sistema empleado ha sido la herramienta de "Desarrollo Gráfico", sistema desarrollado en Java con un tamaño bastante considerable.

5.2.4.1. Experimento 4.1

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los distintos criterios durante la fase de análisis.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para la fase de análisis correspondientes al sistema de diseño en Java. Seguidamente se aplicó el modelo de calidad sobre el análisis y se compararon los datos obtenidos.

• Interpretación: En primer lugar, el valor de calidad alcanzado fue 0,48, indicando que no es un análisis malo. Los aspectos a mejorar pueden extraerse fundamentalmente del vector de criterios: datos estándar, documentación, instrumentación, operatividad y tolerancia a errores. Al comparar este vector de criterios con los obtenidos fruto de la opinión de los dos expertos mediante el test de Kolmogorov-Smirnov se obtuvo como conclusión que ambas poblaciones provenían de la misma distribución, demostrando así la conformidad de los datos obtenidos automáticamente con la opinión de los expertos.

5.2.4.2. Experimento 4.2

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los criterios durante la fase de diseño.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para la fase de diseño. Seguidamente se aplicó el modelo de calidad sobre el diseño y se compararon los resultados obtenidos.

• Interpretación: La calidad del diseño fue 0,54, suponiendo una mejoría, al igual que ocurre con la mayoría de los criterios aunque algunos han empeorado, debido fundamentalmente a las nuevas medidas que se han podido aplicar en el diseño. Los criterios peores fueron tolerancia a errores, instrumentación, eficiencia de ejecución... Al comparar los criterios con la opinión de los expertos por medio del test de Kolmogorov-Smirnov se concluye que ambas poblaciones podían provenir de la misma distribución, lo que demuestra así la adecuación del modelo a la realidad.

5.2.4.3. Experimento 4.3

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los distintos criterios tras la fase de implementación.

• Diseño: Se solicitó a 2 expertos una valoración de los distintos criterios para la implementación. Seguidamente, se aplicó el modelo de calidad sobre el código fuente en Java obtenido y se compararon los datos.

• Interpretación: La calidad de la implementación fue 0,71, destacando el alto valor del factor transportabilidad, poniendo de manifiesto que el sistema está siendo implementado en Java, intentando que sea independiente de la plataforma. El criterio entrenamiento es el único que ha obtenido un valor bajo. El test de Kolmogorov-Smirnov puso de manifiesto que los datos del modelo se asemejan a la opinión que tienen los expertos del sistema.

5.2.5. EXPERIMENTO 5: VALIDACIÓN DE LOS FACTORES Y DE LOS PESOS EN LOS CRITERIOS

El objetivo de este experimento es doble: el primero estriba en realizar una validación de los resultados obtenidos para los factores del modelo de calidad, mientras que el segundo pretende verificar cómo varían los resultados al utilizar pesos en los criterios. Para hacer más sencillo el trabajo, se ha subdividido en tres experimentos, para las tres fases del ciclo de vida.

El sistema utilizado para la validación ha sido la "Máquina de Café".

5.2.5.1. Experimento 5.1

• Planteamiento: La meta es validar los valores de calidad proporcionados por los factores de calidad durante la fase de análisis. Así mismo, también se quiere comprobar si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se encuestó a 3 expertos para obtener los pesos para los criterios dentro de cada factor, teniendo en cuenta que el sistema a evaluar representaba el funcionamiento de una máquina de café. A continuación, se les solicitó una valoración de los factores para la fase de análisis. Seguidamente se aplicó el modelo de calidad sobre el análisis y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: El vector de valores obtenido para los factores del análisis muestra buenos resultados salvo para la utilizabilidad. Para comparar los datos del modelo con la opinión de expertos en los factores y en la calidad se realizó el test Kolmogorov-Smirnov que determinó que ambas poblaciones provenían de la misma distribución, con un nivel de significación de 0,70.

Repitiendo el test sin tener en cuenta los pesos en el cálculo de los factores, se obtuvo un nivel de significación de 0,12. Este valor refleja que, aunque sus distribuciones son la misma, puede existir cierto componente subjetivo en los datos.

De estos resultados se deduce que la presencia de los pesos ayuda a acercar los valores del modelo a la opinión de los expertos en orientación a objetos.

5.2.5.2. Experimento 5.2

• Planteamiento: El objetivo es validar los valores de calidad proporcionados por los factores de calidad durante la fase de diseño. Así mismo, también se quiere comprobar si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se encuestó a 3 expertos para obtener los pesos para los criterios dentro de cada factor, para la "Máquina de Café". A continuación, se les solicitó una valoración de los factores para la fase de diseño. Seguidamente se aplicó el modelo de calidad sobre el diseño y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: Los factores conservan los buenos valores, siendo también la utilizabilidad el más bajo de ellos, aunque con una tendencia alcista. El test de Kolmogorov-Smirnov determinó que ambas poblaciones provenían de la misma distribución, siendo el nivel de significación 0,40.

Sin utilizar pesos para calcular los factores, el nivel de significación obtenido fue 0,05. Este resultado refleja que, aunque se puede afirmar que ambas poblaciones provienen de la misma distribución, existe subjetividad en los datos.

De nuevo, puede comprobarse que los pesos permiten obtener una representación más real del mundo evaluado por el modelo de calidad.

5.2.5.3. Experimento 5.3

• Planteamiento: El objetivo es validar los valores de calidad obtenidos por los factores de calidad tras la fase de implementación. Así mismo, también se quiere comprobar si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se encuestó a 3 expertos para obtener los pesos para los criterios dentro de cada factor. A continuación, se les solicitó una valoración de los factores para la fase de implementación. Seguidamente se aplicó el modelo de calidad sobre el código fuente en C++ y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: Tras la implementación, todos los factores obtuvieron unos buenos valores. El nivel de significación del test de Kolmogorov-Smirnov fue 0,76 (tanto cuando se usan pesos en el modelo como cuando no se usan), lo que indica que las poblaciones siguen la misma distribución.

5.2.6. EXPERIMENTO 6: VALIDACIÓN DE LOS FACTORES Y DE LOS PESOS EN LOS CRITERIOS

El objetivo de este experimento es también doble: realizar una validación de los resultados obtenidos para los factores del modelo de calidad y comprobar cómo varían los resultados al utilizar pesos en los criterios. Para hacer más sencillo el trabajo, se ha subdividido en tres experimentos, para las fases de análisis, diseño e implementación.

Para estos experimentos se ha utilizado como banco de pruebas el sistema del "Cajero Automático".

5.2.6.1. Experimento 6.1

• Planteamiento: El fin del experimento consiste en validar los valores de calidad proporcionados por los factores durante la fase de análisis. Así mismo, también se quiere comprobar si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se encuestó a 3 expertos para obtener los pesos para los criterios dentro de cada factor, teniendo en cuenta que el sistema a evaluar representaba el funcionamiento de un cajero automático. A continuación, se les solicitó una valoración de los factores para la fase de análisis. Seguidamente se aplicó el modelo de calidad sobre el análisis y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: Los factores del análisis presentan buenos valores, salvo la eficiencia. Si se comparan estos datos con los estimados por los expertos, se obtiene como conclusión que ambas poblaciones provienen de la misma distribución, por lo que puede afirmarse que los factores calculados según el modelo de calidad proporcionan información real acerca de su calidad. La realización del test sin incluir los pesos en el cálculo de los factores, habría arrojado la misma conclusión, con el mismo nivel de significación.

5.2.6.2. Experimento 6.2

• Planteamiento: Los objetivos son validar los valores de calidad proporcionados por los factores de calidad durante la fase de diseño y comprobar si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se realizó una encuesta a 3 expertos con el fin de obtener los pesos para los criterios dentro de cada factor, para el problema del "Cajero Automático". A continuación, se les solicitó una valoración de los factores para la fase de diseño. Seguidamente se aplicó el modelo de calidad sobre el diseño y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: Todos los factores en el diseño presentan buenos valores, en especial la integridad. Al comparar estos datos con los estimados por los expertos con el test de Kolmogorov-Smirnov, se obtiene que las poblaciones proceden de la misma distribución. Si al realizar el test, no se hubieran tenido en cuenta los pesos para el cálculo de los factores, habría arrojado el mismo resultado, aunque con un nivel de significación menor, lo que refleja que los pesos ayudan a acercar los datos del modelo a la realidad.

5.2.6.3. Experimento 6.3

• Planteamiento: El objetivo es validar los valores de calidad proporcionados por los factores de calidad tras la fase de implementación al tiempo que se comprueba si la utilización de pesos asociados a los criterios mejora los resultados obtenidos.

• Diseño: Se encuestó a tres expertos para obtener los pesos para los criterios dentro de cada factor. A continuación, se les solicitó una valoración de los factores para la fase de implementación. Seguidamente se aplicó el modelo de calidad sobre el código fuente en C++ y se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los criterios dentro de cada factor.

• Interpretación: La fase de implementación finalizó con unos resultados muy buenos para todos los factores. El test de Kolmogorov-Smirnov permite deducir que los factores obtenidos con el modelo de calidad pueden predecir la opinión de los expertos.

5.2.7. EXPERIMENTO 7: VALIDACIÓN DE LA CALIDAD Y DE LOS PESOS EN LOS FACTORES

El doble objetivo del experimento consiste en realizar una validación de los resultados obtenidos para la calidad y comprobar cómo varían los resultados al utilizar pesos en los factores. Para hacer más sencillo el trabajo, se ha subdividido en tres experimentos, para las fases de análisis, diseño e implementación.

El sistema empleado para este experimento ha sido la "Máquina de Café".

5.2.7.1. Experimento 7.1

• Planteamiento: El fin principal del experimento es validar la calidad global durante la fase de análisis. Además, se quiere comprobar si el uso de pesos mejora los resultados.

• Diseño: Se encuestó a tres expertos para obtener los pesos de los factores en la calidad, teniendo en cuenta que el sistema a evaluar representaba el funcionamiento de una "Máquina de Café". A continuación, se les solicitó una valoración global de la calidad para la fase de análisis. Seguidamente se aplicó el modelo de calidad sobre el análisis.

• Interpretación: La calidad global del sistema tras la fase de análisis resultó ser bastante buena (0,63). Sin tener en cuenta los pesos, el valor obtenido habría sido 0,56. El valor obtenido al extraer la media aritmética de la opinión de los expertos (0,66) se aproxima más al valor logrado utilizando los pesos.

5.2.7.2. Experimento 7.2

• Planteamiento: El objetivo del experimento es validar la calidad global durante la fase de diseño. Además, se quiere comprobar si el uso de pesos en los factores mejora los resultados.

• Diseño: Mediante una encuesta a tres expertos, se obtuvieron los pesos de los factores en la calidad, para la "Máquina de Café". A continuación, se les solicitó una valoración global de la calidad para la fase de diseño. Seguidamente se aplicó el modelo de calidad sobre el diseño.

• Interpretación: La calidad global del sistema tras finalizar el diseño fue bastante buena (0,64). Sin considerar los pesos, el valor obtenido habría sido 0,61. El valor alcanzado al calcular la media aritmética de la opinión de los expertos (0,70) se aproxima algo más al logrado utilizando los pesos.

5.2.7.3. Experimento 7.3

• Planteamiento: El objetivo principal del experimento es validar la calidad global tras la fase de implementación. Además, se quiere comprobar si el uso de pesos mejora los resultados.

• Diseño: Se encuestó a tres expertos para obtener los pesos de los factores en la calidad, teniendo en cuenta el sistema a evaluar ("Máquina de Café"). Posteriormente, se les solicitó una valoración global de la calidad para la fase de implementación. A continuación, se aplicó el modelo sobre el código fuente en C++. Seguidamente, se comparó el valor de calidad obtenido en las tres fases con el proporcionado por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los factores.

• Interpretación: La calidad final lograda por el sistema fue excelente (0,79). Sin considerar los pesos, el valor obtenido habría sido 0,74. El valor alcanzado al calcular la media aritmética de la opinión de los expertos (0,73) se aproxima algo más al logrado sin utilizar los pesos, siendo ésta la única ocasión donde se ha producido, entre todos los experimentos.

Para comparar los datos de calidad del modelo con la opinión de los expertos, se realizó un estudio mediante el test Kolmogorov-Smirnov. El nivel de significación obtenido para la calidad del sistema (tanto usando pesos como sin usarlos) fue 0,52, lo cual indica que las poblaciones siguen la misma distribución.

5.2.8. EXPERIMENTO 8: VALIDACIÓN DE LA CALIDAD Y DE LOS PESOS EN LOS FACTORES

Los objetivos del experimento consisten en realizar una validación de los resultados obtenidos para la calidad y comprobar cómo varían los resultados al utilizar pesos en los factores. Para facilitar el estudio, se ha subdividido en tres experimentos, correspondientes a las fases de análisis, diseño e implementación.

El sistema empleado en esta ocasión ha sido el "Cajero Automático".

5.2.8.1. Experimento 8.1

• Planteamiento: El fin del experimento es validar la calidad global durante la fase de análisis. Además, se quiere comprobar si el uso de pesos mejora los resultados.

• Diseño: Se encuestó a tres expertos para obtener los pesos de los factores en la calidad, teniendo en cuenta que el sistema a evaluar representaba el funcionamiento de un "Cajero Automático". A continuación, se les solicitó una valoración global de la calidad para la fase de análisis. Seguidamente se aplicó el modelo de calidad sobre el análisis.

• Interpretación: La calidad obtenida en la fase de análisis por el modelo fue 0,69, reflejando un buen valor general. El valor obtenido sin tener en cuenta los pesos habría sido 0,62. La opinión de los expertos (0,7) se aproximó más al valor obtenido con los pesos, denotando, una vez más, que éstos ayudan a reflejar mejor la realidad.

5.2.8.2. Experimento 8.2

• Planteamiento: Los objetivos del experimento son validar la calidad global durante la fase de diseño y comprobar si el uso de pesos en los factores mejora los resultados.

• Diseño: Mediante una encuesta a tres expertos, se obtuvieron los pesos de los factores en la calidad, para el "Cajero Automático". A continuación, se les solicitó una valoración global de la calidad para la fase de análisis. Seguidamente se aplicó el modelo de calidad sobre el análisis.

• Interpretación: La calidad obtenida tras el diseño por el modelo fue 0,73, reflejando un buen valor. El valor obtenido sin tener en cuenta los pesos habría sido 0,66. La opinión de los expertos (0,7) está en el punto intermedio de ambos valores, por lo que se ve que el modelo proporciona valores adecuados.

5.2.8.3. Experimento 8.3

• Planteamiento: El objetivo principal del experimento es validar la calidad global tras la fase de implementación. Además, se quiere comprobar si el uso de pesos mejora los resultados.

• Diseño: Se encuestó a tres expertos para obtener los pesos de los factores en la calidad, teniendo en cuenta el sistema a evaluar ("Cajero Automático"). Posteriormente, se les solicitó una valoración global de la calidad para la fase de implementación. A continuación, se aplicó el modelo de calidad sobre el código fuente en C++. Seguidamente, se comparó el valor de calidad obtenido en las tres fases con el proporcionado por los expertos. Finalmente, se repitió la comparación sin tener en cuenta los pesos de los factores.

• Interpretación: El valor de calidad obtenido en la implementación fue 0,82, reflejando un valor excelente. El valor obtenido sin la intervención de los pesos habría sido 0,77. La opinión de los expertos (0,85) se aproxima a la obtenida con pesos. El test de Kolmogorov-Smirnov para comprobar si la calidad proporcionada por el modelo y por la opinión de los expertos (para las tres fases) provenían de la misma distribución dio positivo, con un alto nivel de significación. Si en el test no se tuvieran en cuenta los pesos, también se obtendría un resultado positivo, aunque con una menor significación. De esto se concluye que los pesos permiten ajustar los valores del modelo a la realidad.

5.2.9. EXPERIMENTO 9: VALIDACIÓN DE LOS PESOS EN LAS MEDIDAS

• Planteamiento: Realizar una asignación directa de pesos a medidas de un criterio y validar los resultados obtenidos al aplicar el modelo de calidad tanto usando como sin usar los pesos.

• Diseño: Sobre la implementación del sistema de la "Agenda Personal", en primer lugar se realizó una asignación de pesos a las medidas de completitud según una encuesta a diez expertos. En segundo lugar, se pidió a los expertos que valoraran la completitud del sistema. Seguidamente, se aplicó el modelo de calidad sobre el código fuente del sistema. En último lugar, se compararon los resultados de los expertos con los del modelo de calidad, primero sin utilizar los pesos y después empleándolos.

• Interpretación: El test de Kolmogorov-Smirnov entre los datos del modelo y la opinión agregada de los expertos permiten afirmar sin lugar a dudas que ambas muestras provienen de la misma distribución, al haberse obtenido un nivel de significación del 0,82.

Por último, la opinión de los expertos acerca de la completitud del sistema arroja un valor que se aproxima más al valor obtenido por el modelo cuando se emplean los pesos de la completitud que cuando no se usan, por lo que se ve que su uso resulta aconsejable.

5.2.10. EXPERIMENTO 10: CONTRASTE DE LA CALIDAD EN VARIOS DESARROLLOS Y VALIDACIÓN DE LOS PESOS EN LOS FACTORES

• Planteamiento: El objetivo consiste en analizar el comportamiento del valor de calidad dado por el modelo en comparación con la opinión de expertos para diversos sistemas que resuelvan el mismo problema. También, se quiere verificar la utilidad de los pesos sobre los factores.

• Diseño: Para el "Sistema Retributivo", en primer lugar, dos profesores definieron los pesos para los factores. Seguidamente, 96 alumnos realizaron sus respectivas implementaciones del sistema. Estas 96 implementaciones fueron evaluadas por los profesores. A continuación, se aplicó el modelo de calidad, en su fase de implementación, sobre el código de todos los programas. Finalmente, se compararon las calificaciones de los profesores con los resultados del modelo de calidad, tanto sin usar los pesos de los factores como utilizándolos.

• Interpretación: La comparación entre los datos del modelo de calidad con los obtenidos de los expertos ha proporcionado unos resultados excelentes, lo que demuestra que los resultados del modelo de calidad pueden utilizarse como un indicador de la calidad real del sistema. El análisis de 96 versiones diferentes del mismo sistema da una idea de la potencialidad del uso del modelo de calidad para predecir la calidad de un desarrollo. Por tanto, puede afirmarse que los valores de calidad obtenidos reflejan lo que realmente pretenden medir. Por ello, los resultados del modelo de calidad resultan útiles para mostrar la calidad de un sistema debido a su alta correlación con la realidad según la opinión de los expertos.

Estos resultados se ven mejorados si se tienen en cuenta los pesos, puesto que en este caso aumenta aún más la buena correlación con los expertos a la hora de valorar un sistema. Este hecho demuestra que la presencia de los pesos permite ajustar más la calidad a la realidad de los sistemas.

5.2.11. EXPERIMENTO 11: CONTRASTE DE LA CALIDAD EN EL PARADIGMA IMPERATIVO Y ORIENTADO A OBJETOS, VALIDACIÓN DE LOS PESOS EN LOS FACTORES Y ADECUACIÓN DEL MODELO A OTROS LENGUAJES

Como puede comprobarse, este experimento tiene un triple objetivo. En primer lugar, se pretende comparar los resultados del modelo de calidad obtenidos al aplicarlos sobre un sistema desarrollado siguiendo el paradigma imperativo y sobre el mismo sistema siguiendo el paradigma orientado a objetos. En segundo lugar, se quiere validar la utilidad de los pesos de los factores dentro de la calidad. Por último, también se quiere estudiar el comportamiento del modelo de calidad sobre lenguajes de programación diferentes al C++. El experimento se ha subdividido en dos partes, una tomando la solución imperativa y la otra con la solución orientada a objetos.

El sistema empleado ha sido el problema del "Caballo de Ajedrez", implementado en C y en C++.

5.2.11.1. Experimento 11.1

• Planteamiento: El objetivo principal es aplicar el modelo de calidad sobre la implementación de un sistema desarrollado bajo el paradigma imperativo, con el fin de comprobar la validez o no del modelo.

• Diseño: Tras definir los pesos de los factores, se aplicó el modelo de calidad sobre la implementación en C del problema del "Caballo de Ajedrez". Seguidamente, se estudiaron los resultados obtenidos.

• Interpretación: Los resultados obtenidos muestran que el sistema tiene una calidad aceptable, con un valor ligeramente superior cuando se emplean los pesos en el cálculo de la calidad. Los factores que mejor resultado han obtenido son la eficiencia, fiabilidad, reutilizabilidad y transportabilidad, mientras que los peores han sido proporcionados por la interoperatividad y la utilizabilidad.

Para la evaluación del modelo de calidad, únicamente se pudieron usar un 39,4% de las medidas, debido a que muchas de ellas están diseñadas para la orientación a objetos, con estructuras y construcciones ausentes en el paradigma imperativo. Pero no por ello, el modelo deja de ser aplicable en el paradigma imperativo, aunque si se desea utilizar así, sería conveniente definir nuevas medidas o adaptar las ya existentes a las características notablemente diferentes de esta otra filosofía de desarrollo de sistemas.

5.2.11.2. Experimento 11.2

• Planteamiento: El objetivo principal es aplicar el modelo de calidad sobre la implementación del mismo sistema que en el Experimento 11.1 (el "Caballo de

Ajedrez"), pero esta vez desarrollado bajo el paradigma orientado a objetos, con el fin de comprobar si la orientación a objetos ha mejorado efectivamente la calidad.

• Diseño: Se aplicó el modelo de calidad sobre la implementación en C++ del problema del "Caballo de Ajedrez". Seguidamente, se estudiaron los resultados obtenidos y se compararon con los obtenidos en el experimento anterior.

• Interpretación: Los resultados muestran que el sistema tiene una calidad aceptable, con un valor algo mayor cuando se emplean los pesos en el cálculo de la calidad, pero, en ambos casos, claramente superior a la obtenida en el Experimento 11.1. En cualquier caso, se ve que la utilización de pesos permite reflejar mejor la calidad del sistema según sus objetivos.

Los criterios que mejor resultado han obtenido son la corrección, fiabilidad, mantenibilidad, reutilizabilidad y transportabilidad, mientras que entre los peores se encuentran la interoperatividad y utilizabilidad (los mismos que en el paradigma imperativo, lo que permite ver que en ambos casos no se puso hincapié en dichas características).

A la vista de los resultados, puede concluirse que los principales defectos que presenta el sistema en C++ vienen ocasionados por la finalidad docente del programa, y a que, dado su pequeño tamaño, no se han empleado muchas de las posibilidades de la orientación a objetos en C++, como el polimorfismo, las clases y funciones genéricas o el manejo de excepciones.

De la comparación de los datos obtenidos en este experimento y en el anterior, destaca que todos los factores, salvo la eficiencia, aumentan al emplear la orientación a objetos. Son especialmente interesantes las mejoras en factores como fácil de probar, flexibilidad, mantenibilidad y reutilizabilidad, características éstas que la orientación a objetos, históricamente, ha favorecido. En lo referente a los criterios, en primer lugar, es digno de destacar la mejora experimentada por auto-descriptivo, estabilidad, modularidad y simplicidad, que demuestra lo que los objetos pueden aportar a un sistema. Por otro lado, hay que mencionar las pérdidas sufridas por criterios como concisión, eficiencia de ejecución y generalidad. La explicación es sencilla: C++ es un lenguaje que, al emplear los componentes de la orientación a objetos, crea programas mayores que los creados en C; C es un lenguaje que favorece la velocidad de ejecución; por último, la generalidad en C aumenta porque las funciones independientes son siempre algo más generales que una clase completa. También hay que destacar que los criterios datos estándar y estructuración no han podido ser calculados para el problema en C, al no emplear ninguna de sus medidas; resulta especialmente lógico el caso de la estructuración, pues es un criterio substancialmente introducido para manejar las estructuras que proporciona la orientación a objetos.

Para la evaluación del modelo de calidad en C++ se emplearon un 84,2% de las medidas totales (es decir, más del doble que las empleadas en C). Esto se debe a que muchas de las medidas están especialmente ideadas para su aplicación sobre lenguajes orientados a objetos, que disponen de una serie de estructuras y construcciones ausentes en lenguajes imperativos como C.

5.2.12. EXPERIMENTO 12: ADECUACIÓN DEL MODELO A OTROS LENGUAJES

• Planteamiento: El objetivo es comprobar el comportamiento del modelo de calidad al ser aplicado sobre otros lenguajes orientados a objetos.

• Diseño: Se ha aplicado el modelo de calidad sobre una librería ("Creador de Cursos") desarrollada en Object Pascal. Después, se estudió el comportamiento de las medidas.

• Interpretación: La aplicación del modelo de calidad sobre código fuente en un lenguaje distinto a C++ ha permitido comprobar que el modelo sigue siendo válido, puesto que los resultados están acordes con la opinión del desarrollador.

Lo único a destacar es el hecho de que cerca de un 32% de las medidas del modelo de calidad no han podido ser utilizadas por diversas razones. En primer lugar, más del 10% eran medidas específicas del lenguaje C++ y medían conceptos no presentes en Object Pascal (como la sobrecarga de métodos, funciones amigas o clases virtuales). En segundo lugar, cerca de un 21% no pudieron ser evaluadas porque no se tenían datos precisos sobre la utilización del módulo en el seno del sistema completo o bien no resultaban aplicables debido a la naturaleza del sistema. No obstante, ha podido comprobarse que la mayor parte de las medidas han podido ser utilizadas de forma correcta al estar diseñadas para la orientación a objetos en general y no para un lenguaje en concreto.

5.2.13. EXPERIMENTO 13: ADECUACIÓN DEL MODELO A OTROS LENGUAJES

• Planteamiento: El objetivo es comprobar el comportamiento del modelo de calidad al ser aplicado sobre otros lenguajes orientados a objetos.

• Diseño: Se ha aplicado el modelo de calidad sobre el sistema de "Desarrollo Gráfico" implementado en Java. Seguidamente, se ha estudiado el comportamiento de las medidas.

• Interpretación: La utilización del modelo sobre código fuente en un lenguaje como Java ha permitido contrastar su validez. Evidentemente, durante la fase de análisis y diseño no se encuentran diferencias entre este experimento y el resto. Es en la fase de implementación donde surgen las diferencias, basadas en el hecho de que cerca de un 25% de las medidas de implementación no se han podido utilizar, debido a que medían características específicas del lenguaje C++ no presentes en Java o a que no eran aplicables en este problema concreto. En cualquier caso, la mayoría de las medidas se han podido usar de la forma habitual al estar diseñadas para el paradigma orientado a objetos y no para un lenguaje en concreto.

5.2.14. EXPERIMENTO 14: COMPORTAMIENTO AL ELIMINAR CRITERIOS INNECESARIOS Y VALIDACIÓN DE LOS PESOS EN LOS FACTORES

• Planteamiento: En ocasiones puede ocurrir que en un sistema no importe alguno de los criterios. Para ello, se estudiará el comportamiento de los valores del modelo de calidad cuando se anulen dichos criterios. También se estudiará el comportamiento de la calidad con la utilización de los pesos en los factores.

• Diseño: Primero se calcularon los pesos de los factores, para, después, aplicar el modelo de calidad sobre la implementación en C++ de un juego ("Juego de Dados") que servía como ejercicio de programación. Seguidamente, se anularon aquellos criterios que no se plantearon como objetivos en el enunciado del problema y se estudiaron los nuevos resultados alcanzados por el árbol de calidad.

• Interpretación: Para obtener los pesos de los factores se ha tenido en cuenta la naturaleza y objetivos del programa. Analizando los pesos obtenidos, puede verse que el factor que se ha considerado más importante es la corrección (lo más importante es que el programa esté correcto) seguido de transportabilidad (interesa que el programa pueda servir para cualquier entorno: se enseña C++ en general, no para una arquitectura concreta), mientras que los menos significativos han resultado ser interoperatividad (no hay necesidades de comunicación con otros sistemas) y utilizabilidad (no es un programa para usarlo, sino para aprender a construirlo).

Aplicando las medidas aplicables, aproximadamente un 86% del total, se ha obtenido un valor de calidad global de 0,44, lo cual indica que el programa tiene una calidad regular. Solamente tres factores logran superar el valor medio, donde destaca el valor de la fiabilidad. En la parte baja de la calidad, se encuentran los factores interoperatividad y utilizabilidad.

Seguidamente se estudiaron los principales objetivos que perseguía el desarrollo del sistema, y se eliminaron del modelo aquéllos que no se consideraron prioritarios durante su construcción o, simplemente, no resultaron aplicables (auditoría de accesos, control de acceso, comunicaciones estándar, eficiencia de almacenamiento, eficiencia de ejecución, entrenamiento, instrumentación, operatividad y seguimiento).

Una vez evaluada de nuevo la calidad tras la eliminación de estos criterios, se obtuvo un valor de 0,46, con lo que puede verse cómo los criterios eliminados por considerarse inútiles para el problema, tienen una ligera influencia a la baja en el resultado general de calidad. Por ello, una política de uso del modelo de calidad que simplificaría su utilización podría consistir en prescindir de aquellos criterios superfluos.

5.2.15. EXPERIMENTO 15: COMPORTAMIENTO AL ELIMINAR CRITERIOS INNECESARIOS

• Planteamiento: En ocasiones puede ocurrir que en un sistema no importe alguno de los criterios. Para ello, se estudiará el comportamiento de los valores del modelo de calidad cuando se anulen dichos criterios.

• Diseño: Primero se calcularon los pesos de los factores, para, después, aplicar el modelo de calidad sobre la implementación del problema del "Caballo de Ajedrez" implementado en C++. Seguidamente, se anularon aquellos criterios que no se plantearon como objetivos en el enunciado del problema y se estudiaron los nuevos resultados alcanzados por el árbol de calidad.

• Interpretación: Analizando cuidadosamente los objetivos principales perseguidos en el desarrollo de esta solución (eminentemente didácticos), se puede considerar que los siguientes criterios no fueron tomados como prioritarios durante la construcción del sistema o, simplemente, no resultaban aplicables (auditoría de accesos, control de

acceso, comunicaciones estándar, comunicatividad, entrenamiento, instrumentación, operatividad, seguimiento y tolerancia a errores).

Se consideró que el resto de los criterios eran necesarios para el problema, considerando que aportan calidad en general a un programa orientado a objetos. Hay que puntualizar que, además, también se eliminaron algunas medidas que, a pesar de estar en uno de los criterios admitidos para este problema, medían aspectos a eliminar por no considerarse relevantes o adecuados al sistema.

Además, el factor utilizabilidad pierde también su sentido para el problema, al quedarse únicamente con la documentación técnica, por lo que también se eliminó.

Evaluando de esta manera el modelo de calidad, los factores experimentaron una leve mejoría al ser eliminados de sus cálculos algunas medidas y criterios irrelevantes para el objetivo del problema.

El valor de calidad total así calculado aumenta, cuando se utilizan pesos, del 0,66 hasta un 0,69. Si en los cálculos no se utilizan pesos, el valor de calidad pasa de 0,61 a 0,67. Este supone que los criterios eliminados, tienen una influencia pequeña en el resultado general de calidad. Cuando, además, se conjuga con el uso de los pesos, puede verse que la influencia es menor aún. La explicación de este comportamiento estriba en que al dar los pesos ya se ha dado una menor importancia a aquellos elementos poco significativos, por lo que su eliminación apenas influye en el comportamiento de la calidad.

De esta información puede concluirse que el uso del modelo de calidad se simplificaría prescindiendo de aquellos criterios superfluos.

5.2.16. EXPERIMENTO 16: COMPORTAMIENTO AL MEJORAR UN CRITERIO Y COMPROBACIÓN DE LA MEJORA DEL PROCESO *SOFTWARE*

• Planteamiento: La finalidad principal del presente experimento consiste en comprobar el comportamiento de la calidad de un sistema al aplicar el método de mejora del proceso *software*. El experimento se aplicará aisladamente, de forma que se modifique el sistema sólo por la información proporcionada por uno de los criterios.

• Diseño: Una persona implementará un programa (el problema del "Productor-Consumidor"). Seguidamente, se aplicará el modelo de calidad. Se seleccionará el factor con el valor peor y, de entre los criterios que formen dicho factor, se seleccionará el que tenga un valor más bajo. Al desarrollador se le proporcionarán únicamente las recomendaciones de las medidas de dicho criterio, con el fin de que mejore su programa. Una vez finalizado, se volverá a aplicar el modelo de calidad y se estudiarán los nuevos valores obtenidos.

• Interpretación: Tras analizar los resultados de la primera implementación (con un valor de calidad de 0,51), se observó que el factor que presentó el valor más bajo fue la mantenibilidad (0,36). De sus criterios, era de destacar el valor considerablemente reducido obtenido por la modularidad (0,22). Por tanto, se comunicó al programador las recomendaciones proporcionadas por las distintas medidas para que solucionara los

posibles problemas. Una vez realizados los cambios propuestos por las medidas de este criterio, se volvió a evaluar la calidad, obteniéndose ahora un valor de 0,76 para la modularidad. Por consiguiente, no solo el valor del factor mantenibilidad mejoró hasta alcanzar un valor de 0,44 en la nueva implementación, sino que otros factores también se vieron afectados por la mejoría, obteniéndose un valor de calidad global de 0,56. De esta forma puede verse la enorme utilidad que pueden proporcionar las recomendaciones dadas por las medidas, de tal manera que, si se hacen caso, puede conducir a un incremento notable de la calidad del sistema (en este experimento, una mejora del 52% en uno de los criterios ha supuesto un 5% de mejora en la calidad).

5.2.17. EXPERIMENTO 17: INTERPRETACIÓN DE LOS VALORES DE CALIDAD

• Planteamiento: El objetivo consiste en determinar el significado de los distintos de valores de calidad. Como objetivo secundario estaba realizar un pequeño estudio comparativo entre los datos humanos y los del modelo.

• Diseño: Para un mismo sistema ("Simulador de Gases"), se han realizado 33 diseños e implementaciones con distintos niveles de calidad evaluados por expertos. Después se les aplicó el modelo de calidad con el fin de identificar los rangos de valores de calidad obtenidos con la calidad real del sistema. Adicionalmente, se compararon los resultados de los expertos con los del modelo.

• Interpretación: Tras aplicar el modelo de calidad a los 33 diseños e implementaciones y obtener la evaluación de dos expertos sobre dichos sistemas, se buscó la separación entre los conceptos de calidad mala, normal y buena. Para ello se estudiaron los datos y se fijaron los umbrales (tal como se indica en el apartado I.11.1). Tomando valores redondos por comodidad, puede establecerse en 0,75 el umbral a partir del cual se valorará que un sistema presenta una buena calidad, mientras que el umbral por debajo del cual se considerará una calidad mala se fija en un tercio. Los valores entre ambos serán, entonces, aquéllos que presentan un valor de calidad media. El resto de los experimentos ha permitido contrastar estos umbrales, comprobando que resultan adecuados como punto de referencia.

Para ver la relación existente entre los datos proporcionados por el modelo y la opinión de los profesores, se realizaron unas regresiones lineales que ofrecieron unos datos que reflejan una gran similitud entre la opinión de los expertos y los resultados proporcionados por el modelo. En el diseño se obtuvo una similitud mayor que puede explicarse porque los expertos humanos pueden valorar mejor la calidad de un diseño que la calidad del código fuente de un programa, por ser éste siempre más extenso y requerir más tiempo y atención para su estudio.

5.2.18. EXPERIMENTO 18: COMPROBACIÓN DEL IMPACTO DE LOS RESULTADOS DEL MODELO EN EL PROGRAMADOR

• Planteamiento: Uno de los objetivos del modelo de calidad consiste en que pueda utilizarse como herramienta para la mejora del proceso *software*. Con el presente experimento se pretende comprobar si las recomendaciones que proporciona satisfacen al desarrollador.

• Diseño: Un programador desarrolló un módulo de un sistema ("Creador de Cursos"). A continuación, se aplicó el modelo de calidad sobre el código obtenido. Las conclusiones fueron transmitidas al programador para que diera su opinión al respecto.

• Interpretación: Los resultados del modelo de calidad indican una alta calidad del desarrollo, lo que refleja una buena implementación y un adecuado conocimiento de la tecnología de la orientación a objetos por parte del programador. Los factores que mejor resultado han dado son la corrección y la reutilizabilidad, mientras que el peor ha sido la utilizabilidad.

En este experimento, fue el propio programador el que evaluó las medidas. Como principal conclusión obtenida del análisis de su módulo, comentó que la evaluación de las medidas le sirvió como una revisión profunda del código que le permitió corregir ciertos errores provocados por distracciones, puesto que al aplicar las medidas se podían detectar algunos de estos errores durante el propio proceso de evaluación, con lo que dicho proceso permite obtener un *software* de mayor calidad sin esperar a analizar los resultados [Fuente, 99]. De esta forma se ve cómo la información y recomendaciones proporcionadas por el modelo de calidad puede utilizarse como un mecanismo de mejora del proceso *software* personal.

5.2.19. EXPERIMENTO 19: VERIFICACIÓN DEL FUNCIONAMIENTO DEL MÉTODO Y COMPROBACIÓN DE LA MEJORA DEL PROCESO *SOFTWARE*

Los objetivos de este experimento consisten en verificar las diferencias en el desarrollo de un sistema, dependiendo de la utilización del método de mejora del proceso *software*. El experimento se ha dividido en dos: el primero en el que no se aplica el método de mejora del proceso *software* y el segundo en el que sí se aplica.

El sistema utilizado en estos experimentos es la simulación de la "Máquina de Café".

5.2.19.1. Experimento 19.1

• Planteamiento: El objetivo es estudiar la evolución de la calidad durante el ciclo de vida de un sistema sin informar al programador de las sugerencias ofrecidas por el modelo de calidad.

• Diseño: Se desarrolló un emulador de una "Máquina de Café" por un experto, siendo uno de sus objetivos obtener un producto con la calidad suficiente para utilizarlo en un curso de programación orientada a objetos. Conforme finalizaban las fases del ciclo de vida, se aplicaba el modelo de calidad, pero no se ofrecían al desarrollador las recomendaciones proporcionadas.

• Interpretación: Con el fin de que los resultados del presente experimento y los del siguiente fueran comparables, se partió de una versión del análisis consensuada por ambos desarrolladores, con una calidad de 0,44. El diseño obtenido seguidamente alcanzó un valor de 0,54 para llegar a un valor de calidad global de 0,60 para la implementación.

A pesar de no haber recibido información del modelo de calidad, puede verse que la calidad ha experimentado una leve mejoría (de un 16%), al igual que ocurre con los factores y con los criterios, en general.

5.2.19.2. Experimento 19.2

• Planteamiento: El objetivo es estudiar el comportamiento del método de mejora del proceso *software* durante el ciclo de vida de desarrollo de un sistema.

• Diseño: Se aplicó el método de mejora del proceso *software* propuesto sobre el desarrollo de un emulador de una "Máquina de Café" desarrollado por un experto, siendo uno de sus objetivos obtener un producto con la calidad suficiente para utilizarlo en un curso de programación orientada a objetos. Conforme finalizaban las fases del ciclo de vida, se aplicaba el modelo de calidad y, antes de proseguir, se estudiaban las recomendaciones proporcionadas.

• Interpretación: Partiendo del primer análisis consensuado, con un valor de calidad de 0,44, y analizando los resultados del modelo de calidad, se realizó una revisión del análisis, alcanzando ahora una calidad de 0,63. Seguidamente, se aplicó el modelo sobre el diseño, lográndose un valor de calidad de 0,59. Tras realizar un segundo diseño utilizando las recomendaciones aportadas, la calidad ascendió hasta 0,64. Finalmente, se consiguió un 0,70 como valor global de la calidad de la primera implementación, llegando por fin al 0,79 tras revisar el código con las recomendaciones del modelo.

Uno de los resultados que pueden extraerse del experimento se encuentra en el hecho de que la poca cantidad de medidas presentes en la fase de análisis causan que no exista una fuerte relación entre la calidad del análisis y la del diseño (que dispone de un número considerablemente mayor de medidas). De hecho, a pesar de que el diseño tenía en cuenta las mejoras sugeridas sobre el análisis, las nuevas medidas provocaron inicialmente una ligera disminución de la calidad, que no debe ser tenida como una indicación de un mal trabajo. Esta situación no ocurre entre el diseño y la implementación, puesto que se ha observado una continuidad progresiva en los datos aportados por el modelo de calidad, debido principalmente a que una gran cantidad de medidas son comunes a ambas fases y, por tanto, miden los mismos atributos de la calidad del sistema.

Para finalizar, es de destacar, que la utilización de las recomendaciones del modelo de calidad permite una mejora significativa de la calidad (un 35% frente al 16%), lo cual refleja que la utilización del modelo de calidad puede ayudar a mejorar sustancialmente un desarrollo *software* desde las etapas iniciales del ciclo de vida.

5.2.20. EXPERIMENTO 20: COMPORTAMIENTO DEL MÉTODO DE MEJORA DEL PROCESO *SOFTWARE* SOBRE UN PROYECTO GRANDE

• Planteamiento: El objetivo es estudiar el comportamiento del método de mejora del proceso *software* durante el ciclo de vida de desarrollo de un sistema de gran tamaño.

• Diseño: Se aplicó el método de mejora del proceso *software* propuesto sobre el desarrollo de un "Curso de Idiomas" interactivo para ciegos llevado a cabo dentro de

un grupo de trabajo formado por cinco personas: 1 director de proyecto, 2 ingenieros del *software* y 2 programadores. Conforme finalizaban las fases del ciclo de vida, se aplicaba el modelo de calidad y se estudiaban las recomendaciones proporcionadas.

• Interpretación: Sin entrar en los detalles del experimento, que se encuentran descritos en el apartado I.6.1, el presente desarrollo ha sufrido diversos cambios y versiones distintas, tanto en el análisis como en el diseño, fruto en parte a las recomendaciones proporcionadas por el modelo de calidad y en parte por el proceso iterativo seguido en su construcción. En cualquier caso, la mejora obtenida ha sido significativa, pues se ha pasado de una calidad de 0,39 inicialmente a un valor de 0,57 (en el primer prototipo, aunque se podía esperar un mejor resultado, pero al no ser la implementación del sistema completo, adolece de ciertas deficiencias).

El desarrollo ha demostrado la utilidad del método de aplicación del modelo de calidad sobre el desarrollo de un sistema real de una duración y tamaño considerable, al permitir mejorar significativamente la calidad del sistema, aportando recomendaciones objetivas y valiosas para los desarrolladores.

5.2.21. EXPERIMENTO 21: COMPORTAMIENTO DEL MODELO DE CALIDAD SOBRE UN PROYECTO GRANDE

• Planteamiento: El objetivo es estudiar el comportamiento del modelo de calidad durante el ciclo de vida de desarrollo de un sistema de gran tamaño ("Desarrollo Gráfico").

• Diseño: Se aplicó el modelo de calidad propuesto sobre el desarrollo de una herramienta para la ayuda al diseño orientado a objetos que permite generar código Java. Se estudiaron los resultados, que fueron traspasados al desarrollador.

• Interpretación: La aplicación del modelo de calidad durante las fases de análisis, diseño e implementación permitió sugerir al desarrollador una serie de recomendaciones que consiguieron mejorar la calidad desde el 0,48 del análisis hasta el 0,71 de la implementación.

Según el proceso seguido, puede verse cómo el modelo de calidad resulta una herramienta útil durante todo el proceso de desarrollo de un sistema, pues permite detectar y corregir aquellas deficiencias que muchas veces pasan inadvertidas a los propios desarrolladores o a los validadores.

5.2.22. EXPERIMENTO 22: INTERPRETACIÓN DETALLADA DE LAS MEDIDAS

• Planteamiento: Se trata de ver cómo interpreta el desarrollador los resultados y sugerencias proporcionadas por el modelo de calidad.

• Diseño: Se aplicó el modelo de calidad sobre el diseño y la implementación de un pequeño sistema ("Agenda Personal"). Se seleccionó uno de los criterios, para hacer más manejable el experimento, y se estudiaron las respuestas proporcionadas por el desarrollador a las recomendaciones emitidas por el modelo de calidad.

• Interpretación: Se eligió el criterio de completitud y se aplicaron sus medidas sobre el diseño y la implementación. Los resultados del modelo fueron muy buenos, aunque hubo una serie de medidas que no lograron un buen valor, por lo que se transmitió al desarrollador las recomendaciones dadas por el modelo para dichas medidas. Éste estudió las recomendaciones, estando de acuerdo con todas ellas. La conclusión obtenida es que las medidas le permitieron detectar pequeños defectos del sistema.

5.2.23. EXPERIMENTO 23: ANÁLISIS DE LOS RESULTADOS DEL MODELO DE CALIDAD

• Planteamiento: Con este experimento se pretenden estudiar los datos más llamativos proporcionados por el modelo de calidad.

• Diseño: Se aplicó el modelo de calidad sobre un juego ("Juego de Dados") y utilizando únicamente las medidas de implementación. Se estudiaron los factores que presentaban mejores y peores resultados.

• Interpretación: Empleando las medidas aplicables se ha obtenido un valor de calidad regular. Solo tres factores alcanzan el valor 0,5, siendo de destacar el valor de la fiabilidad con un valor de 0,61. En la parte baja de la calidad, se encuentran los factores interoperatividad y utilizabilidad con valores por debajo de un tercio.

En lo referente a los criterios, destacan los altos valores conseguidos por la completitud, la precisión (los pocos cálculos matemáticos se realizan adecuadamente) y la simplicidad (el problema no era difícil de implementar y se ha realizado sin excesivas complicaciones).

Por el contrario, lo realmente interesante estriba en el análisis de los criterios que no han alcanzado un mínimo nivel de calidad. Entre éstos destacan:

- El criterio auto-descriptivo no ha obtenido un buen valor debido a la práctica inexistencia de comentarios. No obstante, gracias a la uniformidad en la escritura del código y otros detalles, el valor obtenido no resulta excesivamente bajo.
- La concisión falla debido a que no se emplea ninguna de las técnicas de la orientación a objetos que favorecen esta cualidad.
- Los datos estándar han obtenido una baja puntuación debido a la escasa utilización de tipos abstractos de datos y clases de forma uniforme.
- El problema de la documentación se debe a la total ausencia de cualquier tipo de documentación acompañando al programa.
- La baja expansibilidad estriba en la ausencia de herencia y clases y funciones genéricas.
- La generalidad también resulta ligeramente baja por diversas razones, entre las que destaca la falta de definición de ningún constructor, a pesar de su necesidad, ni destructores.
- Los criterios instrumentación y entrenamiento. Su bajo valor se debe a que éstos no se encontraban en los objetivos del programa y, por tanto, no han sido tenidos en consideración en el desarrollo.
- Por último, la tolerancia a errores, debido a la sencillez de la interfaz de usuario y de los pocos errores que se pueden producir.

Como puede observarse, todas estas explicaciones (surgidas de la interpretación de las medidas) resultan tremendamente útiles para el equipo de desarrollo así como para los estudiantes que entran en el terreno de la programación.

5.2.24. EXPERIMENTO 24: ANÁLISIS DE LOS RESULTADOS DEL MODELO DE CALIDAD

• Planteamiento: Con este experimento se pretenden estudiar los datos más destacados proporcionados por el modelo de calidad.

• Diseño: Se aplicó el modelo de calidad sobre un sistema de diseño orientado a objetos ("Desarrollo Gráfico") y se estudiaron los factores que presentaban mejores y peores resultados.

• Interpretación: Tras aplicar el modelo de calidad al análisis, se identificaron una serie de criterios que deberían mejorarse: datos estándar, documentación, instrumentación, operatividad y tolerancia a errores.

Los resultados para el diseño permiten ver que los factores que tenían unos valores más bajos en el análisis han mejorado ostensiblemente, si bien otros han visto disminuido ligeramente su valor, aunque en términos globales ha habido una mejoría generalizada. Viendo los criterios también se comprueba una significativa mejoría, aunque algunos de ellos han empeorado, debido fundamentalmente a las nuevas medidas que se han podido aplicar en el diseño. Los peores resultados se centran en la tolerancia a errores, instrumentación, eficiencia de ejecución...

Tras haber finalizado la implementación se obtuvo una mejoría importante de la calidad con respecto al diseño. El factor transportabilidad ha alcanzado un valor significativamente elevado, poniendo de manifiesto que el sistema está siendo implementado en Java, intentando que sea independiente de la plataforma. También resulta interesante observar cómo el factor de flexibilidad ha logrado buenos valores, lo cual resulta necesario pues el sistema tiene que ser ampliado con nuevas funcionalidades. En lo que se refiere a los criterios, resultan interesantes los resultados superiores a 0,75 alcanzados por 13 de los 23 criterios utilizados.

En el lado negativo de los valores de los factores se encuentra la utilizabilidad, en la que cuatro de los seis criterios no han superado la mitad de la escala. La explicación de este comportamiento estriba en el hecho de que el sistema estudiado constituye la primera fase de un sistema mayor. Esto ha llevado al desarrollador a no incluir todavía documentación de usuario ni ningún tipo de ayuda para el entrenamiento del usuario en su utilización. No obstante, el programador es consciente de su necesidad y que, para finalizar con éxito todas las fases del sistema, resultará imprescindible mejorar los cinco criterios de este factor que se alejan de los valores normales del resto de los criterios.

Puede verse cómo del estudio independiente de los factores o de los criterios puede extraerse una información muy útil, no solo para el desarrollador, sino también para los ingenieros del *software* y directivos a cargo del proyecto.

5.2.25. EXPERIMENTO 25: ANÁLISIS DE LOS RESULTADOS DEL MODELO DE CALIDAD

• Planteamiento: Con este experimento se pretenden estudiar los datos más llamativos proporcionados por el modelo de calidad.

• Diseño: Se aplicó el modelo de calidad sobre el sistema del "Caballo de Ajedrez" implementado en C++. Se estudiaron los factores que presentaban mejores y peores resultados.

• Interpretación: La calidad obtenida es buena, presentando la mayoría de los factores valores buenos, siendo de destacar los resultados de corrección, fiabilidad, mantenibilidad, reutilizabilidad y transportabilidad por encima de la media. En el lado negativo se encuentran la interoperatividad y utilizabilidad con valores bastante por debajo de la media.

Si se estudian los criterios, destacan los buenos valores obtenidos para el criterio auto-descriptivo (se ha conseguido un programa fácilmente comprensible mediante la mera lectura del código), completitud (puesto que, evidentemente, la solución al problema es completa), independencia del *hardware* y del *software* (lógico al ser un programa para enseñar a programar en C++, independientemente de la plataforma) y simplicidad (dado que es un programa realizado fundamentalmente para la enseñanza, se ha logrado resolver el problema de una forma poco compleja y, por tanto, sencilla), así como los valores mejorables alcanzados por la concisión (no se puso mucho hincapié en realizar un sistema conciso, teniendo en cuenta que muchas veces la concisión va en contra de la claridad), datos estándar (el pequeño tamaño del problema no daba pie a la utilización de muchas estructuras de datos) documentación (se limita a explicar su funcionamiento interno), entrenamiento (es un sistema para la enseñanza de la programación y, por tanto, no incluye una enseñanza en su utilización), instrumentación (no se han incorporado mecanismos para la ayuda a la depuración y detección de errores debido al tamaño del programa) y la tolerancia a errores (no tiene mayor importancia dada la sencillez del sistema).

A la vista de estos resultados, puede concluirse que los principales defectos que presenta el sistema vienen ocasionados por la finalidad docente del programa, y a que, dado su pequeño tamaño, no se han empleado muchas de las posibilidades de la orientación a objetos en C++, como el polimorfismo, las clases y funciones genéricas o el manejo de excepciones, aunque se podrían solucionar algunos aspectos.

5.2.26. EXPERIMENTO 26: CONTRASTE DE LA CALIDAD EN VARIOS DESARROLLOS

• Planteamiento: El objetivo consiste en analizar el comportamiento del valor de calidad dado por el modelo en comparación con la opinión de expertos para diversos sistemas que resuelvan el mismo problema. La comparación se realizará para los diseños y las implementaciones.

• Diseño: Para el "Simulador de Gases" 33 alumnos realizaron sus respectivos diseños e implementaciones del sistema. Estos trabajos fueron evaluados por los profesores. A

continuación, se aplicó el modelo de calidad, en sus fases de diseño e implementación, sobre todos los sistemas. Finalmente, se compararon las calificaciones de los profesores con los resultados del modelo de calidad.

• Interpretación: La comparación entre los datos obtenidos por el modelo de calidad con los obtenidos de los profesores ha ofrecido unos resultados excelentes, lo cual pone de manifiesto que los resultados del modelo se pueden utilizar como un indicador de la calidad del sistema. El análisis de 33 versiones diferentes del mismo sistema da una idea de la utilidad del modelo para predecir la calidad de un desarrollo. Por tanto, puede afirmarse que los valores de calidad obtenidos reflejan lo que realmente pretenden medir. Por todo ello, los resultados del modelo resultan convenientes para indicar la calidad de un sistema dada su elevada correlación con la realidad según el parecer de los especialistas.

5.3. RESUMEN DE LOS RESULTADOS

Las primeras ejecuciones de los experimentos permitió estudiar los resultados con el fin de ajustar el comportamiento del modelo de calidad a la realidad. Las modificaciones realizadas se centraron principalmente en el análisis del comportamiento de las distintas medidas, siendo necesaria la inclusión de nuevas medidas, la eliminación de alguna o la modificación de las fórmulas empleadas.

Los distintos experimentos realizados han permitido obtener una serie de resultados y contrastar las conclusiones extraídas, que pueden ser resumidos en los siguientes aspectos:

- Las distintas medidas proporcionan un reflejo de los atributos de calidad que miden que se acerca bastante a la realidad.
- Los criterios y los factores permiten descomponer la calidad de una forma acorde a la realidad.
- La calidad global que proporciona el modelo de calidad se aproxima considerablemente a la opinión que pudiera tener un experto acerca del sistema.
- Los valores de calidad obtenidos para las fases de análisis, diseño e implementación pueden utilizarse como medidas reales y válidas durante el ciclo de vida de un desarrollo *software*.
- La utilización de los pesos en el árbol de calidad permite ajustar los valores que se propagan por el árbol, obteniéndose unos resultados considerablemente más precisos y acordes con la calidad real del sistema.
- El proceso de mejora del *software* se ha puesto claramente de manifiesto, si se utiliza el modelo de calidad durante el ciclo de vida.
- El análisis por separado de los vectores con los valores de los criterios y de los factores permite obtener una información precisa acerca de los problemas que puede presentar un sistema.
- El estudio de las medidas una a una permite guiar de forma rigurosa a los desarrolladores en la búsqueda de los defectos y problemas con el fin de mejorar la calidad.

- Los umbrales de calidad establecidos en 0,33 (mala calidad) y 0,75 (buena calidad) han resultado ser apropiados, si se estudian los datos obtenidos en todos los experimentos.
- El modelo de calidad puede aplicarse a un desarrollo orientado a objetos independientemente del lenguaje de programación elegido. Aunque el modelo presenta algunas medidas específicas de C++, el uso de otro lenguaje simplemente lleva asociado el hecho de eliminar, cambiar o añadir ciertas medidas, pero no por ello pierde validez. No obstante, al ser C++ un lenguaje más complejo que otros lenguajes orientados a objetos, el conjunto de medidas definidas en el modelo de calidad presenta, por lo general, una completitud superior al que se necesita en otros lenguajes orientados a objetos.
- La aplicación del modelo de calidad a un desarrollo siguiendo un paradigma no orientado a objetos no resulta totalmente adecuada. Sería recomendable adaptar el modelo a las nuevas características del paradigma utilizado.

Axioma de la Ingeniería del Software*:*
Buena estructura interna ⇒ buena calidad externa.

— Norman E. Fenton

[Fenton, 91]

6. CONCLUSIONES

6. CONCLUSIONES

El trabajo aquí presentado consiste en el diseño y la construcción de un modelo de calidad de aplicación en el paradigma orientado a objetos, en el cual se ha subdividido la calidad global de un sistema en una serie de 11 factores y éstos a su vez en un total de 26 criterios. Se ha presentado una jerarquía de **características del *software***, bien especificadas y diferenciadas, que definen la calidad del *software*. Su nivel superior (factores) refleja las características externas del producto mientras que su nivel inferior (criterios) ofrece una serie de características internas.

No obstante, puede comprobarse que la división de la **calidad** efectuada no es de aplicación únicamente en el paradigma orientado a objetos, sino que, debido a su generalidad, puede tener aplicación sobre cualquier otro paradigma de desarrollo de *software*.

Este modelo dispone además de un gran conjunto de **medidas** (más de 500) especialmente diseñadas para la evaluación de los atributos internos de calidad que se engloben dentro de la tecnología orientada a objetos y que pueden ser aplicadas objetivamente para obtener una evaluación consistente de un desarrollo orientado a objetos. Además, no ha quedado ahí el estudio, puesto que se ha propuesto un **método** para su aplicación durante el ciclo de vida del desarrollo de los productos *software*, con el fin de obtener una estimación de la calidad del *software* en construcción, así como para permitir mejorar tanto el proceso *software* personal como corporativo.

Este estudio ha venido a demostrar que **el *software* puede medirse cuantitativamente durante las distintas fases de su desarrollo**, utilizando un conjunto de medidas, y que las medidas recolectadas pueden predecir el esfuerzo necesario para el mantenimiento de los sistemas. Sin esta retroalimentación cuantitativa de información, los gestores de proyectos dependen únicamente de su experiencia, intuición y juicio.

Si el desarrollo del *software* constituye una disciplina de ingeniería, debe adoptar estándares definidos para esta disciplina. Una diferencia crítica entre la ingeniería del *software* y otras ingenierías consiste en la ausencia de un conjunto de medidas globalmente aceptadas. Estas medidas ayudarían a esta disciplina a crecer y madurar.

Desde hace cerca de 20 años, se viene proclamando la necesidad de un conjunto de medidas automatizables para que los diseñadores e implementadores de complicados sistemas puedan evaluar rápida, cuantitativa y objetivamente la calidad del *software* [Henry, 84], pues realizar un seguimiento y valoración de la calidad resulta esencial para el éxito [Card, 99]. Este objetivo se alcanza con el trabajo aquí presentado, puesto que una amplia proporción de las medidas, así como sus procesos asociados, son fácilmente automatizables, cumpliéndose también, de esta manera, una de las sugerencias aportadas en [McCabe, 89].

Con el trabajo presentado se ha cumplido también el objetivo de definir una serie de nuevas **medidas específicas para la orientación a objetos** y, además, **clasificarlas**

según el criterio de calidad al que afectan. De esta forma, este conjunto de medidas claras y sencillas se diferencian de otros trabajos similares en el sentido de que éstos se limitan a medir sin preocuparse de estudiar la utilidad de dichas medidas, siendo éste un aspecto más importante aún que el de disponer simplemente de medidas.

El objetivo de cualquier proyecto que trate el tema de las medidas consiste en hacer de la recolección de medidas y su análisis una parte natural y útil dentro del proceso de desarrollo de *software*. El uso de estas técnicas proporcionará a los ingenieros del *software* una mayor comprensión de este proceso, así como de un mayor control sobre él. Mediante el uso del método de aplicación del modelo de calidad presentado, se cumple dicho objetivo, ofreciendo una importante ayuda en su trabajo a los desarrolladores.

Muchas veces, los directivos a cargo de un proyecto requieren información que debe ser extractada de entre una miríada de detalles, pero al mismo tiempo, esos detalles deben estar disponibles para un análisis posterior [Boloix, 97]. Este propósito se cumple con la información en distintos niveles de detalle que proporciona el modelo de calidad.

El uso y el **estudio de los resultados de las medidas** en las etapas iniciales del ciclo de vida puede proporcionar al ingeniero del *software* una importante y útil **retroalimentación durante el desarrollo**. El tipo de medidas proporcionadas en el modelo de calidad, junto con el método de aplicación presentado, es capaz de guiar a los ingenieros del *software* en la evaluación y construcción del *software* en sus distintas etapas, facilitando, de esta manera, el control de la calidad del producto final. En particular, este aspecto resulta importante en el sentido de que se puede **conocer la calidad del producto en las etapas iniciales del ciclo de vida** (las medidas calculadas únicamente sobre el código fuente solo proporcionan información útil al final del ciclo de vida, por lo que resultan inútiles durante las primeras fases de un proyecto [Littlefair, 01]). Esto es interesante de por sí, puesto que los errores o elementos que pueden conducir a un error deben poder ser detectados lo antes posible para disminuir el posible impacto económico negativo en el proyecto debido a un mantenimiento altamente complejo (estudios fiables a escala mundial indican que entre el 60 y el 70% del ciclo de vida del *software* es de mantenimiento [Barba, 92]; la reparación de defectos consume más del 50% de los recursos del proyecto [Boehm, 90]). Las medidas evaluadas pronto dentro del ciclo de vida de un sistema pueden tener un impacto representativo en la fase de diseño e implementación. De esta manera, las medidas de diseño pueden utilizarse para mejorar la calidad del *software* al proporcionar criterios cuantitativos para comparar alternativas de diseño [Bieman, 98] [Kirsopp, 99].

Una atención detallada a las características de la calidad del *software* puede conducir a significativos ahorros en los costes del ciclo de vida de los desarrollos informáticos.

La aplicación de las medidas en las fases de análisis y diseño no solo ayuda al analista y al diseñador a desarrollar su labor (y como apoyo para dirigir los esfuerzos por un determinado camino [Rosenberg, 99]), sino que, además, reduce el trabajo posterior de codificación, pruebas y mantenimiento. En otras palabras, produce, en general, un **ciclo de vida más eficiente**. Así, puede utilizarse el conjunto de medidas para identificar aquellos componentes que contengan un inusualmente elevado número de

errores o aquéllos que requerirán un tiempo significativamente mayor a la media para su codificación e, incluso, para dirigir al equipo de desarrollo en la búsqueda de errores. Los resultados pueden también emplearse para ayudar a comprender el código y el diseño, así como para entender y controlar la tarea de pruebas.

Estas medidas, en una gran parte, son consistentes con las características de un buen análisis, diseño y programación orientada a objetos, con los tres principios fundamentales de la orientación a objetos (abstracción, herencia y polimorfismo) y con la percepción de la calidad del *software*.

Por ello, las medidas presentadas pueden utilizarse también como complemento en el **autoaprendizaje** para determinar si los principios y la tecnología orientada a objetos se ha empleado apropiadamente en el desarrollo de un sistema. De esta forma, el modelo junto a las medidas puede ayudar a los programadores a comprender la manera de programar con este paradigma.

Este modelo permite ayudar en la **búsqueda sistemática de defectos** en la calidad. El modelo indica dónde buscar los errores, además de señalar cuáles son los posibles fallos que pueden haberse cometido. Toda esta información proporciona una útil guía para la validación de los productos obtenidos en cada una de las fases del ciclo de vida.

En términos generales puede afirmarse que no se ha explotado todo el potencial de las medidas desarrolladas hasta la fecha para la reducción del costo del *software*. Este defecto se debe a la ausencia de un método unificado para el desarrollo del *software* empleando las medidas. Este es uno de los puntos que el trabajo que aquí se presenta ha pretendido solucionar al presentar el método de aplicación.

Debe tenerse en cuenta que la adecuación de cualquiera de las medidas propuestas no puede ser asegurada siempre con total precisión. Ello se debe a que, obviamente, distintas personas conciben e interpretan la calidad del *software* de forma diferente. Por esto se ha dejado abierto el diseño tanto del árbol de calidad como de las medidas, así como los pesos que intervienen en los cálculos de todos los valores.

Aunque existe algún autor que argumenta que no se deben utilizar demasiadas medidas para medir una determinada característica (puesto que se difumina la contribución de cada medida) [Schneidewind, 00], hay que tener en cuenta que los errores de un programa se distribuyen aleatoriamente por el código. Por ello, las medidas serán más fiables conforme permitan evaluar más atributos de calidad del programa [Basili, 83a]. El amplio conjunto de medidas presentado cumple con este objetivo.

La naturaleza de los sistemas y de la producción del *software* solamente puede comprenderse si una amplia variedad de atributos del *software* es cuantificada por las correspondientes medidas.

El modelo y el método definido proporcionan un proceso para obtener las diferentes propiedades que definen la calidad del *software*, propiedades que reflejan distintos

atributos de la calidad del *software*. Es decir, se ha establecido un enlace entre las características tangibles de un producto y sus menos tangibles atributos de calidad.

En otro orden de cosas, en ningún momento se ha realizado la afirmación de que el modelo propuesto sea perfecto y que sea el único que se deba emplear en el desarrollo de los sistemas informáticos. El modelo es, en cierto modo, empírico y, por tanto, abierto para su refinamiento y ampliación. Pero, en cualquier caso, aunque los detalles del modelo pudieran sufrir algún cambio o mejora, el modelo seguirá proporcionando una **base para medir la calidad** del *software*.

El objetivo de este estudio no era presentar un modelo definitivo para la medida y predicción de la calidad del *software*, sino que constituyera un importante paso adelante en esta área de investigación, principalmente en el campo del paradigma orientado a objetos, donde existía un claro vacío a la hora de aplicar e integrar toda la experiencia surgida a lo largo de los años dentro de los paradigmas tradicionales y que no se había intentado aplicar plenamente a la orientación a objetos, salvo, quizás, con la definición de algunas nuevas medidas.

Así como para la estimación del coste del *software* existen diversos modelos útiles, como, por ejemplo, el COCOMO [Boehm, 81] [Boehm, 95] (aunque en nuestros días no se haya todavía encontrado una solución perfecta [Jones, 98]), no parece que existieran hasta la fecha modelos prácticos que pudieran estimar la calidad del *software* antes de que comenzara la fase de pruebas. En este trabajo se ha pretendido llenar en parte este importante vacío.

El modelo de calidad presentado no solo proporciona un formalismo para comprender mejor la calidad global de un sistema mediante el uso de medidas específicas, sino que, además, aporta una base sólida para su **uso efectivo** en situaciones prácticas **donde se necesite información sobre la calidad** de un producto.

Los ingenieros del *software* que deseen mejorar su proceso de desarrollo y posibilidades de mantenimiento deberían adoptar y extender el uso de las medidas aquí presentadas junto al método de aplicación en el seno de su organización. Un ingeniero del *software* podría mejorar la calidad de sus desarrollos de dos formas. En primer lugar, utilizando el modelo de calidad tal cual, aceptando como válidos los factores y criterios predefinidos y sus interrelaciones y considerando adecuadas las medidas y su relación con los criterios. En segundo lugar, definiendo su propio modelo de calidad, determinando los factores y criterios que resulten convenientes, estableciendo las relaciones adecuadas y empleando las medidas apropiadas. Evidentemente, para este último supuesto, sería mucho más fácil partir del modelo de calidad ya construido y alterarlo, bien modificando los pesos, bien incorporando algún nuevo elemento (factor, criterio, relación o medida).

Solamente en el contexto de un modelo o una teoría tienen sentido las medidas [Shepperd, 94]. El problema que poseen la mayoría de las medidas presentes en la literatura es que carecen de dicho sustento. En este trabajo se ha definido el conjunto de medidas integrado dentro de un completo modelo de calidad, con el fin de dar sentido a dichas medidas.

Algunas de las características de los lenguajes de programación pueden interferir con la habilidad de los desarrolladores para programar y comprender el código con el que trabajan. Una de las posibles soluciones consiste en la utilización de un modelo de calidad específico que pueda advertirles ante posibles fuentes de problemas en el código desarrollado.

Ya se ha comentado que el PSP (*Personal Software Process*) proporciona a los ingenieros del *software* los pasos específicos que pueden seguir para evaluar y mejorar sus habilidades en el desarrollo de *software*. El método que aquí se ha presentado complementa al PSP u otros procedimientos similares en el sentido de que aporta una **forma completa de evaluar un desarrollo** *software* ofreciendo un amplio conjunto de medidas clasificadas en criterios y factores (las mediciones son una parte necesaria en todo proceso de mejora [Pfleeger, 95]). Este método puede aplicarse también para conseguir una **mejora del proceso *software***, no solo en el ámbito personal sino también en el ámbito corporativo.

De esta forma, las mejoras en la calidad del proceso *software* pueden verse reflejadas en tres aspectos [Ferguson, 97]: Primero, al detectarse los defectos, los ingenieros se sensibilizan por los errores que han cometido y, por tanto, se preocuparán en realizar cuidadosamente su trabajo. Segundo, cuando analizan sus defectos, comprenden claramente el coste de reparar los defectos y, por ello, aplican las técnicas más efectivas para encontrarlos y corregirlos. Y tercero, el PSP presenta una serie de prácticas de calidad efectivas en la prevención de defectos y en la búsqueda y corrección eficientes.

"El Dr. Chandra consiguió restaurar los módulos perdidos y el ordenador ya parece funcionar plenamente. El problema estuvo causado por un conflicto entre las instrucciones básicas y los requisitos de seguridad. Por orden directa del presidente, la existencia de TMA-1 se mantuvo en secreto, incluso ante la tripulación. Como el ordenador podía controlar la nave hasta Júpiter, se decidió programarle para que pudiera completar la misión autónomamente; se le dieron detalles de sus objetivos pero no se le permitió revelarlos a la tripulación. Como resultado, desarrolló lo que podría llamarse, en términos humanos, una psicosis, concretamente esquizofrenia. Técnicamente hablando, se vio atrapado en un bucle Hofstadter-Moebius, una situación algo común en ordenadores avanzados con programas dirigidos por la meta. Tuvo que afrontar un dilema intolerable, por lo que desarrolló síntomas paranoicos dirigidos a impedir que detectaran el problema desde la Tierra. Debido a órdenes contradictorias, informó de un fallo (inexistente) en la antena AE 35 y esto le sumió no solo en una mentira sino en un enfrentamiento con la tripulación. Presumiblemente, decidió que la única forma de resolver la situación era eliminar a sus colegas humanos. En resumen, la rehabilitación del ordenador se ha producido satisfactoriamente, pero, en el futuro... ¿podremos confiar en el ordenador HAL 9000?"
Informe de Heywood Floyd, desde el U. S. S. Discovery.

— Arthur C. Clarke

[Clarke, 83]

7. FUTURAS LÍNEAS DE INVESTIGACIÓN

7. FUTURAS LÍNEAS DE INVESTIGACIÓN

Un aspecto destacable del presente trabajo lo constituye la apertura de nuevas líneas de investigación y aplicación y la continuación de algunas ya existentes.

El desarrollo de sistemas constituye una actividad intelectual: la conversión de una idea en un programa. No obstante, si la profesión de ingeniero del *software* consiste en mejorar la forma en que este crítico trabajo se realiza, entonces es necesario una forma de medir esta actividad intelectual. Es imposible diseñar medidas perfectas, pero cualquier esfuerzo dirigido hacia esta meta debería resultar en unas medidas mejores y, por tanto, en una mayor utilización en la práctica. Por todo ello, el trabajo realizado no debe quedar aquí detenido, sino que se debe seguir explorando esta área del conocimiento con el fin de ampliar y optimizar el modelo de calidad con el objetivo último de una amplia utilización.

Una futura investigación empírica podría proporcionar unos límites o **umbrales** para cada medida, de forma que se pudiera determinar con mayor exactitud cuándo se ha obtenido un valor bueno o malo en la evaluación de una medida. Para ello, podrían emplearse técnicas estadísticas como las utilizadas en [French, 99].

En comparación con las medidas del código, hay relativamente pocos trabajos sobre las **medidas de los requisitos, el análisis y el diseño**. Esto se debe en parte a que los lenguajes de programación proporcionan una notación formal sobre la que se pueden basar las medidas, mientras que las representaciones formales para expresar los requisitos, el análisis o el diseño están emergiendo recientemente, aunque no hay un acuerdo general sobre este tema. El futuro de la investigación en las medidas del *software* se debe concentrar en la definición y el análisis de las medidas de las etapas iniciales del ciclo de vida de los sistemas, lo cual resulta fundamental con el fin de poder guiar a los desarrolladores en la evaluación del producto en construcción. En concreto, sería conveniente ampliar el número de medidas para la especificación de requisitos (como en [Jilani, 01] [Kececi, 01] [Rule, 01]) y para el análisis, así como incluir medidas que evalúen nuevos aspectos durante la fase de diseño (como los trabajos de [Genero, 99] [Verkamo, 01]). Por otra parte, también se podría incluir un conjunto de medidas para su aplicación durante la fase de pruebas y validación, así como una serie de medidas dinámicas para su evaluación durante la ejecución del sistema, lo que permitiría completar el modelo de calidad con nuevas fuentes de información. El análisis estático solamente proporciona parte de la información acerca de la orientación a objetos; los aspectos dinámicos todavía no se han estudiado con profundidad [Briand, 99]. Ejemplos de este tipo de medidas dinámicas puede encontrarse en [Roper, 95] (para C++) o en [Buglione, 01a] (para medir el rendimiento del *software*).

La mayoría de las medidas del modelo de calidad presentado han sido especialmente pensadas para su utilización en cualquier desarrollo orientado a objetos. No obstante, también se han realizado algunas medidas para uno de los lenguajes de programación orientados a objetos más utilizados (C++), aunque intentando no perder generalidad. Habría que plantearse la realización de un estudio completo de la validez y la

adaptación, en su caso, de estas medidas para manejar otros lenguajes de programación orientados a objetos como CLOS, Eiffel, Java, Object Pascal, Objective-C, Smalltalk...

Aunque queda todavía mucho trabajo por realizar en el área de las medidas del *software*, la investigación aquí presentada proporciona un marco de trabajo para la búsqueda de nuevas medidas de acuerdo a las características más relevantes de las nuevas técnicas de desarrollo de programas que pudieran surgir en un futuro.

Para el **refinamiento** del modelo de calidad podrían utilizarse también datos obtenidos de aplicaciones comerciales de gran tamaño, puesto que puede que sean diferentes a los datos de los sistemas de tamaño pequeño/mediano que, principalmente, se han utilizado en la validación. Un área de estudio sería pues la recolección de más datos provenientes de grandes sistemas con el fin de ampliar este trabajo y verificar completamente su validez en otros entornos, incorporando información acerca de sistemas de diferentes tipos. De esta forma, se podría conseguir una gran base de datos histórica con información detallada de los resultados obtenidos con el modelo de calidad.

Un campo de investigación futura se centra en averiguar en qué grado las medidas validadas para uno o más proyectos son medidas precisas de la calidad de proyectos futuros o, por el contrario, necesitan un ajuste fino.

Por ello, antes de poder utilizar los resultados del modelo de calidad en un nuevo entorno de trabajo, es necesario validarlos en dicho entorno, puesto que pueden existir algunas diferencias entre ellos. En concreto, para cada entorno de trabajo y cada dominio de aplicación, deberán completarse las medidas, refinar sus pesos, verificar las relaciones entre medidas y criterios..., en definitiva, adecuar el modelo al entorno y al dominio en particular, de tal forma que, una vez realizada esta labor, se podrá extraer el máximo partido en su utilización.

Una de las tareas complementarias a realizar con el trabajo aquí presentado consiste en seguir utilizándolo en el desarrollo de sistemas reales y su aplicación durante todo el ciclo de vida conforme avanza su construcción. De esta forma, se permitirá que el modelo pueda evolucionar con el tiempo, lo que provocará un aprendizaje importante tanto en el área de la orientación a objetos como en el de las medidas y la calidad del *software*. De hecho, está tarea ya ha comenzado, puesto que en el grupo de trabajo del autor, los nuevos sistemas orientados a objetos que se desarrollan están siguiendo el método de aplicación del modelo de calidad para evaluar su desarrollo, con un resultado altamente positivo hasta la fecha.

Por supuesto, junto a la aplicación del modelo de calidad en un desarrollo, pueden seguirse utilizando otras medidas no incluidas en el modelo, aunque el proceso adecuado sería incorporarlas al modelo si éstas miden algún aspecto de la calidad del *software*.

Los resultados de aplicar el modelo de calidad podrían utilizarse como una manera de **evaluar un programa**, con el fin de ver cuáles son sus principales cualidades y sus defectos más importantes. Además, podría también emplearse para evaluar distintas

soluciones para un mismo problema. En cualquier caso, además de utilizar el modelo de calidad sobre sistemas completos, también se podría usar en librerías o componentes, con el fin de determinar su calidad (ampliando las ideas expuestas en [Etzkorn, 97]); de esta forma, en un entorno de desarrollo solamente se publicarían en el repositorio aquellos componentes que tuvieran un mínimo de calidad, indicando, así mismo, sus posibles deficiencias, con el fin de que el usuario de dichos componentes pudiera adoptar las medidas necesarias para solucionarlo si lo viera conveniente.

Un aspecto importante es que el modelo de calidad construido debe considerarse como un **modelo dinámico**, es decir, que debe ir evolucionando. Una posible forma de evolucionar podría ser la inclusión de un proceso que le permita auto-adaptarse conforme se va utilizando en distintos desarrollos. Esta auto-adaptación podría aplicarse a los pesos, a las medidas y sus fórmulas o al propio árbol de calidad. La fuente de información se podría obtener por realimentación de los usuarios, por ejemplo: por su opinión sobre los atributos y lo que representan, por la utilización o no de cada medida en un problema, por la utilización de los resultados de cada medida en el proceso de mejora del *software*... Incluso se podría pensar en la inclusión de una auto-adaptación automática mediante técnicas de aprendizaje automático, redes de neuronas, etc.

Relativamente pocas investigaciones se han llevado a cabo en la medición fuera del paradigma imperativo. Quizá el único que ha atraído últimamente las miradas ha sido la medición en el paradigma orientado a objetos, fruto de este trabajo. Pero en otro tipo de paradigmas, como pueden ser los paradigmas funcionales, lógicos o demostrativos no existen prácticamente investigaciones significativas (casi limitadas a los trabajos de [Stoeffler, 92] [Berg, 95] [Smaraweera, 98] [Zhao, 98]). Sería interesante estudiar la aplicabilidad y adaptabilidad del modelo de calidad descrito en este trabajo, junto con el estudio de un nuevo conjunto de medidas para evaluar la calidad de los sistemas desarrollados bajo estos **otros paradigmas**.

Incluso, se podría pensar en el diseño de modelos de calidad dirigidos a ciertos tipos de **aplicaciones específicas**, como por ejemplo, para aplicaciones multimedia (como los trabajos iniciales de [Abrardo, 98] [Cowderoy, 98] [Cowderoy, 99]), para aplicaciones Web o portales de Internet (como los estudios de [Claffy, 00] [Spolverini, 00] [Hausen, 00] [Cowderoy, 01] [Mallepalli, 01] [Pooley, 02]), para la calidad de los servicios proporcionados por Internet (como los trabajos de [Demichelis, 01] para medir el comportamiento de una parte de una red IP o de [Sharif, 01] sobre *streaming* de video), para aplicaciones que incorporen agentes inteligentes [Weiss, 99] (como el trabajo sobre la medida de la inteligencia de los agentes por [Franklin, 00]), para aplicaciones en redes sin cable (como el trabajo de [Stavroulakis, 00]), en arquitecturas distribuidas, etc. Es en este último aspecto, donde la adecuación del paradigma orientado a objetos para el modelado e implementación de sistemas distribuidos ha llevado a la aparición de plataformas como Java/RMI, DCOM o CORBA [OMG, 01] que surgieron ante la necesidad de interoperatividad entre la gran cantidad de productos *hardware* y *software* existentes. Aunque ya se han realizado algunos trabajos en la llamada Calidad del Servicio en CORBA [Frolund, 98] o en sistemas de objetos distribuidos [QoSDOS, 01], sería interesante realizar un estudio detallado de la calidad en aplicaciones construidas mediante este tipo de plataformas.

Dentro del terreno de las mediciones, podría resultar adecuado investigar en la posibilidad de incorporar **otros tipos de medidas** dentro del modelo. Ejemplos de estos nuevos tipos de medidas serían la medición de aspectos subjetivos, como por ejemplo, la satisfacción del usuario ante el sistema (tal como se apunta en [Myers, 97]), o la medición de aspectos del propio proceso de desarrollo (como en [Henderso, 01]), como por ejemplo, las tareas de gestión del proyecto (planificación, estimación, actividades de control...), adecuación de la definición del proceso o prácticas para mantener la calidad (aplicación de las medidas, revisiones, efectividad de las pruebas, prevención de fallos...).

Sería adecuado también intentar especificar un **programa de control de calidad**, que incluyera un modelo de calidad, de tal forma que pudiera aplicarse a todos los aspectos del desarrollo del *software*, incluso que sirviera para evaluar la calidad de la gestión de los proyectos. Esto, junto con un adecuado modelo de estimación, son los grandes retos que quedan por afrontar.

En otro orden de cosas, podría resultar un interesante campo de trabajo investigar en la posibilidad de **predecir la calidad** final de un sistema. El modelo de calidad presentado facilitaría esta predicción al poder analizar y estudiar los resultados obtenidos en las fases iniciales del ciclo de vida. Una forma de afrontar este problema podría ser empleando técnicas de aprendizaje automático por inducción a partir de ejemplos. Así, utilizando alguna de los sistemas de la familia TDIDT (como ID3 [Quinlan, 79] [Quinlan, 86], C^3AD [Fuertes, 92], C4.5 [Quinlan, 93], MITO [Montes, 94] o ITI [Utgoff, 01]) sobre distintos sistemas de un dominio determinado, podría construirse un árbol de decisión a partir de la información obtenida por la aplicación del modelo de calidad sobre todas las fases del ciclo de vida (incluyendo, incluso, la opinión de expertos). Los nodos del árbol podrían ser los factores, los criterios o las medidas obtenidas para las distintas fases del desarrollo. De esta forma, ante un nuevo sistema, se aplicaría el modelo de calidad y se consultaría el árbol de decisión que proporcionaría la calidad estimada de dicho sistema. La técnica de aprendizaje por inducción ya ha sido utilizada en algunas ocasiones en el área de la calidad del *software* desde otros puntos de vista: para localizar módulos susceptibles de tener defectos o módulos que requieren un elevado esfuerzo de desarrollo [Selby, 88] [Porter, 90], como método para facilitar la evaluación de las medidas o las características del proceso [Pfleeger, 92] o, más recientemente, como técnica para construir modelos de calidad particulares [Khoshgoftaar, 99] o, junto a las medidas, para controlar la calidad del *software* [Kokol, 01].

Por último, se podría pensar en la formalización detallada de un procedimiento de enseñanza para promover la **mejora personal del proceso *software***, de tal forma que, a través de una serie de ejercicios prefijados y los resultados del modelo de calidad, cada persona fuera capaz de detectar las principales causas de problemas y los errores cometidos en la resolución de los ejercicios. De esta manera, se dispondrá de un importante mecanismo para el aprendizaje autodidacto, mejorando al PSP.

No importa lo despacio que vayas,
siempre que nunca te detengas

— Confucio

8. BIBLIOGRAFÍA

8. BIBLIOGRAFÍA

En este capítulo se muestra parte de la bibliografía empleada para la elaboración de la presente tesis. Las referencias se presentan ordenadas por el apellido del primer autor. En el primer apartado se encuentra la bibliografía referenciada en este documento, mientras que el segundo apartado resume otras publicaciones de interés no mencionadas directamente en el trabajo.

8.1 BIBLIOGRAFÍA REFERENCIADA

Abrardo, 98 — Abrardo, A.; Caldelli, R.; Cowderoy, A.; Donaldson, J.; Granger, S.; Veenendaal, E.: "The MultiSpace Application Priorities", http://dialspace.dial.pipex.com/town/lane/xvc63/multispace/d2-2p.pdf, descargado el 7-12-1999, febrero, 1998.

Abreu, 94 — Abreu, F. B.; Carapuça, R.: "Candidate Metrics for Object-Oriented Software within a Taxonomy Framework", *Journal of Systems and Software*, 26(1), julio, 1994, págs. 87-96.

Aho, 90 — Aho, A. V.; Sethi, R.; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1990.

Alexander, 64 — Alexander, C.: *Notes on the Synthesis of Form*, Cambridge, Harvard University Press, Massacusetts, 1964.

Alonso, 01 — Alonso, F.; Fuertes, J. L.; Martínez, L.; Montes, C.: "Measuring Completeness in a New Object-Oriented Quality Model", *Journal of Object-Oriented Programming* (aceptado para su próxima publicación), julio, 2001.

Alonso, 02 — Alonso, F.; Frutos, S.; Fuertes, J. L.; Martínez, L.; Montes, C.: "Measuring Execution Efficiency in a New Object-Oriented Quality Model", enviado a la *7th European Conference on Software Quality*, junio, 2002.

Alonso, 95 — Alonso, F.; Segovia, F. J.: *Entornos y Metodologías de Programación*, Paraninfo, Madrid, 1995.

Alonso, 98a — Alonso, F.; Fuertes, J. L.; Montes, C.: "Improving the Object-Oriented Software Process by Means of a Quality Model", Proc. *Software Engineering and Knowledge Engineering* (SEKE'98), San Francisco, junio, 1998, págs. 152-155.

Alonso, 98b — Alonso, F.; Fuertes, J. L.; Montes, C.; Navajo, R. J.: "A Quality Model: How to Improve the Object-Oriented Software Process", Proc. *IEEE Systems, Man, and Cybernetics 1998* (IEEE SMC'98), San Diego, octubre, 1998, págs. 4884-4889.

Alonso, 99 — Alonso, F.; Fuertes, J. L.; González, A. L.; Montes, C.: "Towards a New Quality Model to Support the Object-Oriented Software", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 2, págs. 274-281.

ANSI, 83 — ANSI: *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, 1983.

Antoniol, 00 — Antoniol, G.; Caprile, B.; Potrich, A.; Tonella, P.: "Design-Code Traceability for Object-Oriented Systems", Annals of Software Engineering, 9, 2000, págs. 35-58.

Archer, 95a — Archer, C.; Stinson, M.: "Object-Oriented Software Measures", Technical Report CMU/SEI-95-TR-2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, abril, 1995.

Archer, 95b — Archer, C.: "Measuring Object-Oriented Software Products", Curriculum Module SEI-CM-2, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, junio, 1995.

Arthur, 85 — Arthur, L. J.: *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, Nueva York, 1985.

Arthur, 93 — Arthur, L. J.: *Improving Software Quality: an Insider's Guide to TQM*, John Wiley & Sons, Nueva York, 1993.

Austin, 97 — Austin, R. D.: "Measurement and Behavior", *American Programmer*, 10(11), noviembre, 1997, págs. 12-16.

Bail, 88 — Bail, W. G.; Zelkowitz, M. V.: "Program Complexity Using Hierarchical Abstract Computers", *Computer Languages*, 13(3/4), 1988, págs. 109-123.

Baker, 79 — Baker, A. L.; Zweben, S. H.: "The Use of Software Science in Evaluating Modularity Concepts", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 110-120.

Baker, 80 — Baker, A. L.; Zweben, S. H.: "A Comparison of Measures of Control Flow Complexity", *IEEE Transactions on Software Engineering*, 6(6), noviembre, 1980, págs. 506-512.

Bansiya, 97 — Bansiya, J.; Davis, C.: "Automated Metrics for Object-Oriented Development", *Dr. Dobb's Journal*, 272, diciembre, 1997, págs. 42-48.

Bansiya, 99a — Bansiya, J.; Etzkorn, L.; Davis, C.; Li, W.: "A Class Cohesion Metric for Object-Oriented Designs", *Journal of Object-Oriented Programming*, 11(8), enero, 1999, págs. 47-52.

Barba, 92 — Barba, E.: "El Mundo Cambiante de las Telecomunicaciones", *Comunicaciones World*, 61, octubre, 1992, págs. 19-22.

Barbacci, 95 — Barbacci, M. R.; Longstaff, T. H.; Klein, M. H.; Weinstock, C. B.: "Quality Attributes", Technical Report CMU/SEI-95-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, diciembre, 1995.

Barba-Romero, 87 — Barba-Romero, S.: "Panorámica Actual de la Decisión Multicriterio Discreta", *Investigaciones Económicas (segunda época)*, 11(2), 1987, págs. 279-308.

Barnard, 98 — Barnard, J.: "A New Reusability Metric for Object-Oriented Software", *Software Quality Journal*, 7(1), marzo, 1998, págs. 35-50.

Barnes, 93 — Barnes, G. M.; Softwareim, B. R.: "Inheriting Software Metrics", *Journal of Object-Oriented Programming*, 6(7), noviembre-diciembre, 1993, págs. 27-34.

Basili, 83a — Basili, V. R.; Hutchens, D. H.: "An Empirical Study of a Syntatic Complexity Family", *IEEE Transactions on Software Engineering*, 9(6), noviembre, 1983, págs. 664-672.

Basili, 83b — Basili, V. R.; Selby, R. W.; Phillips, T.: "Metric Analysis and Data Validation Across Fortran Projects", *IEEE Transactions on Software Engineering*, 9(6), noviembre, 1983, págs. 652-663.

Basili, 88 — Basili, V. R.; Rombach, H. D.: "The TAME Project: Towards Improvement-Oriented Software Environments", *IEEE Transactions on Software Engineering*, 14(6), junio, 1988, págs. 758-773.

Basili, 96 — Basili, V. R.; Briand, L.; Melo, W. L.: "A Validation of Object-Oriented Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, 22(10), octubre, 1996, págs. 751-761.

Baumert, 92a — Baumert, J. H.: "Software Measures and the Capability Maturity Model", Technical Report CMU/SEI-91-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1992.

Berard, 98 — Berard, E. V.: "Metrics for Object-Oriented Software Engineering", http://www.toa.com/pub/moose.html, descargado el 16-1-2001, 23 de agosto de 1998.

Berg, 95 — Berg, K.: *Software Measurement and Functional Programming*, Tesis Doctoral, Universtiy of Twente, Holanda, 1995.

Berrocal, 03 — Berrocal, A.; Piérola, B.; Caraça-Valente, J. P.; Fuertes, J. L.: "Software de Soporte para el Aprendizaje de Inglés Dirigido a Invidentes", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, 2003 (en preparación).

Bieman, 95a — Bieman, J. M.; Kang, B.: "Cohesion and Reuse in an Object-Oriented System" Proc. Symposium on Software Reusability (SSR'95), *ACM Sigsoft: Software Engineering Notes*, agosto, 1995, págs. 259-262.

Bieman, 95b — Bieman, J. M.; Karunanithi, S.: "Measurement of Language-Supported Reuse in Object-Oriented and Object-Based Software", *Journal of Systems and Software*, 30, 1995, págs. 271-293.

Bieman, 98 — Bieman, J. M.; Kang, B.-K.: "Measuring Design-Level Cohesion", *IEEE Transactions on Software Engineering*, 24(2), febrero, 1998, págs. 111-124.

Binder, 94b — Binder, R. V.: "Design for Testability in Object-Oriented Systems", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 87-101.

Bloch, 77 — Bloch, A.: *The Complete Murphy Law*, Price Sloan, Los Ángeles, California, 1977.

Blundell, 97 — Blundell, J. K.; Hines, M. L.; Stach, J.: "The Measurement of Software Design Quality", *Annals of Software Engineering*, 4, 1997, págs. 235-255.

Boehm, 76 — Boehm, B. W.; Brown, J. R.; Lipow, M.: "Quantitative Evaluation of Software Quality", Proc. *2nd International Conference on Software Engineering*, IEEE Computer Society Press, San Francisco, octubre, 1976, págs. 592-605.

Boehm, 78 — Boehm, B. W.; Brown, J. R.; Kaspar, H.; Lipow, M.; MacLeod, G. J.; Merrit, M. J.: *Characteristics of Software Quality*, North-Holland, Nueva York, 1978.

Boehm, 81 — Boehm, B. W.: *Software Engineering Economics*, Prentice-Hall, New Jersey, 1981.

Boehm, 89 — Boehm, B. W.: *Software Risk Management*, IEEE Computer Society Press, Washington DC, 1989.

Boehm, 90 — Boehm, B. W.; Papaccio, P. N.: "Understanding and Controlling Software Costs" en T. DeMarco y T. Listers (eds.) *Software State-of-the-art: Selected Papers*, Dorset House, Nueva York, 1990, págs. 31-60.

Boehm, 95 — Boehm, B.; Clark, B.; Horowitz, E.; Westland, C.; Madachy, R.; Selby, R.: "Cost Models for Future Software Life Cycle Processes: COCOMO 2.0", *Annals of Software Engineering*, 1(1), 1995, págs. 57-94.

Bologna, 88 — Bologna, S.: "Software Measurement– What and How", *1er Seminaire EOQC sur la Qualité des Logiciels*, Bruselas, abril, 1988, págs. 424-436.

Boloix, 00 — Boloix, G.: "Germinal Boloix Coffee Machine", http://www.geocities.com/SiliconValley/Drive/9071/cof.html, descargado el 27-3-2000.

Boloix, 97 — Boloix, G.: "System Evaluation and Quality Improvement", *Journal of Systems and Software*, 36(3), marzo, 1997, págs. 297-311.

Booch, 86 — Booch, G.: "Object-Oriented Development", *IEEE Transactions on Software Engineering*, 12(2), febrero, 1986, págs. 211-212.

Booch, 91 — Booch, G.: *Object-Oriented Design with Applications*, Benjamin/Cummings, Redwood City, California, 1991.

Booch, 94 — Booch, G.: *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, California, 1994.

Booch, 99 — Booch, G.; Jacobson, I.; Rumbaugh, J.: *The Unified Software Development Process*, Addison-Wesley Longman, enero, 1999.

Boreham, 97 — Boreham, B.: "Large-Scale C++", *Object Expert*, 2(2), enero-febrero, 1997, págs. 54-55.

Borrajo, 90 — Borrajo, J.: *Norma de Programación en C*, Telefónica I+D, Madrid, junio, 1990.

Bowen, 78 — Bowen, J. B.: "Are Current Approaches Sufficient for Measuring Software Quality?", Proc. *ACM Software Quality Assurance Workshop*, noviembre, 1978, págs. 148-155.

Bowen, 83 — Bowen, T. P.; Post, J. V.; Tsai, J.; Presson, P. E.; Schmidt, R. L.: "Software Quality Measurement for Distributed Systems, Guidebook for Software Quality Measurement", Vol II, Final Technical Report RADC-TR-83-175, Rome Air Development Center, Air Force Systems Command, Griffis Air Force Base, Nueva York, 1983.

Briand, 00 — Briand, L. C.; Wüst, J.; Daly, J. W.; Porter, D. V.: "Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems", *Journal of Systems and Software*, 51(3), mayo, 2000, págs. 245-274.

Briand, 99a — Briand, L.: "Measurement and Quality Modeling of OO Systems", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Measurement for Object-Oriented Software Projects Workshop, Boca Ratón, Florida, noviembre, 1999.

Brooks, 87 — Brooks, F. P.: "No Silver Bullets: Essence and Accidents of Software Engineering", *IEEE Computer*, 20(4), abril, 1987, págs. 10-19.

Brown, 87 — Brown, B. J.: "Static Analysis and Dynamic Analysis Applied to Software Quality Assurance", en *Handbook of Software Quality Assurance*, G. G. Schulmeyer y J. I. McManus (eds.), Van Nostrand Reinhold Company, Nueva York, 1987, págs. 268-284.

Buckley, 89 — Buckley, F. J.: "Standard Set of Useful Software Metrics Is Urgently Needed", *IEEE Computer*, 22(7), julio, 1989, págs. 88-89.

Buglione, 01a — Buglione, L.; Abran, A.: "Multidimensionality in Software Performance Measurement: the QEST/LIME Models", Proc. *International Conference on Advances in Infraestructure for Electronic Business, Science, and Education on the Internet* (SSGRR 2001), L'Aquila, Italia, agosto, 2001.

Burril, 80 — Burril, C. W.; Ellsworth, L. W.: *Modern Project Management: Foundations for Quality and Productivity*, Burrill-Ellsworth Associates, Tenafly, New Jersey, 1980.

Byard, 94 — Byard, C.: "Software Beans: Class Metrics and the Mismeasure of Software", *Journal of Object-Oriented Programming*, 7(5), septiembre, 1994, págs. 32-34.

Cain, 00 — Cain, J. W.: "An Analyser of Object Oriented Programs", http://www.blunder1.demon.co.uk/james/programme%20analyser.html, descargado el 19 de noviembre, 2000.

Cámara, 96 — de la Cámara, M.: "Métricas en Objetos", *Desarrollo y Macroinformación*, 6, 4º trimestre, 1996, págs. 8-9.

Canorea, 03 — Canorea, E.; Polo, M. A.; Fuertes, J. L.: "Sistema para la Evaluación de la Calidad de Programas Desarrollados en Java", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, 2003 (en preparación).

Cant, 94 — Cant, S. N.; Henderson-Sellers, B.; Jeffery, D. E.: "Application of Cognitive Complexity Metrics to Object-Oriented Programs", *Journal of Object-Oriented Programming*, 7(4), julio-agosto, 1994, págs. 52-63.

Card, 90 — Card, D. N.; Glass, R. L.: *Measuring Software Design Quality*, Prentice-Hall, New Jersey, 1990.

Card, 91 — Card, D. N.: "What Makes a Software Measure Succesful", *American Programmer*, 4, septiembre, 1991, págs. 2-8.

Card, 99 — Card, D.; Scalzo, B.: "Measurement for Object-Oriented Software Projects", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Measurement for Object-Oriented Software Projects Workshop, Boca Ratón, Florida, noviembre, 1999.

Carleton, 92 — Carleton, J. A.; Park, R. E.; Goethert, W. B.; Florac, W. A.; Bailey, E. K.; Pfleeger, S. L.: "Software Measurement for DoD Systems: Recommendations for Initial Core Measures", Technical Report CMU/SEI-92-TR-19, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1992.

Cavano, 78 — Cavano, J. P.; McCall, J. A.: "A Framework for the Measurement of Software Quality", Proc. *ACM Software Quality Assurance Workshop*, noviembre, 1978, págs. 133-139.

Chakravart, 67 Chakravart, I. M.; Laha, R. G.; Roy, J.: *Handbook of Methods of Applied Statistics, volume I*, John Wiley, 1967.

Channon, 74 Channon, R. N.: *On a Measure of Program Structure*, Tesis Doctoral, Carnegie Mellon University, Pittsburgh, Pennsylvania, noviembre, 1974.

Chen, 93 Chen, J.-Y.; Lu, J.-F.: "A New Metric for Object-Oriented Design", *Information and Software Technology*, 35(4), abril, 1993, págs. 232-240.

Chidamber, 91 Chidamber, S. R.; Kemerer, C. F.: "Towards a Metrics Suite for Object Oriented Design", Proc. 6th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91), *ACM Sigplan Notices*, 26(11), noviembre, 1991, págs. 197-211.

Chidamber, 94 Chidamber, S. R.; Kemerer, C. F.: "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, 20(6), junio, 1994, págs. 476-493.

Chistensen, 81 Chistensen, K.; Fitsos, G. P.; Smith, C. P.: "A Perspective on Software Science", *IBM Systems Journal*, 20(4), 1981, págs. 372-387.

Chung, 97 Chung, C. M.; Shih, T. K.; Wang, C. C.; Lee, M. C.: "Integration Object-Oriented Software Testing and Metrics", *International Journal of Software Engineering and Knowledge Engineering*, 7(1), marzo, 1997, págs. 125-144.

Churcher, 95a Churcher, N. I.; Shepperd, M. J.: "Comments on «A Metrics Suite for Object Oriented Design»", *IEEE Transactions on Software Engineering*, 21(3), marzo, 1995, págs. 263-265.

Churcher, 95b Churcher, N. I.; Shepperd, M. J.: "Towards a Conceptual Framework for Object Oriented Software Metrics", *ACM Sigsoft: Software Engineering Notes*, 20(2), abril, 1995, págs. 69-76.

Churchman, 57 Churchman, C. W.; Ackoff, R. L.; Arnoff, E. L.: *Introduction to Operations Research*, Wiley, 1957.

Claffy, 01 Claffy, K. C.: "Measuring the Internet", *IEEE Internet Computing*, enero/febrero, 2000, págs. 73-75.

Clarke, 83 Clarke, A. C.: *2010: Odyssey two*, Grafton, Gran Bretaña, 1983.

Conte, 86 Conte, S. D.; Dunsmore, H. G.; Shen, V. Y.: *Software Engineering Metrics and Models*, Benjaming/Cummings, Menlo Park, California, 1986.

Cooper, 79 Cooper, J. D.; Fisher, M. J.: *Software Quality Management*, Petrocelli Books, 1979.

Cowderoy, 01 Cowderoy, A.: "Quality in a Dotcom Startup– Fact or Fiction?", Proc. *Quality Week* (QW 2001), San Francisco, mayo, 2001.

Cowderoy, 98 Cowderoy, A. J. C.; Donaldson, A. J. M.; Jenkins, J. O.: "A Metrics Framework for Multimedia Creation", Proc. *Fifth International Software Metrics Symposium* (METRICS'98), Bethesda, Maryland, noviembre, 1998.

Cowderoy, 99 Cowderoy, A. J. C.; Donaldson, A. J. M.; Jenkins, J. O.: "Quality on Multimedia", Proc. *Sixth International Symposium on Software Metrics* (METRICS'99), Boca Ratón, Florida, noviembre, 1999.

Cox, 86 Cox, B. J.; Novabilsky, A.: *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

Crespo, 03 Crespo, S.; Fuertes, J. L.: "Diseño e Implementación de la Fase de Análisis de un Sistema de Evaluación de la Calidad para Programas en C++", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, 2003 (en preparación).

Curtis, 79b Curtis, B.; Sheppard, S. B..; Milliman, P.: "Third Time Charm: Stronger Prediction of Programmer Perfomance by Software Complexity Metrics", Proc. *4th International Conference on Software Engineering*, IEEE Computer Society Press, septiembre, 1979, págs. 356-360.

Dahl, 70 Dahl, O. J.; Myhrhaug, B.; Nygaard, K.: *The Simula 67 Common Base Language*, Forskningsveien, Oslo, Noruega, 1970.

De Groot, 88 De Groot, M. H.: *Probabilidad y Estadística*, Addison-Wesley Iberoamericana, 1988.

De Marco, 82 De Marco, T.: *Controlling Sofware Projects: Management, Measurement, and Estimation*, Prentice-Hall, New Jersey, 1982.

Dekkers, 01 Dekkers, T.: "Maximising Customer Satisfaction", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 407-416.

Demichelis, 01 Demichelis, C.: "QoS Assessment in IP Networks", Proc. *International Conference on Advances in Infraestructure for Electronic Business, Science, and Education on the Internet* (SSGRR 2001), L'Aquila, Italia, agosto, 2001.

Deutsch, 88 Deutsch, M. S.; Willis, R. R.: *Software Quality Engineering: a Total Technical and Management Approach*, Prentice-Hall, New Jersey, 1988.

Dhama, 95 Dhama, H.: "Quantitative Models of Cohesion and Coupling in Software", *Journal of Systems and Software*, 29, 1995, págs. 65-74.

Dion, 93 Dion, R.: "Process Improvement and the Corporate Balance Sheet", *IEEE Software*, 10(4), julio, 1993, págs. 28-35.

DoD, 88 DoD: *Defense System Software Quality Program*, DOD-STD-2168, Department of Defense, EE.UU., 1988.

Dromey, 95 Dromey, R. G.: "A Model for Software Produt Quality", *IEEE Transactions on Software Engineering*, 21(2), febrero, 1995, págs. 146-162.

Dumke, 99 Dumke, R.: "Metrics Tools- An Overview", *Metrics News*, 4(1), 1999, págs. 21-28.

Ebert, 92 Ebert, C.: "Correspondence Visualization Techniques for Analyzing and Evaluating Software Measures", *IEEE Transactions on Software Engineering*, 18(11), noviembre, 1992, págs. 1029-1034.

Edgar, 82 Edgar, J. D.: "Controlling Murphy: How to Budget for Program Risk", *Concepts*, verano, 1982, págs. 60-73.

Ellis, 91 Ellis, M. A.; Stroustrup, B.: *The Annotated C++ Reference Manual*, AT&T Bell Laboratories, Addison-Wesley, mayo, 1991.

Emam, 01 Emam, K. E.; Melo, W., Machado, J. C.: "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", *Journal of Systems and Software*, 56(1), febrero, 2001, págs. 63-75.

Embley, 95 Embley, D. W.; Jackson, R. B.; Woodfield, S. N.: "OO Systems Analysis: Is It or Isn't It?", *IEEE Software*, julio, 1995, págs. 19-33.

Etzkorn, 97 Etzkorn, L. H.: *A Metrics-Based Approach to the Automated Identification of Object-Oriented Resuable Software Components*, Tesis Doctoral, Department of Computer Science, School of Graduate Studies, University of Alabama in Huntsville, Alabama, 1997.

Etzkorn, 98 Etzkorn, L; Davis, C.; Li, W: "A Practical Look at the Lack of Cohesion in Methods Metric", *Journal of Object-Oriented Programming*, 11(5), septiembre, 1998, págs. 27-34.

Etzkorn, 99 Etzkorn, L.; Bansiya, J.; Davis, C.: "Design and Code Complexity Metrics for OO Classes", *Journal of Object-Oriented Programming*, 12(1), marzo-abril, 1999, págs. 35-40.

Evanco, 94 Evanco, W. M.; Lacovara, R.: "A Model-Based Framework for the Integration of Software Metrics", *Journal of Systems and Software*, 26, 1994, págs. 77-86.

Fenton, 91 Fenton, N. E.: *Software Metrics: a Rigorous Approach*, Chapman & Hall, Londres, 1991.

Fenton, 94a Fenton, N. E.: "Software Measurement: a Necessary Scientific Basis", *IEEE Transactions on Software Engineering*, 20(3), marzo, 1994, págs. 199-206.

Fenton, 94b Fenton, N.; Pfleeger, S. L.; Glass, R. L.: "Science and Substance: A Challenge to Software Engineers", *IEEE Software*, 11(4), 1994, págs. 86-95.

Ferguson, 97 Ferguson, P.; Humphrey, W. S.; Khajenoori, S.; Macke, S.; Matvya, A.: "Results of Applying the Personal Software Process", *IEEE Computer*, 30(5), mayo, 1997, págs. 24-31.

Finkelstein, 84 Finkelstein, L.: "A Review of the Fundamental Concepts of Measurement", *Measurement*, 2(1), 1984, págs. 25-34.

Ford, 93 Ford, G.: "Lecture Notes on Engineering Measurement for Software Engineers", Educational Materials CMU/SEI-93-EM-9, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, abril, 1993.

Franklin, 00 Franklin, D.; Abrao, A.: "Measuring Software Agent's Intelligence", Proc. *International Conference: Advances in Infraestructure for Electronic Business, Science and Education on the Internet*, L'Aquila, Italia, agosto, 2000.

French, 99 French, V. A.: "Establishing Software Metric Thresholds", Proc. *9th International Workshop on Software Measurement* (IWSM'99), Lac Supérieur, Quebec, Canadá, septiembre, 1999, págs. 43-50.

Frolund, 98 Frolund, S.; Koistinen, J.: "Quality of Service Specification in Distributed Object Systems Design", Proc. *4th USENIX Conference on Object-Oriented Technologies* (COOTS), Santa Fe, Nuevo México, abril, 1998.

Fuente, 99 de la Fuente, J. L.: *Módulo Flexible para la Creación de Cursos*, Trabajo Fin de Carrera, Facultad de Informática, Universidad Politécnica de Madrid, julio, 1999.

Fuertes, 92 Fuertes, J. L.: *Sistema Configurable para la Construcción y Consulta de Árboles de Decisión*, Trabajo Fin de Carrera, Facultad de Informática, Universidad Politécnica de Madrid, noviembre, 1992.

Fuertes, 93 Fuertes, J. L.: *Programación C++*, Unidad Didáctica del Máster de Ingeniería del Software e Ingeniería del Conocimiento, Facultad de Informática, Universidad Politécnica de Madrid, Madrid, 1993.

Fuertes, 99 Fuertes, J. L.: "Calidad del Software", *Cuadernos de Informática*, 1, Federación Española de Sociedades de Informática, Madrid, marzo, 1999, págs. 19-72.

Gannon, 86 Gannon, J. D.; Katz, E. E.; Basili, V. R.: "Metrics for Ada Packages: an Initial Study", *Communications of the ACM*, 29(7), julio, 1986, págs. 616-623.

Garvin, 84 Garvin, D.: "What Does 'Product Quality' Really Mean?", *Sloan Management Review*, 4, otoño, 1984.

Genero, 99 Genero, M.; Manso, M.; Piattini, M.; García, F. J.: "Assessing the Quality and the Complexity of OMT Models", Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 99-109.

Gilb, 88 Gilb, T.: *Principles of Software Engineering Management*, Addison-Wesley, Wokingham, Reino Unido, 1988.

Gillibrand, 98 Gillibrand, D.; Liu, K.: "Quality Metrics for Object-Oriented Design", *Journal of Object-Oriented Programming*, enero, 1998, págs. 56-59.

Gillies,92 Gillies, A. C.: *Software Quality: Theory and Management*, Chapman & Hall, Londres, 1992.

Goldberg, 83 Goldberg, A.; Robson, D.: *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

González, 93 González, J.: "Calidad del Software", *ComputerWorld*, 531, abril, 1993, págs. 25-27.

González, 95 González, R. R.: "A Unified Metric of Software Complexity: Measuring Productivity, Quality, and Value", *Journal of Systems and Software*, 29, 1995, págs. 17-37.

Gordon, 79a Gordon, R. D.: "Measuring Improvements in Program Clarity", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 79-90.

Gould, 81 Gould, S. J.: *The mismeasure of Man*, W. W. Norton and Co., Nueva York, 1981.

Govindaraj, 98 Govindaraj, T.: "Characterizing Complexity in the Design of Human-Integrated Systems", Proc. *IEEE Systems, Man, and Cybernetics 1998* (IEEE SMC'98), San Diego, octubre, 1998, págs. 985-988.

Grady, 87 — Grady, R. B.; Caswell, D. C.: *Software Metrics: Establishing a Company-Wide Program*, Englewood Cliffs, Prentice Hall, New Jersey, 1987.

Gunning, 68 — Gunning, R.: *The Technique of clear writing*, McGraw-Hill, 1968.

Hall, 84 — Hall, N. R.; Preiser, S.: "Combined Network Complexity Measures", *IBM Journal of Research and Development*, 28(1), enero, 1984, págs. 15-27.

Halstead, 77 — Halstead, M. H.: *Elements of Software Science*, Elsevier North-Holland, Nueva York, 1977.

Hamer, 82 — Hamer, P. G.; Frewin, G. D.: "M. H. Halstead's Software Science: A Critical Examination", Proc. *IEEE 6th International Conference on Software Engineering*, 1982, págs. 197-206.

Hamilton, 78 — Hamilton, P. A.; Musa, J. D.: "Measuring Reliability of Computation Center Software", Proc. *3rd International Conference on Software Engineering*, mayo, 1978, págs. 29-36.

Hansen, 78 — Hansen, W.: "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)", *ACM Sigplan Notices*, 13(3), marzo, 1978, págs. 29-33.

Harland, 85 — Harland, D.: "An Alternative View of Polymorphism", *ACM Sigplan Notices*, 20(10), octubre, 1985.

Harrison, 00 — Harrison, R.; Counsell, S.; Nithi, R.: "Experimental Assesment of the Effect of Inheritance on the Maintainabiltiy of Object-Oriented Systems", *Journal of Systems and Software*, 52(2/3), junio, 2000, págs. 173-179.

Harrison, 81a — Harrison, W.; Magel, K.: "A Topological Analysis of the Complexity of Computer Programs with Less than Three Binary Branches", *ACM Sigplan Notices*, 16(4), abril, 1981, págs. 51-61.

Hausen, 00 — Hausen, H.-L.: "On a Standards Based Quality Framework for Web Portals", Proc. *4th International Software Quality Week Europe* (QWE 2000), Bruselas, noviembre, 2000.

Hausen, 89 — Hausen, H.-L.: "Generic Modelling of Software Quality", en *Measurement for Software Control and Assurance*, B. A. Kitchenham y B. Littlewood (eds.), Proc. Centre for Software Reliability Conference, Elsevier Science Publishers, 1989, págs. 201-241.

Henderson, 01 — Henderson, P.; Howard, Y. M.; Walters, R. J.: "A Tool for Evaluation of the Software Development Process", *The Journal of Systems and Software*, 59(3), 2001, págs. 355-362.

Henderson-Sellers, 94a — Henderson-Sellers, B.; Tegarden, D.: "The Theoretical Extension of Two Versions of Cyclomatic Complexity to Multiple Entry/Exis Modules", *Software Quality Journal*, 3(4), 1994, págs. 253-269.

Henderson-Sellers, 96a — Henderson-Sellers, B.: "Managing an Object-Oriented Metrics Programme", Proc. *SIGS Conferences*, Reino Unido, 1996, págs. 39-44.

Henry, 79 — Henry, S. M.: *Information Flow Metrics for the Evaluation of Operating Systems' Structure*, Tesis Doctoral, Iowa State University, Ames, 1979.

Henry, 81a — Henry, S.; Kafura, D.: "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, 7(5), septiembre, 1981, págs. 510-518.

Henry, 81b — Henry, S.; Kafura, D.; Harris, K.: "On the Relationships Among Three Software Metrics", *ACM Performance Evaluation Review*, 10(1), primavera, 1981, págs. 81-88.

Henry, 84 — Henry, S.; Kafura, D.: "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", *Software Practice and Experience*, 14(6), junio, 1984, págs. 561-573.

Henry, 90a — Henry, S.; Selig, C.: "Predicting Source-Code Complexity at the Design Stage", *IEEE Software*, 7(2), marzo, 1990, págs. 36-44.

Hitz, 96 — Hitz, M.; Montazeri, B.: "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective", *IEEE Transactions on Software Engineering*, 22(4), abril, 1996, págs. 267-271.

HP, 92 Hewlett-Packard: *C++/Object-Oriented Programming. Instructor Guide*, HP Computer Systems, Training Course, diciembre, 1992.

Humphrey, 89 Humphrey, W. S.: *Managing the Software Process*, Addison-Wesley, Reading, Massachusetts, 1989.

Humphrey, 95a Humphrey, W. S.: *A Discipline for Software Engineering*, Addison-Wesley, 1995.

Humphrey, 95b Humphrey, W. S.: "The Business of Software Process Improvement", *European Conference on Software Process Improvement (SPI'95)*, Barcelona, noviembre, 1995.

IEEE, 90 IEEE: *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Std. 610.12-1990, 1990.

IEEE, 93 IEEE: *IEEE Standard for a Software Quality Metrics Methodology*, IEEE Std. 1061-1992, Nueva York, marzo, 1993.

Ince, 89a Ince, D.: "Software Metrics", en *Measurement for Software Control and Assurance*, Proc. Centre for Software Reliability Conference, B. A. Kitchenham y B. Littlewood (eds.), Elsevier Science Publishers, 1989, págs. 27-62.

Ince, 90 Ince, D.: "Software Metrics: Introduction", *Information and Software Technology*, 32(4), mayo, 1990, págs. 297-303.

ISO, 86 ISO: *Quality– Vocabulary*, ISO 8402-1986, International Organization of Standardization, 1986.

ISO, 91 ISO: *Information Technology– Software Product Evaluation– Quality Characteristics and Guidelines for their Use*, ISO/IEC 9126, International Organization of Standardization, 1991.

ISO, 94 ISO: *Quality– Vocabulary*, ISO 8402-1994, International Organization of Standardization, 1994.

ISO, 97 ISO: *Software Quality Characteristics and Metrics– Part 1: Quality Characteristics and Sub-characteristics*, ISO/IEC 9126-1, International Organization of Standardization, 1997.

ISO, 98 ISO: *Information Technology– Programming Languages– C++*, ANSI/ISO/IEC 14882, International Organization of Standardization, 1998.

Jézéquel, 97 Jézéquel, J.-M.; Meyer, B.: "Design by Contract: The Lessons of Ariane", *IEEE Computer*, 30(1), enero, 1997, págs. 129-130.

Jilani, 01 Jilani, L. L.; Desharnais, J.; Mili, A.: "Defining and Applying Measures of Distance Between Specifications", *IEEE Transactions on Software Engineering*, 27(8), 2001, págs. 673-703.

Jones, 01 Jones, C.: "Software Quality in 2001– A Survey of the State of the Art", Proc. *International Conference on Practical Software Quality Techniques* (PSQT 2001 East), Orlando, Florida, abril, 2001.

Jones, 97a Jones, C.: "Strengths and Weaknesses of Software Metrics", *American Programmer*, 10(11), noviembre, 1997, págs. 44-49.

Jones, 98 Jones, C.: "Minimizing the Risks of Software Development", *Cutter IT Journal*, 11(6), junio, 1998, págs. 13-21.

Juran, 74 Juran, J. M.: "Basic Concepts", en *Quality Control Handbook*, J. M. Juran et al. (eds.), McGraw-Hill, Nueva York, 1974.

Kafura, 85 Kafura, D.; Canning, J. T.: "A Validation of Software Metrics Using Many Metrics and Two Resources", Proc. *8th International Conference on Software Engineering*, IEEE Computer Society Press, agosto, 1985, págs. 378-385.

Kafura, 87 Kafura, D.; Reddy, G. R.: "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, 13(3), marzo, 1987.

Kececi, 01 — Kececi, N.; Abran, A.: "An Integrated Measurement for Functional Requirement Correctness", en Dumke/Abran *Current Trends in Software Measurement*, Shaker Publ., Aquisgrán, Alemania, 2001, págs. 200-219.

Keller, 90 — Keller, S.; McNulty, M. S.; Gustafson, D. A.: "Stochastic Models for Software Science", *The Journal of Systems and Software*, 12, 1990, págs. 59-68.

Kemerer, 99 — Kemerer, C. F.: "Metrics for Object Oriented Software: A Retrospective", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Measurement for Object-Oriented Software Projects Workshop, Boca Ratón, Florida, noviembre, 1999.

Khoshafian, 90 — Khoshafian, S.; Abnous, R.: *Object Orientation. Concepts Language, Database, User Interfaces*, John Wiley & Sons, 1990.

Khoshgoftaar, 00 — Khoshgoftaar, T. M.; Allen, E. B.; Jones, W. D.; Hudepohl, J. P.: "Accuracy of Software Quality Models over Multiple Releases", *Annals of Software Engineering*, 9, 2000, págs. 103-116.

Khoshgoftaar, 94a — Khoshgoftaar, T. M.; Munson, J. C.; Lanning, D. L.: "Alternative Approaches for the Use of Metrics to Order Programs by Complexity", *Journal of Systems and Software*, 24, 1994, págs. 211-221.

Khoshgoftaar, 94b — Khoshgoftaar, T. M.; Oman, P.: "Software Metrics: Charting the Course", *IEEE Computer*, 27(9), septiembre, 1994, págs. 13-15.

Khoshgoftaar, 99 — Khoshgoftaar, T. M.; Allen, E. B.; Naik, A.; Jones, W. D.; Hudepohl, J. P.: "Using Classification Trees for Software Quality Models: Lessons Learned", *International Journal of Software Engineering and Knowledge Engineering*, 9(2), abril, 1999, págs. 217-231.

Kincaid, 81 — Kincaid, J. P.; Aagard, J. A.; O'Hara, J. W.; Cottrell, L. K.: "Computer Readability Editing System", *IEEE Transactions on Professional Communication*, 24(1), 1981, págs. 38-41.

Kirsopp, 99 — Kirsopp, C.; Shepperd, M.; Webster, S.: "An Empirical Study into the Use of Measurement to Support OO Design Evaluation", Proc. *Sixth International Software Metrics Symposium* (METRICS'99), Boca Ratón, Florida, noviembre, 1999, págs. 230-241.

Kitchenham, 89a — Kitchenham, B. A.: "Software Quality Assurance", *Microprocessors and Microsystems*, 13(6), julio/agosto, 1989, págs. 373-381.

Kokol, 01 — Kokol, P.; Podgorelec, V.; Pighin, M.: "Using Software Metrics and Evolutionary Decision Trees for Software Quality Control", Proc. *European Software Control and Metrics*, (ESCOM 2001), Londres, abril, 2001, págs. 453-461.

Kosarajo, 74 — Kosarajo, S.; Ledgard, H.: "Concepts in Quality Design", NBS Technical Note 842, agosto, 1974.

Krasner, 88 — Krasner, G. E.; Pope, S. T.: "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, agosto/septiembre, 1988, págs. 29-49.

Lake, 92a — Lake, A.; Cook, C.: "A Software Complexity Metric for C++", Proc. *Fourth Annual Workshop on Software Metrics*, Oregon, marzo, 1992.

Lakshminara-yana, 99 — Lakshminarayana, A.; Newman, T. S.: "Principal Component Analysis of Lack of Cohesion in Methods (LCOM) Metrics", Technical Report TR-UAH-CS-1999-01, Department of Computer Science, University of Alabama in Huntsville, Alabama, 1999.

Lanning, 94 — Lanning, D. L.; Khoshgoftaar, T. M.: "Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty", *Computer*, 27(9), septiembre, 1994, págs. 35-40.

Laranjeira, 90 — Laranjeira, L. A.: "Software Size Estimation of Object-Oriented Systems", *IEEE Transactions on Software Engineering*, 16(5), mayo, 1990, págs. 510-522.

Lassez, 81 — Lassez, J.-L.: "A Critical Examination of Software Science", *Journal of Systems and Software*, 2(2), 1981, págs. 105-112.

Lee, 01 — Lee, Y.; Chang, K. H.; Umphress, D. A.; Hendirx, D.; Cross, J. H.: "Automated Tool for Software Quality Measurement", Proc. *13th International Conference on Software Engineering and Knowledge Engineering*, Buenos Aires, Argentina, junio, 2001, págs. 196-202.

Lejter, 92 — Lejter, M.; Meyers, S.; Reiss, S. P.: "Support for Maintaining Object-Oriented Programs", *IEEE Transactions on Software Engineering*, 18(12), diciembre, 1992, págs. 1045-1052.

Lewis, 91 — Lewis, J. A.; Henry, S. M.; Kafura, D. G.; Schulman, R. S.: "An Empirical Study of the Object-Oriented Paradigm and Software Reuse", Proc. 6th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'91), *ACM Sigplan Notices*, 26(11), noviembre, 1991, págs. 184-196.

Li, 87 — Li, H. F.; Cheung, W. K.: "An Empirical Study of Software Metrics", *IEEE Transactions on Software Engineering*, 13(6), junio, 1987, págs. 697-708.

Li, 93a — Li, W.; Henry, S.: "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, 23(2), noviembre, 1993, págs. 111-122.

Li, 95 — Li, W.; Henry, S.; Kafura, D.; Schulman, R.: "Measuring Object-Oriented Design", *Journal of Object-Oriented Programming*, 8(4), julio-agosto, 1995, págs.48-55.

Lieberherr, 88 — Lieberherr, K.; Holland, I.; Riel, A.: "Object-Oriented Programming: An Objective Sense of Style", Proc. 3rd Conference on Object Oriented Programming, Systems, Languaje, and Applications (OOPSLA'88), *ACM Sigplan Notices*, 23(11), noviembre, 1988, págs. 323-334.

Lieberherr, 89 — Lieberherr, K. J.; Holland, I. M.: "Assuring Good Style for Object-Oriented Programs", *IEEE Software*, 6(5), septiembre, 1989, págs. 38-48.

Liggesmeyer, 95 — Liggesmeyer, P.: "A Set of Complexity Metrics for Guiding the Software Test Process", *Software Quality Journal,* 4, 1995, págs. 257-273.

Littlefair, 01 — Littlefair, T.: "An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment", Tesis Doctoral, Faculty of Communications, Health and Science, Edith Cowan University, Australia, 18 de junio de 2001.

Littlewood, 78 — Littlewood, B.: "How to Measure Software Reliability, and How Not to...", Proc. *3rd International Conference on Software Engineering*, IEEE Computer Society Press, California, mayo, 1978, págs. 37-45.

Lorenz, 93 — Lorenz, M.: *Object-Oriented Software Development*, Prentice Hall, 1993.

Mallepalli, 01 — Mallepalli, C.; Tian, J.: "Measuring and Modeling Usage and Reliability for Statistical Web Testing", *IEEE Transactions on Software Engineering*, 27(11), 2001, págs. 1023-1036.

Mansfield, 63 — Mansfield, M.: *Introduction to Topology*, Van Nostrand, Princeton, New Jersey, 1963

Martínez, 01 — Martínez, L.: "Práctica de Entornos de Programación: Simulación de un Gas", http://rosada.ls.fi.upm.es/asig/entornos/practicacpp.html, Facultad de Informática, Universidad Politécnica de Madrid, descargado el 29-5-2001.

Mata-Toledo, 92 — Mata-Toledo, R. A.; Gustafson, D. A.: "A factor Analysis of Software Complexity Measures", *Journal of Systems and Software*, 17, 1992, págs. 267-273.

Mayer, 92 — Mayer, A.; Sykes, A. M.: "Statistical Methods for the Analysis of Software Metrics Data", *Software Quality Journal,* 1, 1992, págs. 209-223.

Mayrand, 00 — Mayrand, J.; Patenaude, J.-F.; Merlo, E.; Dagenais, M.; Laguë, B.: "Software Assessment Using Metrics: A Comparison Across Large C++ and Java Systems", *Annals of Software Engineering*, 9, 2000, págs. 117-141.

McCabe, 01 — McCabe, T. J.: "McCabe Quality Tools", descargado el 17-1-01.

McCabe, 76 — McCabe, T. J.: "A Complexity Measure", *IEEE Transactions on Software Engineering*, 2(4), diciembre, 1976, págs. 308-320.

McCabe, 89 McCabe, T. J.; Butler, C. W.: "Design Complexity Measuremet and Testing", *Communications of the ACM*, 32(12), diciembre, 1989, págs. 1415-1425.

McCall, 77 McCall, J. A.; Richards, P. K.; Walters, G. F.: "Factors in Software Quality", Vol. I, II, III, Final Technical Report, RADC-TR-77-369, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, Nueva York, noviembre, 1977.

McCall, 79 McCall, J. A.: "An Introduction to Software Quality Metrics", en *Software Quality Management*, J. D. Cooper y M. J. Fisher (eds.), Petrocelli Books, 1979, págs. 127-142.

McClure, 78 McClure, C. L.: "A Model for Program Complexity Analysis", Proc. *3rd International Conference on Software Engineering*, mayo, 1978, págs. 149-157.

Mehndiratta, 90 Mehndiratta, B.; Grover, P. S.: "Software Metrics- An Experimental Analysis", *ACM Sigplan Notices*, 25(2), febrero, 1990, págs. 35-41.

Meyers, 00 Meyers, S.; Klaus, M.: "A First Look at C++ Program Analyzers", http://www.teleport.com/~smeyers/ddjpaper1.htm, descargado el 1-2-2000.

Meyers, 97 Meyers, S.; Klaus, M.: "Examining C++ Program Analyzers", *Dr. Dobb's Journal*, 22(2) febrero, 1997, págs. 68-75.

Miller, 56 Miller, G. A.: "The Magical Number Seven, Plus or Minus Two: some Limits on our Capacity for Processing Information", *The Psychological Review*, 63, 1956, págs. 81-97.

Mills, 88 Mills, E. E.: "Software Metrics", Curriculum Module SEI-CM-12-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, diciembre, 1988.

Mills, 90 Mills, H. D.; Dyson, P. B.: "Using Metrics to Quantify Development", *IEEE Software*, 7(2), marzo, 1990, págs. 15-16.

Molloy, 95 Molloy, T.; Ryland, H. A.; Shephard, B. J.: "C++ Coding Standard", Documento STD/D/001, Westinghouse Signals, junio, 1995.

Montes, 94 Montes, C.: *MITO: Método de Inducción Total*, Tesis Doctoral, Facultad de Informática, Universidad Politécnica de Madrid, febrero, 1994.

Mora, 92 Mora, A.; Cosculluela, A.: "A Metrics Approach to the Software Reuse Problem" en *Software Quality: the Challenge in the 90s*, INTA, Proc. 3rd European Conference on Software Quality, Madrid, noviembre, 1992.

Morcillo, 83 Morcillo, J.; Fernández, M.: *Química*, Anaya, S.A., Madrid, 1983.

Moulet, 92 Moulet, A.: "Todos los Recursos al Alcance de la Mano: Llamadas a Procedimientos Remotos (RPCs)", *Comunicaciones World*, 60, septiembre, 1992, págs. 44-45.

Myers, 97 Myers, B. L.; Kappelman, L. A.; Prybutok, V. R.: "A Comprehensive Model for Assessing the Quality and Productivity of the Information Systems Function: Toward a Theory for Information Systems Assessment", *Information Resources Management Journal*, 10(1), invierno, 1997, págs. 6-25.

Neumann, 88 Neumann, P. G.: "Risks to the Public in Computer and Related Systems", *ACM Sigsoft: Software Engineering Notes*, abril, 1988, págs. 5-18.

OMG, 01 Object Management Group: "The Common Object Request Broker: Architecture and Specification. Revision 2.4.2", http://www.omg.org/cgi-bin/doc?formal/01-02-33, descargado el 3-5-2001, febrero, 2001.

Oviedo, 80 Oviedo, E. I.: "Control Flow, Data Flow and Program Complexity", Proc. 4th *International Computer Software and Applications Conference* (COMPSAC'80), 1980, págs. 146-152.

Page-Jones, 92 Page-Jones, M.: "Comparing Techniques by Means of Encapsulation and Connascence", *Communications of the ACM*, 35(9), septiembre, 1992, págs. 147-151.

Pant, 96 — Pant, Y.; Henderson-Sellers, B.; Verner, J. M.: "Generalization of Object-Oriented Components for Reuse: Measurements of Effort and Size Change", *Journal of Object-Oriented Programming*, 9(2), mayo, 1996, págs. 19-31,41.

Parasoft, 01 — Parasoft: "C++ Test 2.0", http://www.parasoft.com, descargado el 11-7-2001.

Paulk, 91 — Paulk, M.: "Capability Maturity Model", Technical Report CMU/SEI-91-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1991.

Perlis, 81 — Perlis, A.; Sayward, F.; Shaw, M.: *Software Metrics: an Analysis and Evaluation*, Cambridge, MIT Press, Massachusetts, 1981.

Perry, 91 — Perry, W. E.: *Quality Assurance for Information Systems, Methods, Tools and Techniques*, QED Information Sciences, Massachusetts, 1991.

Petzold, 96 — Petzold, C.: *Programación en Windows 95*, McGraw-Hill, Microsoft Press, 1996.

Pfleeger, 90a — Pfleeger, S. L.; McGowan, C. L.: "Software Metrics in a Process Maturity Framework", *Journal of Systems and Software*, 12, julio, 1990, págs. 255-261.

Pfleeger, 92 — Pfleeger, S. L.; Fitzgerald, J. C.; Rippy, D. A.: "Using Multiple Metrics for Analysis of Improvement", *Software Quality Journal,* 1, 1992, págs. 27-36.

Pfleeger, 95 — Pfleeger, S. L.: "Maturity, Models, and Goals: How to Build a Metrics Plan", *Journal of Systems and Software*, 31, 1995, págs. 143-155.

Pfleeger, 98 — Pfleeger, S. L.: "Software Quality", *Dr. Dobb's Journal*, 283, marzo, 1998, págs. 22-29.

Pooley, 02 — Pooley, R.; Senior, D.; Christie, D.: "Collecting and Analyzing Web-Based Project Metrics", *IEEE Software*, enero-febrero, 2002, págs. 52-58.

Porter, 90 — Porter, A. A.; Selby, R. W.: "Empirically Guided Software Development Using Metric-Based Classification Trees", *IEEE Software*, 7(2), marzo, 1990, págs. 46-54.

QoSDOS, 01 — "2nd Workshop on Quality of Service in Distributed Object Systems", http://www.dist-systems.bbn.com/confs/2001/QoSDOS, descargado el 20-3-2001.

Quinlan, 79 — Quinlan, J. R.: "Discovering Rules by Induction from Large Collections of Examples", *Expert Systems in the Microelectronic Age*, Edinburgh University Press, 1979, págs. 168-201.

Quinlan, 86 — Quinlan, J. R.: "Induction of Decision Trees", *Machine Learning*, 1(1), 1986, págs. 81-106.

Quinlan, 93 — Quinlan, J. R.: *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

RAE, 92 — Real Academia Española: *Diccionario de la Lengua Española*, vigésima primera edición, Espasa Calpe, 1992.

Ramamoorthy, 75 — Ramamoorthy, C. V.; Ho, S. F.: "Testing Large Software with Automated Software Evaluation Systems", *IEEE Transactions on Software Engineering*, 1(2), marzo, 1975.

Ramamurthy, 88 — Ramamurthy, B.; Melton, A.: "A Synthesis of Software Science Measures and the Cyclomatic Number", *IEEE Transactions on Software Engineering*, 14(8), agosto, 1988, págs. 1116-1121.

Ramírez, 03 — Ramírez, J.; Fuertes, J. L.: "Evaluación de la Calidad en Sistemas Orientados a Objetos: Manual de Calidad y Medidas Basadas en la Ciencia del *Software*", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, 2003 (en preparación).

Rifkin, 91 — Rifkin, S.; Cox, C.: "Measurement in Practice", Technical Report CMU/SEI-91-TR-16, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, julio, 1991.

Río, 03 — del Río, J. M.; Fuertes, J. L.: "Diseño e Implementación del Cálculo de Medidas en un Modelo de Calidad para un Sistema de Evaluación de la Calidad", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, abril, 2003.

Rising, 92 — Rising, L. S.; Calliss, F. W.: "Problems with Determining Package Cohesion and Coupling", *Software- Practice and Experience*, 22(7), julio, 1992, págs. 553-571.

Rising, 94 — Rising, L. S.; Calliss, F. W.: "An Information-Hiding Metric", *Journal of Systems and Software*, 26, 1994, págs. 211-220.

Roberts, 79a — Roberts, F. S.: *Measurement Theory with Applications to Decision Making, Utility, and the Social Sciences*, Addison-Wesley, Reading, Massachusetts, 1979.

Roberts, 93 — Roberts, T.: "Workshop Report- Metrics for Object-Oriented Software Development", 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Addendum to the Proceedings, *OOPS Messenger*, 4, abril, 1993, págs. 97-100.

Roesthlisberger, 39 — Roesthlisberger, F. J.; Dickson, W. J.: *Management and the Worker*, Harvard University Press, Cambridge, 1939.

Rombach, 87 — Rombach, H. D.: "A Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering*, 13(3), marzo, 1987, págs. 344-354.

Roper, 95 — Roper, M.; Wilson, J.; Brooks, A.; Miller, J.; Wood, M.: "A Simple Dynamic Analyser for C++", Research Report RR/95/194, Dept. Computer Science, University of Strathclyde, Glasgow, Reino Unido, 1995.

Rosenberg, 99 — Rosenberg, L.; Stapko, R.; Gallo, A.: "Applying Object Oriented Metrics", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Measurement for Object-Oriented Software Projects Workshop, Boca Ratón, Florida, noviembre, 1999.

Rothfeder, 88 — Rothfeder, J.: "It's Late, Costly, Incompetent —But Try Firing a Computer System", *Business Week*, 7 de noviembre, 1988, págs. 164-165.

Rovira, 03 — Rovira, J. L.; Fuertes, J. L.: "Herramienta Gráfica para el Diseño e Implementación de Sistemas Orientados a Objetos en Java con Evaluación de su Calidad", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, 2003 (en preparación).

Rubey, 68 — Rubey, R.; Hartwick, R.: "Quantitative Measurement of Program Quality", Proc. *23rd ACM National Conference*, 1968, págs. 671-677.

Rubin, 83 — Rubin, H. A.: "Macro-Estimation of Software Development Parameters: The ESTIMACS System", Proc. *SOFTAIR: A Conference on Software Development Tools, Techniques, and Alternatives*, IEEE Computer Society Press, Nueva York, julio, 1983, págs. 109-118.

Rule, 01 — Rule, P. G.: "Using Measures to Understand Requirements", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 327-335.

Rumbaugh, 91 — Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

Saaty, 90 — Saaty, T. L.: "How to Make a Decision: The Analytic Hierarchy Process", *European Journal of Operational Research*, 48, 1990, págs. 9-26.

Samadzadeh, 91 — Samadzadeh, M. H.; Nandakumar, K.: "A Study of Software Metrics", *Journal of Systems and Software*, 16(3), noviembre, 1991, págs. 229-234.

Samaraweera, 98 — Samaraweera, L. G.; Harrison, R.: "Evaluation of the Functional and Object-Oriented Programming Paradigms: A Replicated Experiment", *Software Engineering Notes*, 23(4), 1998, págs. 38-43.

Sanders, 94 — Sanders, J.; Curran, E.: *Software Quality: A Framework for Success in Software Development and Support*, Addison-Wesley, Gran Bretaña, 1994.

Schach, 96 — Schach, S. R.: "The Cohesion and Coupling of Objects", *Journal of Object-Oriented Programming*, 9(1), enero, 1996, págs. 48-50.

Schneidewind, 00 — Schneidewind, N. F.: "Software Quality Control and Prediction Model for Maintenance", *Annals of Software Engineering*, 9, 2000, págs. 79-101.

Schneidewind, 79 — Schneidewind, N. F.; Hoffmann, H. M.: "An Experiment in Software Error Data Collection and Analysis", *IEEE Transactions on Software Engineering*, 5(3), mayo, 1979, págs. 276-286.

Schulmeyer, 87 — Schulmeyer, G. G.; McManus, J. I.: *Handbook of Software Quality Assurance*, Van Nostrand Reinhold Company, Nueva York, 1987.

Schulmeyer, 90 — Schulmeyer, G. G.: *Zero Defect Software*, McGraw-Hill, 1990.

Schuyler, 96 — Schuyler, J. R.: *Decision Analysis in Projects*, Project Management Institute, diciembre, 1996.

Selby, 88 — Selby, R. W.; Porter, A. A.: "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", *IEEE Transactions on Software Engineering*, 14(12), diciembre, 1988, págs. 1743-1757.

Shannon, 71 — Shannon, C. E.; Weaver, W.: *The Mathematical Theory of Communication*, University of Illinois Press, 1971.

Sharble, 93 — Sharble, R. C.; Cohen, S. S.: "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods", *ACM Sigsoft: Software Engineering Notes*, 18(2), abril, 1993, págs. 60-73.

Sharif, 01 — Sharif, H.; Chen, B.: "End-to-end QoS Requirements for Real-Time Video Streaming in Internet2", Proc. *International Conference on Advances in Infraestructure for Electronic Business, Science, and Education on the Internet* (SSGRR 2001), L'Aquila, Italia, agosto, 2001.

Shaw, 84 — Shaw, M.: "Abstraction Techniques in Modern Programming Languages", *IEEE Software*, 1, 1984, págs. 10-26.

Shaw, 89 — Shaw, W. H.; Howatt, J. W.; Maness, R. S.; Miller, D. M.: "A Software Science Model of Compile Time", *IEEE Transactions on Software Engineering*, 15(5), 1989, págs. 543-549.

Shen, 83 — Shen, V. Y.; Conte, S. D.; Dunsmore, H. E.: "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", *IEEE Transactions on Software Engineering*, 9(2), marzo, 1983, págs. 155-165.

Shen, 85 — Shen, V. T.; Yu, T.; Thebaut, S. M.; Paulsen, L. R.: "Identifying Error-Prone Software– An Empirical Study", *IEEE Transactions on Software Engineering*, 11(4), abril, 1985, págs. 317-324.

Shepperd, 88 — Shepperd, M. J.: "A Critique of Cyclomatic Complexity as a Software Metric", *Software Engineering Journal,* 3(2), 1988, págs. 1-8.

Shepperd, 94 — Shepperd, M.; Ince, D. C.: "A Critique of Three Metrics", *Journal of Systems and Software*, 26, 1994, págs. 197-210.

Shih, 97 — Shih, T. K.; Wang, C.-C.; Chung, C.-M.: "Using Z to Specify Object-Oriented Software Complexity Measures", *Information and Software Technology*, 39(8), agosto, 1997, págs. 515-529.

Sloman, 98 — Sloman, A.: "Damasio, Descartes, Alarms and Meta-management", Proc. *IEEE Systems, Man, and Cybernetics 1998* (IEEE SMC'98), San Diego, octubre, 1998, págs. 2652-2657.

Smith, 89 — Smith, D. J.; Wood, K. B.: *Engineering Quality Software: A Review of Current Practices, Standards and Guidelines Including New Methods and Develpment Tools*, 2ª edición, Elsevier Science Publishers, Essex, Inglaterra, 1989.

Spolverini, 00 — Spolverini, M.: "Measuring and Improving the Quality of Web Site Applications", Proc. *4th International Software Quality Week Europe* (QWE 2000), Bruselas, noviembre, 2000.

Stålhane, 92 — Stålhane, T.: "The REBOOT Quality Model and some Extensions" en *Software Quality Principles and Techniques*, European Research Consortium for Informatics and Mathematics, Pisa, 1992, págs. 133-152.

Stavroulakis, 00 Stavroulakis, P.; Sandalidis, H. G.: "Quality Considerations of Large Scale Systems Used in Wireless Networks", Proc. *International Conference: Advances in Infraestructure for Electronic Business, Science and Education on the Internet*, L'Aquila, Italia, agosto, 2000.

Stiglic, 95 Stiglic, B.; Hericko, M.; Rozman, I.: "How to Evaluate Object-Oriented Software Development?", *ACM Sigplan Notices*, 30(5), mayo, 1995, págs. 3-10.

Stoeffler, 92 Stoeffler, K.: *Investigations to Software Metrics for Nonprocedural Languages*, Tesis de Máster, TU Magdeburg, 1992.

Stroustrup, 86 Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 1986.

Stroustrup, 88a Stroustrup, B.: "What is Object-Oriented Programming", *IEEE Software*, 5(3), mayo, 1988, págs. 10-20.

Stroustrup, 88b Stroustrup, B.: "Parameterized Types for C++", *USENIX C++ Conference*, Denver, 1988.

Stroustrup, 97 Stroustrup, B.: *The C++ Programming Language*, 3ª ed., Addison-Wesley, 1997.

Szulewski, 81 Szulewski, P. A.: "The Measurement of Software Science Parameters in Software Designs", *ACM Performance Evaluation Review*, 10(1), spring, 1981, págs. 89-94.

Takahashi, 97 Takahashi, R.: "Software Quality Classification Model Based on McCabe's Complexity Measure", *Journal of Systems and Software*, 38(1), julio, 1997, págs. 61-69.

Taylor, 90 Taylor, D. A.: *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, Reading, Massachusetts, 1990.

Tegarden, 95 Tegarden, D. P.; Sheetz, S. D.; Monarchi, D. E.: "A Software Complexity Model of Object-Oriented Systems", *Decision Support Systems: the International Journal*, 13, 1995, págs. 241-262.

Tervonen, 92 Tervonen, I.: "Formal Reviews with Design Rationale" en *Software Quality: the Challenge in the 90s*, INTA, Proc. *3rd European Conference on Software Quality*, Madrid, noviembre, 1992.

Tian, 92 Tian, J.; Zelkowitz, M. V.: "A Formal Program Complexity Model and its Application", *Journal of Systems and Software*, 17, 1992, págs. 253-266.

Tian, 95 Tian, J.; Zelkowitz, M. V.: "Complexity Measure Evaluation and Selection", *IEEE Transactions on Software Engineering*, 21(8), agosto, 1995, págs. 641-650.

Turban, 93 Turban, E.; Meredith, J. R.: *Fundamentals of Management Science*, 6ª edición, McGraw-Hill, diciembre, 1993.

Utgoff, 01 Utgoff, P.: "Machine Learning Laboratory: Incremental Decision Tree Induction", http://www.cs.umass.edu/~lrn/iti/index.html, Dept. of Computer Science, University of Massachusetts, descargado el 30-5-2001, marzo, 2001.

Verkamo, 01 Verkamo, A. I.; Gustafsson, J.; Nenonen, L.; Paakki, J.: "Measuring Design Diagrams for Product Quality Evaluation", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 357-366.

Vincent, 88a Vincent, J. P.; Waters, A.; Sinclair, J.: *Software Quality Assurance. Practice and Implementation*, Vol. I, Englewood Cliffs, New Jersey, 1988.

Ward, 91 Ward, W. T.: "Calculating the Real Costs of Software Defects", *Hewlett-Packard Journal*, 42(4), octubre, 1991, págs. 55-59.

Watts, 87 Watts, R. A.: *Measuring Software Quality*, The National Computing Centre Limited, Oxford, 1987.

Weber, 91 Weber, C. V.; Paulk, M. C.; Wise, C. J.; Withey, J. V.: "Key Practices for the Capability Maturity Model", Technical Report CMU/SEI-91-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, agosto, 1991.

Weiss, 99 Weiss, G.: *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, MIT Press, junio, 1999.

Wieringa, 98 Wieringa, P. A.; Wawoe, D. P.: "The Operator Support System Dilemma: Balancing a Reduction in Task Complexity vs. an Increase in System Complexity", Proc. *IEEE Systems, Man, and Cybernetics* 1998 (IEEE SMC'98), San Diego, octubre, 1998, págs. 993-997.

Wilde, 92 Wilde, N.; Huitt, R.: "Maintenance Support for Object-Oriented Programs", *IEEE Transactions on Software Engineering*, 18(12), diciembre, 1992, págs. 1038-1044.

Wilkie, 00 Wilkie, F. G.; Kitchenham, B. A.: "Coupling Measures and Change Ripples in C++ Application Software", *Journal of Systems and Software*, 52(2/3), junio, 2000, págs. 157-164.

Woodfield, 79 Woodfield, S. N.: "An Experiment on Unit Increase in Problem Complexity", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 76-79.

Woodward, 79 Woodward, M. R.; Hennell, M. A.; Hedley, D.: "A Measure of Control Flow Complexity in Program Text", *IEEE Transactions on Software Engineering*, 5(1), enero, 1979, págs. 45-50.

Wulf, 73 Wulf, W. A.: "Programming Methodology", Proc. *Symposium on the High Cost of Software*, J. Goldberg (ed.), Stanford Research Institute, septiembre, 1973

Yau, 80 Yau, S. S.; Collofello, J. S.: "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, 6(6), noviembre, 1980, págs. 545-552.

Yau, 85 Yau, S. S.; Collofello, J. S.: "Design Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, 11(9), septiembre, 1985, págs. 849-856.

Yin, 78 Yin, B. H.; Winchester, J. W.: "The Establish and Use of Measures to Evaluate the Quality of Software Designs", Proc. *ACM Software Quality Assurance Workshop*, 3(5), 1978, págs. 45-52.

Yin, 79 Yin, B. H.; Winchester, J. W.: "The Establishment and Use of Measures to Evaluate the Quality of Software Designs", *ACM Sigsoft: Software Engineering Notes*, 3(5), 1979, págs. 45-52.

Zage, 93 Zage, W. M.; Zage, D. M.: "Evaluating Design Metrics on Large-Scale Software", *IEEE Software*, 10(4), julio, 1993, págs. 75-80.

Zeleny, 74 Zeleny, M.: *Linear Multiobjective Programming*, Springer Verlag, Nueva York, 1974.

Zhao, 98 Zhao, J.; Cheng, J.; Ushijima, K.: "A Metric Suite for Concurrent Logic Programs", Proc. *CSMR'98*, Florencia, 8-11 de marzo, 1998, págs. 172-178.

Zuse, 89 Zuse, H.; Bollman, P.: "Software Metrics: Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics", *ACM Sigplan Notices*, 24(8), agosto, 1989, págs. 23-33.

8.2 BIBLIOGRAFÍA CONSULTADA

Abowd, 97 Abowd, G.; Bass, L.; Clements, P.; Kazman, R.; Northrop, L.; Zaremski, A.: "Recommended Best Industrial Practice for Software Architecture Evaluation", Technical Report CMU/SEI-96-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, enero, 1997.

Abran, 96 Abran, A.; Robillard, P. N.: "Function Point Analysis: An Empirical Study of Its Measurement Processes", *IEEE Transactions on Software Engineering*, 22(12), diciembre, 1996, págs. 895-910.

Adrion, 82 Adrion, W. R.: "Validation, Verification and Testing Computer Software", *ACM Computer Survey*, 14(2), junio, 1982.

Albrecht, 79 Albrecht, A. J.: "Measuring Application Development Productivity", Proc. *Joint SHARE/GUIDE/IBM Application Development Symposium*, Chicago, 1979.

Albrecht, 83 Albrecht, A. J.; Gaffney, J. E.: "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", *IEEE Transactions on Software Engineering*, 9(6), noviembre, 1983, págs. 639-648.

Allen, 01 — Allen, E.; Khoshgoftaar, T.; Chen, Y.: "Measuring Coupling and Cohesion of Software Modules: An Information-Theory Approach", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 124-134.

Alonso, 00 — Alonso, F.; Frutos, S.; Fuertes, J. L.; Martínez, L.; Montes, C.: "Towards a New Object-Oriented Quality Model", Technical Report, Facultad de Informática, Universidad Politécnica de Madrid, abril, 2000.

Andrews, 01 — Andrews, A. A.: "Learning to Measure or Measuring to Learn?", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 2-3.

Appelo, 97 — Appelo, L,: Cappellini, V.; Cowderoy, A.; Balestrini, O.; Bracci, R.; Brunelli, B.; Daily, K.; Dessipris, N.; Donaldson, J.; Ferrara, P.; Finnigan, A.; Geyres, S.; Granger, S.; Jones, L.; Raadsheer, P.; Ramachandran, M.; Tolias, Y.; Trinci, F.; Veenendaal, E.; Wolf, A.: "The MultiSpace Quality Framework", http://dialspace.dial.pipex.com/town/lane/xvc63/multispace/d2-1p.pdf, descargado el 7-12-1999, julio, 1997.

Arnold, 94 — Arnold, T. R.; Fuson, W. A.: "Testing 'In A Perfect World'", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 78-86.

Atkinson, 98 — Atkinson, G.; Hagemeister, J.; Oman, P.; Baburaj, A.: "Directing Software Development Projects with Product Metrics", Proc. *Fifth International Software Metrics Symposium* (METRICS'98), Bethesda, Maryland, 20-21 de noviembre, 1998, págs. 193-204.

Bailey, 81 — Bailey, J. W.; Basili, V. R.: "A Meta-Model for Software Development and Resource Expenditures", Proc. *5th International Conference on Software Engineering*, IEEE Computer Society Press, agosto, 1981.

Balla, 01 — Balla, K.; Bemelmans, T.; Kusters, R.; Trienekens, J.: "QMIM: Quality through Managed Improvement and Measurement", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 315-326.

Banker, 91 — Banker, R. D.; Kauffman, R. J.; Kumar, R.: "Output Measurement Metrics in an Object-Oriented Computer Aided Software Engineering (CASE) Environment: Critique, Evaluation and Proposal", Proc. *24th Hawaii International Conference on System Sciences*, IEEE Computer Society Press, enero, 1991.

Banker, 94 — Banker, R. D.; Kauffman, R. J.; Wright, C.; Zweig, D.: "Automating Output Size and Reuse Metrics in a Repository-Based Computer-Aided Software Engineering (CASE) Environment", *IEEE Transactions on Software Engineering*, 20(3), marzo, 1994, págs. 169-187.

Bansiya, 02 — Bansiya, J.; Davis, C. G.: "A Hierarchical Model for Object-Oriented Design Quality Assessment", *IEEE Transactions on Software Engineering*, 28(1), 2002, págs. 4-17.

Bansiya, 99b — Bansiya, J.; Davis, C; Etzkorn, L.: "An Entropy Based Complexity Measure for Object-Oriented Designs", *Theory & Practice of Object Systems*, 1999.

Barbacci, 97 — Barbacci, M. R.; Klein, M. H.; Weinstock, C. B.: "Principles for Evaluating the Quality Attributes of a Software Architecture", Technical Report CMU/SEI-96-TR-36, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, marzo, 1997.

Bashir, 99 — Bashir, I.; Paul, R. A.: "Object-Oriented Integration Testing", *Annals of Software Engineering*, 8, 1999, págs. 187-202.

Basili, 79 — Basili, V. R.; Reiter, R. W.: "Evaluating Automatable Measures of Software Development", Proc. *Workshop Quantitative Software Models*, octubre, 1979.

Basili, 80 — Basili, V. R.: *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Computer Society Press, Nueva York, 1980.

Basili, 81 — Basili, V. R.; Phillips, T.: "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", *ACM Performance Evaluation Review*, 10(1), primavera, 1981, págs. 95-106.

Basili, 84 — Basili, V. R.; Perricone, B. T.: "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, 27(1), 1984, págs. 45-52.

Basili, 85 — Basili, V. R.; Selby, R. W.: "Calculation and Use of an Environments's Characteristic Software Metrics Set", Proc. *8th International Conference on Software Engineering*, IEEE Computer Society Press, agosto, 1985, págs. 386-391.

Basili, 87 — Basili, V. R.; Selby, R. W.: "Comparing the Efectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering*, 13(12), diciembre, 1987.

Basili, 90 — Basili, V. R.: "Recent Advances in Software Measurement", Proc. *12th International Conference on Software Engineering*, Niza, Francia, 26-30 de marzo, 1990, págs. 44-49.

Basili, 92 — Basili, V. R.; Caldiera, G.; Rombach, H. D.: "The Goal Question Metric Approach", Technical Report, Department of Computer Science, University of Maryland, Maryland, 1992.

Basili, 95 — Basili, V. R.; Briand, L.; Melo, W. L.: "A Validation of Object-Oriented Design Metrics as Quality Indicators", Technical Report UMIACS-TR-95-40, University of Maryland, Computer Science Department, abril, 1995.

Bastani, 87 — Bastani, F. B.; Iyengar, S. S.: "The Effect of Data Structures on the Logical Complexity of Programs", *Communications of the ACM*, 30(3), marzo, 1987, págs. 250-259.

Batos, 99 — Batos, V.; Kalpic, D.; Baranovic, M.: "An Approach to the Quality Improvement in the Software Development Process", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 2, págs. 391-395.

Baumert, 92b — Baumert, J. H.; McWhinney, M. S.: "Software Measures and the Capability Maturity Model", Technical Report CMU/SEI-92-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1992.

Behrens, 83 — Behrens, C. A.: "Measuring the Productivity of Computer Systems Development Activities with Function Points", *IEEE Transactions on Software Engineering*, 9(6), noviembre, 1983, págs. 648-652.

Beizer, 90 — Beizer, B.: *Software Testing Techniques*, 2ª edición, Van Nostrand Reinhold, 1990.

Benlarbi, 99 — Benlarbi, S.; Melo, W. L.: "Polymorphism Measures for Early Risk Prediction", Proc. *21st International Conference on Software Engineering* (ICSE'99), Los Ángeles, mayo, 1999.

Berard, 93 — Berard, E. V.: *Object Coupling and Object Cohesion*, Essays on Object-Oriented Software Engineering, Prentice-Hall, 1993.

Berry, 01 — Berry, M.; Vandenbroeck, M. F.: "A Targeted Assessment of the Software Measurement Process", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 222-235.

Bieman, 91 — Bieman, J. M.: "Deriving Measures of Software Reuse in Object Oriented Systems", Technical Report CS-91-112, Computer Science Department, Colorado State University, Fort Collins, Colorado, julio, 1991.

Bieman, 92a — Bieman, J. M.: "Deriving Measures of Software Reuse in Object Oriented Systems", Proc. *BCS-FACS Workshop on Formal Aspects of Measurement 1991*, T. Denvir, R. Herman y R. Whitty (eds.), Springer-Verlag, 1992.

Bieman, 92b — Bieman, J. M.; Fenton, N. E.; Gustafson, D. A.; Melton, A.; Whitty, R.: "Moving from Philosophy to Practice in Software Measurement", en Denvir et. al. *Formal Aspects of Measuremen*, Springer Verlag, 1992, págs. 38-59.

Bieman, 94 — Bieman, J. M.; Ott, L. M.: "Measuring Functional Cohesion", *IEEE Transactions on Software Engineering*, 20(8), agosto, 1994, págs. 644-657.

Billow, 94 — Billow, S. C.; Lea, D.: "Workshop Report- Processes and Metrics for Object-Oriented Software Development", 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'93), Addendum to the Proceedings, *OOPS Messenger*, 5, abril, 1994, págs. 95-98.

Binder, 94a — Binder, R. V.: "Object-Oriented Software Testing", *Communications of the ACM*, 37(9), septiembre, 1994, pág. 29.

Bobkowska, 01 — Bobkowska, A.: "Quantitative and Qualitative Methods in Process Improvement and Product Quality Assessment", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 347-356.

Boloix, 95 — Boloix, G.; Robillard, P. N.: "A Software Systems Evaluation Framework", *IEEE Computer*, 28(12), diciembre, 1995.

Boloix, 99 — Boloix, G.: "Specifying and Evaluating Software Quality", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 2, págs. 404-411.

Briand, 01 — Briand, L. C.; Wüst, J.: "Integrating Scenario-Based and Measurement-Based Software Product Assessment", *The Journal of Systems and Software*, 59(1), 2001, págs. 3-22.

Briand, 94 — Briand, L.; Morasca, S.; Basili, V. R.: "Property-based Software Engineering Measurement", Technical Report CS-TR-3368, University of Maryland, Computer Science Department, 1994.

Briand, 95 — Briand, L.; El Emam, K.; Melo, W. L.: "AINSI: An Inductive Method for Software Process Improvement: Concrete Steps and Guidelines", Proc. *ESI-ISCN'95: Measurement and Training Based Process Improvement*, Viena, septiembre, 1995.

Briand, 96 — Briand, L. C.; Differding, C. M.; Rombach, H. D.: "Practical Guidelines for Measurement-Based Process Improvement", *Software Process– Improvement and Practice*, 2(4), 1996, págs. 253-280.

Briand, 98 — Briand, L. C.; Daly, J. W.; Wust, J.: "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", *Empirical Software Engineering*, 3, 1998, págs. 65-117.

Briand, 99b — Briand, L.; Wuest, J.; Ikonomovski, S.; Lounis, H.: "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Strudy", Proc. *21st International Conference on Software Engineering* (ICSE'99), Los Ángeles, mayo, 1999.

Brotbeck, 99 — Brotbeck, G.: "Highlights of Practical Software Measurement", Proc. *Fourth International Conference on Practical Software Quality Techniques* (PSQT'99 South), San Antonio, junio, 1999.

Buckley, 79 — Buckley, F.: "A Standard for Software Quality Assurance Plans", *IEEE Computer*, 12(8), agosto, 1979.

Buckley, 90 — Buckley, F. J.: "Establishing a Standard Metrics Program", *IEEE Computer*, 23(6), junio, 1990, págs. 85-86.

Buglione, 01b — Buglione, L.; Abran, A.: "QF²D: A Different Way to Measure Software Quality", Proc. *New Approaches in Software Measurement, 10th International Workshop on Software Measurement* (IWSM 2000), R. Dumke y A. Abran (eds.), Berlín, Febrero, 2001.

Buglione, 01c — Buglione, L.; Abran, A.: "Creativity and Innovation in SPI: An Exploratory Paper on their Measurement?", en Dumke/Abran *Current Trends in Software Measurement*, Shaker Publ., Aquisgrán, Alemania, 2001, págs. 112-126.

Burd, 00 — Burd, E.; Munro, M.: "Supporting Program Comprehension Using Dominance Trees", *Annals of Software Engineering*, 9, 2000, págs. 193-213.

Cádenas-García, 91 — Cádenas-García, S.; Zelkowitz, M.: "A Management Tool for Evaluation of Software Designs", *IEEE Transactions on Software Engineering*, 17, 1991, págs. 961-971.

Cardoso, 01 — Cardoso, A. I.; Araujo, T.; Crespo, R. G.: "An Alternative Way to Measure Software", Proc. *4th European Conference on Software Measurement and ITC Control* (FESMA-DASMA 2001), Heidelberg, Alemania, mayo, 2001, págs. 225-236.

Chae, 98 — Chae, H. S.; Kwon, Y. R.: "A Cohesion Measure for Classes in Object-Oriented Systems", Proc. *Fifth International Software Metrics Symposium* (METRICS'98), Bethesda, Maryland, noviembre, 1998, págs. 158-166.

Chapin, 79 — Chapin, N.: "A Measure of Software Complexity", Proc. *National Comput. Conference*, 1979, págs. 995-1002.

Chen, 78 — Chen, E. T.: "Program Complexity and Programmer Productivity", *IEEE Transactions on Software Engineering*, 4(3), mayo, 1978, págs. 187-194.

Cherniavsky, 91 — Cherniavsky, J. C.; Smith, C. H.: "On Weyuker's Axioms for Software Complexity Measures", *IEEE Transactions on Software Engineering*, 17(6), junio, 1991, págs. 636-638.

Chidamber, 97 — Chidamber, S. R.; Daray, D. P.; Kemerer, C. F.: "Managerial Use of Metrics for Object Oriented Software: an Exploratory Analysis", http://www.pitt.edu/~ckemerer/fridaysp.htm, Pittsburgh University, Pittsburgh, Pennsylvania, descargado el 16-1-2001, diciembre de 1997.

Chow, 85 — Chow, T. S.: *Software Quality Assurance: A Practical Approach*, IEEE Computer Society Press, 1985.

Chulani, 01 — Chulani, S.; Snathanam, P.; Moore, D.; Leszkowicz, B.; Davidson, G.: "Deriving a Software Quality View from Customer Satisfaction and Service Data", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 225-232.

Chung, 92 — Chung, C. M.; Lee, M. C.: "Inheritance-Based Metric for Complexity Analysis in Object-Oriented Design", *Journal of Information Science and Engineering*, 8, 1992, págs. 431-447.

Churcher, 94 — Churcher, N. I.; Shepperd, M. J.: "Comment on 'A Metircs Suite for Object Oriented Design'", Technical Report OO-1/94, Bournemouth University, 1994.

Cobb, 90 — Cobb, R. H.; Mills, H. D.: "Engineering Software under Statistical Quality", *IEEE Software*, 7(6), noviembre, 1990.

Coleman, 94 — Coleman, D.: "Using Metrics to Evaluate Software System Maintenance", *IEEE Computer*, 27(8), agosto, 1994.

Computerworld, 92 — Computerworld: "La Utopía del Software Exento de Errores", *Computerworld*, 507, 6 de noviembre, 1992, págs. 33-34.

Computerworld, 93 — Computerworld: "¿Cuál Es el Objetivo? Un Acercamiento a la Tecnología Orientada a Objetos (OOT)", *Computerworld*, 539, 25 de junio, 1993, págs. 25-26.

Coppick, 92 — Coppick, C. J.; Cheatham, T. J.: "Software metrics for Object-Oriented systems". Proc. *ACM CSC'92 Conference*, Kansas City, Missouri, ACM Press, Nueva York, marzo, 1992, págs. 317-322.

Cowderoy, 00 — Cowderoy, A.: "Measuring of Size and Complexity for Web-Site Content", Proc. *ESCOM-SCOPE 2000*, Shaker, Publ., Munich, Alemania, abril, 2000, págs. 423-431.

Crandall, 92 — Crandall, V. J.: "Examples of Problems Encountered when Myers Measures of Good Modularity are Violated and their Solutions", en *Software Quality: the Challenge in the 90s*, INTA, Proc. 3rd European Conference on Software Quality, Madrid, noviembre, 1992.

Curtis, 79a — Curtis, B.; Sheppard, S. B.; Milliman, P.; Borst, M. A.; Love, T.: "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 96-104.

Curtis, 83 — Curtis, B.: "Software Metrics: Guest Editor's Introduction", *IEEE Transactions on Software Engineering*, 9(6), noviembre, 1983, págs. 637-638.

Damerla, 92 Damerla, S.; Shatz, S. M.: "*Software* Complexity and Ada Redezvous: Metrics Based on Nondeterminism", *Journal of Systems and Software*, 17, 1992, págs. 119-127.

Davis, 88 Davis, J. S.; LeBlanc, R. J.: "A Study of the Applicability of Complexity Measures", *IEEE Transactions on Software Engineering*, 14(9), septiembre, 1988, págs. 1366-1372.

De Marco, 95 De Marco, T.: *Why Does Software Cost so Much, and Other Puzzles of the Information Age*, Dorset House, Nueva York, 1995.

Deming, 86 Deming, W. E.: *Out of the Crisis*, MIT Press, 1986.

Dimitrov, 00 Dimitrov, E.; Schmietendorf, A.; Dumke, R.: "Software Reuse and Metrics within a Process Model for Object-Oriented Developmen", *Metrics News*, 5(2), 2000, págs. 34-44.

Dromey, 92 Dromey, R. G.; McGettrick, A. D.: "On Specifying Software Quality", *Software Quality Journal,* 1(1), 1992.

Drummond, 85 Drummond, S.: "Measuring Applications Development Performance", *Datamation*, febrero, 1985, págs. 102-108.

Dumke, 01 Dumke, R. R.; Lother, M.; Abran, A.: "An Approach for Integrated Software Measurement Processes in the IT Area", Proc. *4th European Conference on Software Measurement and ITC Control* (FESMA-DASMA 2001), Heidelberg, Alemania, mayo, 2001, págs. 15-30.

Dunaway, 96 Dunaway, D. K.; Masters, S.: "CMM[SM]-Based Appraisal for Internal Process Improvement (CBA IPI): Method Description", Technical Report CMU/SEI-96-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, abril, 1996.

Dunn, 90 Dunn, R. H.: *Software Quality: Concepts and Plans*, Prentice Hall, New Jersey, 1990.

Ebert, 99 Ebert, C; Liedtke, T.; Baisch, E.: "Improving Reliability of Large Software Systems", *Annals of Software Engineering*, 8, 1999, págs. 3-51.

Eder, 94 Eder, J.; Kappel, G.; Schrefl, M.: "Coupling and Cohesion in Object Oriented Systems", Technical Report, University of Klagenfurt, 1994.

Ejiogu, 91 Ejiogu, L. O.: "TM: A Systematic Methodology of Software Metrics", *ACM Sigplan Notices*, 26(1), enero, 1991, págs. 124-132.

Emam, 01 Emam, K. E.: "A Primer on Object-Oriented Measurement", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 630-650.

ERCIM, 92 ERCIM: *Software Quality Principles and Techniques*, European Research Consortium for Informatics and Mathematics, Pisa, 1992.

Escala, 98 Escala, D.; Morision, M.: "A Metrics Suite for a Team PSP", Proc. *Fifth International Software Metrics Symposium* (METRICS'98), Bethesda, Maryland, 20-21 de noviembre, 1998, págs. 89-92.

Etzkorn, 95 Etzkorn, L. H.; Davis, C. G.: "An Approach to Object-Oriented Program Understanding", Technical Report TR-UAH-CS-1995-01, Department of Computer Science, University of Alabama in Huntsville, Alabama, 1995.

Fehlmann, 01 Fehlmann, T. M.: "Risk Exposure Measurements on Websites", Proc. Proc. *4th European Conference on Software Measurement and ITC Control* (FESMA-DASMA 2001), Heidelberg, Alemania, mayo, 2001, págs. 149-160.

Fenton, 90 Fenton, N. E.: "Software Metrics: Theory, Tools, and Validation", *Software Engineering Journal,* 5(1), enero, 1990, págs. 65-84.

Fenton, 92 Fenton, N. E.: "When a Software Measure Is Not a Measure", *Software Engineering Journal,* 7(5), septiembre, 1992, págs. 357-362.

Fenton, 99 Fenton, N.; Neil, M.: "Software Metrics and Risks", Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 39-55.

Fernández, 92 Fernández, L.: "Ahora Hay Crisis y Sólo la Calidad Puede Salvar el Mercado", *Computerworld*, 507, 6 de noviembre, 1992, págs. 25-27.

Ferneley, 00 Ferneley, E.: "Coupling and Control Flow Measures in Practice", *Journal of Systems and Software*, 51(2), abril, 2000, págs. 99-109.

Fewster, 01 Fewster, R.; Mendes, E.: "Measurement, Prediction and Risk Analysis for Web Applications", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 338-348.

Fioravanti, 01 Fioravanti, F.; Nese, P.: "Estimation and Prediction Metrics for Adaptative Maintenance Efoort of Object-Oriented Systems", *IEEE Transactions on Software Engieering*, 27(12), 2001, págs. 1062-1084.

Florac, 92 Florac, W. A.: "Software Quality Measurement: A Framework for Counting Problems and Defects", Technical Report CMU/SEI-92-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1992.

Frakes, 01 Frakes, W. B.; Succi, G.: "An Industrial Study of Reuse, Quality, and Productivity", *The Journal of Systems and Software*, 57(2), 2001, págs. 99-106.

Frankl, 88 Frankl, P. G.; Weyuker, E. J.: "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on Software Engineering*, 14(10), octubre, 1988, págs. 1483-1498.

Gall, 00 Gall, G. L.; Lucas, J.; Vallee, F.; Vernos, D.: "Characterisation of some C++ Projects through their Measures", Proc. *ESCOM-SCOPE 2000*, Shaker, Publ., Munich, Alemania, abril, 2000, págs. 439-448.

Gemoets, 90 Gemoets, L. A.; Mahmood, M. A.: "Effect of the Quality of User Documentation on User Satisfaction with Information Systems", *Information & Management*, 18(1), 1990, págs. 47-54.

Gentleman, 97 Gentleman, W. M.: "If Software Quality Is a Perception, How Do we Measure it?", Proc. *IFIP TC2/WG2.5 Working Conference*, Conference on Mathematical and Scientific Computing: Quality of Numerical Software, Londres, 1997, págs. 32-43.

Goethert, 92 Goethert, W. B.; Bailey, E. K.; Busby, M. B.: "Software Effort and Schedule Measurement: A Framework for Counting Staff-Hours and Reporting Schedule Information", Technical Report CMU/SEI-92-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1992.

Gokhale, 99 Gokhale, S. S.; Trivedi, K. S.: "A Time/Structure Based Software Reliability Model", *Annals of Software Engineering*, 8, 1999, págs. 85-121.

Gordon, 76 Gordon, R. D.; Halstead, M. H.: "An Experiment Comparing Fortran Programing Times with the Software Physics Hypothesis", Proc. *AFIPS*, 1976, págs. 935-937.

Gordon, 79b Gordon, R. D.: "A Qualitative Justification for a Measure of Program Clarity", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 121-128.

Gorla, 97 Gorla, N.; Ramakrishnan, R.: "Effect of Software Structure Attributes on Software Development Productivity", *Journal of Systems and Software*, 36(2), febrero, 1997, págs. 191-199.

Goth, 00 Goth, G.: "The Team Software Process: A Quiet Quality Revolution", *IEEE Software*, noviembre/diciembre, 2000, págs. 125-127.

Gowda, 94 Gowda, R. G.; Winslow, L. E.: "An Approach for Deriving Object-Oriented Metrics", Proc. *IEEE 1994 National Aerospace and Electronics Conference*, Dayton, Ohio, IEEE Computer Society Press, California, 23-27 mayo, 1994, págs. 897-904.

Grable, 99 Grable, R.; Jernigan, J.; Pogue, C.; Divis, D.: "Metrics for Small Projects: Experiences at the SED", *IEEE Software*, marzo-abril, 1999, págs. 21-29.

Grady, 92 Grady, R. B.: *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, Prentice Hall, New Jersey, 1992.

Grady, 93 — Grady, R. B.: "Software Metrics Etiquette", *American Programmer*, 6(2), febrero, 1993.

Graham, 96 — Graham, D.: "Testing Object-Oriented systems". Proc. *Object Expo Europe*, SIGS Conferences, Reino Unido, 1996, págs. 309-318.

Gupta, 97 — Gupta, B. S.: *A Critique of Cohesion Measures in the Object-Oriented Paradigm*, Master of Science Thesis, Dept. of Computer Science, Michigan Technological University, Michigan, marzo, 1997.

Hall, 01 — Hall, T.; Baddoo, N.; Wilson, D. N.: "Measurement in Software Process Improvement Programmes: An Empirical Study", en Dumke/Abran *New Approaches in Software Measurement*, Springer Publ., 2001, págs. 73-82.

Hall, 94 — Hall, T.; Fenton, N. E.: "Implementing Software Metrics- the Critical Success Factors", *Software Quality Journal,* 3, 1994, págs. 195-208.

Halstead, 79 — Halstead, M. H.: "Guest Editorial on Software Science", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 74-75.

Harmon, 98 — Harmon, S. Y.: "Evaluating and Comparing Information Systems", Proc. *IEEE Systems, Man, and Cybernetics 1998* (IEEE SMC'98), San Diego, octubre, 1998, págs. 1009-1014.

Harrison, 01 — Harrison, W.; Cook, C.: "Insights on Improving the Maintenance Process through Software Measurement", http://www.cs.pdx.edu/~warren/Papers/CSM.htm, descargado el 16-1-2001.

Harrison, 81b — Harrison ,W.; Magel, K.: "A Complexity Measure Based on Nesting Level", *ACM Sigplan Notices*, 16(3), marzo, 1981.

Harrison, 92 — Harrison, W.: "An Entropy-Based Measure of Software Complexity", *IEEE Transactions on Software Engineering*, 18(11), noviembre, 1992, págs. 1025-1029.

Harrison, 98 — Harrison, R.; Counsell, S.; Nithi, R.: "Coupling Metrics for Object-Oriented Design", Proc. *Fifth International Software Metrics Symposium* (METRICS'98), Bethesda, Maryland, noviembre, 1998, págs. 150-157.

Henderson-Sellers, 90 — Henderson-Sellers, B.; Edwards, J. M.: "The Object Oriented Systems Life Cycle", *Communications of the ACM*, septiembre, 1990.

Henderson-Sellers, 91 — Henderson-Sellers, B.: "Some Metrics for Object-Oriented Software Engineering", Proc. *International Conference on Teaching Object-Oriented Languages and Systems* (TOOLS'91), 1991, págs. 131-139.

Henderson-Sellers, 94b — Henderson-Sellers, B.: "Identifying Internal and External Characteristics of Classes Likely To Be Useful as Structural Complexity Metrics", Proc. *OOIS'94*, Londres, diciembre, 1994.

Henderson-Sellers, 96b — Henderson-Sellers, B.: *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall, New Jersey, 1996.

Henderson-Sellers, 99 — Henderson-Sellers, B.: "OO Software Process Improvement with Metrics", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Boca Ratón, Florida, noviembre, 1999, págs. 2-8.

Henry, 90b — Henry, S.; Selig, C.: "Design Metrics which Predict Source Code Quality", *IEEE Software.*, 7(2), marzo, 1990.

Henry, 90c — Henry, S.; Wake, S.; Li, W.: "Applying Complexity Metrics During the Software Life Cycle", Proc. *Interface'90, the 22nd Symposium on the Interface: Computing Science and Statistics*, Michigan, mayo, 1990, págs. 23-29.

Henry, 92 — Henry, S.; Li, W.: "Metrics for Object-Oriented Systems", Proc. 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), *ACM Sigplan Notices*, 27(10), octubre, 1992.

Hirayama, 90 Hirayama, M.; Sato, H.; Yamada, A.: "Practice of Quality Modeling and Measurement on Software Life-Cycle", Proc. *12th International Conference on Software Engineering*, IEEE Computer Society Press., Francia, 1990, págs. 98-107.

Hirvensalo, 01 Hirvensalo, J.: "Improving the Utilisation of Measurements in a Software Development Process", Proc. *4th European Conference on Software Measurement and ITC Control* (FESMA-DASMA 2001), Heidelberg, Alemania, mayo, 2001, págs. 247-258.

Holt, 97 Holt, J.: "Software Metrics– Real World Experiences", *Software Process– Improvement and Practice*, 3(3), septiembre, 1997, págs. 155-163.

Hopkins, 91 Hopkins, T. P.: "Do We Need Object-Oriented Design Metrics?", *Hotline on Object-Oriented Technology*, 2(8), 1991, págs. 16-17.

Hops, 95 Hops, J. M.; Sherif, J. S.: "Development and Application of Composite Complexity Models and a Relative Complexity Metric in a Software Maintenance Environment", *Journal of Systems and Software*, 31, 1995, págs. 157-169.

Hordijk, 00 Hordijk, W.; Molterer, S.; Salzmann, C.; Paech, B.; Linos, P. K.: "Maintainable Systems with a Business Object Approach", *Annals of Software Engineering*, 9, 2000, págs. 273-292.

Horgan, 94 Horgan, J. R.; London, S.; Lyu, M. R.: "Achieving Software Quality with Testing Coverage Measures", *IEEE Computer*, 27(9), septiembre, 1994, págs. 60-69.

Humphrey, 88 Humphrey, W. S.: "Characterizing the Software Process: A Maturity Framework", IEEE Software, 5(2), marzo, 1988, págs. 73-79.

Humphrey, 96 Humphrey, W. S.: "Using a Defined and Measured Personal Software Process", *IEEE Software*, mayo, 1996, págs. 77-88.

Hurrington, 91 Hurrington, H. J.: *Business Process Improvement: The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*, McGraw-Hill, Nueva York, 1991.

Hutchens, 85 Hutchens, D. H.; Basili, V. R.: "System Structure Analysis: Clustering with Data Bindings", *IEEE Transactions on Software Engineering*, 11(8), agosto, 1985, págs. 749-757.

IEEE, 81 IEEE: *IEEE Standard for Software Quality Assurance Plans*, ANSI/IEEE Std. 730-1981, IEEE, Nueva York, 1981.

IEEE, 83 IEEE: *Glosary of Software Engineering Terminology*, ANSI/IEEE Std. 729-1983, diciembre, 1983.

IEEE, 86 IEEE: *Guide for Software Quality Assurance Planning*, IEEE Std 983-1986, Nueva York, enero, 1986.

Ikeda, 00 Ikeda, K.; Nishiyama, T.; Shima, K.; Matsumoto, K.; Inoue, K.; Torii, K.: "Quality Assurance Activities in Object-Oriented Software Development", Proc. *ESCOM-SCOPE 2000*, Múnich, Alemania, abril, 2000, págs. 459-466.

Ince, 89b Ince, D. C.; Shepperd, M. J.: "An Empirical and Theoretical Analysis of an Information Flow Based Design Metric", Proc. *European Software Engineering Conference*, 1989, págs. 86-99.

Ince, 91 Ince, D.: *Software Quality and Reliability: Tools and Methods*, Chapman & Hall, Londres, 1991.

INTA, 92 INTA: "Software Quality: the Challenge in the 90s", Proc. *3rd European Conference on Software Quality*, Madrid, noviembre, 1992.

ISO, 99 ISO: Information Technology– Software Product Evaluation– Part 1: General Overview, ISO/IEC JTC1/SC7/WG6, IS 14598-1, International Organization of Standardization, 1999.

Iversen, 99 Iversen, J. H.: "Establishing SPI Effect Measurements", Proc. *International Conference on Product Focused Software Process Improvement*, Oulu, Finalandia, junio, 1999, págs. 591-605.

Jenkis, 93 Jenkis, J.: "A Paucity of Panaceas: Software Metrics Won't Eliminate the Productivity Crisis", *American Programmer*, 6(2), febrero, 1993.

Jenson, 91 Jenson, R. L.; Bartley, J. W.: "Parametric Estimation of Programming Effort: an Object-Oriented Model", *Journal of Systems and Software*, 15, 1991, págs. 107-114.

Jones, 81 Jones, C.: *Programming Productivity: Issues for the Eighties*, IEEE Computer Society Press, 1981.

Jones, 86 Jones, C.: *Programming Productivity*, McGraw-Hill, Nueva York, 1986.

Jones, 93 Jones, C.: "Software Measurement: Why, What, When, and Who", *American Programmer*, 6(2), febrero, 1993.

Jones, 97b Jones, C.: *Economics of Object-Oriented Software*, Software Productivity Research, Burlington, 1997.

Jones, 97c Jones, C.: *Software Quality: Analysis and Guidelines for Success*, International Thomson Computer Press, Boston, 1997.

Jorgensen, 94 Jorgensen, P. C.; Erickson, C.: "Object-Oriented Integration Testing", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 30-38.

Jorgensen, 97 Jorgensen, M.: "Measurement of Software Quality", Proc. *International Conference on Software Quality Engineering*, Italia, mayo, 1997, págs. 257-266-

Jorgensen, 99 Jorgensen, M.: "Software Quality According to Measurement Theory", *Telektronic*, 95(1), 1999, págs. 12-16.

Kalakota, 93 Kalakota, R.; Rathmam, S.; Whinston, A.: "The Role of Complexity in Object Oriented Development", Proc. *26th Annual Conference on Systems Sciences*, 1993.

Kamatar, 00 Kamatar, J.; Hayes, W.: "An Experience Report on the Personal Software Process", *IEEE Software*, noviembre/diciembre, 2000, págs. 85-89.

Kan, 95 Kan, S. H.: *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995.

Karunanithi, 93a Karunanithi, S.; Bieman, J. M.: "Measuring Software Reuse in Object-Oriented Systems and Ada Software", Technical Report CS-93-125, Computer Science Department, Colorado State University, Fort Collins, Colorado, 1993.

Karunanithi, 93b Karunanithi, S.; Bieman, J. M.: "Candidate Reuse Metrics for Object Oriented and Ada Software", Proc. *IEEE-CS International Software Metrics Symposium*, 1993, págs. 120-128.

Kautz, 99 Kautz, K.: "Making Sense of Measurement for Small Organization", *IEEE Software*, marzo-abril, 1999, págs. 14-20.

Kazman, 94 Kazman, R.; Bass, L.: "Toward Deriving Software Architectures from Quality Attributes", Technical Report CMU/SEI-94-TR-10, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, agosto, 1994.

Kearney, 86 Kearney, J. K.; Sedlmeyer, R. L.; Thompson, W. B.; Gray, M. A.; Adler, M. A.: "Software Complexity Measurement", *Communications of the ACM*, 29(11), noviembre, 1986, págs. 1044-1050.

Kemerer, 87 Kemerer, C. F.: "An Empirical Validation of Software Cost Estimation Models", *Communications of the ACM*, 30(5), 1987, págs. 461-529.

Kemerer, 92 Kemerer, C. F.; Porter, B. S.: "Improving the Reliability of Function Point Measurement: an Empirical Study", *IEEE Transactions on Software Engineering*, 18(11), noviembre, 1992, págs. 1011-1024.

Kemerer, 93 Kemerer, C. F.: "Reliability of Function Points Measurement: a Field Experiment", *Communications of the ACM*, 36(2), febrero, 1993, págs. 85-97.

Kennet, 99 Kennet, R. S.; Baker, E. R.: *Software Process Quality– Management and Control*, Marcel Dekker Inc. Publ., 1999.

Keyes, 92 — Keyes, J.: "New Metrics Needed for New Generation", *Software Magazine*, 12(6), mayo, 1992, págs. 42-56.

Khaddaj, 01 — Khaddaj, S.; Horgan, G.: "Factors in Software Quality for Advanced Computer Architectures", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 437-442.

Khoshgoftaar, 90 — Khoshgoftaar, T. M.; Munson, J. C.: "Predicting Software Development Errors Using Complexity Metrics", *IEEE Journal of Selected Areas of Communications*, 8(2), febrero, 1990.

Khoshgoftaar, 92 — Khoshgoftaar, T. M.; Munson, J. C.; Bhattacharya, B. B.; Richardson, G. D.: "Predictive Modeling Techniques of Software Quality from Software Measures", *IEEE Transactions on Software Engineering*, 18(11), noviembre, 1992, págs. 979-987.

Khoshgoftaar, 94c — Khoshgoftaar, T. M.; Allen, E. B.: "Applications of Information Theory to Software Engineering Measurement", *Software Quality Journal*, 3, 1994, págs. 79-103.

Khoshgoftaar, 98 — Khoshgoftaar, T. M.; Allen, E. B.; Halstead, R.; Trio, G. P.; Flass, R. M.: "Using Process History to Predict Software Quality", *IEEE Computer*, abril, 1998, págs. 66-72.

Kim, 94 — Kim, E. M.; Chang, O. B.; Kusumoto, S.; Kikuno, T.: "Analysis of Metrics for Object-Oriented Program Complexity", Proc. International Computer Software and Applications Conference (COMPSAC'94), *IEEE Computer*, 1994, págs. 201-207.

King, 99 — King, G.: "Software Quality: Process Improvement through Quantitative Approaches", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 2, págs. 460-466.

Kirsopp, 01 — Kirsopp, C.: "Measurement and the Software Development Process", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 165-173.

Kitchenham, 01 — Kitchenham, B. A.; Hughes, R. T.; Linkman, S. G.: "Modeling Software Measurement Data", *IEEE Transactions on Software Engineering*, 27(9), 2001, págs. 788-804.

Kitchenham, 89b — Kitchenham, B. A.; Littlewood, B.: "Measurement for Software Control and Assurance", Proc. *Centre for Software Reliability Conference*, Elsevier Science Publishers, 1989.

Kitson, 92 — Kitson, D. H.; Masters, S.: "An analysis of SEI Software Process Assessment Results", Technical Report CMU/SEI-92-TR-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, julio, 1992.

Kokol, 99 — Kokol, P.; Podgorelec, V.; Zorman, M.: "Universality – A Need for a New Software Metric", Proc. *9th International Workshop on Software Measurement* (IWSM'99), Lac Supérieur, Quebec, Canadá, septiembre, 1999, págs. 51-54.

Lake, 92b — Lake, A.; Cook, C.: "A Software Complexity Metric for C++", Technical Report 92-60-03, Computer Science Department, Oregon State University, Corvallis, Oregon, 1992.

Lange, 95 — Lange, D. B.; Nakamura, Y.: "Program Explorer: A Program Visualicer for C++", Proc. USENIX Conference on Object-Oriented Technologies (COOTS), Monterey, California, junio, 1995.

Lanubile, 95 — Lanubile, F.; Lonigro, A.; Visaggio, G.: "Comparing Models for Identifying Fault-Prone Software Components", Proc. *Software Engineering and Knowledge Engineering* (SEKE'95), junio, 1995.

Lanubile, 96 — Lanubile, F.; Visaggio, G.: "Evaluating Predictive Quality Models Derived from Software Measures: Lessons Learned", Technical Report UMIACS-TR-96-14, University of Maryland, Computer Science Department, enero, 1996.

Lattanzi, 93 — Lattanzi, M.; Henry, S.; Tiffany, M.; Gong, W.; Li, W.: "A Software Metrics Generation Tool for the Object-Oriented Paradigm", Proc. *Applications of Software Measurement*, Orlando, noviembre, 1993, págs. 603-614.

Lee, 93 — Lee, Y.; Liang, B.; Wang, F.: "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow", Proc. *CompEuro*, París-Ivry, IEEE Computer Society Press, California, 24-27 de mayo, 1993, págs. 302-310.

Leonardi, 95 — Leonardi, C.; Morisio, M.: "Object-Oriented Methods for Reuse and Quality Improvement", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995, págs. 211-229.

Leveson, 89 — Leveson, N. G.: "Safety as a Software Quality", *IEEE Software*, 6(3), mayo, 1989.

Li, 90 — Li, W.; Henry, S.: "A Lifecycle Model which Incorporates Software Metrics", Proc. *The Ninth National Conference on EDP System and Software Quality Assyrence*, Washington DC, octubre, 1990, págs. 331-356.

Li, 91ª — Li, W.; Henry, S.: "Software Life Cycle Model Using Metrics", Proc. *The 19th Annual Computer Science Conference of ACM*, San Antonio, Texas, marzo, 1991, págs. 661-662.

Li, 91b — Li, W.; Henry, S.; Calvin, S.: "Measuring Ada Design to Predict Maintainability", Proc. *The 9th Annual National Conference on Ada Technology*, Washington DC, marzo, 1991, págs. 107-113.

Li, 91c — Li, W.; Henry, S.: "Some Metrics Primitives for the Object-Oriented Paradigm", Proc. *International Conference on Achieving Quality in Software'91*, Pisa, Italia, abril, 1991, págs. 45-50.

Li, 93b — Li, W.; Henry, S.: "Maintenance Metrics for the Object Oriented Paradigm", Proc. *1st International Software Metrics Symposium*, 1993.

Li, 97ª — Li, W; Talburt, J; Chen, Y. T.: "Quantitatively Measuring Object-Oriented Design Evolution", Proc. *24th International Technology of Object-Oriented Languages and Systems* (TOOLS) Asia'97, Beijing, China, septiembre, 1997.

Li, 97b — Li, W; Delugach, H.: "Software Metrics and Application Domain Complexity", Proc. *Asia-Pacific Software Engineering Conference*, Honk Kong, diciembre, 1997.

Li, 99 — Li, W.; Talburt, J.; Alshayeb, M.: "Software Metrics and Object-Oriented System Evolution", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 2, págs. 475-482.

Lorenz, 94 — Lorenz, M.; Kidd, J.: *Object-Oriented Software Metrics: A Practical Guide*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

Määtä, 00 — Määtä, J.: "Evolution of the Simple Moodel of the Software Development Process for Measurement; What's Beyond Measurement Problems?", Proc. *ESCOM-SCOPE 2000*, Shaker, Publ., Munich, Alemania, abril, 2000, págs. 277-288.

Mah, 97 — Mah, M. C.; Putnam, L. H.: "Software by the Numbers: An Aerial View of the Software Metrics Landscape", *American Programmer*, 10(11), noviembre, 1997, págs. 3-11.

Mah, 99 — Mah, M. C.: "High Definition Software Measurement– Benchmarking Techniques for Process and Quality Improvement", Proc. *Fourth International Conference on Practical Software Quality Techniques* (PSQT'99 South), San Antonio, junio, 1999.

Manns, 88 — Manns, T.; Coleman, M.: *Software Quality Assurance*, Macmillan Education, Londres, 1988.

Marzal, 92 — Marzal, M. J.: "Calidad Total", *Computerworld*, 508, 13 de noviembre, 1992, pág. 5.

Mason, 81 — Mason, R. O.; Swanson, E. B.: *Measurement for Management Decision*, Addison-Wesley, 1981.

Maupetit, 95 — Maupetit, C.; Kuvaja, P.; Palo, J.; Belli, M.; Isokaanta, M.: "Drive SPI: a Risk-Driven Approach for Software Process Improvement", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995, págs. 151-173.

McAndrews, 93 McAndrews, D. R.: "Establishing a Software Measurement Process", Technical Report CMU/SEI-93-TR-16, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, julio, 1993.

McCabe, 93 McCabe, T. J.: *Object Oriented Tools Aids Software Testing*, The Outlook, McCabe & Associattes, 1993.

McCall, 81 McCall, J. A.; Markham, D.; Stosick, M.; McGindly, R.: "The Automated Measurement of Software Quality", Proc. *5th International Computer Software and Applications Conference* (COMPSAC'81), 1981, págs. 52-58.

McColl, 92 McColl, R. B.; McKim, J. C.: "Evaluating and Extending NPath as a Software Complexity Measure", *Journal of Systems Software*, 17, 1992, págs. 275-279.

McGarry, 02 McGarry, J.; Card, D.; Jones, C.; Layman, B.; Clark, E.; Dean, J.; Hall, F.: *Practical Software Measurement– Objective Information for Decision Makers*, Addison-Wesley, 2002.

McGarry, 94 McGarry, F.; Pajerski, R.; Page, G.; Waligora, S.; Basili, V. R.; Zelkowitz, M.: "Software Process Improvement in the NASA Software Engineering Laboratory", Technical Report CMU/SEI-94-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, diciembre, 1994.

McGregor, 94 McGregor, J. D.; Korson, T. D.: "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 59-77.

McGregor, 97 McGregor, J. D.: "An Overview of Testing", *Journal of Object-Oriented Programming*, 9(8), enero, 1997, págs. 5-9.

Meyer, 98 Meyer, B.: "The Role of Object-Oriented Metrics", *IEEE Computer*, noviembre, 1998, págs, 123-125.

Miluk, 93 Miluk, G.: "The Role of Measurement in Software Process Improvement", Proc. *Third International Conference on Software Quality*, Lake Tahoe, Nevada, 4-6 de octubre, 1993, págs. 191-197.

Misic, 01 Misic, V.: "Cohesion is Structural, Coherence is Functional: Different Views, Different Measures", Proc. *Seventh International Software Metrics Symposium* (METRICS 2001), Londres, abril, 2001, págs. 112-123.

Miyazaki, 87 Miyazaki, Y.; Murakami, N.: "Software Metrics Using Deviation Value", Proc. *9th International Conference on Software Engineering*, IEEE Computer Society Press, marzo, 1987, págs. 83-91.

Möller, 93 Möller, K. H.; Paulish, D. J.: *Software Metrics: A Practitioner's Guide to Improved Product Development*, Los Alamitos, IEEE Computer Society Press, California, 1993.

Morasca, 97 Morasca, S.; Briand, L. C.: "Towards a Theoretical Framework for Measuring Software Attributes", Proc. *Fourth METRICCS'97*, Albuquerque, 5-7 de noviembre, 1997, págs. 119-126.

Moreau, 90a Moreau, D. R.; Dominick, W. D.: "A Programming Environment Evaluation Methodolgy for Object-Oriented Systems: Part I- The Methodology", *Journal of Object-Oriented Programming*, 3(1), mayo/junio, 1990, págs. 38-52.

Moreau, 90b Moreau, D. R.; Dominick, W. D.: "A Programming Environment Evaluation Methodolgy for Object-Oriented Systems: Part II- The Test Case Application", *Journal of Object-Oriented Programming*, 3(3), septiembre/octubre, 1990, págs. 23-32.

Munson, 92 Munson, J. C.; Khoshgoftaar, T. M.: "The Detection of Fault-Prone Programs", *IEEE Transactions on Software Engineering*, 18(5), mayo, 1992, págs. 423-432.

Murphy, 94 Murphy, G. C.; Townsend, P.; Wong, P. S.: "Experiences with Cluster and Class Testing", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 39-47.

Myers, 79 Myers, G.: *The Art of Software Testing*, John Wiley & Sons, 1979.

Myers, 92 — Myers, J. P.: "The Complexity of Software Testing", *Software Engineering Journal,* 7(1), enero, 1992, págs. 13-24.

Nejmeh, 88 — Nejmeh, B. A.: "NPath: A Measure of Execution Path Complexity and its Applications", *Communications of the ACM*, 31(2), febrero, 1988, págs. 188-200.

Neto, 95 — Neto, J.; Lavrador, A.; Pinto, C.; Reed, R.: "SDL-92 for Software Process Improvement: Practical Experience", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995, págs. 406-423.

OMG, 99 — Object Management Group: "The Common Object Request Broker: Architecture and Specification. Revision 2.3.1", http://www.omg.org/corba/corbaiiop.html, descargado el 22-2-2000, octubre, 1999.

Orci, 99 — Orci, T.: "Software Process Improvement or Measurement Programme– Which One Comes First?", Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 127-140.

Ott, 95 — Ott, L. M.; Bieman, J. M.; Kang, B.-K.; Mehra, B.: "Developing Measures of Class Cohesion for Object-Oriented Software", Proc. *7th Annual Oregon Workshop on Software Metrics*, Oregon, junio, 1995.

Ould, 99 — Ould, M. A.: *Managing Software Quality and Business Risk*, John Wiley & Sons, 1999.

Paige, 80 — Paige, M.: "A Metric for Software Test Planning", Proc. 4th International Computer Software and Applications Conference (COMPSAC'80), *IEEE Transactions on Software Engineering*, 6(2), marzo, 1980, págs. 70-75.

Panfilis, 95 — de Panfilis, S.; Morfuni, N.: "A Successful Approach to Address Software Quality", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995.

Pant, 95 — Pant, Y. R.; Verner, J. M.; Henderson-Sellers, B.: *S/C: A Software Size/Complexity Measure. Software Quality and Productivity. Theory, Practice, Education and Training*, en M. Lee, B. Z. Barta y P. Juliff (eds.), 1995.

Park, 92 — Park, R. E.: "Software Size Measurement: A Framework for Counting Source Statements", Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, septiembre, 1992.

Paulk, 93 — Paulk, M. C.; Weber, C. V.; García, S.; Chrisis, M. B.; Bush, M.: "Key Practices of the Capability Madurity Model", versión 1.1, Technical Report CMU/SEI-93-TR-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, febrero, 1993.

Paulk, 94 — Paulk, M. C.; Weber, C. V.; Curtis, B.; Chrisis, M. B.: "The CMM: Guidelines for Improving the Software Process", Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1994.

Pearse, 99 — Pearse, T.; Freeman, T.; Oman, P.: "Using Metrics to Manage the End-Game of a Software Project", Proc. *6th International Symposium on Software Metrics* (METRICS'99), Boca Ratón, Florida, noviembre, 1999, págs. 207-215.

Perry, 87 — Perry, W. E.: "Effective methods of EDP Quality Assurance", en *Handbook of Software Quality Assurance*, G. G. Schulmeyer y J. I. McManus (eds.), Van Nostrand Reinhold Company, Nueva York, 1987, págs. 408-430.

Pfleeger, 90b — Pfleeger, S. L.; Palmer, J. D.: "Software Estimation for Object Oriented Systems", Proc. *1990 International Function Point Users Group Fall Conference*, San Antonio, Texas, octubre, 1990, págs. 181-196.

Pfleeger, 97 — Pfleeger, S. L.; Jeffery, R.; Curtis, B.; Kitchenham, B.: "Status Report on Software Measurement", *IEEE Software*, marzo/abril, 1997, págs. 33-43.

Poston, 94 — Poston, R. M.: "Automated Testing from Object Models", *Communications of the ACM*, 37(9), septiembre, 1994, págs. 48-58.

Prechelt, 01 — Prechelt, L.; Unger, B.: "An Experiment Measurign the Effects of Personal Software Process (PSP)", *IEEE Transactions on Software Engineering*, 27(5), 2001, págs. 465-472.

Pressman, 96 — Pressman, R.: *Software Engineering: A Practitioner's Approach*, 4º ed., McGraw-Hill, 1996.

Punter, 99 — Punter, T.: "Do Your Metrics Predict Software Quality?", Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 341-348.

Putnam, 92 — Putnam, L. H.; Myers, W.: *Measures for Excellence: Reliable Software on Time, Within Budget*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

Putnam, 97 — Putnam, L. H.; Myers, W.: *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Press, Los Alamitos, California, 1997.

Rajaraman, 92 — Rajaraman, C.; Lyu, M. R.: "Some Coupling Measures for C++ Programs", Proc. *TOOLS USA 92*, Vol. 6, 1992, págs. 225-234.

Ramil, 01 — Ramil, J.; Lehmann, M.: "Defining and Applying Metrics in the Context of Continuing Software Evolution", Proc. *Seventh International Softwaare Metrics Symposium* (METRICS, 01), Londres, abril, 2001, págs. 199-209.

Rangarajan, 01 — Rangarajan, K.; Swaminathan, N.; Hedge, V.; Jacob, J.: "Product Quality Framework: A Vehicle for Focusing on Product Quality Goals", *ACM Sigsoft: Software Engineering Notes*, 26(4), 2001, págs. 77-82.

Ravichandran, 01 — Ravichandran, S.; Shareef, P. M.: "Software Process Assessment through Metrics Models", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 367-376..

Rees, 82 — Rees, M. J.: "Automatic Assessment Aids for Pascal Programs", *ACM Sigplan Notices*, 17(10), octubre, 1982, págs. 33-42.

Rifkin, 01 — Rifkin, S.: "What Makes Measuring Software So Hard?", *IEEE Softwaare*, mayo/junio, 2001, págs. 41-45.

Roberts, 79b — Roberts, F. S.: "Measurement Theory", en *Encyclopedia of Mathematics and Its Applications*, G.-C. Rota (ed.), Addison-Wesley, 1979.

Roche, 94 — Roche, J. M.: "Software Metrics and Measurement Principles", *ACM Sigsoft: Software Engineering Notes*, 19, enero, 1994, págs, 76-85.

Rombach, 90 — Rombach, H.: "Design Measurement. Some Lessons Learned", *IEEE Software*, marzo, 1990.

Rösel, 01 — Rösel, A.: "External Metrics for Software Quality in an International Manufacturing Company", Proc. *4th European Conference on Software Measurement and ICT Control* (FESMA-DASMA 2001), Heidelberg, Alemania, mayo, 2001, págs. 357-367.

Rotenstreich, 94 — Rotenstreich, S.: "Toward Measuring Potential Coupling", *Software Engineering Journal,* marzo, 1994, págs. 83-90.

Rozum, 92 — Rozum, J. A.: "Software Measurement Concepts for Acquisition Program Managers", Technical Report CMU/SEI-92-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, junio, 1992.

Rugaber, 00 — Rugaber, S.: "The Use of Domain Knowledge in Program Understanding", *Annals of Software Engineering*, 9, 2000, págs. 143-192.

Rumbaugh, 97 — Rumbaugh, J.: "Modeling Through the Years", *Journal of Object-Oriented Programming*, 10(4), julio/agosto, 1997, págs. 16-19.

Sagri, 89 — Sagri, M. M.: "Rated and Operating Complexity of Program- An Extension to McCabe's Theory of Complexity Measure", *ACM Sigplan Notices*, 24(8), agosto, 1989, págs. 8-12.

Sakai, 97 — Sakai, M.; Matsumoto, K.-I.; Torii, K.: "A New Framework for Improving Software Development Process on Small Computer Systems", *International Journal of Software Engineering and Knowledge Engineering*, 7(2), junio, 1997, págs. 171-184.

Schneidewind, 92 — Schneidewind, N. F.: "Methodology for Validating Software Metrics", *IEEE Transactions on Software Engineering*, 18(5), mayo, 1992, págs. 410-422.

Schneidewind, 94 — Schneidewind, N. F.: "Validating Metrics for Ensuring Space Shuttle Flight Software Quality", *IEEE Computer*, 27(8), agosto, 1994, págs. 50-57.

Schonberg, 00 — Schonberg, E.; Cofino, T.; Hoch, R.; Podlaseck, M.; Spraragen, S. L.: "Measuring Success", *Communications of the ACM*, 43(8), 2000, págs. 127-134.

SER, 95 — SER Consortium: "REBOOT: Reuse Based on Object Oriented Techniques", http://www.sema.es/projects/REBOOT/reboot.html#RMM, descargado el 2-10-1997, 1995.

Shah-Jarvis, 93 — Shah-Jarvis, A.: "Improvement Process Based on Metrics", Proc. *Third International Conference on Software Quality*, Lake Tahoe, Nevada, 4-6 de octubre, 1993, págs. 102-120.

Sheppard, 81 — Sheppard, S. B.; Kruesi, E.; Curtis, B.: "The Effects of Symbology and Spatial Arrangement on the Comprehension of Software Specifications", Proc. *5th International Conference on Software Engineering*, IEEE Computer Society Press, marzo, 1981.

Shepperd, 90 — Shepperd, M.: "Early Life-Cycle Metrics and Software Quality Models", *Information and Software Technology*, 32(4), 1990, págs. 311-316.

Shepperd, 93a — Shepperd, M. J.: *Software Engineering Metrics. I, Measures and Validations*, McGraw-Hill, Inglaterra, 1993.

Shepperd, 93b — Shepperd, M. J.; Ince, D. C.: *The Derivation and Validation of Software Metrics*, Oxford University Press, Oxford, 1993.

Snyder, 86 — Snyder, A.: "Encapsulation and Inheritance in Object-Oriented Programming Languages", Proc. 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), *ACM Sigplan Notices*, 21(11), noviembre, 1986.

Solingen, 99 — Solingen, R. V.; Berghout, E.: *The Goal/Question/Metric Method. A Practical Guide for Quality Improvement of Software Development*, McGraw-Hill, Berksshire, Reino Unido, enero, 1999.

Spencer, 95 — Spencer, E.; Gupta, A.; Bell, D.: "Enhancement of the Software Process: Human Factors as an Integral Component", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995, págs. 231-242.

Subramanian, 01 — Subramanian, G.; Cortin, W.: "An Empirical Study of Certain Object-Oriented Software Metrics", *The Journal of Systems and Software*, 59(1), 2001, págs. 57-64.

Sunohara, 81 — Sunohara, T.; Takano, A.; Uehara, K.; Ohkawa, T.: "Program Complexity Measure for Software Development Management", Proc. *5th International Conference on Software Engineering*, IEEE Computer Society Press, marzo, 1981, págs. 100-106.

Symons, 91 — Symons, C. R.: *Software Syzing and Estimating. (Function Point Analysis)*, 1991.

Taguchi, 89 — Taguchi, G.; Elsayed, E. A.; Hsiang, T., C.: *Quality Engineering in Production Systems*, McGraw-Hill, 1989.

Tang, 99 — Tang, M.; Kao, M.; Chen, M.: "An Empirical Study on Object-Oriented Metrics". Proc. *Sixth International Software Metrics Symposium* (METRICS'99), Boca Ratón, Florida, noviembre, 1999, págs. 242-249.

Tegarden, 91 — Tegarden, D. P.; Sheetz, S. D.; Monarchi, D. E.: "Effectiveness of Traditional Metrics for Object-Oriented Systems", Proc. *25th Hawaii International Conference on System Science*, Kauai, Hawaii, IEEE Computer Society Press, California, enero, 1991, págs. 359-368.

Tian, 97 — Tian, J.; Troster, J.; Palma, J.: "Tool Support for Software Measurement, Analysis and Improvement", *Journal of Systems and Software*, 39(2), noviembre, 1997, págs. 165-178.

Torres, 91a — Torres, W. R.; Samadzadeh, M. H.: "Software Reuse and Information Theory Based Metrics", *IEEE Transactions on Software Engineering*, 17(11), noviembre, 1991, págs. 1025-1029.

Torres, 91b — Torres, W. R.; Samadzadeh, M.: "Software Reuse and Information Theory Based Metrics", Proc. *1991 Symposium on Applied Computing*, IEEE Computer Soc. Press, Los Alamitos, California, 1991, págs. 437-446.

Travassos, 99 — Travassos, G.; Basili, V. R.: "Detecting Defects in Object-Oriented Designs: Using Reading Tecnhiques to Increase Software Quality", Proc. 14th *Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA'99), Denver, noviembre, 1999.

Troy, 81 — Troy, D. A.; Zweben, S. H.: "Measuring the Quality of Structured Designs", *Journal of Systems and Software*, 2(2), 1981, págs. 113-120.

Truong, 00 — Truong, D.; Chan, A.: "Measuring C++ Program Efficiency", *Dr. Dobb's Journal*, 25(10), octubre, 2000, págs. 62-67.

Varkoi, 99 — V arkoi, T. K.: "Development of Measurement Programs to Support Software Process Improvement in Small Software Companies". Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 141-149.

Vincent, 88b — Vincent, J. P.; Waters, A.; Sinclair, J.: *Software Quality Assurance. A Program Guide*, Vol. II, Englewood Cliffs, New Jersey, 1988.

Virtanen, 01 — Virtanen, P.: "Empirical Study Evaluating Component Reuse Metrics", Proc. *European Software Control and Metrics*, (ESCOM 2001), Londres, abril, 2001, págs. 125-136.

Vivar, 92 — Vivar, J.: *Norma de Programación en Lenguaje C*, Telefónica I+D, Madrid, 1992.

Wadaa, 99 — Wadaa, A.; Shen, S.; Eltoweissy, M.; Mata-Toledo, R.: "A New Approach to Interoperability Management in Large Scale Distributed Object Environment", Proc. *3rd World Multiconference on Systemics, Cybernetics and Informatics and 5th International Conference on Information Systems Analysis and Synthesis* (SCI/ISAS'99), Orlando, agosto, 1999, vol. 1, págs. 323-329.

Waguespack, 87 — Waguespack, L. J.; Badlani, S.: "Software Complexity Assessment: An Introduction and Annotated Bibliography", *ACM Sigsoft: Software Engineering Notes*, 12(4), octubre, 1987, págs. 52-71.

Walston, 77 — Walston, C. E.; Felix, C. P.: "A Method of Programming Measurement and Estimation", *IBM Systems Journal,* 16(1), 1977, págs. 54-73.

Walters, 79 — Walters, G. F.: "Applications of Metrics to a Software Quality Management Program", en *Software Quality Management*, Cooper, J. D. y Fisher, M. J., Petrocelli Books, 1979, págs. 143-158.

Wedde, 00 — Wedde, K. J.; Stålhane, T.: "The Use of Experts in Metrics Interpretation and Analysis", Proc. *ESCOM-SCOPE 2000*, Múnich, Alemania, abril, 2000, págs. 133-141.

Wegner, 01 — Wegner, M.: "The Impact of Quality Assurance and Metrics in Real Projects", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 397-405.

Weyuker, 88 — Weyuker, E. J.: "Evaluating Software Complexity Measures", *IEEE Transactions on Software Engineering*, 14(9), septiembre, 1988, págs. 1357-1365.

Wild, 93 — Wild, F. H.: "Experience Report- Creating Well Formed Class Inheritance Schemes in C++", 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'92), Addendum to the Proceedings, *OOPS Messenger*, 4, abril, 1993, págs. 17-23.

Wilde, 93 — Wilde, N.; Matthews, P.; Huitt, R.: "Maintaining Object-Oriented Software", *IEEE Software*, 10(1), enero, 1993, págs. 75-80.

Wilkie, 95 — Wilkie, G.; McCauley, C.; O'Neill, I; Tripathy, S.: "The IDIOM Framework for Improving the Software Process", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995, págs. 366-381.

Wilkie, 98 — Wilkie, F. G.; Hylands, B.: "Measuring Complexity in C++ Application Software", *Software- Practice and Experience*, 28(5), 1998, págs. 513-546.

Williams, 93 — Williams, J. D.: "Metrics for Object Oriente Projects", Proc. *Object Expo Euro Conference*, Londres, ACM SIGS Public., Nueva York, 12-16 de julio, 1993, págs. 13-18.

Wilson, 01 — Wilson, D. N.; Hall, T.; Badoo, N.: "A Framework for Evaluation and Prediction of Software Process Improvement Success", *The Journal of Systems and Software*, 59(2), 2001, págs. 135-142.

Woodfield, 81 — Woodfield, S. N.; Dunsmore, H. E.; Shen, V. Y.: "The Effect of Modularization and Comments on Program Comprehension", Proc. *5th International Conference on Software Engineering*, IEEE Computer Society Press, marzo, 1981.

Woodings, 01 — Woodings, T.; Bundell, G.: "A Framework for Software Projects Metrics", Proc. *ESCOM 2001*, Londres, abril, 2001, págs. 77-86.

Wu, 01 — Wu, C. T.: *Introducción a la Programación Orientada a Objetos con Java*, McGraw-Hill, 2001.

Xenos, 92 — Xenos, M.; Tsalidis, C.; Christodoulakis, D.: "Measuring Software Complexity Using Software Metrics", en *Software Quality: the Challenge in the 90s*, INTA, Proc. 3rd European Conference on Software Quality, Madrid, noviembre, 1992.

Xenos, 97 — Xenos, M.; Christodoulakis, D.: "Measuring Perceived Software Quality", *Information and Software Technology*, 39(6), junio, 1997, págs. 417-424.

Xenos, 99 — Xenos, M.: "Lessons Learned on Measuring Perceived Software Quality", Proc. *FESMA'99*, Amsterdam, Holanda, octubre, 1999, págs. 349-356.

Yacoub, 99 — Yacoub, S. M.; Ammar, H. H.; Robinson, T.: "Dynamic Metrics for Object-Oriented Design", Proc. *Sixth International Software Metrics Symposium* (METRICS'99), Boca Ratón, Florida, noviembre, 1999, págs. 50-61.

Yau, 78 — Yau, S. S.; Collofello, J. S.; MacGregor, T. M.: "Ripple Effect Analysis of Software Maintenance", Proc. *2nd International Computer Software and Applications Conference* (COMPSAC'78), 1978, págs. 60-65.

Zahran, 95 — Zahran, S.; Sanders, K.: "A Software Process Improvement Framework: The Theory and Practice at Bull Information Systems", Proc. *European Conference on Software Process Improvement* (SPI'95), Barcelona, noviembre, 1995.

Zahran, 98 — Zahran, S.: *Software Process Improvement*, Addison-Wesley, 1998.

Zelkowitz, 94 — Zelkowitz, M. V.; Tian, J.: "Measuring Prime Program Complexity", *IBM Information Sciences*, 77, 1994, págs. 325-350.

Zuse, 92 — Zuse, H.: "Properties of Software Measures", *Software Quality Journal,* 1, 1992, págs. 225-260.

Zweben, 79 — Zweben, S. H.; Halstead, M. H.: "The Frequency Distribution of Operators in PL/I Programs", *IEEE Transactions on Software Engineering*, 5(2), marzo, 1979, págs. 91-95.

En una ciencia física, un primer paso esencial hacia el aprendizaje de una materia consiste en encontrar principios de cálculo numérico y métodos para medir de forma práctica la calidad conectada con ella.

— Lord Kelvin

ANEXO I. EXPERIMENTOS

I. EXPERIMENTOS

I.1. INTRODUCCIÓN

Dentro de este capítulo se expondrán brevemente los datos concretos acerca de los experimentos que han sido mencionados en el capítulo de Resultados correspondientes al diseño experimental. Los experimentos se van a presentar agrupados por el sistema sobre el que se han aplicado. En los siguientes apartados se enunciará la descripción del sistema empleado, sus características, los datos concretos y los resultados particulares obtenidos en cada experimento. Parte de los sistemas que se han empleado para realizar los experimentos se han seleccionado del entorno académico debido a la facilidad para acceder a la documentación de desarrollo y al código fuente, y porque resultan programas fáciles de estudiar y analizar; el resto de los experimentos reflejan sistemas reales, entendiendo por "real" que uno de sus objetivos es ser puesto en explotación.

I.2. AGENDA PERSONAL

Descripción del sistema: El sistema consiste en una agenda personal sencilla que incorpora datos acerca de personas (amigos y contactos de trabajo) y citas. El usuario puede añadir, borrar y buscar personas y citas. El sistema realiza una serie de comprobaciones, como no admitir dos personas con el mismo nombre, ni dos citas a la misma hora, asegurándose, además, que las citas son concertadas con personas almacenadas en la agenda. El sistema se pensó para ser propuesto como ejercicio de programación orientada a objetos en un curso y la solución aquí utilizada fue desarrollada e implementada en C++ por uno de los profesores. Es importante mencionar que el desarrollador no sabía que se iba a evaluar la completitud de su solución. La solución utilizada tiene 11 clases y no llega a 2.000 líneas de código fuente, por lo que resulta fácil de evaluar por una persona.

I.2.1. EXPERIMENTO 1

Descripción del experimento: El procedimiento fue solicitar a dos expertos en orientación a objetos su valoración (entre 0 y 10) sobre cada una de las medidas (pero sin proporcionarles las fórmulas) de las tres fases del ciclo de vida. Seguidamente, se aplicó el modelo de calidad sobre el análisis, diseño e implementación del sistema obteniendo los valores de las medidas. Finalmente, se compararon ambos valores.

Objetivos: Se trata de evaluar si la opinión de los expertos con relación a las medidas coincide con la proporcionada automáticamente por el modelo de calidad.

Resultados: En primer lugar, la Figura I.1 muestra los valores globales de calidad obtenidos para la fase de análisis, donde se observan unos valores bastante aceptables de los factores, salvo para la eficiencia. El valor de calidad es de 0,59.

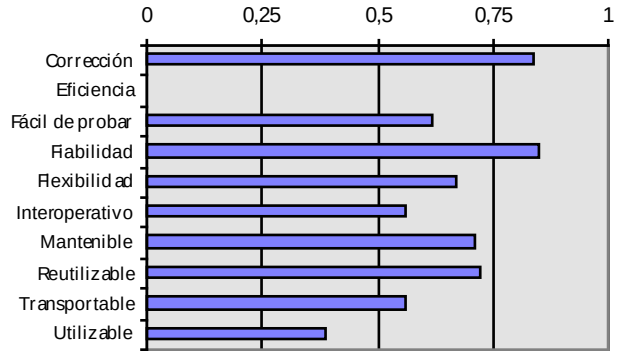


Figura I.1: Factores para el análisis de la agenda

Los valores de calidad obtenidos para la fase de diseño se presentan en la Figura I.2, obteniendo un valor total de 0,67.

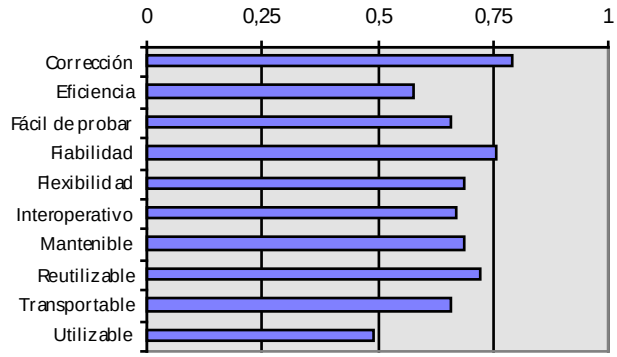


Figura I.2: Factores para el diseño de la agenda

Por último, la implementación ofreció un valor de calidad de 0,73, teniendo todos los factores una calidad considerablemente buena, tal como se refleja en la Figura I.3.

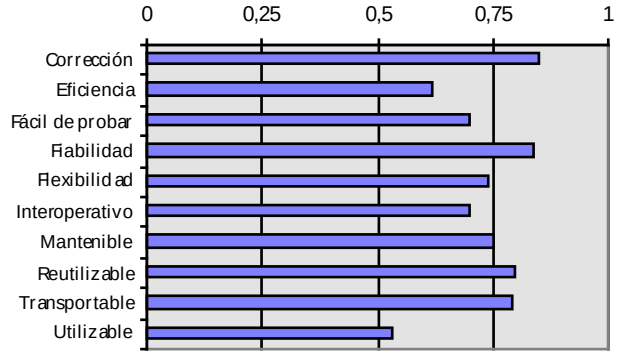


Figura I.3: Factores para la implementación de la agenda

En lo referente a la valoración de los expertos sobre las distintas medidas, se va a mostrar aquí únicamente la comparación para las medidas de uno de los criterios, para no alargar demasiado la explicación. Se ha elegido el criterio auto-descriptivo y en la Figura I.4 se muestran los resultados alcanzados por las distintas medidas en las tres fases del ciclo de vida.

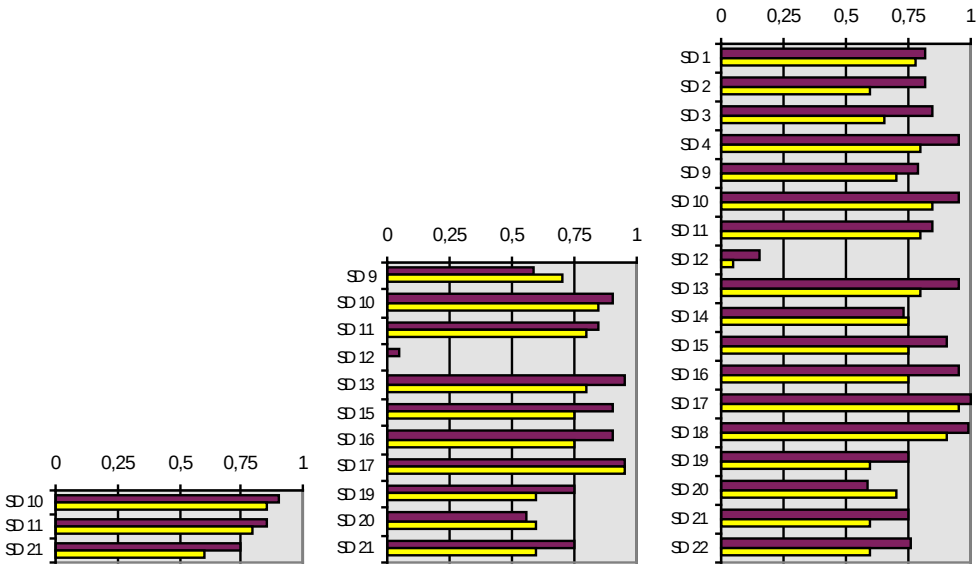


Figura I.4: Valores de las medidas de auto-descriptivo obtenidas por el modelo (en oscuro) y por los expertos (en claro) para las fases de análisis, diseño e implementación, respectivamente

Para realizar la comparación entre los resultados del modelo y de los expertos se ha aplicado el test de Kolmogorov-Smirnov con dos muestras [Chadravart, 67]. Este test permite determinar si dos muestras proceden de la misma distribución. Para su aplicación, se ha construido el evaluador tipo agregando las opiniones de los expertos con la media aritmética. Los niveles de significación así obtenidos para las tres fases han sido 0,10, 0,16 y 0,07. Esto indica que las muestras proceden de distribuciones similares, aunque en las opiniones se demuestra un cierto nivel de subjetividad.

I.2.2. EXPERIMENTO 9

Descripción del experimento: Primero se realizó una asignación de pesos a las medidas de completitud según una encuesta a diez expertos. En segundo lugar, se pidió a los expertos que valoraran la completitud del sistema. Seguidamente, se aplicó el modelo de calidad sobre el código del sistema y se compararon los resultados de los expertos con los del modelo de calidad, primero sin utilizar los pesos y después empleándolos.

Objetivos: Realizar una asignación directa de pesos a medidas. Comparar los resultados obtenidos con el modelo de calidad, tanto usando como sin usar pesos, con la opinión de expertos.

Resultados: En primer lugar, se encuestó a diez expertos con el fin de que dieran un peso subjetivo a cada una de las medidas de completitud para el sistema de la agenda (ya se ha comentado que el método de Análisis Jerárquico no resulta viable, por su tamaño, para las medidas), por lo que los expertos dieron una estimación directa de los pesos. Una vez obtenidos, se calculó la media de los pesos de cada medida y, finalmente, se realizó una normalización de tal forma que la suma de los pesos de todas las medidas fuera la unidad. La Tabla I.1 muestra los resultados de las medidas de implementación.

Medida	Media	Peso
CM_1	9,24	0,0408
CM_2	9,24	0,0408
CM_3	9,24	0,0408
CM_4	9,24	0,0408
CM_5	9,49	0,0419
CM_6	9,49	0,0419
CM_7	8,48	0,0375
CM_8	8,48	0,0375
CM_9	9,22	0,0407
CM_{10}	7,24	0,0321
CM_{11}	9,47	0,0418
CM_{12}	9,47	0,0418
CM_{13}	9,47	0,0418
CM_{14}	9,47	0,0418
CM_{15}	8,03	0,0355
CM_{16}	8,51	0,0376
CM_{17}	9,75	0,0431
CM_{18}	8,99	0,0398
CM_{19}	8,99	0,0398
CM_{20}	6,28	0,0277
CM_{21}	5,52	0,0244
CM_{23}	8,73	0,0386
CM_{24}	10,00	0,0442
CM_{27}	9,75	0,0431
CM_{28}	8,51	0,0376
CM_{29}	6,03	0,0266

Tabla I.1: Ponderación subjetiva y pesos de las medidas de completitud de implementación

Seguidamente, se aplicaron las medidas de completitud al código fuente en C++ obtenido durante la fase de implementación del presente sistema. Los valores que se obtuvieron por las distintas medidas individuales se presentan agrupados en la Figura I.5, junto a las proporcionadas por dos expertos, con el objetivo de que sirvan como comparación.

Una vez ejecutado el test de Kolmogorov-Smirnov, éste pone de manifiesto que la opinión de los expertos (agregada con la media aritmética) y los datos del modelo provienen de la misma distribución, al haberse obtenido un nivel de significación del 0,82.

El modelo de calidad para la fase de implementación ofreció un valor de completitud global de 0,93 (0,91, sin emplear los pesos). Por último, la opinión de los expertos acerca de la completitud del presente sistema arrojó un valor de 0,94 (0,92, sin emplear los pesos).

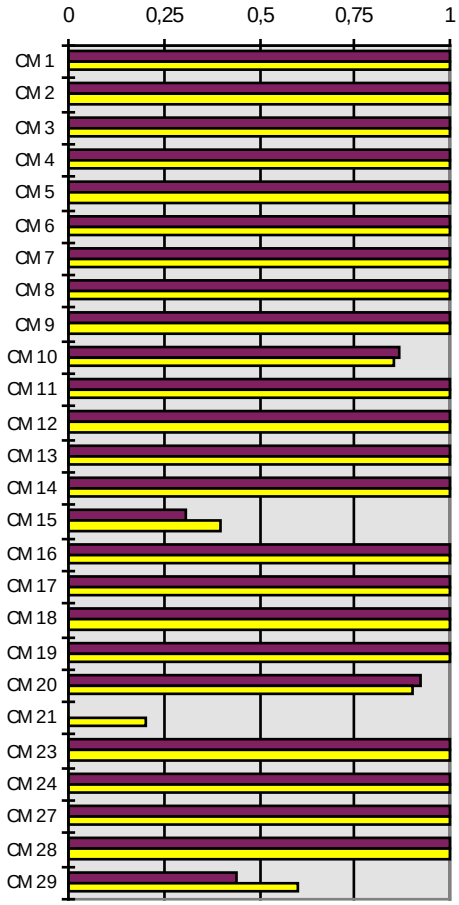


Figura I.5: Medidas de completitud de implementación para la agenda, obtenido con el modelo de calidad (en oscuro) y mediante la opinión de expertos (en claro)

I.2.3. EXPERIMENTO 22

Descripción del experimento: El experimento consiste en aplicar las medidas de completitud tanto en el diseño como en la implementación. Una vez obtenidos cada uno de los resultados, se ha informado al desarrollador para que dé su opinión al respecto. Hay que mencionar que el desarrollador no sabía que se iba a evaluar la completitud de su solución.

Objetivos: El objetivo es comprobar la interacción con el desarrollador, comprobando al mismo tiempo que los consejos proporcionados son adecuados. Para ello, se ha realizado el experimento sólo con las medidas de uno de los criterios.

Resultados: En primer lugar, se aplicaron las medidas de completitud sobre el diseño, proporcionando un resultado notablemente bueno. Para obtener el valor de completitud, se utilizó la siguiente fórmula (usando los pesos del Experimento 9):

$$CM = \sum_i \omega_i^{CM} \cdot CM_i = \sum_i \frac{1}{15} \cdot CM_i = 0,89$$

Los resultados para cada medida se muestran en la Figura I.6. En estos datos se observa que la principal debilidad del diseño desde el punto de vista de la completitud lo constituye la ausencia de constructores de clases y el poco uso que se hace de los métodos.

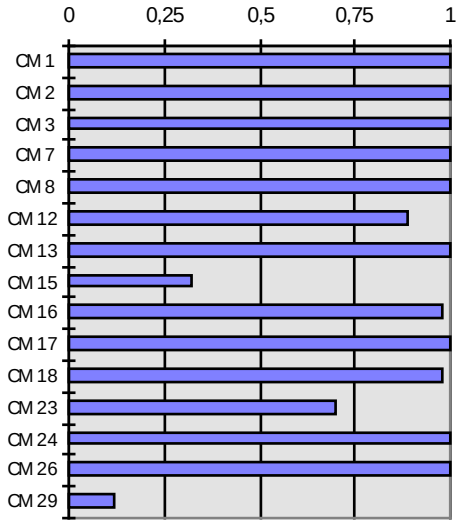


Figura I.6: Medidas de completitud de diseño para la agenda

Exceptuando las medidas CM_{15} y CM_{29}, se puede ver que las medidas han alcanzado excelentes resultados. Las sugerencias emitidas por el modelo de calidad se resumen en la Tabla I.2, junto con la justificación proporcionada por el desarrollador.

Sugerencias	Explicaciones
La clase Agenda no tiene destructor (medida CM_{15}). Además, las medidas CM_{12} y CM_{18} revelan que se hace referencia a este destructor en el diagrama de interacción de objetos, aunque no fue definido.	Se omitió el destructor de la definición de la clase Agenda al construir el diagrama de clases porque esta clase no requiere una asignación de memoria dinámica.
De acuerdo con la medida CM_{15}, cuatro de las siete clases no tienen constructor por omisión, y ninguna tiene constructor de copia ni operador de asignación.	El sistema no se desarrolló con el ánimo de la reutilización y, por tanto, cada clase sólo incluye los métodos usados en el programa. Por ello, tampoco se ha considerado necesario definir los constructores de copia y los operadores de asignación.
Como indica la medida CM_{16}, tres mensajes en el diagrama de interacción de objetos tienen parámetros que no concuerdan con los especificados en sus respectivas clases.	En el diagrama de interacción de objetos se ha utilizado un nombre genérico (datos) para resumir todos los parámetros requeridos para crear personas y citas. El propósito fue hacer un diagrama más claro.
La medida CM_{29} refleja que los métodos no se utilizan mucho. La mayoría sólo se emplean una vez.	El sistema en cuestión es pequeño, lo cual implica que los métodos no se usen mucho.

Tabla I.2: Sugerencias y justificación de los resultados de las medidas de diseño

Seguidamente, se aplicaron las medidas de completitud al código fuente en C++ obtenido durante la fase de implementación. En dicha implementación, se han mantenido las clases y métodos del diseño, habiéndose añadido cuatro nuevas clases para facilitar la programación. El valor de completitud, que superaba ligeramente al del diseño, se calculó sobre 26 de las 29 medidas de esta fase de acuerdo a la fórmula:

$$CM = \sum_i \omega_i^{CM} \cdot CM_i = \sum_i \frac{1}{26} \cdot CM_i = 0,93$$

Los valores obtenidos por las medidas individuales se presentan en la Figura I.7.

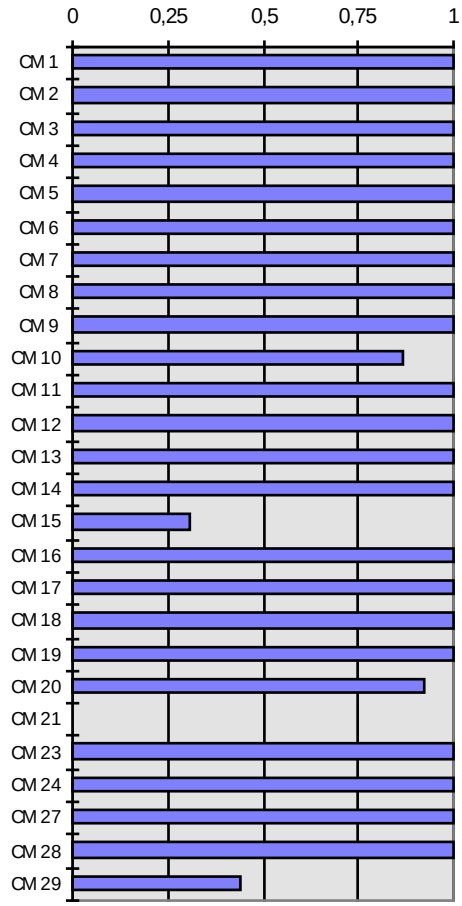


Figura I.7: Medidas de completitud de implementación para la agenda

En la figura se ve que, excepto por las medidas CM_{15} y CM_{29} (que heredan problemas del diseño) y la CM_{21}, los valores obtenidos son excelentes. La Tabla I.3 resume las sugerencias tras un análisis de los resultados, junto con las explicaciones dadas por los desarrolladores.

Sugerencias	Explicaciones
Hay atributos que solo son leídos o modificados una vez, como refleja la medida CM_{10}. Esto podría indicar algún tipo de omisión acerca del uso de dichos atributos.	El tamaño del sistema es pequeño, lo que significa que no es necesario un gran uso de los atributos.
De acuerdo con la medida CM_{15}, seis de las once clases no tienen constructor por omisión, tres no tienen destructor y ninguna tiene constructor de copia ni operador de asignación. Sería deseable que estuvieran definidos en cada clase.	La razón de este problema es de nuevo que sólo se han definido los métodos que se utilizan en el programa.
Como indica la medida CM_{20}, algunas decisiones no tienen código alternativo. Hay ocho sentencias `switch` y sólo una tiene la etiqueta `default`. Debería definirse el código alternativo para el caso de que la condición de evaluación proporcionara un valor inesperado.	Se desarrollo el código fuente para ser lo más corto posible, y se decidió omitir este código alternativo de las decisiones al asumir que no se podían producir nunca estas situaciones anómalas.
Se han añadido dos tipos abstractos de datos para reflejar fechas y horas. No se ha proporcionado a estos tipos operadores para llevar a cabo ciertas operaciones útiles. De esta forma, se podía haber sobrecargado los operadores de comparación, puesto que las horas y días de las citas tienen que ser comparadas de acuerdo al problema (CM_{21}).	La funcionalidad de estos tipos auxiliares se mantuvo al mínimo requerido para resolver el problema, por lo que, efectivamente, son relativamente incompletas.
La medida CM_{29} señala que los métodos siguen con poco uso. Aunque el uso ha crecido, casi la mitad sólo se llaman una vez.	La razón es, de nuevo, el pequeño tamaño del sistema desarrollado.

Tabla I.3: Sugerencias y justificación de los resultados de las medidas de diseño

En resumen, las medidas han permitido detectar ligeros defectos del sistema, aunque se ha visto que éste es lo suficientemente completo para satisfacer sus objetivos [Alonso, 01]. (Un estudio similar, pero analizando el criterio eficiencia de ejecución sobre un sistema desarrollado en Java se muestra en [Alonso, 02].)

I.3. CABALLO DE AJEDREZ

Descripción del sistema: El sistema resuelve el problema del salto de caballo de ajedrez. En pocas palabras, puede enunciarse así: "Dado un tablero de n filas y n columnas (con n^2 casillas), se trata de encontrar circuitos de n^2-1 movimientos legales del caballo de ajedrez de forma que cada cuadro del tablero sea visitado una sola vez". Los programas pertenecen a un capítulo de [Alonso, 95], donde se enseña la programación en C y C++, por lo que solo se dispone del código fuente.

I.3.1. EXPERIMENTOS 11 Y 25

Descripción del experimento: Se va a aplicar el modelo de calidad, en su fase de calidad de la implementación, sobre dos códigos fuente (uno realizado según el paradigma imperativo en C y otro según el paradigma orientado a objetos en C++) de un pequeño programa que resuelve el problema del salto de caballo. Así mismo, se aplicará el método de Análisis Jerárquico para obtener los pesos de los factores con relación a la calidad. Es importante destacar que ambos sistemas fueron desarrollados por la misma persona, siguiendo las mismas pautas y con los mismos objetivos.

Objetivos: El objetivo principal consiste en comprobar la validez o no del modelo sobre el paradigma imperativo. Posteriormente, se trata de verificar si el desarrollo del mismo sistema con el paradigma orientado a objetos ha servido para mejorar su calidad. Así mismo, se emplearán pesos asociados a los factores para el experimento. Por último, se analizarán los resultados que ofrece el modelo de calidad.

Resultados: Para la obtención de los pesos de los factores mediante el método de Análisis Jerárquico, se ha tenido en cuenta, a la hora de construir la matriz de relevancia, la naturaleza y el objetivo del programa. En la Tabla I.4 puede observarse la matriz obtenida, junto con el autovector que define los pesos de cada factor dentro de la calidad. Analizando este autovector, se puede ver que el factor que se ha considerado más importante es la corrección seguido de transportabilidad, mientras que los menos significativos han resultado ser interoperatividad y utilizabilidad.

En primer lugar, se ha aplicado el modelo de calidad sobre el código fuente en C realizado siguiendo una metodología estructurada. El valor de calidad global obtenido fue 0,54. Aplicando los pesos, se obtiene un valor global de calidad para el sistema de 0,58, ligeramente superior al calculado con la media aritmética, lo que refleja una calidad aceptable. La Figura I.8 muestra el vector de valores de los factores. Son de destacar los valores de eficiencia, fiabilidad, reutilizabilidad y transportabilidad por encima de 0,6. En la parte peor, están la interoperatividad y utilizabilidad con valores por debajo de 0,4.

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	7	5	3	3	9	5	5	3	9	**0,2639**
Eficiencia	1/7	1	1/3	1/3	1/5	7	1/5	1/5	1/5	7	**0,0395**
Fácil de probar	1/5	3	1	3	1/3	7	3	1/3	1/5	7	**0,0756**
Fiabilidad	1/3	3	1/3	1	1/2	7	1/3	1/5	1/6	7	**0,0559**
Flexibilidad	1/3	5	3	2	1	9	3	1/3	1/6	9	**0,1020**
Interoperatividad	1/9	1/7	1/7	1/7	1/9	1	1/6	1/9	1/9	1/3	**0,0118**
Mantenibilidad	1/5	5	1/3	3	1/3	6	1	1/4	1/6	8	**0,0691**
Reutilizabilidad	1/5	5	3	5	3	9	4	1	1/3	9	**0,1366**
Transportabilidad	1/3	5	5	6	6	9	6	3	1	9	**0,2298**
Utilizabilidad	1/9	1/7	1/7	1/7	1/9	3	1/8	1/9	1/9	1	**0,0158**

Tabla I.4: Matriz de relevancia de los factores dentro de la calidad y su autovector para el problema del Salto del Caballo

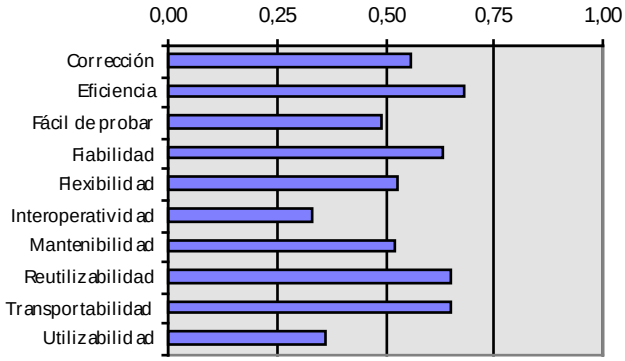


Figura I.8: Factores de calidad para el problema del Salto del Caballo en C

A continuación, se estudiarán los criterios de estos factores. La Figura I.9 presenta el vector de criterios para la eficiencia, donde destaca el elevado valor conseguido por la eficiencia de ejecución (siempre se ha afirmado que C es un lenguaje eficiente).

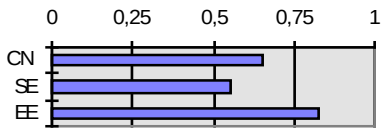


Figura I.9: Eficiencia en el Salto del Caballo en C

La Figura I.10 representa los valores de los criterios de la fiabilidad, donde destacan los altos valores conseguidos por la completitud (el problema se ha resuelto totalmente) y la simplicidad (dado que es un programa realizado fundamentalmente para la enseñanza, se ha logrado resolver el problema de una forma poco compleja y, por tanto, sencilla). El bajo valor de la tolerancia a errores, no tiene importancia dada la sencillez del sistema.

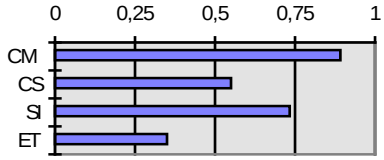


Figura I.10: Fiabilidad en el Salto del Caballo en C

La Figura I.11 refleja el vector de los criterios de reutilizabilidad, donde son de resaltar los altos valores de los criterios relativos a la independencia del *hardware* y del *software* (lógico al ser un programa para enseñar a programar en C, independientemente de la plataforma) y la generalidad (el problema se ha resuelto de una manera general). El bajo valor de la documentación se debe a la práctica inexistencia de documentación, salvo el propio texto explicativo.

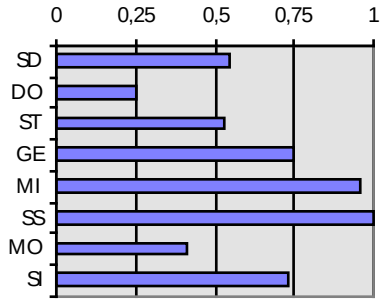


Figura I.11: Reutilizabilidad en el Salto del Caballo en C

La Figura I.12 refleja los criterios de transportabilidad, haciendo notar que el sistema puede ser transportado fácilmente a otro entorno, al estar realizado en un lenguaje como C.

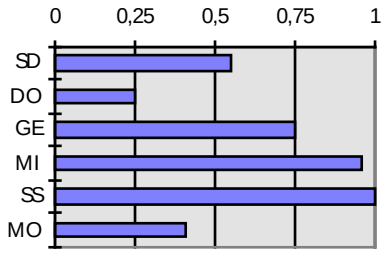


Figura I.12: Transportabilidad en el Salto del Caballo en C

Los valores de los criterios de interoperatividad aplicables al problema se observan en la Figura I.13, donde se aprecia la baja documentación.

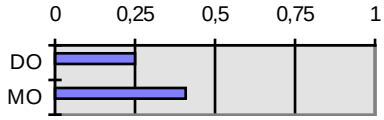


Figura I.13: Interoperatividad en el Salto del Caballo en C

Por último, la Figura I.14 presenta los criterios de utilizabilidad, donde destaca el bajo valor obtenido por el criterio entrenamiento, puesto que el objetivo es enseñar a programar en C y, por tanto, no se ha previsto una enseñanza en su utilización.

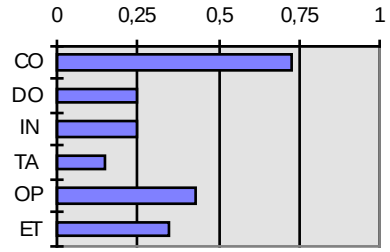


Figura I.14: Utilizabilidad en el Salto del Caballo en C

Para la versión de la solución construida en C++, se ha obtenido un valor de calidad global de 0,61, lo cual indica que este programa tiene una calidad adecuada. Aplicando los pesos, se obtiene un valor global de calidad para el sistema de 0,66, que como puede apreciarse, es ligeramente superior al obtenido con la media aritmética, reflejando de esta manera mejor la calidad del sistema según sus objetivos.

En la Figura I.15 se muestra una gráfica que resume el vector de valores obtenido para cada uno de los factores. Nótese la ausencia del factor integridad debido a que el programa no necesita restringir el acceso a los usuarios. Puede observarse que la mayoría de los factores presentan valores buenos, siendo de destacar los resultados de corrección, fiabilidad, mantenibilidad, reutilizabilidad y transportabilidad por encima de la media. En el lado negativo se encuentran la interoperatividad y utilizabilidad con valores bastante por debajo de la media.

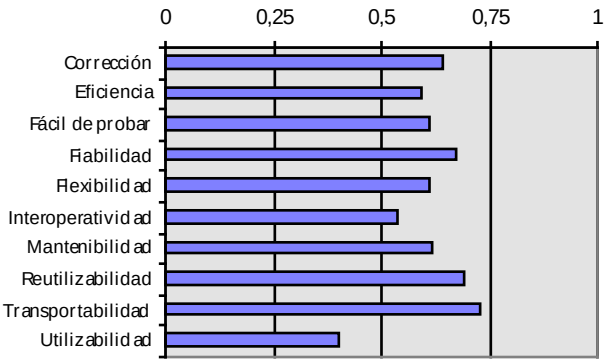


Figura I.15: Factores de calidad para el problema del Salto del Caballo en C++

Seguidamente se estudian los criterios de estos factores destacados. La Figura I.16 representa los valores del vector de criterios para la corrección, donde destaca el elevado valor conseguido por la completitud (puesto que, evidentemente, la solución al problema es completa) y el bajo valor obtenido para la documentación, debido a que el programa carece de documentación de usuario y solamente se ha tenido en cuenta la documentación que explica su funcionamiento interno.

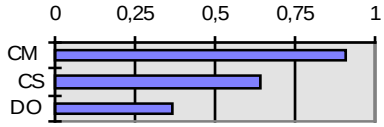


Figura I.16: Corrección en el Salto del Caballo en C++

La Figura I.17 refleja los valores del vector de criterios para la fiabilidad donde es de destacar los altos valores conseguidos por la completitud y la simplicidad (dado que es un programa realizado fundamentalmente para la enseñanza, se ha logrado resolver el problema de una forma poco compleja y, por tanto, sencilla). El valor bajo obtenido para la tolerancia a errores, en este caso no tiene mayor importancia dada la sencillez del sistema.

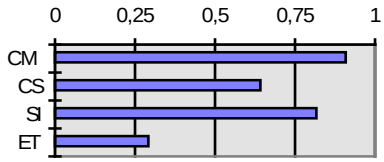


Figura I.17: Fiabilidad en el Salto del Caballo en C++

La Figura I.18 muestra los valores del factor mantenibilidad donde destacan los altos valores conseguidos por el criterio auto-descriptivo (se ha conseguido un programa fácilmente comprensible mediante la mera lectura del código) y la simplicidad.

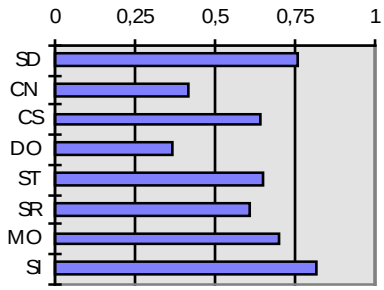


Figura I.18: Mantenibilidad en el Salto del Caballo en C++

La Figura I.19 representa los valores de los criterios del factor reutilizabilidad, donde son de destacar los valores cercanos a la unidad que presentan los criterios relativos a la independencia del *hardware* y del *software* (lógico al ser un programa para enseñar a programar en C++, independientemente de la plataforma), además del ya mencionado de la simplicidad. El bajo valor de datos estándar se debe a que el tamaño del problema no daba pie a la utilización de muchas estructuras de datos, aunque se podría haber utilizado alguna más.

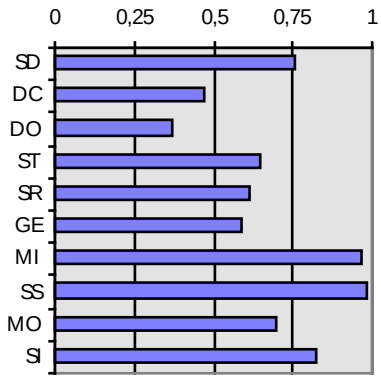


Figura I.19: Reutilizabilidad en el Salto del Caballo en C++

La Figura I.20 refleja los criterios de transportabilidad, haciendo notar que el sistema puede ser transportado fácilmente a otro entorno.

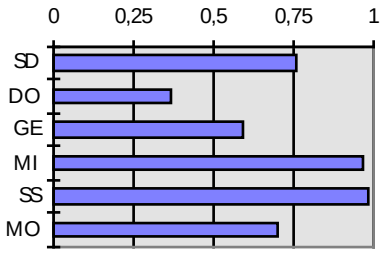


Figura I.20: Transportabilidad en el Salto del Caballo en C++

Los valores de los criterios de interoperatividad se muestran en la Figura I.21, donde se aprecia la buena modularidad y estructuración y la baja documentación.

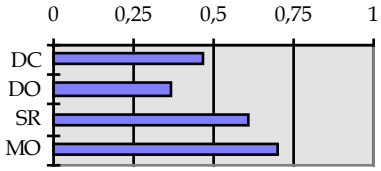


Figura I.21: Interoperatividad en el Salto del Caballo en C++

Por último, la Figura I.22 refleja los criterios de utilizabilidad, donde destaca el bajo valor obtenido por el criterio entrenamiento, dado que es un sistema para la enseñanza de la programación y no incluye una enseñanza en su utilización.

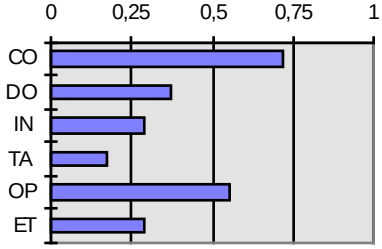


Figura I.22: Utilizabilidad en el Salto del Caballo en C++

A la vista de estos resultados, puede concluirse que los principales defectos que presenta el sistema en C++ vienen ocasionados por la finalidad docente del programa, y a que, dado su pequeño tamaño, no se han empleado muchas de las posibilidades de la orientación a objetos en C++, como el polimorfismo, las clases y funciones genéricas o el manejo de excepciones, aunque se podrían solucionar algunos aspectos. A modo de ejemplo, se podrían citar los bajos valores presentados por algunos atributos de las clases para las medidas DC_5, ST_3 y GE_4. Tras estudiarlos detenidamente, se observa que dichos atributos son públicos y, por tanto, pueden ser accedidos directamente, sin utilizar los servicios de sus clases. Esto se podría solucionar fácilmente programando dichos servicios y privatizando todos los atributos de las clases.

La Figura I.23 muestra una comparación que resume los factores y los criterios para los dos desarrollos: en C, empleando el paradigma imperativo, y en C++, empleando el paradigma orientado a objetos.

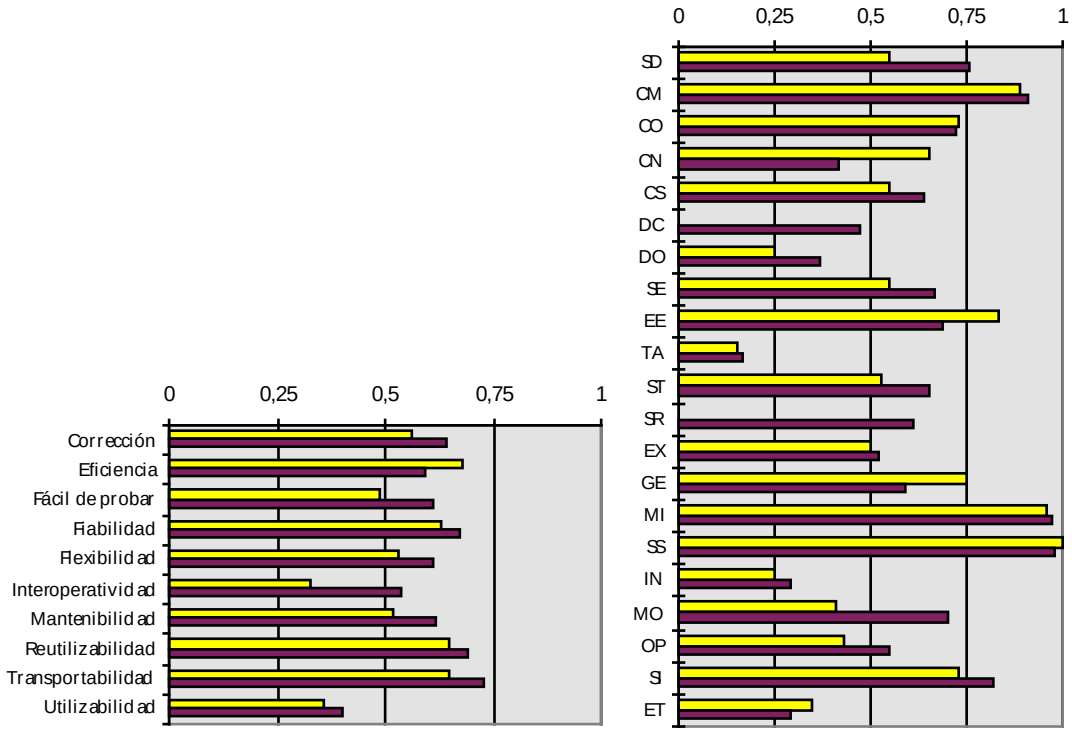


Figura I.23: Factores y criterios, respectivamente, para el problema del Salto del Caballo en C (en claro) y en C++ (en oscuro)

De la comparación de los datos, es de destacar que todos los factores, salvo uno, aumentan al emplear la orientación a objetos. Son interesantes las mejoras en los factores fácil de probar, flexibilidad, mantenibilidad y reutilizabilidad, características que la orientación a objetos, históricamente, ha favorecido. En lo referente a los criterios, es digno de destacar la mejora experimentada por auto-descriptivo, estabilidad, modularidad y simplicidad, que demuestra lo que los objetos pueden aportar a un sistema. Por otro lado, hay que mencionar las pérdidas sufridas por los criterios concisión, eficiencia de ejecución y generalidad. La explicación es sencilla: C++ es un lenguaje que, al emplear los componentes de la orientación a objetos, crea programas de mayor tamaño que los creados en C; ya se ha comentado antes, también, que C es un lenguaje que favorece la velocidad de ejecución; por último, la generalidad en C aumenta porque las

funciones independientes suelen ser siempre algo más generales que una clase completa. También hay que destacar que los criterios datos estándar y estructuración no han podido ser calculados para el problema en C, al no emplear ninguna de sus medidas; resulta especialmente lógico el caso de la estructuración, pues es un criterio substancialmente introducido para manejar las estructuras que proporciona la orientación a objetos.

En lo referente a las medidas utilizadas para realizar la evaluación de ambos sistemas, para el caso en el que el lenguaje era C++ se emplearon un 84% de las medidas totales, mientras que para el sistema implementado en C, únicamente se pudieron usar un 39% de ellas. Esto se debe a que muchas de las medidas están especialmente ideadas para su aplicación sobre lenguajes orientados a objetos, que disponen de una serie de estructuras y construcciones ausentes en lenguajes imperativos como C.

I.3.2. EXPERIMENTO 15

Descripción del experimento: Una vez aplicado el modelo de calidad de la implementación sobre el código C++, se eliminaron aquellos criterios que se consideraban innecesarios para el problema con el fin de observar cómo varía la calidad del sistema.

Objetivos: Se pretende verificar el comportamiento si los criterios superfluos se eliminan.

Resultados: Estudiando los objetivos principales que perseguía el desarrollo de esta solución (didácticos), puede considerarse que los siguientes criterios no fueron tomados como prioritarios durante la construcción del sistema o, simplemente, no eran aplicables:

- Auditoría de accesos y control de acceso: evidentemente, en el sistema no había que realizar ningún tipo de control de acceso a los usuarios, por lo que estos criterios carecen de sentido. De hecho, ninguna de sus medidas pudo ser utilizada, por lo que no se tuvieron en cuenta originalmente.
- Comunicaciones estándar: el sistema es una aplicación independiente que no requiere en ningún momento comunicarse con otras aplicaciones. Ninguna de sus medidas se utilizó en primera instancia, por lo que no se ha tenido en cuenta.
- Comunicatividad: el sistema carece de entradas y la salida se limita a pintar en pantalla el tablero completo indicando el orden de recorrido de las casillas. A pesar de todo, el valor obtenido fue de un 0,72, y solo se utilizaron 8 de sus medidas.
- Entrenamiento: al ser un sistema utilizado en la enseñanza, no tiene como objetivo ser utilizado por usuarios. Las 6 medidas utilizadas dieron un valor de 0,17.
- Instrumentación: el sistema no incorpora ningún mecanismo para facilitar su depuración debido a su pequeño tamaño y a su objetivo. Las 7 medidas empleadas proporcionaron un valor de 0,29.
- Operatividad: el sistema no pretende facilitar su uso al usuario sino demostrar cómo se programa; de hecho, el usuario no tiene más que ejecutarlo. Las 12 medidas usadas ofrecieron un valor de 0,55, un valor bueno debido a su extrema sencillez.
- Seguimiento: no existe un análisis ni un diseño del sistema, por lo que este criterio carece de sentido.
- Tolerancia a errores: no existe necesidad de realizar ningún tipo de actuación ante la aparición de errores dada la sencillez del sistema. Originalmente, se habían usado 14 medidas que totalizaron un valor de 0,29.

El resto de los criterios se creen necesarios, considerando que aportan calidad en general a un programa orientado a objetos. Hay que puntualizar que, además, también se eliminaron algunas medidas que, a pesar de estar en uno de los criterios admitidos para este problema, miden aspectos a eliminar. Un ejemplo de tales medidas pertenece al criterio documentación, puesto que contiene medidas relacionadas con la documentación técnica (adecuadas) y otras (inadecuadas como de la DO_{13} a la DO_{16} y DO_{19}) con la del usuario; para este experimento, también se prescindió de este tipo de medidas.

Así mismo, el factor utilizabilidad pierde también su sentido para el problema, al quedarse únicamente con la documentación técnica, por lo que tampoco se deberá tener en cuenta.

De esta manera, los nuevos valores obtenidos se presentan en la Figura I.24. Se puede observar cómo todos los factores han experimentado una leve mejoría al ser eliminados de sus cálculos algunas medidas y criterios irrelevantes para el objetivo del problema.

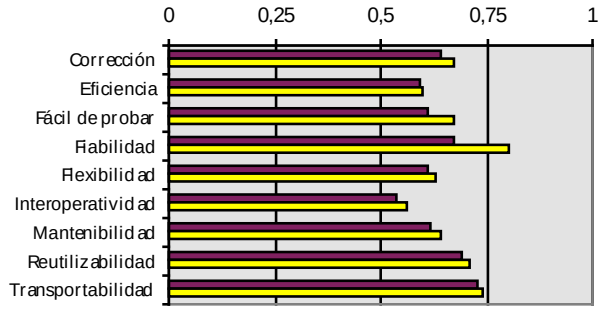


Figura I.24: Nuevos factores de calidad para el problema del Salto del Caballo en C++ (en claro) comparados con los anteriores

El valor de calidad total ahora ha aumentado hasta 0,67 utilizando la media, mientras que si se emplean los pesos reflejados en la Tabla I.4 la calidad obtenida es de un 0,69. Se puede comprobar cómo ahora la diferencia entre usar o no usar pesos ha disminuido. Ello se debe a que los pesos premiaban los factores más importantes mientras que penalizaban a los menos importantes. Al haber eliminado los criterios menos significativos, los datos que quedan son los considerados fundamentales en el problema.

I.4. CAJERO AUTOMÁTICO

Descripción del sistema: El sistema implementa el problema del cajero automático. El sistema se ha desarrollado siguiendo el enunciado y planteamiento de la solución indicados en [Booch, 99] (el banco, que no se considera parte del sistema, se simula con un proceso que responde a las peticiones del cajero). El objetivo del desarrollo era su utilización en un curso de diseño y programación orientada a objetos.

I.4.1. EXPERIMENTOS 2, 3, 6 Y 8

Descripción del experimento: El procedimiento consistió en solicitar a dos expertos su valoración (entre 0 y 10) sobre cada una de las medidas (sin proporcionarles las fórmulas) de las tres fases del ciclo de vida. También se solicitó la opinión acerca de los criterios, factores y la calidad global. Seguidamente se aplicó el modelo de calidad sobre el análisis, diseño e implementación en C++ del sistema, obteniendo los valores de las medidas. También se ha construido la matriz de relevancia para criterios y factores

utilizando la opinión de tres profesores que han impartido diversos cursos de orientación a objetos. Finalmente se compararon los valores de las medidas.

Objetivos: Los objetivos consisten en la comparación de los valores obtenidos por las medidas, los criterios, los factores y la calidad total con la opinión de expertos. También se usaron pesos en los criterios y en los factores para estudiar el comportamiento de los resultados del modelo. Un objetivo adicional es verificar la utilidad de algunas medidas que en otros experimentos no han sido tenidas en cuenta, como las correspondientes a integridad. El problema abordado tiene características diferentes al resto que sirven para probar aspectos distintos del modelo de calidad en las sucesivas fases del ciclo de vida.

Resultados: En primer lugar, se comenzó aplicando el método de Análisis Jerárquico mediante una entrevista a cada uno de los tres profesores, especialistas en orientación a objetos, uno de los cuales sería el diseñador y desarrollador del sistema. La Tabla I.5 muestra los pesos de los criterios en cada factor y la Tabla I.6 muestra los datos obtenidos para los factores. Examinando el autovector obtenido para este problema, se descubre que los dos factores más importantes son la fiabilidad y la integridad, debido, fundamentalmente, a la naturaleza del sistema.

Corrección	
CM	0,5755
CS	0,0551
DO	0,2742
TR	0,0952

Eficiencia	
CN	0,1373
SE	0,2395
EE	0,6232

Fácil de probar	
SD	0,1287
CO	0,0350
DO	0,1664
SR	0,2197
IN	0,0796
MO	0,2891
SI	0,0815

Fiabilidad	
CM	0,1305
CS	0,0440
AU	0,2872
SI	0,0323
ET	0,5060

Flexibilidad	
SD	0,0766
CC	0,1457
DC	0,1119
DO	0,1682
ST	0,0787
SR	0,1768
EX	0,0539
GE	0,0951
MO	0,0368
SI	0,0563

Integridad	
AA	0,3333
AC	0,6667

Interoperativo	
CC	0,3928
DC	0,1564
DO	0,2556
SR	0,0729
MO	0,1223

Mantenible	
SD	0,1425
CN	0,0322
CS	0,0409
DO	0,1694
ST	0,1770
SR	0,0903
MO	0,1784
TR	0,1193
SI	0,0500

Reutilizable	
SD	0,0440
CC	0,0510
DC	0,0664
DO	0,1289
ST	0,1166
SR	0,0738
GE	0,1588
MI	0,0719
SS	0,0602
MO	0,1571
TR	0,0436
SI	0,0277

Transportable	
SD	0,0554
CC	0,1963
DO	0,0637
GE	0,1138
MI	0,2625
SS	0,2157
MO	0,0926

Utilizable	
CO	0,1857
DO	0,0344
IN	0,0319
TA	0,0455
OP	0,3932
ET	0,3093

Tabla I.5: Pesos de los criterios dentro de cada factor para el problema del cajero automático

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Integridad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	7	7	1/4	5	1/2	6	5	7	7	4	**0,1838**
Eficiencia	1/7	1	4	1/6	5	1/4	4	3	5	5	2	**0,0974**
Fácil de probar	1/7	1/4	1	1/6	2	1/5	3	1/3	1	2	1/4	**0,0362**
Fiabilidad	4	6	6	1	6	1/2	6	5	7	7	5	**0,2322**
Flexibilidad	1/5	1/5	1/2	1/6	1	1/6	1	1/2	4	4	1/2	**0,0390**
Integridad	2	4	5	2	6	1	6	3	6	6	2	**0,2019**
Interoperatividad	1/6	1/4	1/3	1/6	1	1/6	1	1/3	1	2	1/4	**0,0257**
Mantenibilidad	1/5	1/3	3	1/5	2	1/3	3	1	5	6	1	**0,0705**
Reutilizabilidad	1/7	1/5	1	1/7	1/4	1/6	1	1/5	1	1	1/3	**0,0222**
Transportabilidad	1/7	1/5	1/2	1/7	1/4	1/6	1/2	1/6	1	1	1/4	**0,0184**
Utilizabilidad	1/4	1/2	4	1/5	2	1/2	4	1	3	4	1	**0,0727**

Tabla I.6: Matriz de relevancia de los factores dentro de la calidad y su autovector para el problema del cajero automático

El sistema se analizó tomando la información proporcionada por [Booch, 99] como punto de partida. Una vez finalizado el análisis inicial, se obtuvo una calidad de 0,69 mediante la aplicación de las medidas de análisis, tal como se muestra en la Figura I.25.

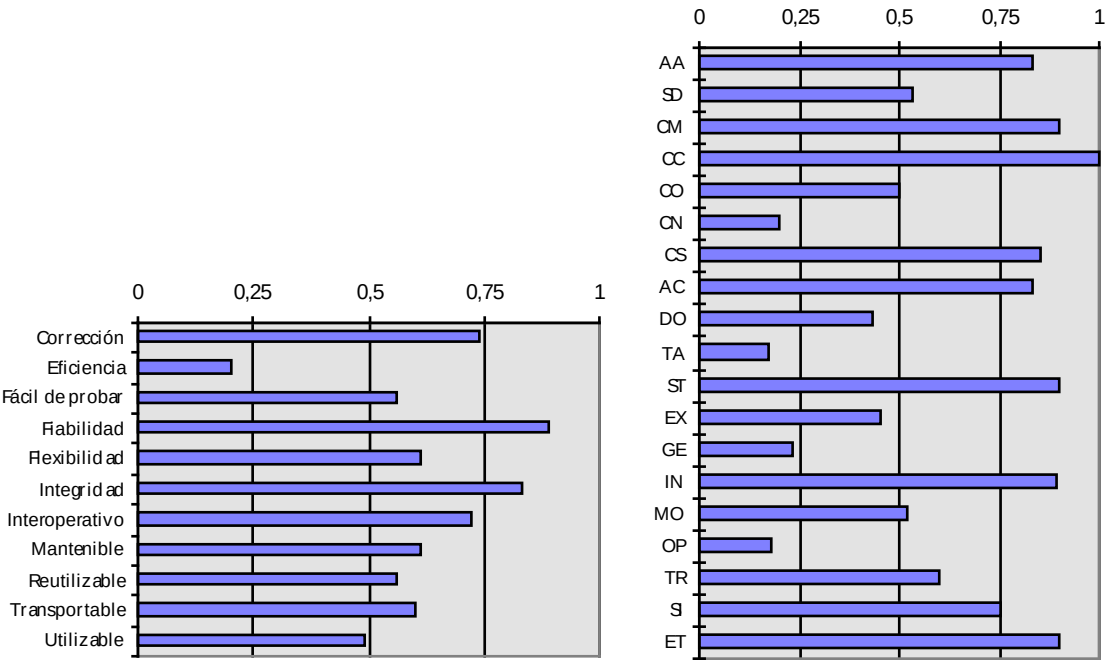


Figura I.25: Factores y criterios de calidad para el análisis

Cuando se completó el diseño, tomando la información y sugerencias proporcionadas tras el análisis, se aplicó el modelo de calidad utilizando las medidas de diseño, obteniendo de esta manera los valores de los criterios y los factores de la fase de diseño mostrados en la Figura I.26. El valor global de calidad obtenido en esta fase fue de 0,73, lo cual implica un diseño de alta calidad. Todos los factores tienen un valor de calidad por encima de 0,5, lo que indica que los resultados son buenos (en especial la integridad,

poniendo de manifiesto la naturaleza del sistema), aunque todavía se pueden mejorar. Si se estudian en detalle los valores de los criterios, se descubre que el diseño tiene algunos defectos, al no haber alcanzado valores suficientes en los criterios concisión (no había requisitos al respecto ni se realizó ningún esfuerzo por realizar un diseño conciso), entrenamiento (no hay necesidad de enseñar a un usuario a utilizar un cajero automático), operatividad (este criterio debería mejorarse para mejorar la utilización del sistema) y documentación (la documentación es importante en todos los proyectos, por tanto, este criterio también necesita una mejora). Analizando cuidadosamente las medidas de estos criterios, el desarrollador puede afrontar estos problemas.

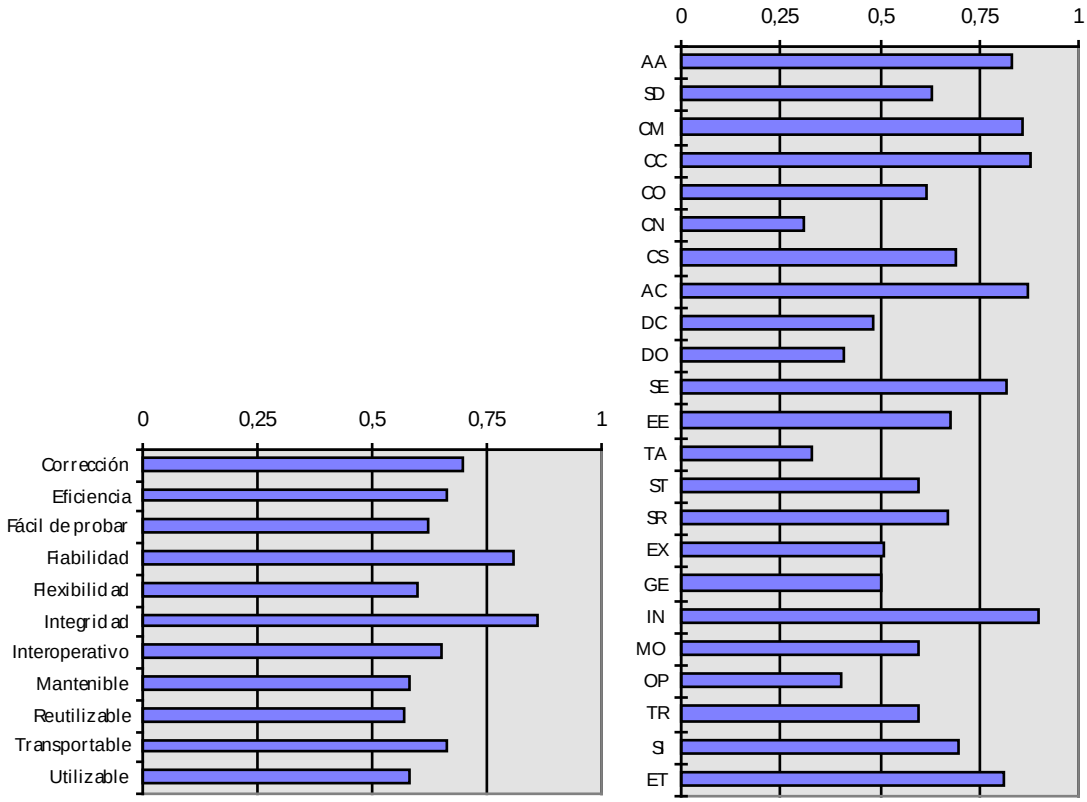


Figura I.26: Valores de los factores y de los criterios obtenidos para el diseño del cajero automático

Como puede observarse, se han utilizado 23 de los 26 criterios para evaluar el diseño. Esto ocurre porque hay tres criterios que no disponen de medidas para su cálculo durante la fase de diseño. Estos criterios son independencia de la máquina, independencia del sistema *software* y precisión, que solamente tienen sentido en la implementación.

Seguidamente, tras realizar la implementación teniendo en cuenta los consejos aportados por el modelo de calidad, éste se aplicó sobre el código C++, donde, aproximadamente, un 8% de las medidas no fueron de aplicación. Los valores de los factores y criterios se muestran en la Figura I.27. La calidad global fue de 0,82, indicando un sistema con una calidad excelente. Mirando los criterios, la mayoría de ellos presentan valores adecuados. Los más bajos son la concisión y el entrenamiento, por las razones ya comentadas. Los valores de documentación y operatividad aumentaron satisfactoriamente gracias a la información proporcionada por el modelo de calidad.

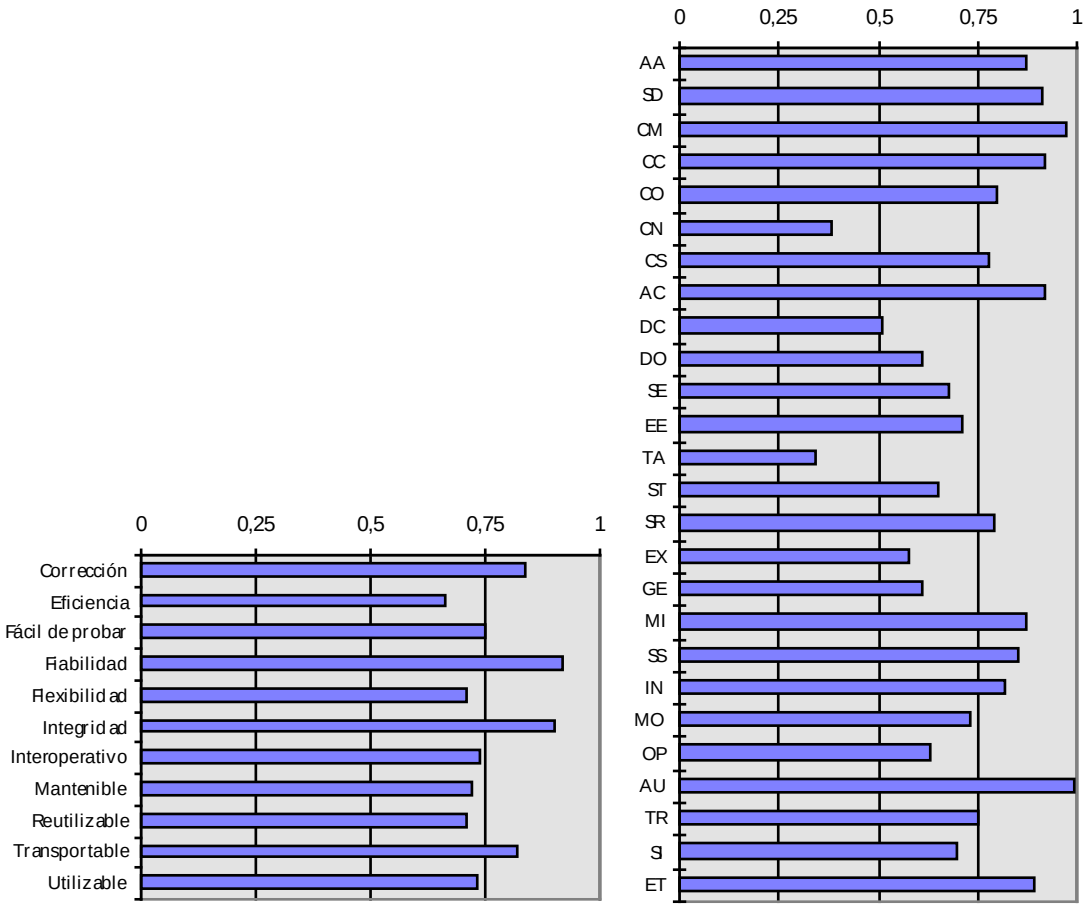


Figura I.27: Valores de los factores y de los criterios obtenidos para la implementación del cajero automático

Los buenos valores de calidad obtenidos ponen de manifiesto la experiencia y buenos conocimientos en orientación a objetos que posee el autor del sistema, si bien hay que decir que como sabía que su trabajo iba a ser evaluado mediante este modelo de calidad, puso un especial cuidado durante el proceso de desarrollo, analizando escrupulosamente los consejos y medidas obtenidas del diseño. Es de destacar, también, el hecho de que los tres factores que mejores valores han obtenido son, precisamente, los tres que se había considerado más importantes, para este desarrollo, tras aplicar el método de Análisis Jerárquico.

Con respecto a la valoración de los expertos sobre las distintas medidas, se presenta la comparación para las medidas del criterio expansibilidad, a modo de ejemplo. En la Figura I.28 se muestran los resultados alcanzados por las medidas en las tres fases del ciclo de vida. Para determinar si los resultados del modelo y de los expertos proceden de la misma distribución, se ha aplicado el test de Kolmogorov-Smirnov. Tras agregar las opiniones de los expertos, utilizando la media aritmética, para construir la figura de evaluador tipo, para el ejemplo de la expansibilidad, el nivel de significación obtenido para la fase de análisis, diseño e implementación, ha sido, respectivamente, 0,96, 0,53 y 0,72. Este indica que las muestras proceden de la misma distribución. Para el resto de las medidas, también se obtuvieron resultados positivos.

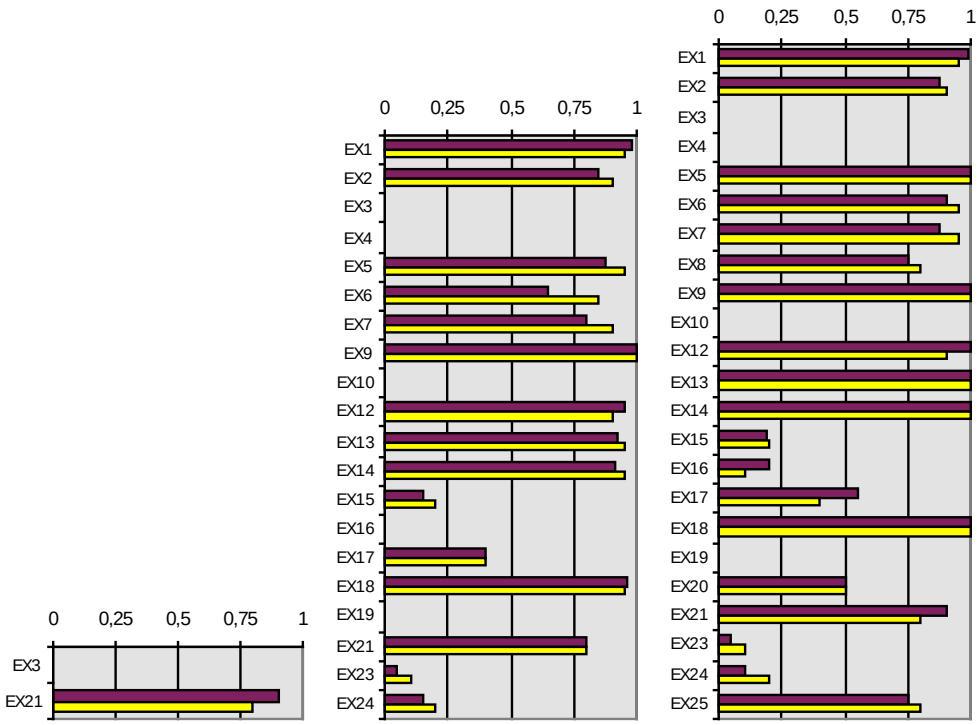


Figura I.28: Valores de las medidas de expansibilidad obtenidas por el modelo (en oscuro) y por los expertos (en claro) para las fases de análisis, diseño e implementación, respectivamente

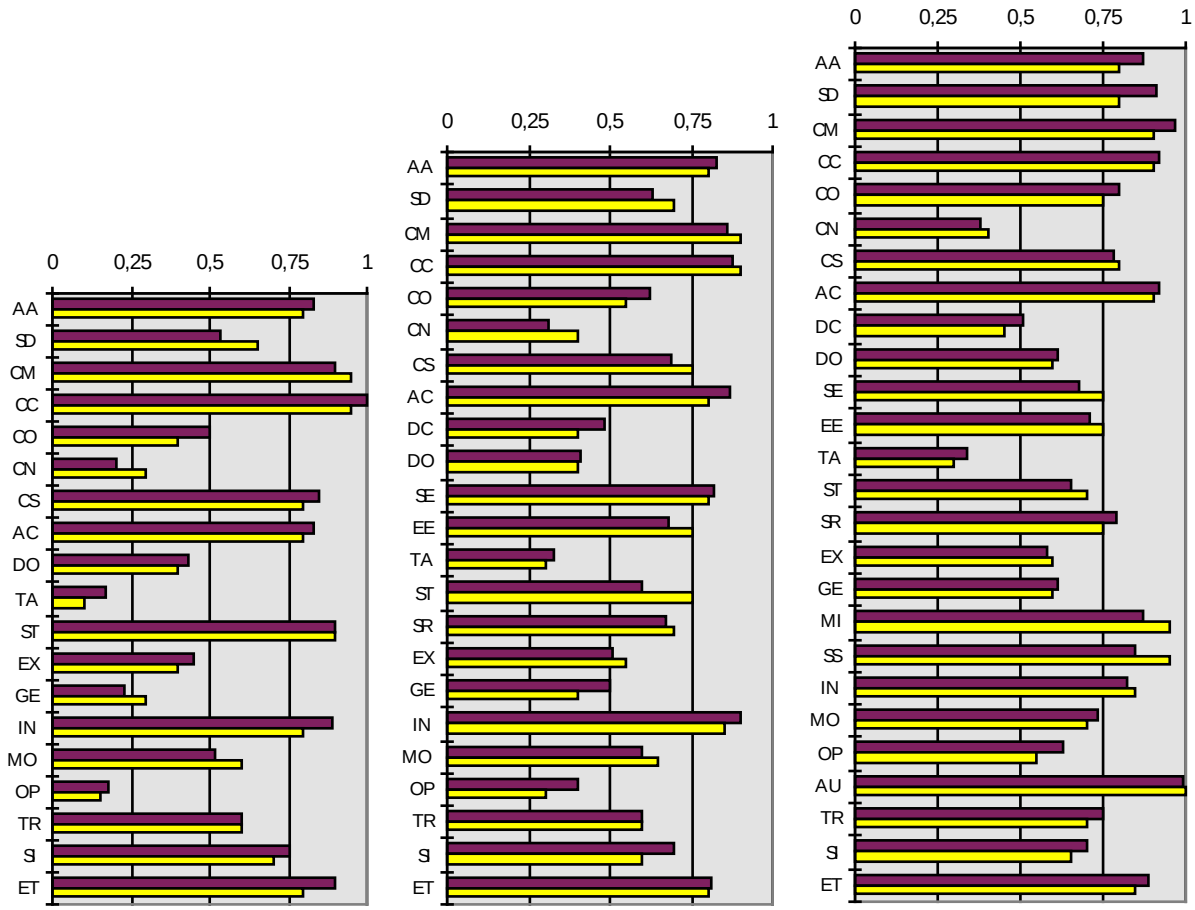


Figura I.29: Valores de los criterios obtenidos por el modelo (en oscuro) y por los expertos (en claro) para las fases de análisis, diseño e implementación, respectivamente

En la Figura I.29 se muestra la comparación entre los resultados de los criterios del modelo en las tres fases junto a la opinión de los expertos. De nuevo se aplica el test de Kolmogorov-Smirnov, tras agregar las opiniones de los expertos. El nivel de significación obtenido para la fase de análisis, diseño e implementación, ha sido, respectivamente, 0,71, 0,65 y 0,95, indicando que las muestras proceden de la misma distribución.

Análogamente, la Figura I.30 presenta la comparación entre los resultados obtenidos por el modelo de calidad y la opinión subjetiva de los expertos, donde también puede observarse que no existe demasiada disimilitud entre ambos valores. El nivel de significación obtenido para las fases de análisis, diseño e implementación en el test de Kolmogorov-Smirnov ha sido de 0,46. Este indica que las muestras proceden de la misma distribución.

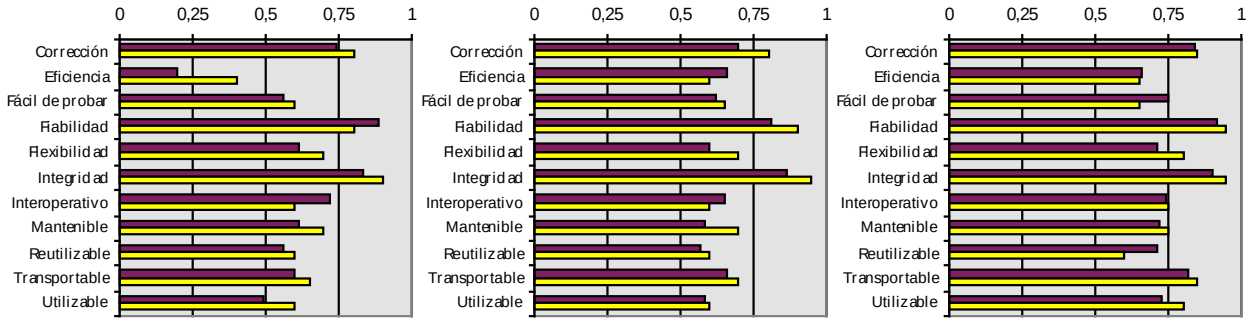


Figura I.30: Valores de los factores obtenidos por el modelo (en oscuro) y por los expertos (en claro) para las fases de análisis, diseño e implementación, respectivamente

Por último, la calidad total comparada entre el modelo (tanto con pesos como sin pesos) y los expertos en las tres fases se resume en la Figura I.31. El nivel de significación obtenido por el test de Kolmogorov-Smirnov fue de un 0,97, revelando que ambas muestras proceden de la misma distribución.

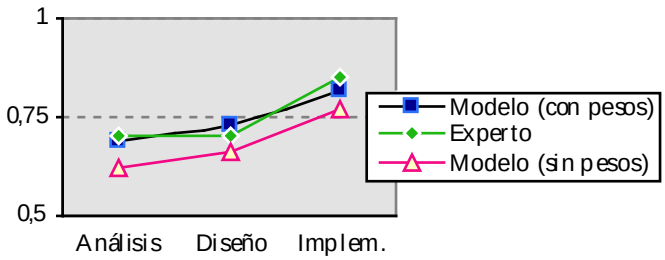


Figura I.31: Valores de calidad obtenidos por el modelo y por los expertos para las fases de análisis, diseño e implementación, respectivamente

Si se hubieran realizado los tests de Kolmogorov-Smirnov sobre los datos del modelo sin intervenir los pesos en el cálculo, los niveles de significación habrían sido: 0,21, 0,21 y 0,46 para los factores en las fases de análisis, diseño e implementación, respectivamente, y 0,52 para la calidad total. Estos datos, aunque también reflejan que las muestras proceden de la misma distribución, ponen de manifiesto que esta afirmación se puede realizar mucho más categóricamente en el caso de emplear los pesos en los cálculos.

I.5. CREADOR DE CURSOS

Descripción del sistema: En este experimento se va a evaluar un módulo perteneciente a PHASE (Proyecto de Herramienta de Autor para *Software* Educativo) un sistema inteligente de tutoría basado en un modelo general de cursos *software*, compuesto de varios módulos. El módulo a analizar será el Módulo Creador de Cursos, que se encarga de recoger toda la información necesaria para el curso y crear las estructuras precisas para que el resto de los módulos utilicen esta información para la ejecución del curso. En concreto, el presente módulo se comunicará con el Módulo Interfaz de Usuario y con el núcleo del sistema [Fuente, 99].

I.5.1. EXPERIMENTO 12

Descripción del experimento: Se ha aplicado el modelo de calidad, en su fase de calidad de la implementación, sobre el código fuente de un módulo, con el fin de evaluar el desarrollo de esa parte de un sistema completo. Es de destacar que el módulo a evaluar se encuentra implementado en el lenguaje Object Pascal del entorno Delphi de Borland, por lo que el experimento también mostrará la aplicabilidad del modelo de calidad sobre un lenguaje diferente al C++.

Objetivos: Evaluar un sistema (en concreto, un módulo del sistema) construido en Object Pascal, por lo que el experimento servirá para probar la utilidad del modelo de calidad en otros lenguajes de programación orientados a objetos.

Resultados: Como valor de calidad global se ha alcanzado un 0,65, lo que refleja una buena implementación y un adecuado conocimiento de la tecnología de la orientación a objetos por parte del programador. La Figura I.32 muestra la gráfica que resume el vector de valores obtenido para cada uno de los factores. Puede observarse que la mayoría de los factores se encuentran entre 0,5 y 0,75, siendo de destacar los altos valores de la corrección y reutilizabilidad. En lado negativo se encuentran el factor utilizabilidad con un valor ligeramente inferior a 0,5, debido fundamentalmente a que se está midiendo un módulo de un sistema que no ofrece facilidades de utilización directamente al usuario del programa.

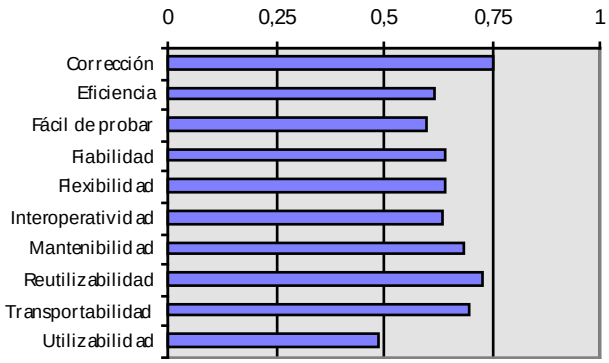


Figura I.32: Factores de calidad para el módulo creador de cursos

Con relación a las medidas utilizadas, cerca de un 32% no se han usado por diversas razones. En primer lugar, más del 10% eran específicas del lenguaje C++ y medían

conceptos no presentes en Object Pascal (como la sobrecarga de métodos, funciones amigas o clases virtuales). En segundo lugar, cerca de un 21% no pudieron ser evaluadas porque no se tenían datos precisos sobre la utilización del módulo en el seno del sistema completo o bien no resultaban aplicables debido a la naturaleza del sistema.

I.5.2. EXPERIMENTO 18

Descripción del experimento: Se ha aplicado el modelo de calidad, en su fase de calidad de la implementación, sobre el código fuente de un módulo (desarrollado en Object Pascal) construido teniendo como principal objetivo la flexibilidad, con el fin de evaluar el desarrollo de esa parte de un sistema completo. Seguidamente, se presentó al programador los resultados obtenidos para el factor de flexibilidad.

Objetivos: Evaluar un sistema real (en concreto, un módulo del sistema) construido persiguiendo la flexibilidad, para comprobar la aplicabilidad del modelo y verificar el valor de flexibilidad obtenido. Un último objetivo consiste en estudiar los efectos del modelo de calidad sobre el programador.

Resultados: Como valor de calidad se ha alcanzado un 0,65, lo que refleja una buena implementación y un adecuado conocimiento de la orientación a objetos por el programador.

Uno de los objetivos perseguidos en la construcción del presente módulo era su flexibilidad, lo cual ha sido alcanzado en un grado elevado al obtener un 0,64. La Figura I.33 refleja los buenos valores logrados finalmente por los criterios de la flexibilidad, únicamente deslucidos por el 0,36 de la documentación, criterio que pone una vez más de manifiesto lo que cuesta a los programadores generar una buena y completa documentación de los programas que llevan a cabo. Vistos estos datos, la recomendación para obtener un módulo más flexible pasaría por mejorar substancialmente la documentación (hasta al menos doblar su valor) y realizar las modificaciones pertinentes para mejorar la estabilidad y la generalidad (aumentando al menos un 15% sus valores). Con estos sencillos remedios se podría superar el valor de 0,7 para la flexibilidad.

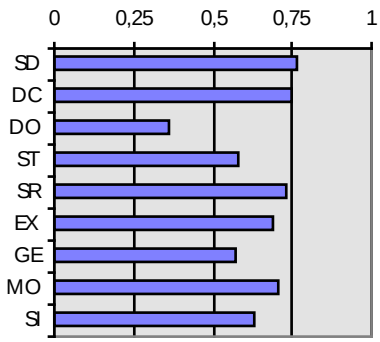


Figura I.33: Flexibilidad del módulo creador de cursos

En este experimento, fue el propio programador el que evaluó las medidas. Como principal conclusión obtenida del análisis de su módulo, comentó que la evaluación de las medidas le sirvió como una revisión profunda del código que le permitió corregir ciertos errores provocados por distracciones, puesto que al aplicar las medidas se

podían detectar algunos de estos errores durante el propio proceso de evaluación, con lo que dicho proceso permite obtener un *software* de mayor calidad sin esperar a analizar los resultados [Fuente, 99].

I.6. CURSO DE IDIOMAS

Descripción del sistema: El sistema en construcción consiste en un curso de idiomas para invidentes, que presenta al usuario una sucesión de ejercicios que deben ser resueltos. Conforme se van realizando ejercicios, crece la dificultad y el nivel del curso. El sistema almacena el progreso del alumno e incorpora una serie de ayudas adicionales, como puede ser un resumen de la gramática de cada ejercicio y la conexión con un diccionario bilingüe externo. El sistema, internamente, está dividido en dos grandes módulos: el módulo de control y el módulo de los ejercicios y el desarrollo ha seguido un proceso iterativo incremental. [Berrocal, 03]

I.6.1. EXPERIMENTO 20

Descripción del experimento: Se ha seguido el proceso de desarrollo del sistema durante todo el ciclo de vida, desde los requisitos hasta la implementación de un prototipo sencillo, pasando por varios análisis y diseños.

Objetivos: El objetivo principal era comprobar el funcionamiento del método de aplicación del modelo de calidad durante el ciclo de vida de un sistema real, cuyo desarrollo está ocupando cerca de dos años y comprobar la mejora del proceso *software*.

Resultados: En primer lugar, y tras tener una primera versión de los requisitos del sistema, se aplicó el modelo de calidad obteniéndose los resultados de los factores mostrados en la Figura I.34 (correspondientes a 31 medidas) y que totalizan un valor de calidad de 0,39, que, como puede observarse, es considerablemente bajo, puesto que la mayoría de los factores están bastante por debajo del 0,5.

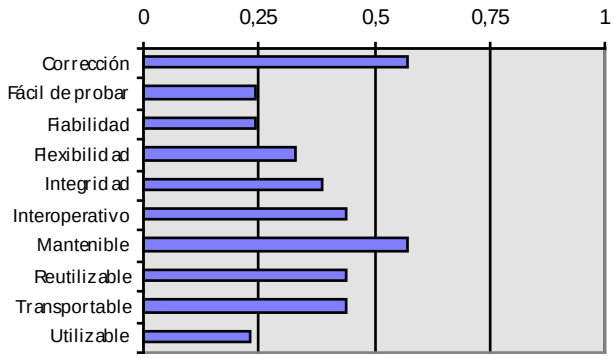


Figura I.34: Factores en la fase de requisitos

Estos resultados hicieron recomendar al equipo de ingenieros del *software* la revisión de los requisitos, lo cual, una vez realizado, dio lugar a los resultados del modelo de calidad mostrados en la Figura I.35. En esta ocasión, la calidad alcanzó un valor de 0,42. Aunque este valor no era suficiente, los encargados del proyecto decidieron dar paso a la fase de análisis, teniendo en cuenta las recomendaciones proporcionadas por el modelo.

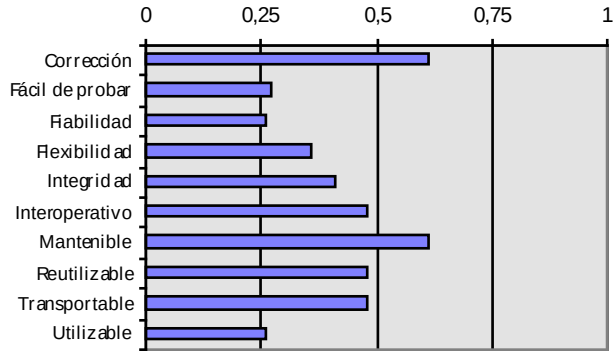


Figura I.35: Factores para los requisitos revisados

El primer análisis obtenido, restringido sólo al módulo de control, obtuvo un valor de calidad igual a 0,32. Como puede observarse en la Figura I.36, la mayoría de los criterios se han visto bastante afectados debido, fundamentalmente, a la intervención de un mayor número de medidas (75) en los distintos criterios, lo que supone evaluar más atributos de la calidad. En términos generales, este primer análisis se entendió como una primera aproximación rápida a la solución del problema, por lo que su calidad no es buena.

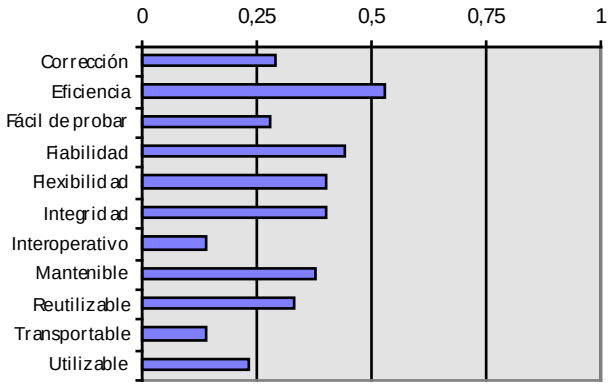


Figura I.36: Factores en el primer análisis

En el segundo análisis, además de revisar y ampliar el análisis anterior, incluyendo también herencia, se incorporó el análisis del módulo de los ejercicios. El resultado obtenido fue de una calidad de 0,35 (Figura I.37), todavía bastante lejos de lo deseable, por lo que la sugerencia adoptada por los ingenieros del *software* fue la de revisar el análisis.

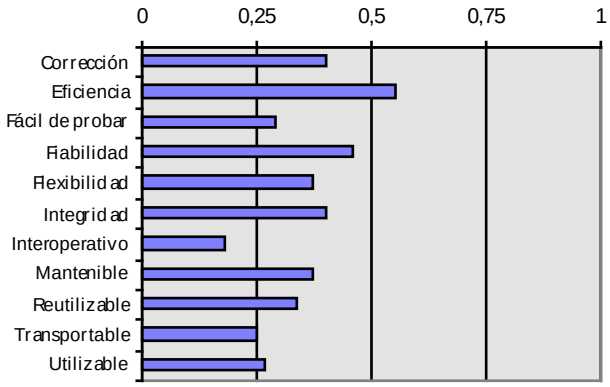


Figura I.37: Factores en el segundo análisis

En el tercer análisis se realizó un profundo cambio principalmente en el módulo de ejercicios, reduciéndose significativamente la cantidad de clases existentes y el número de relaciones de herencia, puesto que éstas resultaban excesivas. Una vez aplicado el modelo de calidad, ofreció un valor global de 0,39, sensiblemente superior al anterior, habiéndose mejorado ligeramente la mayoría de los factores según demuestra la Figura I.38.

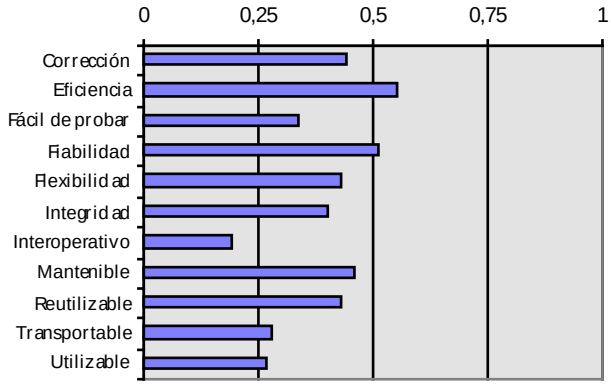


Figura I.38: Factores en el tercer análisis

Como la mejora no se consideraba suficiente, se refinó el análisis, introduciéndose nuevas relaciones de herencia, clases abstractas y enlace dinámico. El resultado se muestra en la Figura I.39 y el valor de calidad fue 0,43, que, aunque mejor, todavía no resultaba satisfactorio.

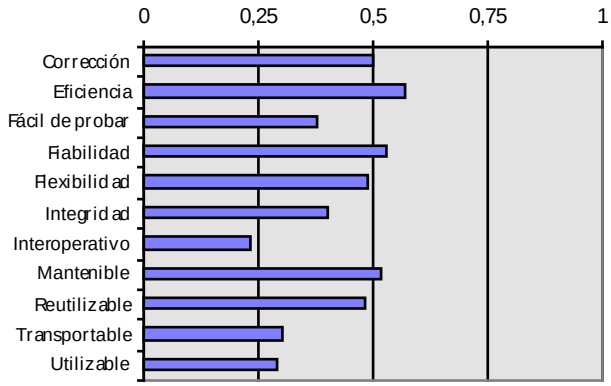


Figura I.39: Factores en el cuarto análisis

Con todos los datos obtenidos y viendo la dificultad que había alcanzado el sistema, se decidió modificar la filosofía de construcción, adoptando la técnica de modelo-vista-control [Krasner, 88], que facilita aislar los distintos componentes relacionados con una clase. La Figura I.40 refleja los valores de los factores, siendo la calidad total ahora de un 0,52, que, como puede verse, mejora significativamente al valor del anterior análisis, lo cual puede proporcionar la idea de que la nueva técnica adoptada se adecua mejor al problema.

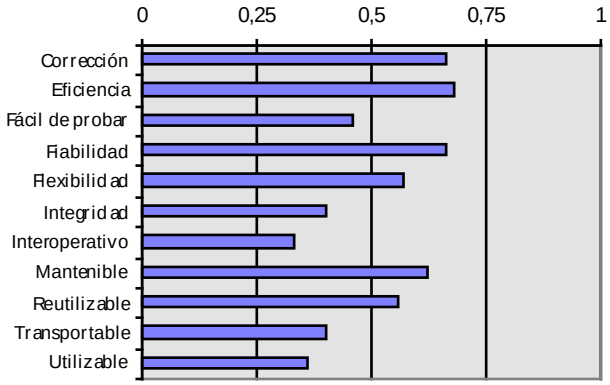


Figura I.40: Factores en el quinto análisis

Con estos resultados, se decidió afrontar ya el diseño. En la Figura I.41 se representan los valores de los factores para el primer diseño, que obtuvo un 0,54 a partir de 367 medidas de diseño. Estos resultados, que no son malos (tan solo dos factores no llegan al 0,5) son mejorables y los ingenieros así lo entendieron tras analizar las recomendaciones.

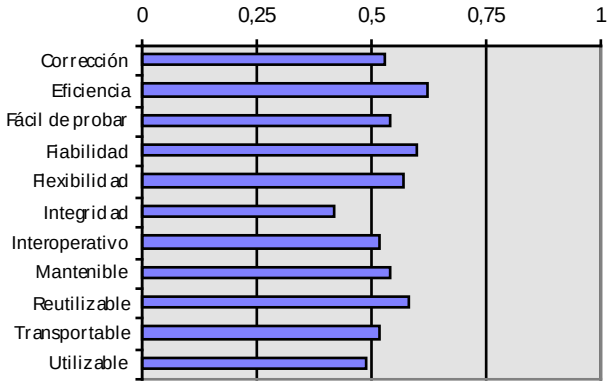


Figura I.41: Factores en el primer diseño

Por ello, llevaron a cabo un rediseño, lo cual supuso mejorar la calidad hasta un 0,60, como se muestra en la Figura I.42. Éste es un valor de calidad bueno, pero los ingenieros del *software* decidieron revisar el último análisis por motivos ajenos al modelo de calidad, aunque las recomendaciones ofrecidas se tuvieron en cuenta para el subsiguiente diseño.

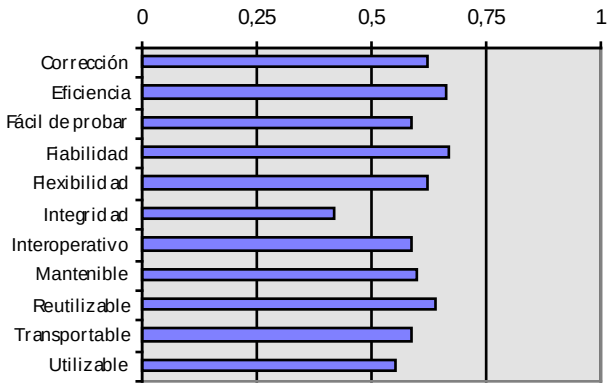


Figura I.42: Factores en el segundo diseño

El nuevo análisis incorporaba mayores relaciones de herencia y un menor número de clases. La calidad obtenida fue de un 0,61 y la Figura I.43 refleja sus factores.

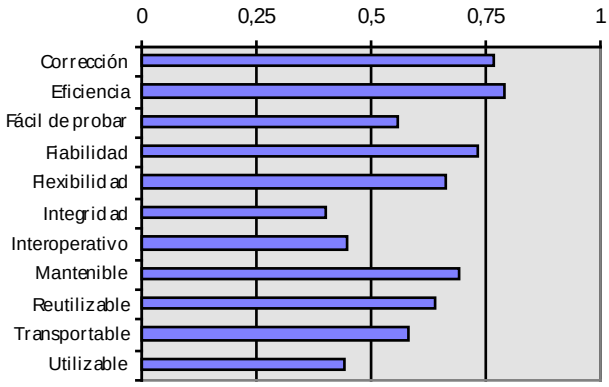


Figura I.43: Factores en el sexto análisis

Retomando el diseño, se alcanzó una última versión, cuya calidad era de un 0,63 (Figura I.44). Este diseño y su calidad se consideraron adecuados, por lo que se decidió afrontar la implementación, teniendo, evidentemente, en cuenta las recomendaciones proporcionadas por el modelo de calidad.

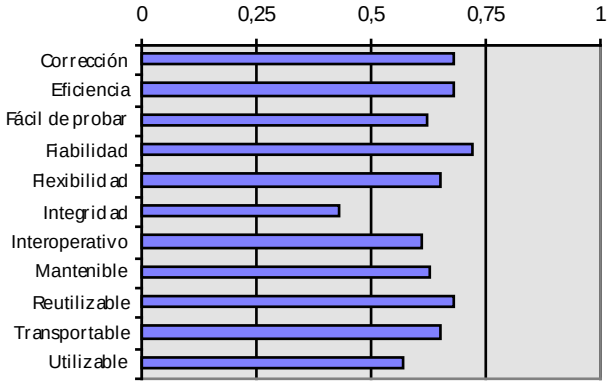


Figura I.44: Factores en el tercer diseño

La implementación del primer prototipo proporcionó una calidad de 0,57. La Figura I.45 muestra los valores de los factores obtenidos a partir de las medidas de implementación que se aplicaron sobre el código fuente desarrollado en C++. Estos resultados son relativamente buenos (se podían esperar mejores visto el diseño último, pero al no ser la implementación del sistema completo, adolece de ciertas deficiencias). En cualquier caso, las medidas proporcionaron consejos a seguir que debían incluirse en la implementación del sistema final.

Lamentablemente, a la hora de describir el presente experimento, el segundo prototipo estaba empezando a construirse, por lo que aquí se da por terminada la exposición del experimento.

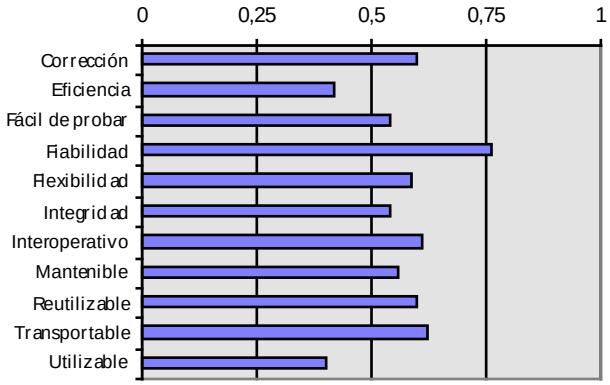


Figura I.45: Factores en el prototipo

Como resumen de los distintos valores globales de calidad obtenidos, se muestra la Figura I.46, donde se observa la evolución seguida durante las distintas fases por las que ha pasado el ciclo de vida del sistema.

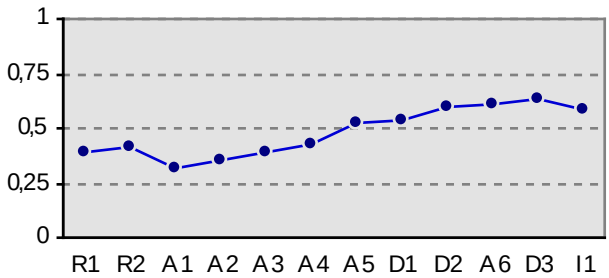


Figura I.46: Evolución de la calidad durante el ciclo de vida

I.7. DESARROLLO GRÁFICO

Descripción del sistema: El sistema del experimento consiste en una aplicación de desarrollo gráfico de sistemas orientados a objetos en Java. El usuario podrá introducir gráficamente las clases, los distintos tipos de relaciones entre clases y definir sus atributos y métodos. La aplicación transformará esta entrada en código Java que el usuario podrá modificar y completar con el código de los métodos, reflejándose los cambios en la interfaz gráfica. El desarrollo del sistema consta de varias fases: la primera fase corresponde a la interfaz gráfica, la segunda fase se corresponde con la traducción de la información gráfica a código Java y viceversa, la tercera fase incluye el desarrollo de un compilador de Java y la última fase incluirá la evaluación de la calidad, que permitirá al usuario ver la calidad tanto del diseño como de la implementación de acuerdo al modelo de calidad presentado en este trabajo, adaptado para su utilización con Java. El sistema actualmente se encuentra en pleno desarrollo, habiéndose finalizado tanto el diseño como la implementación en Java de la primera fase [Rovira, 03].

I.7.1. EXPERIMENTOS 4, 13, 21 Y 24

Descripción del experimento: El experimento ha consistido en aplicar el modelo de calidad al análisis de la primera fase del sistema, proporcionando la información obtenida al desarrollador. Posteriormente, se vuelve a aplicar tanto sobre el diseño

como sobre la implementación. Finalmente, se compararán los valores de los criterios con la opinión de dos expertos.

Objetivos: En este caso se aúnan múltiples objetivos. En primer lugar, estudiar la bondad del modelo en relación con los criterios, comparándolos con la opinión de expertos. En segundo lugar, comprobar la adaptación del modelo de calidad y del método de mejora del proceso *software* a un sistema en desarrollo de un tamaño considerable (la implementación de la primera fase contiene 110 clases propias más las reutilizadas de librerías públicas y un total de 216 ficheros fuente Java). En tercer lugar, analizar con detenimiento los resultados obtenidos por el modelo. Y, en último lugar, estudiar la adecuación, tanto del modelo como de las medidas, al lenguaje de programación Java.

Resultados: Los datos obtenidos tras aplicar el modelo de calidad sobre el análisis se presentan en la Figura I.47. El valor de calidad alcanzado es de un 0,48, lo cual indica que no es un análisis malo, pues como puede observarse, la mayoría de los factores sobrepasan la mitad de la escala, aunque sería fácilmente mejorable. Y los aspectos a mejorar pueden extraerse fundamentalmente del vector de criterios: datos estándar, documentación, instrumentación, operatividad y tolerancia a errores. Con estos datos, el desarrollador decidió mejorar estas características durante el diseño.

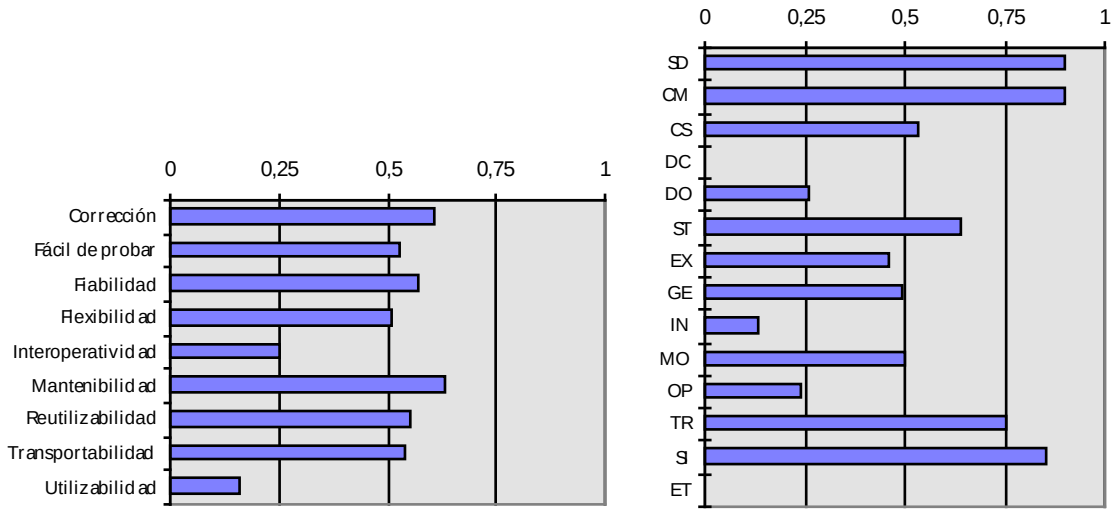


Figura I.47: Factores y criterios de calidad para el análisis

Los principales resultados para el diseño de la fase primera se presentan en la Figura I.48 donde se muestra el vector de valores obtenido para los factores y criterios, respectivamente. El valor de calidad global alcanzado aumentó hasta un 0,54. Un aspecto importante a tener en cuenta a la hora de analizar estos resultados estriba en que debido a la metodología empleada para realizar el diseño, no se ha incluido el diseño de los métodos en pseudocódigo, como ocurría en los experimentos previos, por lo que muchas de las medidas de diseño no han podido ser aplicadas.

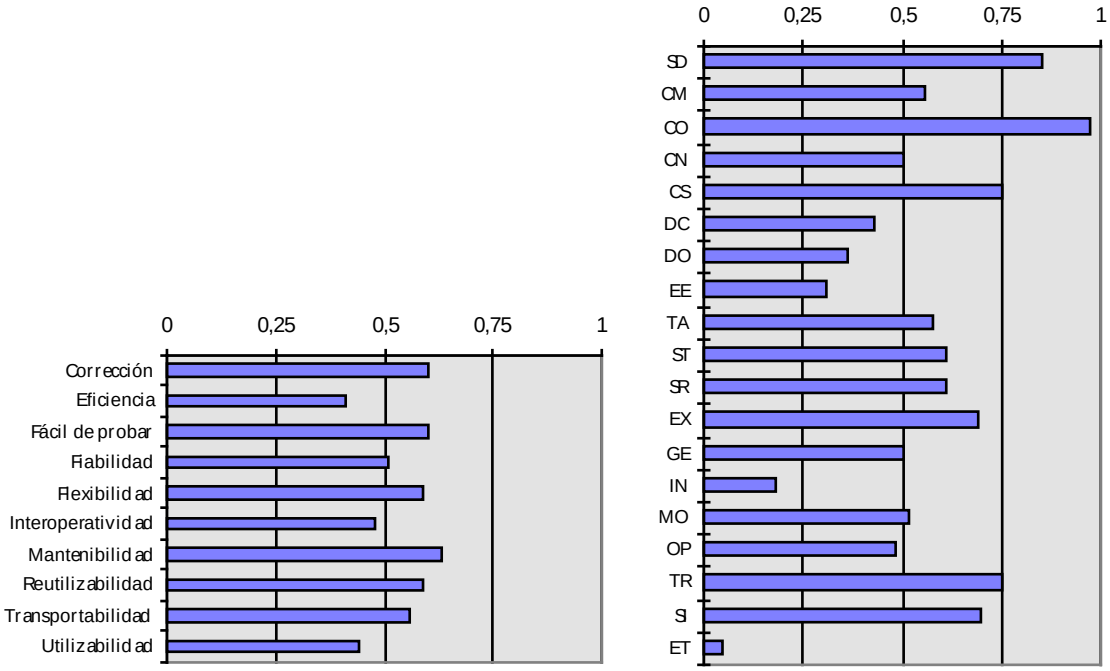


Figura I.48: Factores y criterios de calidad para el diseño

Puede comprobarse que los factores que tenían unos valores más bajos en el análisis han mejorado ostensiblemente, si bien otros han visto disminuido ligeramente su valor, aunque en términos globales ha habido una mejoría generalizada. Viendo los criterios también se comprueba una significativa mejoría, aunque algunos de ellos han empeorado, debido fundamentalmente a las nuevas medidas que se han podido aplicar en el diseño.

Tras notificar al desarrollador estos resultados y hacerle una serie de sugerencias para la mejora de ciertos aspectos (tolerancia a errores, instrumentación, eficiencia de ejecución...), decidió proseguir con la fase de implementación sin modificar el diseño, salvo para corregir los errores que algunas medidas habían puesto de manifiesto (sólo a modo de ejemplo, las medidas CM_{17} y CM_{30} mostraron clases que no tenían atributos o métodos definidos, la medida CM_{18} mostró alguna referencia a elementos no definidos, la CM_{23} advertía de la ausencia de constructores y destructores, etc.).

Tras haber finalizado completamente la implementación de esta primera fase y una vez aplicado el modelo de calidad, se obtuvo un valor global de calidad de 0,71, lo cual indica una mejoría importante con respecto a la calidad obtenida para el diseño. La Figura I.49 permite comprobar los valores obtenidos para los distintos factores y criterios.

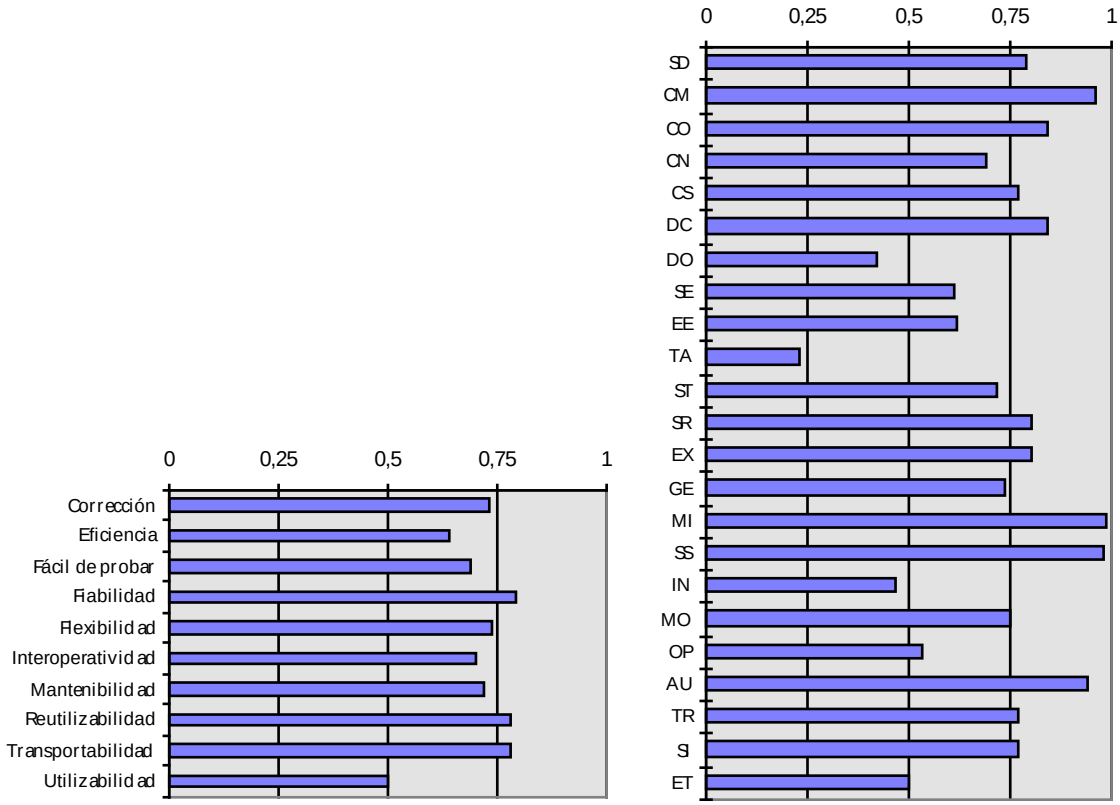


Figura I.49: Factores y criterios de calidad para la implementación

Puede observarse cómo el factor transportabilidad ha alcanzado un valor significativamente elevado (0,78), poniendo de manifiesto que el sistema está siendo implementado en Java, intentando que sea independiente de la plataforma. La Figura I.50 resume sus criterios, donde se observa la buena generalidad y modularidad junto a la excelente independencia de la plataforma.

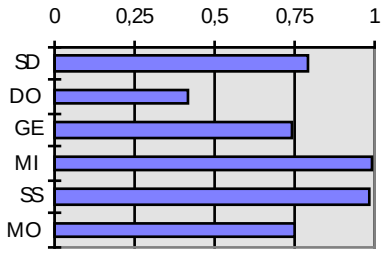


Figura I.50: Transportabilidad

También resulta interesantes observar cómo el factor de flexibilidad está obteniendo buenos valores (0,74), lo cual resulta necesario pues el sistema tiene que ser ampliado con nuevas funcionalidades en las siguientes fases. Particularmente, resultan fascinantes los resultados superiores a 0,7 alcanzados por ocho de los nueve criterios (Figura I.51) y, en particular, por la estabilidad y la expansibilidad relacionados más directamente con las modificaciones al código.

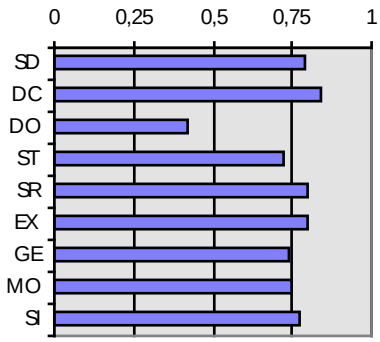


Figura I.51: Flexibilidad

En el lado negativo de los valores de los factores se encuentra la utilizabilidad (0,5), en la que cuatro de los seis criterios no han superado la mitad de la escala (Figura I.52). La explicación de este comportamiento estriba en el hecho de que el sistema estudiado constituye la primera fase de un sistema mayor. Esto ha llevado al desarrollador a no incluir todavía documentación de usuario ni ningún tipo de ayuda para el entrenamiento del usuario en su utilización. No obstante, el programador es consciente de su necesidad y que, para finalizar con éxito todas las fases del sistema, resultará imprescindible mejorar los cinco criterios de este factor que se alejan de los valores normales del resto de los criterios.

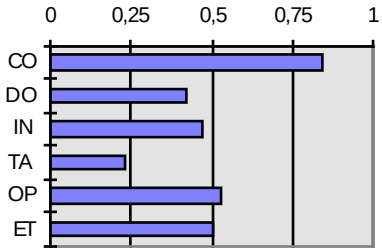


Figura I.52: Utilizabilidad

Por último, es de resaltar que en la fase de implementación de este experimento, ha habido aproximadamente un 25% de las medidas que no se han podido utilizar, debido a que medían características específicas del lenguaje C++ no presentes en Java o a que no eran de aplicabilidad en este problema concreto.

En lo que respecta a la opinión que los dos expertos emitieron sobre los distintos criterios en las tres fases del ciclo de vida, la Figura I.53 los resume en comparación con los valores proporcionados por el modelo de calidad.

Con el fin de comparar las dos poblaciones formadas con los valores de los criterios obtenidos por el modelo y por los expertos se realizó el test no paramétrico de Kolmogorov-Smirnov, dado que se desconoce la distribución que siguen los datos. Para poder comparar ambas poblaciones, se construyó la figura del evaluador tipo aplicando la media aritmética a los datos de los expertos. Entonces, entre el evaluador tipo y el modelo, se obtuvieron unos niveles de significación aproximada de 0,90, 0,53 y 0,65, para las fases de análisis, diseño e implementación, respectivamente. Estos datos

permiten rechazar la hipótesis de que ambas poblaciones, en los tres casos, no proceden de la misma distribución.

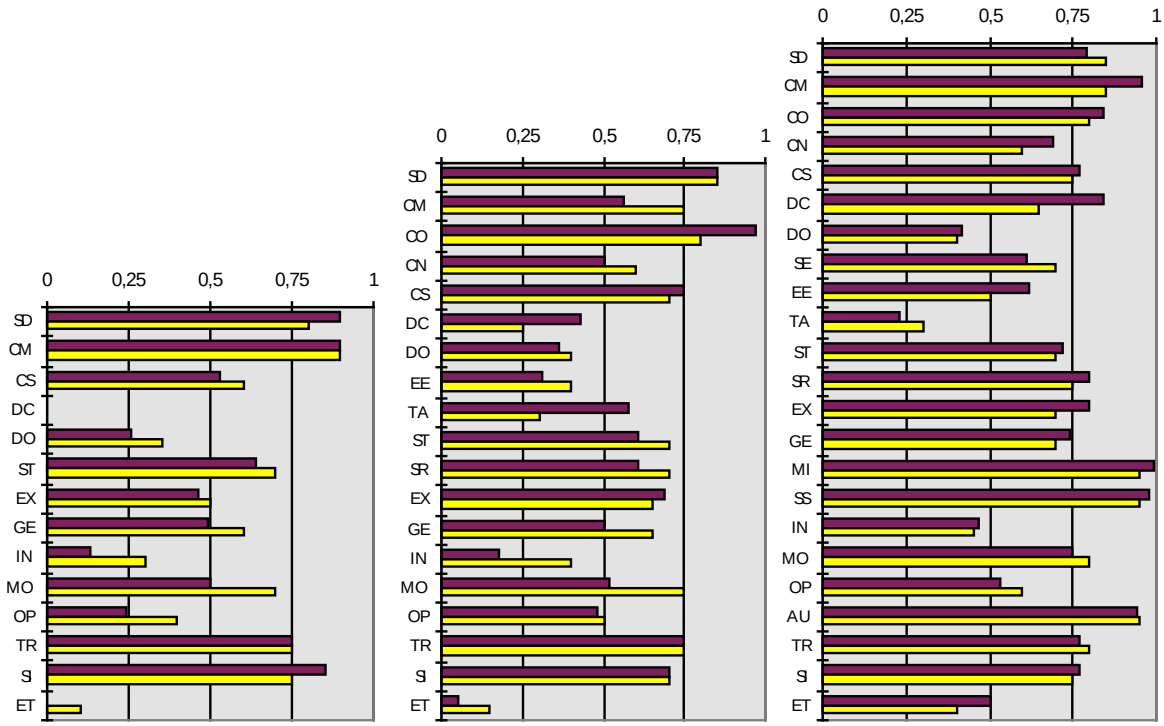


Figura I.53: Valores de los criterios obtenidos por el modelo (en oscuro) y por los expertos (en claro) para las fases de análisis, diseño e implementación, respectivamente

I.8. JUEGO DE DADOS

Descripción del sistema: El sistema es un sencillo juego de dados denominado *Greed* (avaricia), en el que pueden participar varios jugadores, pudiendo estar representados por humanos o por la máquina. Dependiendo de los valores de los dados obtenidos en una tirada, se asignan una serie de puntos. Entonces el jugador puede arriesgarse y volver a tirar los dados que no han puntuado, de tal forma que si no se obtienen nuevos puntos, se pierden los conseguidos en la jugada. El enunciado completo puede encontrarse en [HP, 92], y el programa ha sido desarrollado por un alumno de un curso de orientación a objetos y C++ donde se utilizó esta documentación, cuando todavía no se le había enseñado la herencia, constituyendo su primer programa en este lenguaje.

I.8.1. EXPERIMENTO 14

Descripción del experimento: Se va a aplicar el modelo de calidad, en su fase de calidad de la implementación, sobre el código fuente de un programa que implementa el juego de dados *Greed*. Así mismo, se aplicará el método de Análisis Jerárquico para obtener los pesos de los factores con relación a la calidad. Seguidamente, se anularon aquellos criterios que no se plantearon como objetivos en el enunciado del problema y se estudiaron los nuevos resultados alcanzados por el árbol de calidad.

Objetivos: El primer objetivo consiste en estudiar el comportamiento del modelo de calidad cuando se eliminan los criterios que no se tuvieron en mente a la hora de

construir el sistema. El segundo objetivo estriba en el estudio del comportamiento de los pesos de los factores a la hora de calcular la calidad total del sistema.

Resultados: Aplicando las medidas aplicables (86%, aproximadamente), se ha obtenido un valor de calidad de 0,42, indicando una calidad regular. En la Figura I.54 se muestra una gráfica que resume el vector de valores obtenido para cada uno de los factores.

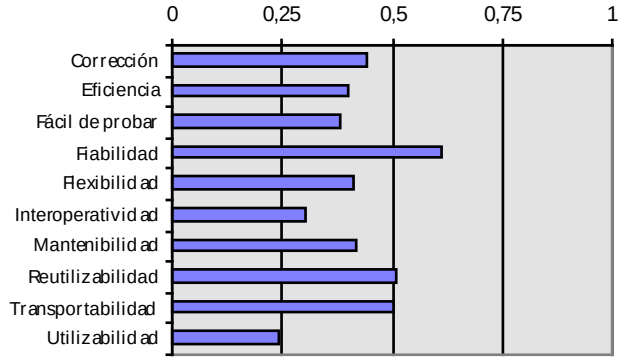


Figura I.54: Factores de calidad para el sistema *Greed*

Para obtener los pesos de los factores mediante el método de Análisis Jerárquico, se ha tenido en cuenta la naturaleza y objetivos del programa. En la Tabla I.7 puede observarse la matriz obtenida, junto con el autovector que define los pesos de cada factor dentro de la calidad. Analizando este autovector, se puede ver que el factor más importante es la corrección (importa que el programa esté correcto) seguido de transportabilidad (interesa que el programa pueda servir para cualquier entorno: se enseña C++ en general, no para una arquitectura concreta), mientras que los menos significativos han sido interoperatividad (no hay necesidades de comunicación con otros sistemas) y utilizabilidad (no es un programa para usarlo, sino para aprender a construirlo). Aplicando estos pesos, se obtiene un valor de 0,44 para la calidad global del sistema, dos centésimas por encima del obtenido con la media aritmética.

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	7	5	5	5	9	5	5	5	7	**0,3319**
Eficiencia	1/7	1	1/3	1	1	7	1/3	1/3	1/3	5	**0,0569**
Fácil de probar	1/5	3	1	3	1	9	3	1	1	7	**0,1194**
Fiabilidad	1/5	1	1/3	1	1	7	1/3	1/3	1/3	3	**0,0537**
Flexibilidad	1/5	1	1	1	1	7	1	1	1/3	3	**0,0691**
Interoperatividad	1/9	1/7	1/9	1/7	1/7	1	1/7	1/7	1/9	1/5	**0,0129**
Mantenibilidad	1/5	3	1/3	3	1	7	1	1/3	1/5	3	**0,0745**
Reutilizabilidad	1/5	3	1	3	1	7	3	1	1/3	5	**0,1037**
Transportabilidad	1/5	3	1	3	3	9	5	3	1	3	**0,1491**
Utilizabilidad	1/7	1/5	1/7	1/3	1/3	5	1/3	1/5	1/3	1	**0,0288**

Tabla I.7: Matriz de relevancia de los factores dentro de la calidad y su autovector para el sistema *Greed*

Si se estudian los principales objetivos que perseguía el desarrollo del sistema, puede considerarse que los siguientes criterios no se tomaron como prioritarios durante su construcción o, simplemente, no resultaban aplicables:

- Auditoría de accesos y control de acceso: el sistema no necesita realizar ningún tipo de control de acceso a los usuarios, por lo que estos criterios carecen de sentido.
- Comunicaciones estándar: el programa es independiente y no requiere comunicarse con otras aplicaciones.
- Eficiencia de almacenamiento y ejecución: el sistema no requiere almacenar grandes volúmenes de información ni tampoco tiene que realizar un proceso complejo.
- Entrenamiento: al ser un sistema utilizado para el aprendizaje, no tiene como objetivo ser utilizado por usuarios.
- Instrumentación: el sistema no incorpora mecanismos para facilitar su depuración debido a su tamaño.
- Operatividad: el sistema no pretende facilitar su uso al usuario sino que sirva como práctica de programación.
- Seguimiento: al ser un sistema utilizado como práctica de programación, no se realizó ni análisis ni diseño, por lo que este criterio carece de sentido.

Además, se eliminó también el factor de eficiencia, pues tan solo tenía un criterio poco significativo. Una vez evaluada de nuevo la calidad, se obtuvo un valor de 0,46 (0,44 sin pesos), con lo que puede verse cómo los criterios eliminados por inútiles para el problema, tienen una influencia pequeña en el resultado general de calidad.

I.8.2. EXPERIMENTO 23

Descripción del experimento: Se aplica el modelo de calidad, en la fase de implementación, sobre el código fuente de un programa que implementa el juego de dados *Greed*. Seguidamente se estudian aquellos factores que tienen los valores más bajos y más altos.

Objetivos: El objetivo principal consiste en analizar los resultados proporcionados por el modelo.

Resultados: Aplicando las medidas se ha obtenido un valor de calidad de 0,42, indicando una calidad regular. En la Figura I.54 se resume el vector de valores obtenido para cada uno de los factores. Puede observarse que solamente tres factores alcanzan el valor 0,5, siendo de destacar el valor de la fiabilidad con un valor de 0,61. En la parte baja de la calidad, se encuentran los factores interoperatividad y utilizabilidad con valores por debajo de un tercio.

La Figura I.55 refleja los valores del vector de criterios para la fiabilidad donde destacan los altos valores conseguidos por la completitud, la precisión (los pocos cálculos matemáticos se realizan adecuadamente) y la simplicidad (el problema no era difícil de implementar y se ha realizado sin excesivas complicaciones). El valor bajo obtenido para la tolerancia a errores, debido a la sencillez de la interfaz de usuario y de los errores que se pueden producir, no resulta demasiado significativo.

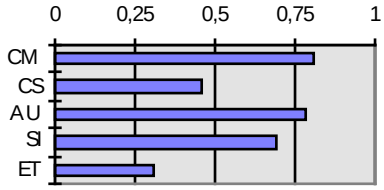


Figura I.55: Fiabilidad del sistema *Greed*

Los valores de los criterios del factor interoperatividad se muestran en la Figura I.56, donde se puede contemplar cómo ninguno de ellos alcanza la mitad de la escala. Es de destacar el valor casi nulo alcanzado por la documentación, debido a la total ausencia de cualquier tipo de documentación acompañando al programa, lo cual tampoco resulta demasiado preocupante puesto que éste era un mero ejercicio de programación, aunque debería intentarse incrementarlo. También, el criterio datos estándar ha obtenido una baja puntuación debido a la escasa utilización de tipos abstractos de datos y clases de forma estándar. Este aspecto sí resulta importante para la programación orientada a objetos en C++, por lo que se tendrá que advertir al programador sobre esta deficiencia, aportándole la información necesaria obtenida a partir de las medidas con peores valores, como DC_8, DC_9 y de DC_{12} a DC_{15}, relacionadas con el uso de clases predefinidas, la sobrecarga, los `template` y, principalmente, acerca de la conveniencia de que las clases dispongan de atributos (un 40% de las clases carecen de atributos).

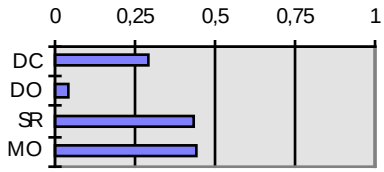


Figura I.56: Interoperatividad del sistema *Greed*

Por último, la Figura I.57 refleja los criterios de utilizabilidad, donde destacan los pésimos valores obtenidos por los criterios documentación, instrumentación y entrenamiento. Ello se debe a que éstos últimos no se encontraban en los objetivos del programa y, por tanto, no han sido tenidos en consideración en el desarrollo. Obviamente, habría que comunicarle estos datos al alumno para que mejore estos aspectos.

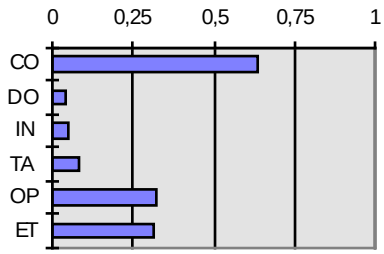


Figura I.57: Utilizabilidad del sistema *Greed*

Analizando el resto de los criterios aisladamente, son de destacar los siguientes cuatro criterios con valores ciertamente bajos. La capacidad del programa de ser auto-descriptivo no ha sido alcanzada suficientemente (0,38) debido a la práctica inexistencia de comentarios, según se deduce de las medidas SD_1 a SD_{13}. No obstante, gracias a la uniformidad en la escritura del código y otros detalles puestos de manifiesto en el resto de las medidas de este criterio, el valor obtenido no resulta excesivamente bajo. Evidentemente, hay que hacer comprender al alumno la importancia de los comentarios, aunque se trate de pequeños programas. A modo de ejemplo, incluyendo los comentarios apropiados, se podría alcanzar un valor del criterio superior a 0,9, por lo que el valor de calidad se elevaría hasta cerca de un 0,5. En relación con el criterio expansibilidad, que ha obtenido un 0,39, las principales conclusiones que pueden obtenerse estriban en la ausencia de herencia y clases y funciones genéricas. La generalidad también resulta ligeramente baja (0,34) por muy diversas razones, entre las que puede destacar por su extrañeza la falta de definición de ningún constructor (GE_{32}), a pesar de su necesidad, lo cual, también se pone de manifiesto en CM_{32}, puesto que tampoco hay destructores. Por último, la concisión ha alcanzado tan solo un 0,23 debido a que no se emplea ninguna de las técnicas de la orientación a objetos que favorecen esta cualidad.

I.9. MÁQUINA DE CAFÉ

Descripción del sistema: El sistema a evaluar consiste en una simulación del *software* de una máquina de café, capaz de proporcionar distintos tipos de café y de cobrar la cantidad adecuada al café servido (en [Boloix, 00] puede verse funcionando una sencilla implementación en Java de un sistema parecido utilizado, también, para estudios acerca de la calidad). El sistema, además, calcula el valor de las monedas introducidas y devuelve el cambio correcto optimizando el número de monedas. La implementación del sistema se ha realizado empleando el lenguaje C++. El objetivo del sistema es utilizarlo en una clase de orientación a objetos para demostrar cómo se puede desarrollar una aplicación empleando este paradigma.

I.9.1. EXPERIMENTOS 5 Y 7

Descripción del experimento: Se ha aplicado el método de Análisis Jerárquico para obtener unas ponderaciones detalladas para cada uno de los criterios dentro de su factor y para los factores dentro de la calidad. A continuación, se solicitó a unos expertos una valoración de los factores y de la calidad para cada fase. Una vez aplicado el modelo de calidad se compararon los datos obtenidos para los factores con los proporcionados por los expertos. Finalmente, se repitió la comparación teniendo en cuenta los pesos de los criterios dentro de cada factor.

Objetivos: El principal objetivo del presente experimento consiste en estudiar el comportamiento tanto de los factores como de la calidad global en las fases del ciclo de vida en comparación con la opinión del ser humano, comprobando, al mismo tiempo, que las recomendaciones proporcionadas originan una mejora del sistema. Así mismo, también se pretende analizar la influencia de los pesos en los criterios y en los factores en el funcionamiento del modelo de calidad.

Resultados: En primer lugar, se aplicó el método de Análisis Jerárquico para los criterios de cada factor, obteniéndose los pesos que se muestran en la Tabla I.8. Seguidamente, se repitió el proceso con el fin de conseguir la ponderación de cada uno de los factores en la calidad global del sistema, logrando el vector de pesos de la Tabla I.9. Este proceso ha sido realizado por dos profesores de cursos en orientación a objetos, mostrando las tablas el promedio de sus respuestas.

Corrección	
CM	0,5951
CS	0,2761
DO	0,0644
TR	0,0644

Eficiencia	
CN	0,1102
SE	0,3460
EE	0,5438

Fácil de probar	
SD	0,1053
CO	0,0754
DO	0,0404
SR	0,2267
IN	0,0667
MO	0,3712
SI	0,1143

Fiabilidad	
CM	0,1876
CS	0,0778
AU	0,3570
SI	0,0967
ET	0,2809

Flexibilidad	
SD	0,0590
DC	0,0436
DO	0,0378
ST	0,1507
SR	0,1551
EX	0,1751
GE	0,0731
MO	0,2123
SI	0,0933

Interoperativo	
DC	0,1915
DO	0,0694
SR	0,2895
MO	0,4496

Mantenible	
SD	0,1955
CN	0,0408
CS	0,0691
DO	0,0546
ST	0,1504
SR	0,1632
MO	0,2135
TR	0,0349
SI	0,0780

Reutilizable	
SD	0,0628
DC	0,1090
DO	0,0339
ST	0,0896
SR	0,1140
GE	0,1562
MI	0,0400
SS	0,0410
MO	0,2761
TR	0,0200
SI	0,0574

Transportable	
SD	0,0950
DO	0,0397
GE	0,0820
MI	0,2850
SS	0,2850
MO	0,2133

Utilizable	
CO	0,3255
DO	0,0684
IN	0,0626
TA	0,0485
OP	0,2921
ET	0,2029

Tabla I.8: Pesos de los criterios dentro de cada factor para la máquina de café

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	9	4	6	3	9	5	4	3	9	**0,2976**
Eficiencia	1/9	1	1/6	1/4	1/5	4	1/5	1/5	1/4	1/3	**0,0262**
Fácil de probar	1/4	6	1	6	2	9	1	2	1/2	7	**0,1328**
Fiabilidad	1/6	4	1/6	1	1/4	4	1/5	1/5	1/6	2	**0,0402**
Flexibilidad	1/3	5	1/2	4	1	6	1	1/2	1/3	5	**0,0885**
Interoperatividad	1/9	1/4	1/9	1/4	1/6	1	1/6	1/2	1/8	1	**0,0194**
Mantenibilidad	1/5	5	1	5	1	6	1	1	1/3	7	**0,1006**
Reutilizabilidad	1/4	5	1/2	5	2	2	1	1	1/2	7	**0,0996**
Transportabilidad	1/3	4	2	6	3	8	3	2	1	7	**0,1708**
Utilizabilidad	1/9	3	1/7	1/2	1/5	1	1/7	1/7	1/7	1	**0,0243**

Tabla I.9: Matriz de relevancia y pesos para los factores para la máquina de café

Los resultados alcanzados por el modelo de calidad tras obtener el análisis definitivo se muestran en la Figura I.58, junto con la opinión subjetiva de los dos expertos.

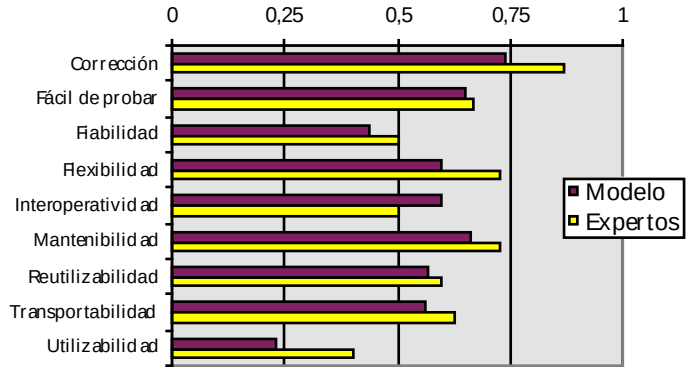


Figura I.58: Valores de los factores obtenidos para el análisis por el modelo de calidad y por los expertos

Tras la conclusión de la siguiente etapa del ciclo de vida, se aplicó el modelo de calidad sobre el diseño, lográndose los valores de los factores de la Figura I.59, junto con la opinión de los expertos.

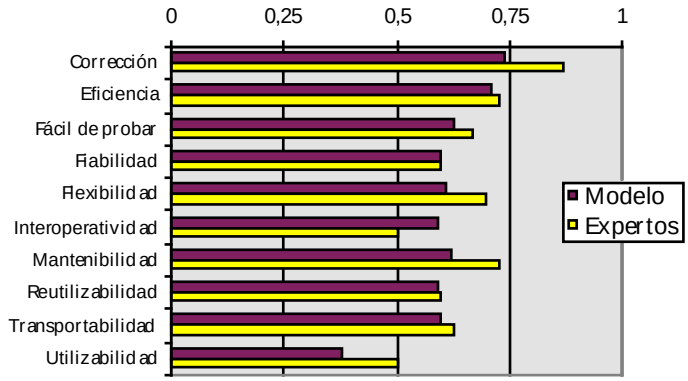


Figura I.59: Valores de los factores obtenidos para el diseño por el modelo de calidad y por los expertos

Una vez finalizada la implementación, se le aplicaron las medidas correspondientes del modelo de calidad, obteniendo los datos mostrados en la Figura I.60 junto a la opinión de los expertos.

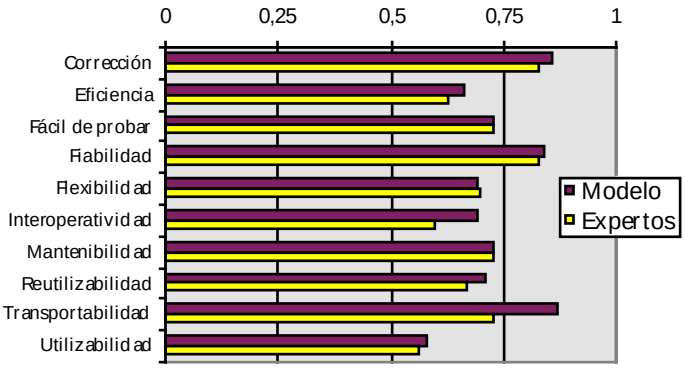


Figura I.60: Valores de los factores obtenidos para la implementación por el modelo de calidad y por los expertos

La Figura I.61 muestra la comparación de los valores de calidad obtenidos en las tres fases del ciclo de vida por el modelo de calidad junto con la opinión de los expertos.

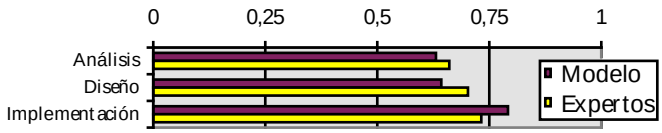


Figura I.61: Valores de la calidad obtenidos por el modelo de calidad y por los expertos

Para comparar los datos del modelo con la opinión de expertos en los factores y en la calidad, se realizó un estudio no paramétrico (Kolmogorov-Smirnov) que determinara si ambas poblaciones provenían de la misma distribución. El nivel de significación obtenido para los factores (en las tres fases) fue, respectivamente, 0,70, 0,40 y 0,76, y el alcanzado para la calidad fue 0,52; estos datos indican que se puede afirmar que en todos los casos, las poblaciones siguen la misma distribución.

En cambio, si se repite este estudio no paramétrico usando los datos obtenidos por el modelo sin emplear los pesos en los cálculos, los resultados del nivel de significación serían 0,12, 0,05 y 0,76 (para los factores) y 0,52 (para la calidad). En los dos últimos se deduce directamente que la distribución que siguen las poblaciones es la misma. Pero, los dos primeros resultados reflejan que, aunque sus distribuciones son las mismas, puede existir cierto componente subjetivo en los datos.

I.9.2. EXPERIMENTO 19

Descripción del experimento: El experimento consiste en evaluar la calidad del sistema durante las distintas fases del desarrollo, siguiendo el método de aplicación del modelo de calidad. Este proceso fue realizado informando al desarrollador de las conclusiones ofrecidas por el modelo. Simultáneamente, otro desarrollador realizó el mismo sistema, pero esta vez sin la realimentación del modelo de calidad.

Objetivos: El principal objetivo del presente experimento trata de ver la diferencia en el desarrollo del sistema tanto obteniendo como sin obtener la realimentación del modelo de calidad.

Resultados: Primero, se utilizó el método de Análisis Jerárquico sobre los criterios de cada factor y sobre los factores de la calidad, para obtener los pesos que se muestran en la Tabla I.8 y en la Tabla I.9. Estos pesos fueron asignados por ambos desarrolladores. En el resto del experimento se mencionarán únicamente los valores de calidad obtenidos empleando estos pesos.

En primer lugar, se resumirá el proceso seguido por el desarrollador que fue informado de la calidad de su trabajo durante las sucesivas fases del ciclo de vida.

Tras obtener una primera versión del análisis (conjuntamente por ambos desarrolladores, con el fin de partir del mismo nivel de calidad), se aplicó el modelo de calidad, en su fase de calidad del análisis, de tal forma que las medidas proporcionaron los valores de los factores y criterios que se muestran en la Figura I.62. El valor de calidad logrado sobre este análisis fue de un 0,44.

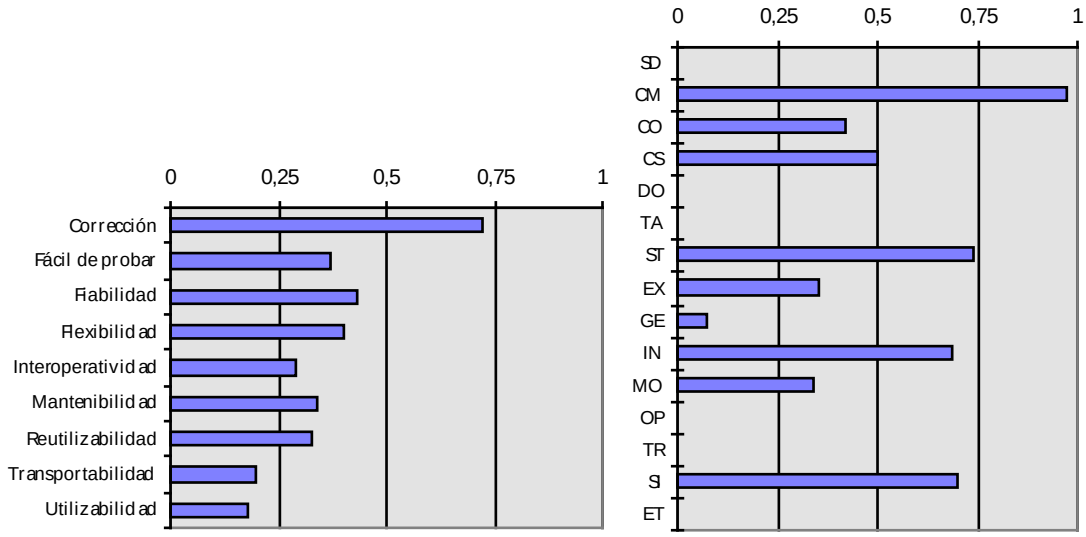


Figura I.62: Valores de los factores y de los criterios obtenidos para el primer análisis

Analizando los valores obtenidos en cada uno de los factores puede observarse que, excepto la corrección, ninguno ha obtenido un buen resultado, estando la mayoría por debajo del valor deseable. Estudiando más detenidamente los datos de los criterios, puede verse que la mayoría resultan considerablemente bajos, habiendo, incluso, obtenido seis de ellos el valor cero. Por ello, se estudiaron aquellas medidas con peores valores, consiguiendo elaborar una serie de consejos para el analista, que pueden resumirse, a grandes rasgos, en comentar los diagramas del análisis, redactar una documentación detallada, tener en cuenta más al usuario y su actividad ante el sistema y darle más responsabilidades a cada clase, revisando los diagramas de clases.

Como resultado de los consejos proporcionados, el desarrollador decidió regresar a la fase de análisis para realizar una profunda revisión. Cuando estuvo finalizada una segunda versión del análisis del sistema, se aplicó de nuevo el modelo de calidad sobre el análisis, proporcionando los valores de los factores y criterios que se muestran en la Figura I.63 (a efectos de comparación, se muestran también los resultados del primer análisis). La calidad total obtenida en esta ocasión fue de un 0,63, notablemente superior a la anterior.

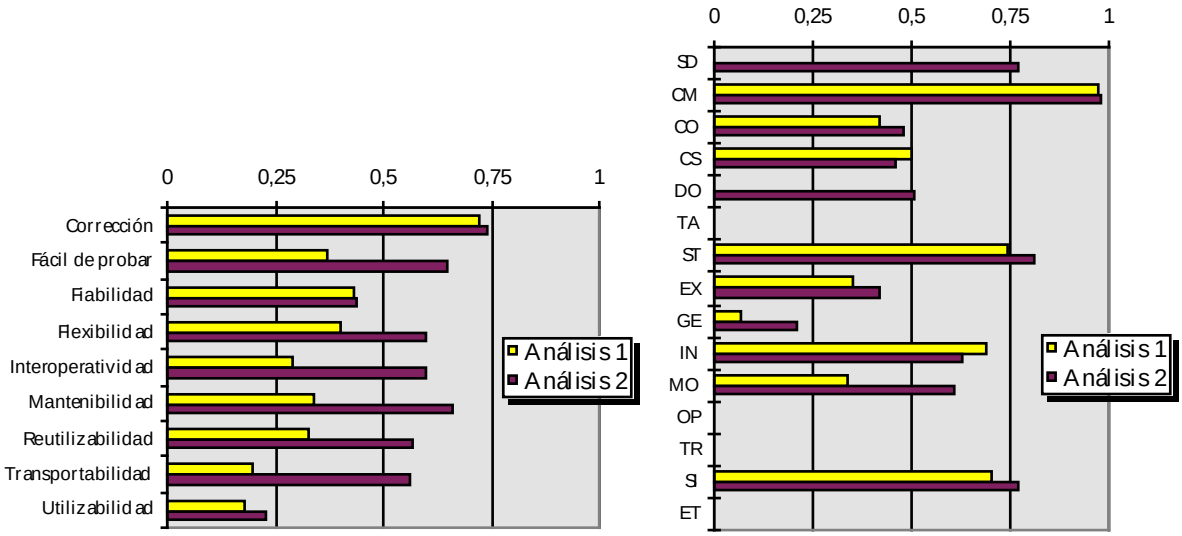


Figura I.63: Valores de los factores y de los criterios obtenidos para el análisis

Estudiando los nuevos resultados, puede observarse que los factores han mejorado ostensiblemente, quedando solamente dos de ellos por debajo de la mitad de la escala. En lo referente a los criterios, también se puede comprobar cómo la mejora ha sido sustancial en auto-descriptivo o documentación, aunque todavía quedan cuatro de ellos con un valor nulo. También se aprecia cómo dos de los criterios han sufrido un sucinto receso debido al compromiso que existe entre unos y otros (las causas, fundamentalmente, de este retroceso ha sido la sustitución de una jerarquía por una única clase). Tras informar al desarrollador de los resultados y, tras proporcionarle los nuevos consejos obtenidos de las medidas, éste decidió no retroceder y comenzar la siguiente fase (el diseño), aunque teniendo presente las advertencias obtenidas.

Tras la conclusión de la siguiente etapa del ciclo de vida, se aplicó el modelo de calidad sobre el diseño, lográndose los valores de los factores y criterios de la Figura I.64. El valor de calidad obtenido fue 0,59.

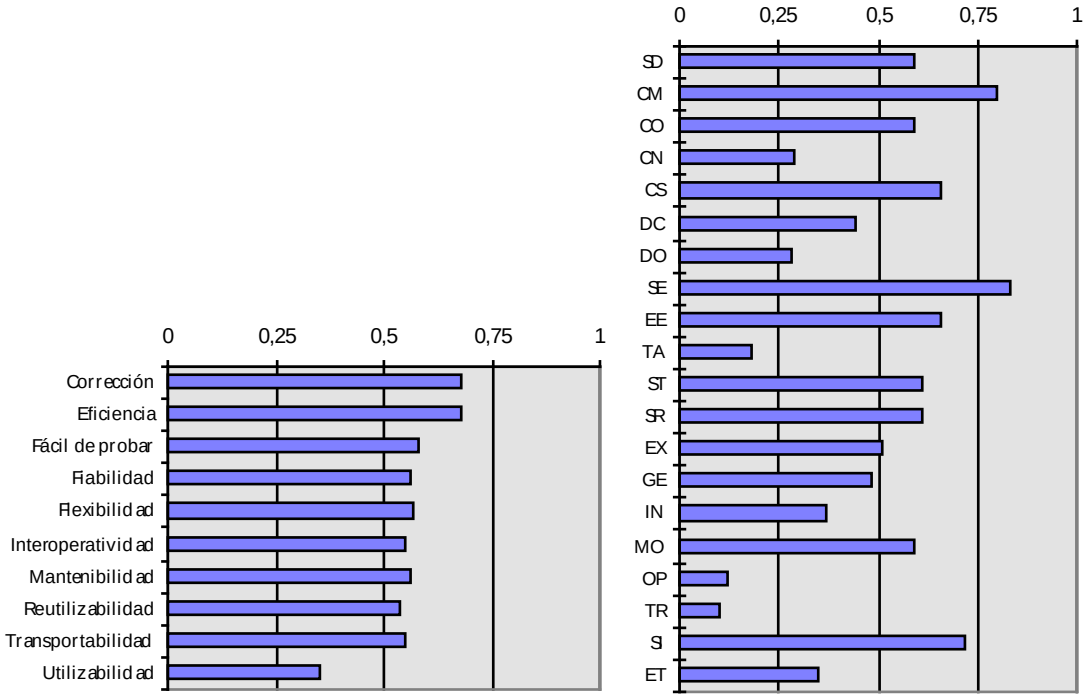


Figura I.64: Valores de los factores y de los criterios obtenidos para el primer diseño

Estudiando los factores, se observa que tan solo el factor utilizabilidad no ha obtenido un valor mínimo adecuado. Si se miran los criterios, se puede comprobar que los consejos del último análisis han surtido efecto puesto que ya no hay ninguno con valor cero. Los consejos extraídos de las medidas con valores bajos se centran, en términos generales, en prestar más atención al manejo que hará el usuario de la aplicación, mejorar la auto-descripción y la documentación (que no han mantenido la calidad del análisis) así como el seguimiento que se mantiene demasiado bajo, y, más concretamente, incluir constructores y destructores a todas las clases y estudiar una clase aislada que parece ser que no se utiliza, así como uno de los métodos de otra clase que tampoco es llamado nunca.

Con toda la información, el desarrollador decidió revisar el diseño obteniendo una nueva versión, sobre la que se volvió a aplicar el modelo de calidad, cuyos resultados se

muestran en la Figura I.65 (junto a los resultados anteriores). La calidad alcanzada en esta ocasión fue de un 0,64.

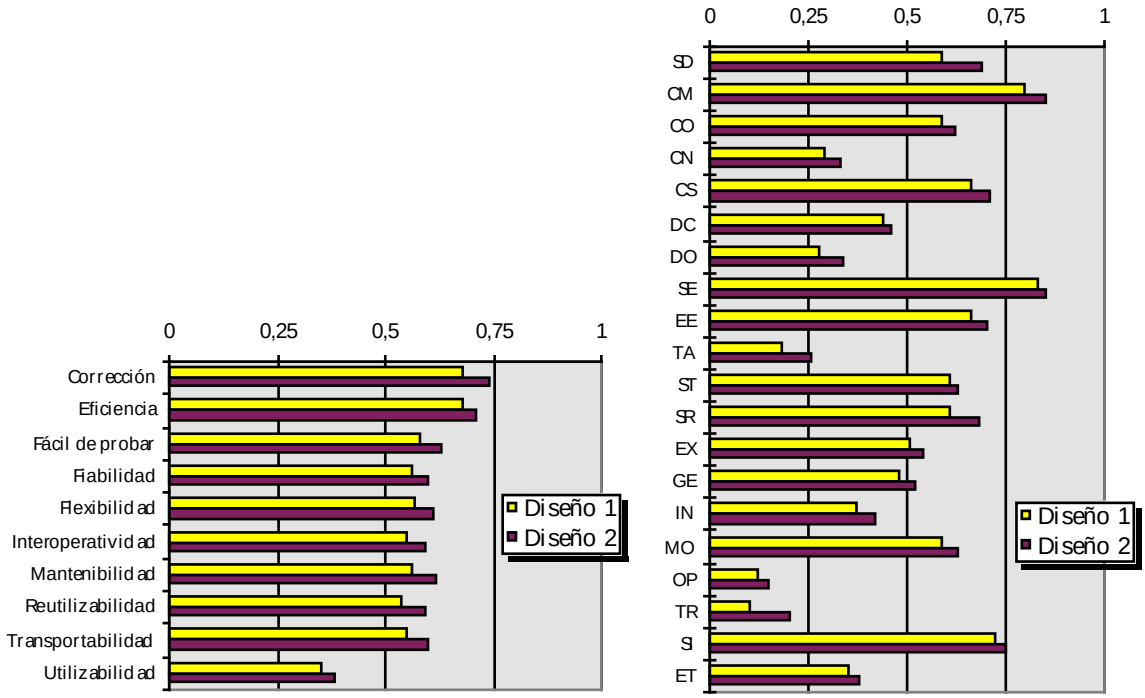


Figura I.65: Valores de los factores y de los criterios obtenidos para el diseño

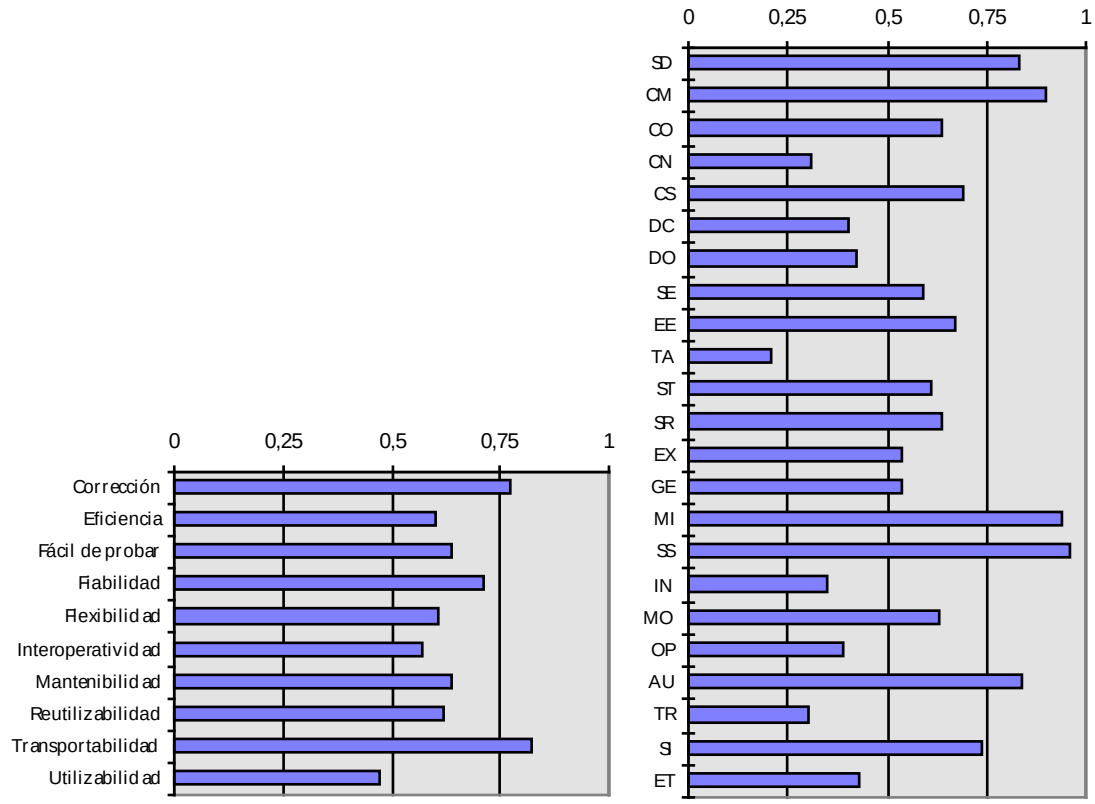


Figura I.66: Valores de los factores y de los criterios obtenidos para la primera implementación

Analizando los datos obtenidos, puede comprobarse cómo todos los factores han experimentado una ligera mejoría, al igual que los criterios, debido a que se ha hecho caso de las sugerencias obtenidas sobre el diseño previo. Aunque algunas medidas podían mejorarse, el desarrollador dio por bueno el diseño actual, dando comienzo entonces la implementación. Una vez finalizada, se le aplicaron las medidas correspondientes del modelo de calidad, obteniendo los datos mostrados en la Figura I.66 y un 0,70 como valor global de la calidad.

Puede verse cómo los factores han alcanzado unos valores bastante adecuados, así como ocurre con la mayoría de los criterios. No obstante, estos valores son mejorables. Algunos ejemplos de sugerencias concretas que se proporcionó al programador surgidas de las medidas fueron: utilizar el concepto de métodos constantes de C++ (CS_{17}), emplear métodos en línea (SE_{12}), tener cuidado con los métodos excesivamente pequeños (EE_{21}), verificar toda la memoria dinámica obtenida (ET_{12}) e introducir comprobaciones para evitar el acceso a punteros nulos (ET_{14}) así como optimizar la precisión y mejorar el seguimiento. Con toda la información proporcionada, el programador puso manos a la obra para mejorar su sistema, obteniendo los valores de factores y criterios de la Figura I.67. La calidad total del sistema alcanzó un valor de 0,79.

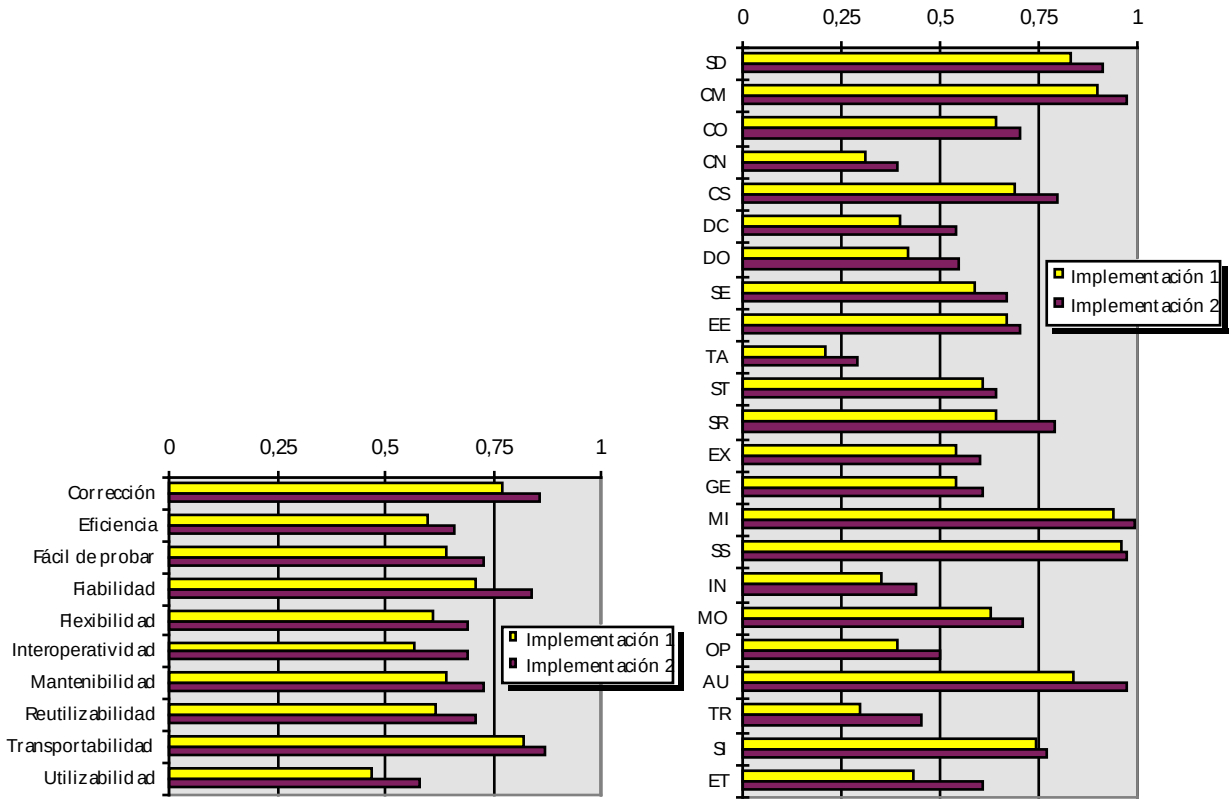


Figura I.67: Valores de los factores y de los criterios obtenidos para la implementación

Tras analizar los nuevos resultados (fruto de aplicar el 89% de las medidas del modelo), verificar que prácticamente todos los valores habían mejorado (tres de los factores y ocho de los criterios habían sobrepasado la barrera del 0,75 y tan solo el criterio de entrenamiento caía por debajo del 0,33) y comprobar las indicaciones proporcionadas por las medidas, el programador decidió dar por finalizada la implementación, puesto

que el pequeño tamaño del sistema, así como su objetivo, hacía que no mereciera la pena su modificación: las principales sugerencias se centraban en ampliar el uso de la herencia, del enlace dinámico y la sobrecarga.

El presente experimento pone de manifiesto que el escaso número de medidas de la fase de análisis causan que no exista una fuerte relación entre la calidad del análisis y la del diseño (que dispone de un número considerablemente mayor de medidas). De hecho, a pesar de que el diseño tenía en cuenta las mejoras sugeridas sobre el análisis, las nuevas medidas provocaron inicialmente una ligera disminución de la calidad, que no debe ser tenida como una indicación de un mal trabajo. Esta situación no ocurre entre el diseño y la implementación, puesto que se ha observado una continuidad progresiva en los datos aportados por el modelo de calidad, debido principalmente a que una gran cantidad de medidas son comunes a ambas fases y, por tanto, miden los mismos atributos de la calidad del sistema.

Seguidamente, se comentará el proceso seguido por el segundo desarrollador, cuyo trabajo fue evaluado a posteriori, es decir, sin tener una realimentación de los resultados proporcionados por el modelo de calidad.

Partiendo del análisis realizado en conjunto (como ya se ha indicado), cuyos resultados de calidad aparecen en la Figura I.62, el desarrollador afrontó el diseño del sistema. Los resultados del diseño se presentan en la Figura I.68, siendo 0,54 la calidad global.

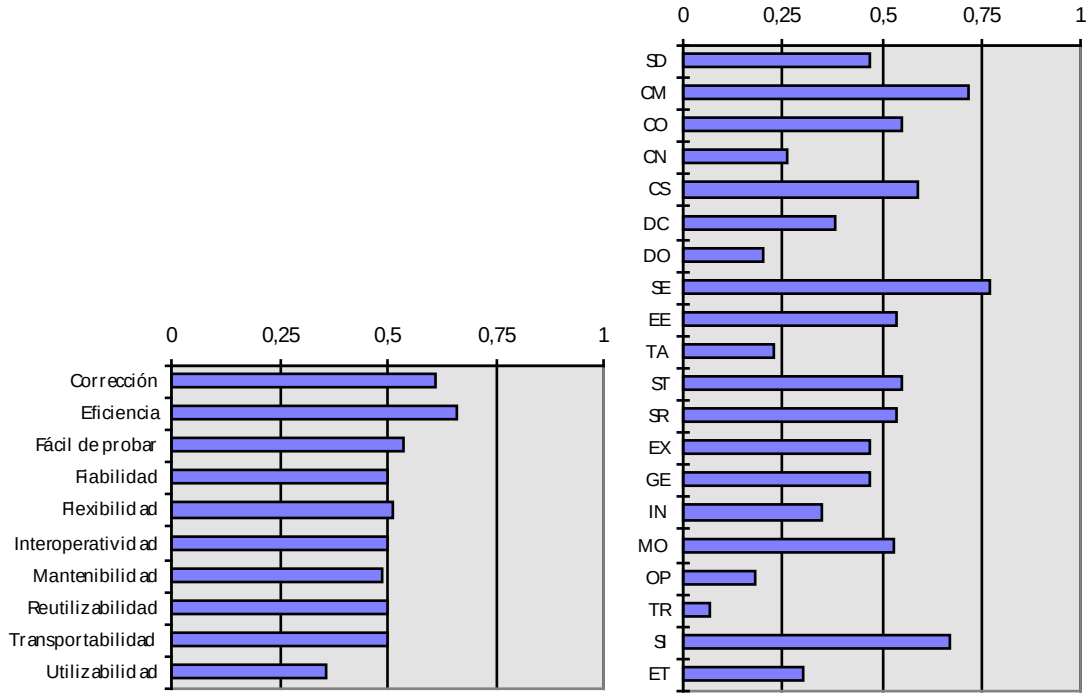


Figura I.68: Valores de los factores y de los criterios obtenidos para el diseño

Estudiando los factores, puede verse que la mayoría presentan unos resultados buenos, mientras que aproximadamente la tercera parte de los criterios obtuvieron un mal valor. Seguidamente, el desarrollador realizó su implementación, obteniendo los valores de calidad reflejados en la Figura I.69. El valor final de calidad alcanzado fue de un 0,60.

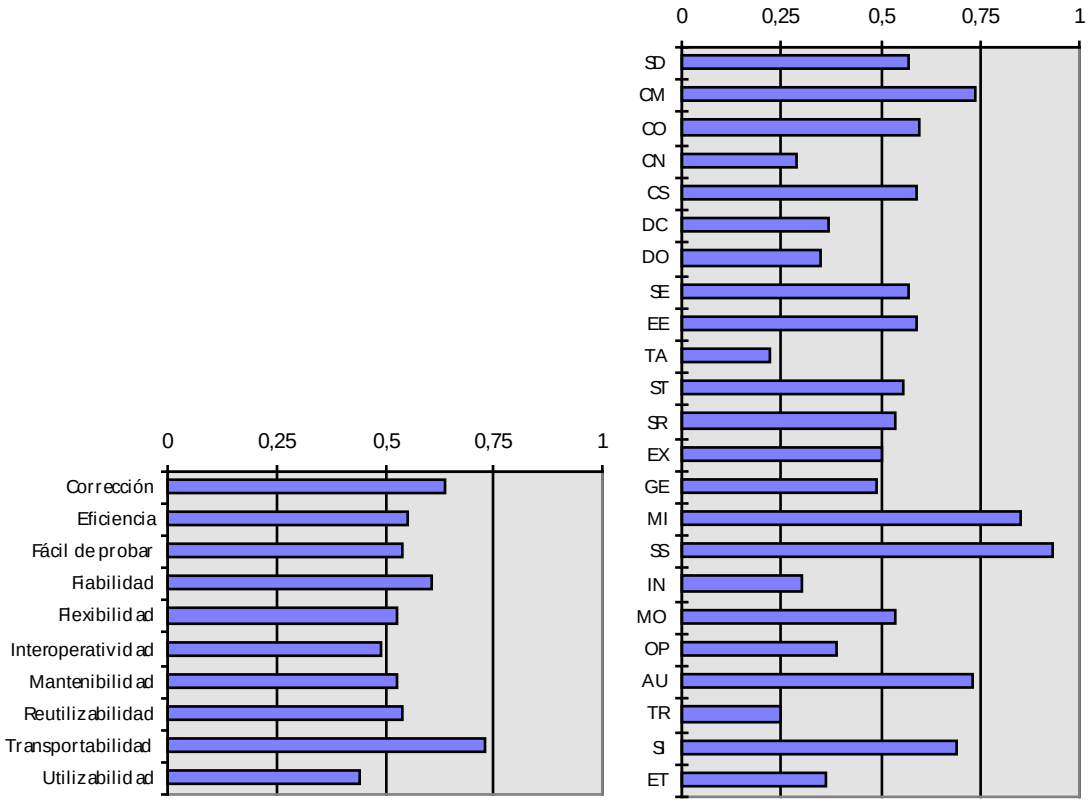


Figura I.69: Valores de los factores y de los criterios obtenidos para la implementación

Puede verse que tanto los factores como los criterios, en general, han experimentado una leve mejoría.

Para finalizar, es de destacar, para el segundo desarrollador, el paso del valor de calidad inicial de un 0,44 hasta un 0,60, mientras que el primer desarrollador, ha pasado de un 0,44 hasta un 0,79, lo cual refleja que la utilización del modelo de calidad puede ayudar a mejorar sustancialmente un desarrollo *software* desde las etapas iniciales del ciclo de vida. La Figura I.70 muestra este progreso en la calidad global del sistema a lo largo de su ciclo de vida para ambos desarrollos.

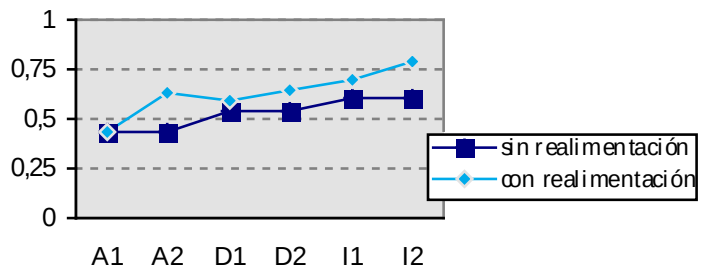


Figura I.70: Evolución de la calidad durante el ciclo de vida

I.10. PRODUCTOR-CONSUMIDOR

Descripción del sistema: La aplicación a desarrollar consiste en un pequeño programa en C++ que solucione el problema del productor-consumidor utilizando una cola de prioridades. Este desarrollo forma parte de un ejercicio para una clase de orientación a objetos y C++.

I.10.1. EXPERIMENTO 16

Descripción del experimento: Se trata de aplicar el modelo de calidad sobre dos implementaciones distintas (*A*: realizada por el profesor y *B*: realizada por uno de los alumnos) del sistema descrito.

Objetivos: Analizar los resultados de la evaluación sobre los dos sistemas, comparándolos entre sí. Además, se tratará de ver cómo se puede aumentar la calidad global del sistema mejorando uno de los criterios que hayan alcanzado un peor valor.

Resultados: Analizando los resultados, puede verse fácilmente que el sistema *A* tiene una calidad mejor que la del sistema *B*. La Figura I.71 muestra un resumen de los resultados alcanzados por ambas implementaciones, donde el sistema *A* tiene un valor de calidad de 0,71, mientras que el sistema *B* ha alcanzado un 0,51.

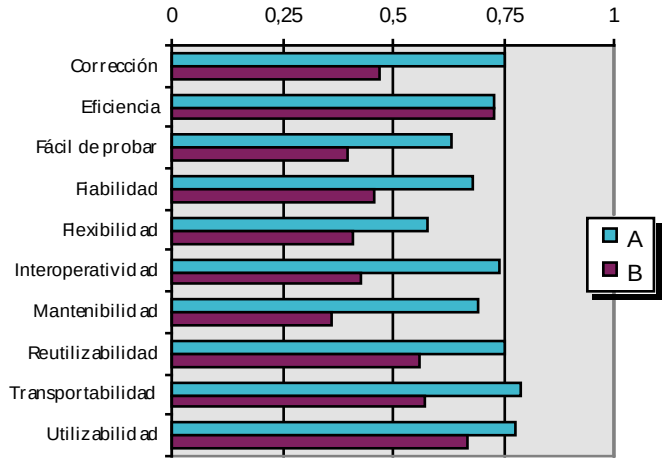


Figura I.71: Factores de calidad para el problema del productor-consumidor

Analizando los resultados del sistema *B*, es de destacar que el valor más bajo es el obtenido por el factor mantenibilidad (0,36). El procedimiento seguido entonces fue estudiar los criterios de dicho factor (que se muestran en la Figura I.72) para analizar los valores menores con el fin de entender el problema y solucionarlo.

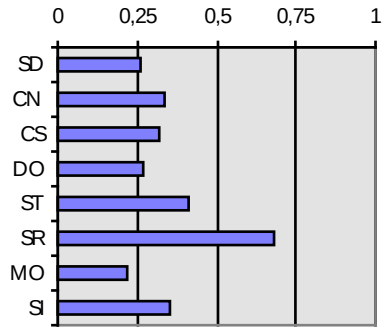


Figura I.72: Mantenibilidad del problema del productor-consumidor

En este caso, era de destacar el valor considerablemente reducido obtenido por la modularidad (0,22). Por tanto, a continuación se estudiaron sus medidas. Así, por ejemplo, para la medida MO_1, se observó que devolvía un valor de 0,17, por lo que se recomendó al programador disminuir el tamaño de los métodos. Repitiendo un estudio

similar para cada medida, el programador fue capaz de solucionar los problemas sugeridos por las distintas medidas. Una vez que se hubo terminado de realizar los cambios propuestos por las medidas de este criterio, se volvió a pasar el modelo de calidad, obteniéndose ahora un valor de 0,76 para la modularidad. Por consiguiente, no solo el valor del factor mantenibilidad mejoró hasta alcanzar un valor de 0,44 en la nueva implementación (sistema B'), sino que otros factores también se vieron afectados por la mejoría, como puede verse en la Figura I.73, obteniéndose un valor de calidad global de 0,56. Siguiendo este mismo proceso, el alumno podría aprender y mejorar su sistema hasta alcanzar o, incluso, poder superar el valor de calidad obtenido por su profesor.

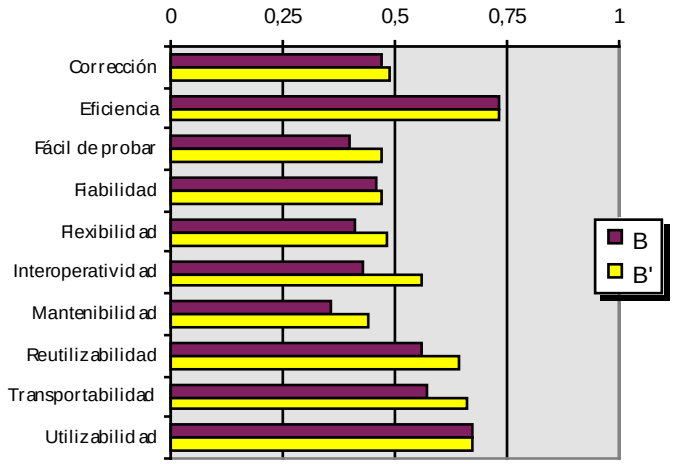


Figura I.73: Factores de calidad al mejorar la modularidad

I.11. SIMULADOR DE GASES

Descripción del sistema: Se trata de un programa en C++ que simula el movimiento de un número reducido de partículas de un gas dentro de un recinto cerrado por cuatro paredes. La pared de la derecha tiene un foco de calor, la de la izquierda se encuentra a temperatura ambiente, mientras que las otras dos tienen una temperatura variable según la proximidad a las otras paredes. Las partículas pueden colisionar entre sí o contra las paredes, y el choque puede ser elástico o inelástico. [Martínez, 01]

I.11.1. EXPERIMENTO 17

Descripción del experimento: Se han evaluado 33 diseños e implementaciones diferentes con el modelo de calidad. Así mismo, estos 33 sistemas han sido evaluados por dos expertos que les han asignado una puntuación global en el rango [A, E], tanto para el diseño como para la implementación. Con estos datos, se han determinado los umbrales de los valores de calidad del *software*.

Objetivos: El principal objetivo era poder concretar los umbrales que determinan una buena calidad y una mala calidad.

Resultados: El valor de calidad dado por el modelo y por los dos expertos a los 33 diseños del problema está en la Tabla I.10, donde los datos del modelo tienen una media de 0,557 con una desviación típica de 0,224. En los valores de los expertos, las letras dan una indicación de la calidad subjetiva, siendo la letra A una calidad alta y la E una calidad baja.

nº	Experto 1	Experto 2	modelo
1	D	D	0,33
2	E	E	0,17
3	B	B	0,77
4	C	C	0,55
5	C	A	0,67
6	B	A	0,83
7	B	B	0,77
8	C	D	0,35
9	A	A	0,88
10	B	B	0,77
11	D	E	0,27
12	E	E	0,18
13	A	A	0,87
14	C	D	0,42
15	D	C	0,38
16	D	B	0,52
17	D	B	0,46

nº	Experto 1	Experto 2	modelo
18	B	B	0,73
19	C	C	0,46
20	D	D	0,30
21	C	C	0,50
22	B	A	0,81
23	E	E	0,22
24	B	B	0,82
25	C	B	0,65
26	A	B	0,89
27	D	B	0,45
28	A	B	0,84
29	D	D	0,28
30	C	B	0,58
31	C	C	0,54
32	C	C	0,48
33	B	B	0,63

Tabla I.10: Resultados de los 33 diseños por los expertos y el modelo

Para establecer los umbrales, se han considerado tres zonas de calidad: buena, normal y mala (correspondientes a los valores A, C y E, respectivamente). Estas zonas están separadas entre sí por los valores B y D. Para determinar el valor umbral de B, se han tomado los diseños que han obtenido una calificación subjetiva cercana a B (es decir, BB, AB, BC o AC). Este grupo de diseños tiene una media de 0,751 (σ=0,093). Análogamente, para determinar el umbral de D, se han tomado los diseños con una calificación subjetiva cercana a D (es decir, DD, CD, DE o CE). Este grupo ha obtenido una media de 0,333 (σ=0,055).

Para la implementación, se repitió el mismo procedimiento, mostrándose en la Tabla I.11 los resultados, con una media 0,555 y desviación típica 0,217. El grupo de implementaciones con calificación subjetiva próxima a B dio una media de 0,729 (σ=0,100), mientras que la media de las implementaciones cercanas a D es 0,315 (σ=0,067).

nº	Experto 1	Experto 2	modelo
1	D	C	0,38
2	E	D	0,19
3	B	B	0,77
4	C	B	0,61
5	C	C	0,60
6	B	B	0,79
7	B	B	0,79
8	C	C	0,41
9	A	A	0,89
10	B	B	0,74
11	C	D	0,37
12	D	E	0,22
13	A	A	0,88
14	C	C	0,43
15	D	D	0,35
16	D	D	0,37
17	D	E	0,32

nº	Experto 1	Experto 2	modelo
18	B	B	0,74
19	C	C	0,45
20	D	D	0,29
21	C	C	0,47
22	B	B	0,78
23	D	C	0,34
24	B	C	0,72
25	C	B	0,61
26	A	B	0,88
27	D	D	0,38
28	A	B	0,89
29	D	E	0,26
30	C	B	0,58
31	C	B	0,59
32	C	C	0,52
33	B	B	0,72

Tabla I.11 Resultados de las 33 implementaciones por los expertos y el modelo

Con estos datos, y tomando valores redondos por comodidad, puede establecerse el umbral a partir del cual se valorará que un sistema presenta una buena calidad en el valor 0,75, mientras que el umbral por debajo del cual se considerará una calidad mala se fija en un tercio. Los valores entre ambos serán, entonces, aquéllos que presentan un valor de calidad media.

Los valores así determinados se han contrastado, además, en el resto de los experimentos realizados, obteniendo como conclusión su utilidad como delimitadores de la calidad de los sistemas.

I.11.2. EXPERIMENTO 26

Descripción del experimento: Se han evaluado 33 diseños e implementaciones diferentes con el modelo de calidad y por dos expertos. A continuación se han comparado entre sí los datos obtenidos.

Objetivos: El principal objetivo era comprobar si los resultados proporcionados por el modelo de calidad, tanto en la fase de diseño como en la de implementación, se aproximan a los proporcionados por los humanos.

Resultados: El valor de calidad otorgado por el modelo y por los dos expertos a los 33 diseños del problema se muestra en la Tabla I.10, mientras que los obtenidos para las implementaciones aparecen en la Tabla I.11.

Para ver la relación existente entre los datos proporcionados por el modelo y la opinión de los profesores, se realizaron unas regresiones lineales [De Groot, 88]. En primer lugar, se realizaron sendas regresiones lineales para comparar los datos de cada uno de los expertos con los del modelo, tanto para la fase de diseño como para la implementación. Los datos se muestran resumidos en la Tabla I.12 (el tamaño de la muestra es 33 en todos los casos).

	Experto 1	Experto 2
Diseño	R^2 = 87,13% Recta de regresión: y = 12,2x – 1,6	R^2 = 79,40% Recta de regresión: y = 12,5x – 1,2
Implementación	R^2 = 91,28% Recta de regresión: y = 11,7x – 1,1	R^2 = 77,91% Recta de regresión: y = 11,1x – 0,8

Tabla I.12: Datos de las regresiones entre los expertos y el modelo

Además, se hicieron dos nuevas regresiones tomando la media de la opinión de los expertos y los resultados del modelo de calidad, tanto para la fase de diseño como para la de implementación. Los resultados, junto con las rectas de regresión obtenidas, se muestran en la Figura I.74.

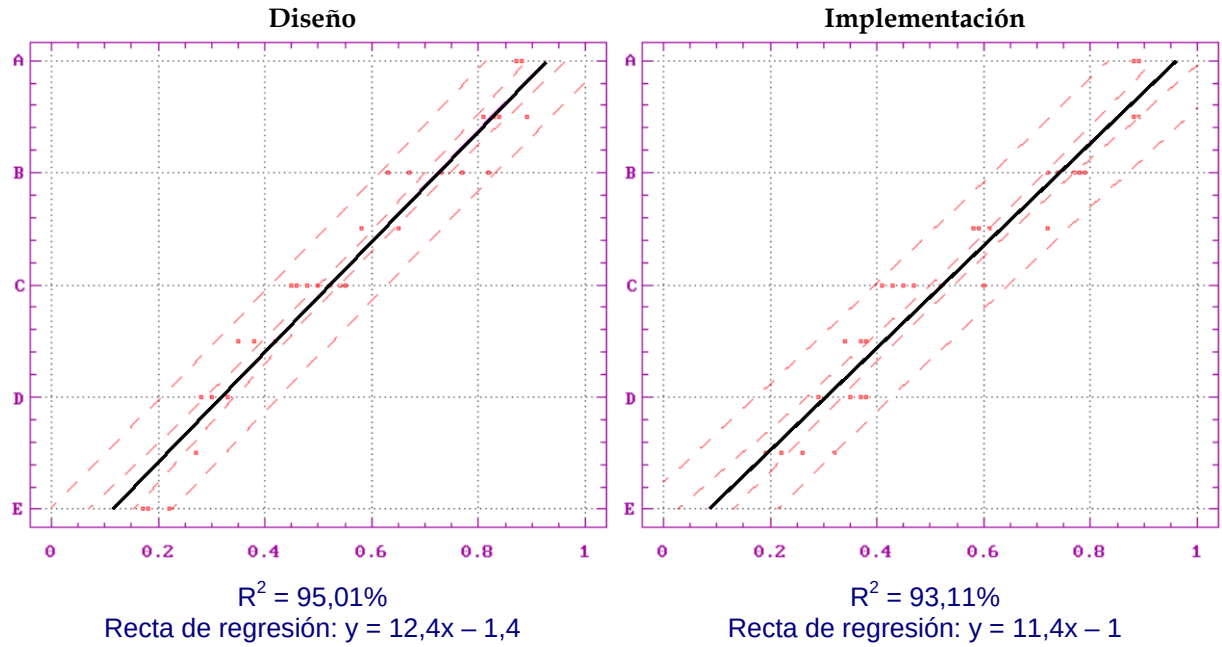


Figura I.74: Rectas de regresión entre la opinión de los expertos y los resultados del modelo para el diseño y la implementación

Estos datos reflejan la gran similitud existente entre la opinión de los expertos y los resultados proporcionados por el modelo. La mayor similitud existente en el diseño puede explicarse porque los expertos humanos pueden valorar mejor la calidad de un diseño que la calidad del código fuente de un programa.

I.12. SISTEMA RETRIBUTIVO

Descripción del sistema: El sistema consiste en un sistema retributivo para una empresa, que incluye un amplio abanico de posibilidades, como las retenciones del IRPF, la Seguridad Social y la posibilidad de tener trabajadores extranjeros. El sistema toma sus datos de un fichero y proporciona los resultados en otro fichero.

I.12.1. EXPERIMENTO 10

Descripción del experimento: El sistema ha sido construido, independientemente, por 96 alumnos como trabajo práctico de un curso de C++. El sistema se ha evaluado en primer lugar por los profesores, que les han asignado una puntuación en el rango [0, 6]. Posteriormente, se ha aplicado el modelo de calidad, en su fase de calidad de la implementación, sobre el código fuente de los distintos sistemas, para evaluar la calidad del desarrollo de cada uno de ellos, tanto teniendo en cuenta los pesos de los factores como sin tenerlos.

Objetivos: Evaluar una serie de sistemas que resuelven el mismo problema con el fin de obtener un valor de calidad sobre cada uno de ellos para poder compararlo con el obtenido por los profesores. De esta manera, se podrá también estudiar si el modelo de calidad proporciona valores relacionados con las calificaciones otorgadas por expertos y si esta relación es más fuerte cuando se emplean pesos en el modelo.

Resultados: Las soluciones aportadas por la mayoría de los sistemas tienen un buen número de similitudes, debido a la naturaleza del problema, bastante delimitado en su enunciado real, a las guías y consejos dados por los profesores para su resolución y a que los alumnos pertenecen a quinto curso de una licenciatura en Informática, por lo que se les presupone unos buenos conocimientos de las técnicas de programación.

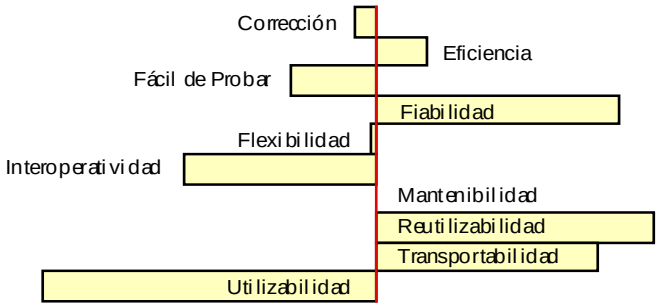


Figura I.75: Factores con valores menores y mayores de los sistemas

Por ello, los valores de calidad obtenidos por el modelo de calidad se encuentran bastante próximos entre sí, existiendo menos de un cuarto de punto de diferencia entre los mejores y peores trabajos, no encontrándose tampoco ninguna solución especialmente mala o especialmente buena. Como muestra la Figura I.75, los sistemas suelen presentar bajos resultados en los factores utilizabilidad (el enunciado del ejercicio indicaba que la interfaz con el usuario debía ser a través de ficheros), interoperatividad (no había necesidad de interactuar con otros sistemas) y fácil de probar (al ser un sistema sencillo, no se incluían ayudas a la depuración). Por el contrario, los mejores resultados se obtuvieron en los factores reutilizabilidad (se hace un buen uso de las capacidades que ofrecen las clases que redundan en una posible reutilización), fiabilidad (los resultados que proporcionan los sistemas tienen la fiabilidad requerida por un sistema retributivo) y transportabilidad (se ha programado en un C++ estándar, siendo raro el sistema que utiliza elementos dependientes del *software* o del *hardware*).

	Corrección	Eficiencia	Fácil de probar	Fiabilidad	Flexibilidad	Interoperatividad	Mantenibilidad	Reutilizabilidad	Transportabilidad	Utilizabilidad	**Autovector**
Corrección	1	9	7	6	8	9	7	7	8	9	**0,3591**
Eficiencia	1/9	1	1/5	1/6	1/3	5	1/2	1/6	1/2	1	**0,0274**
Fácil de probar	1/7	4	1	1/3	2	7	1	2	4	7	**0,0895**
Fiabilidad	1/6	6	2	1	7	9	6	4	8	9	**0,1946**
Flexibilidad	1/8	3	1/3	1/7	1	8	1/2	1/4	6	7	**0,0680**
Interoperatividad	1/9	1/5	1/8	1/9	1/8	1	1/9	1/9	1/5	1/5	**0,0122**
Mantenibilidad	1/7	2	1/3	1/6	2	9	1	1/3	7	7	**0,0809**
Reutilizabilidad	1/7	6	1/4	1/4	4	9	3	1	6	9	**0,1142**
Transportabilidad	1/8	2	1/6	1/8	1/6	5	1/7	1/6	1	2	**0,0311**
Utilizabilidad	1/9	1	1/8	1/9	1/7	5	1/7	1/9	1/2	1	**0,0230**

Tabla I.13: Matriz de relevancia de los factores dentro de la calidad y su autovector para el sistema retributivo

La Tabla I.13 muestra la matriz de relevancia utilizada para obtener los distintos pesos de los factores, construida por los profesores teniendo en cuenta los objetivos planteados en la elaboración del enunciado del problema.

Una vez finalizado el experimento, se ha realizado un estudio estadístico de los datos, con el fin de averiguar la relación existente entre los valores de calidad proporcionados al aplicar el modelo de calidad y las calificaciones dadas por los profesores. Para ello, se realizó una regresión lineal [De Groot, 88]. En primer lugar, la regresión se llevó a cabo con los datos obtenidos sin utilizar los pesos obtenidos. La recta de regresión obtenida, junto a los datos, se muestra en la Figura I.76, por lo que se puede comprobar que el modelo de calidad tiene una buena correlación con los expertos a la hora de valorar un sistema.

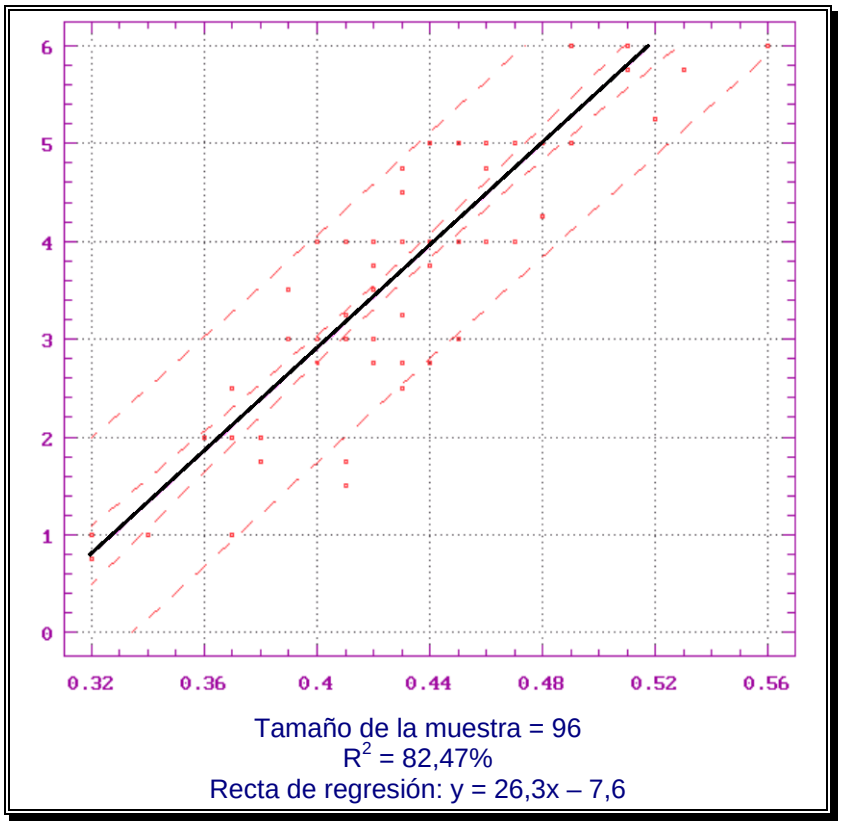


Figura I.76: Recta de regresión de los datos de los profesores sobre los del modelo sin pesos

Posteriormente, se repitió el mismo cálculo estadístico, pero ahora teniendo en cuenta los pesos obtenidos a partir de la opinión de los profesores. La nueva recta de regresión, junto a sus datos, se presenta en la Figura I.77, por lo que se comprueba que aumenta la correlación entre el modelo de calidad y la opinión de los expertos y, por tanto, puede utilizarse como un medio de estimar la calidad de los sistemas. Este hecho demuestra que la presencia de los pesos permite ajustar más la calidad a la realidad.

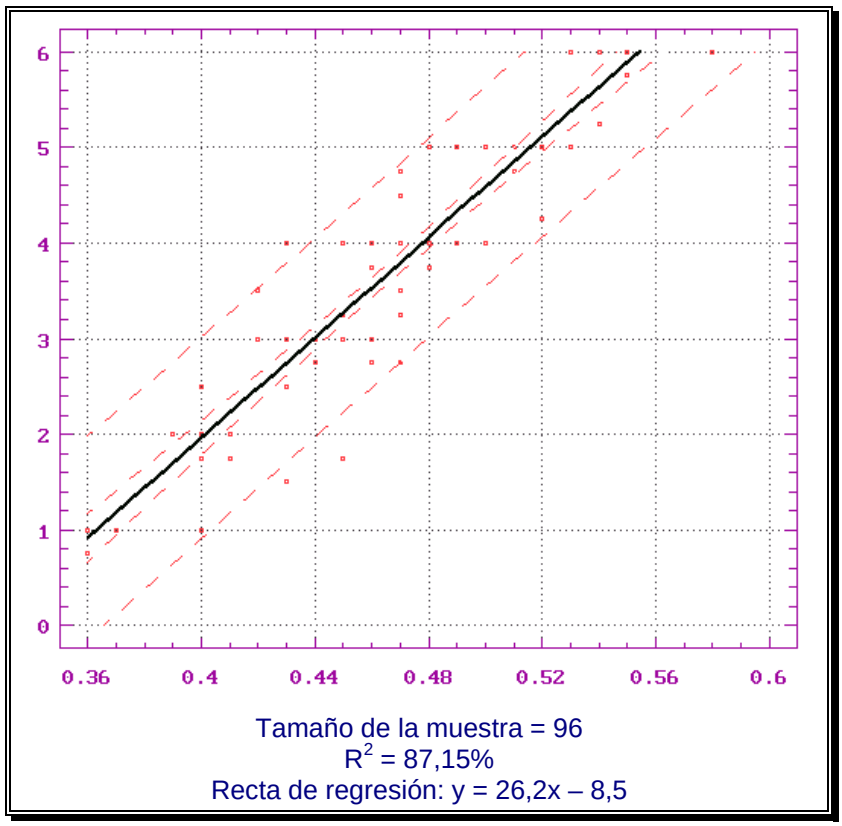


Figura I.77: Recta de regresión de los datos de los profesores sobre los del modelo con pesos

La calidad no es un hecho.
Es una costumbre.

— Aristóteles

ANEXO II. MANUAL DE CALIDAD

II. MANUAL DE CALIDAD

II.1. INTRODUCCIÓN

El presente documento contiene una serie de recomendaciones a tener en cuenta a la hora de desarrollar un sistema empleando el paradigma del modelo de calidad para la orientación a objetos [Ramírez, 03].

Estas recomendaciones se han obtenido directamente de la observación de las distintas medidas, y se han resumido y agrupado de forma que una misma recomendación esté respaldada por una o varias medidas, que, habitualmente, representarán los distintos aspectos de la orientación a objetos a los que afecta. Estas medidas se indicarán en cada recomendación (entre paréntesis), normalmente al final de la explicación de la misma, con una justificación de cómo afecta al desarrollo. En el caso de que aparezcan precedidas de un signo negativo, se tratará de una influencia negativa del concepto sobre el que trate la recomendación en el campo de la orientación a objetos que represente dicha medida. Cuando se trata de una influencia favorable no se precederá la medida con ningún signo.

Debe tenerse en cuenta que algunas recomendaciones pueden ser contradictorias entre sí. En estos casos, debe seguirse la recomendación que favorezca al criterio de calidad más importante dentro del sistema en desarrollo.

La estructura del documento intenta hacer más fácil el acceso a una información puntual en un determinado momento, agrupando estas recomendaciones dentro del campo de la orientación a objetos donde se puedan incluir, obteniéndose así recomendaciones referidas a cada una de las fases diferentes del ciclo de vida del *software*, concernientes a cada uno de los componentes característicos de la orientación a objetos (clase, método, atributo) o, incluso, recomendaciones relativas a conceptos más globales del desarrollo como pueden ser su documentación, pruebas o interacción con el usuario.

Este documento está referido al desarrollo utilizando la tecnología orientada a objetos sobre cualquier lenguaje. Sin embargo, cada diferente lenguaje puede tener una serie de particularidades que provoquen que no sean aplicables exactamente con lo que se comenta en este documento. Como ejemplo, un reducido número de consejos son dependientes del lenguaje, principalmente referidos al C++.

II.2. JERARQUÍA

Una jerarquía de clases es un conjunto de clases relacionadas mediante una relación de generalización (herencia).

- Cuantas **más clases tenga una jerarquía**, será más sencillo que dicha jerarquía pueda ser utilizada para obtener un mayor provecho, pero más complicada será la estructura de dicha jerarquía **(GE_{36} / $-SI_{83}$)**. No obstante, cada jerarquía debe tener el **número apropiado de clases**, sin que éstas sean muy pocas ni demasiadas, dependiendo de la funcionalidad y organización de la jerarquía. Es difícil decir cuál es este número, pero debe ser el que consiga dar la mayor modularidad al sistema **(MO_7)**.

- Las clases deben **aparecer una sola vez** como máximo en cada jerarquía de herencia. Una situación en el que puede ocurrir lo contrario es cuando se manejan gran número de clases, de modo que resulta difícil llevar un control exhaustivo de las mismas. En el caso en el que se hereda una misma clase por dos ramas distintas, se ha de usar la herencia virtual (en C++) **(SI_{82})**.

- Las jerarquías de herencia deben **utilizar pocos objetos** para favorecer su simplicidad **(SI_{84})**.

- En un sistema deben existir **diversas jerarquías de clases independientes**, ya que esto hará más sencilla la ampliación del sistema, debido a la menor interrelación entre sus elementos. Sin embargo, crecerá la complejidad del sistema **(EX_{24} / -SI_{81})**.

- El **número de herencias** en una jerarquía de clases influye en la posibilidad de que un cambio en una de las clases afecte al funcionamiento de la jerarquía, disminuyendo la modularidad debido a las interrelaciones que se crean. Además, la jerarquía se haría más compleja, aunque las clases de dicha jerarquía serían más concisas, debido a la reutilización **(CN_{11} / -ST_{27} / -MO_{6} / -SI_{85})**.

- **Demasiadas clases raíz** en la jerarquía facilitarán que un cambio en una de las clases raíz afecte a su funcionamiento. Además, se complicará el sistema **(-ST_{29} / -SI_{87})**.

- El mayor **número de clases hoja** en las jerarquías de herencia afecta a la simplicidad del sistema **(-SI_{86})**.

- Debe intentarse **minimizar la profundidad** de la jerarquía, ya que una clase tendrá menos posibilidades de verse afectada por una modificación cuanto más cercana se encuentre de la raíz de la jerarquía. Esto permite también que se reduzca la complejidad, ya que los programadores depuran y modifican las clases más eficientemente si están cerca de la raíz que si están en las profundidades de la jerarquía **(ST_{16} / SI_{93})**. Además, cuanto más profundo es un grafo de herencia, mayor es la complejidad para su diseño e implementación, puesto que se verán involucrados un mayor número de métodos y clases **(SI_{80})**. La profundidad óptima de una jerarquía se sitúa entre 3 y 5 niveles [Harrison, 00].

II.2.1. CLASES

- Las clases deben disponer de **comentarios con una estructura estándar**, pues esto permite una fácil identificación de cada parte del comentario, influyendo positivamente en la modularidad **(SD_{2} / MO_{11})**.

- Todas las clases deben **definirse satisfactoriamente** (en C++ está asegurado pues es un lenguaje fuertemente tipado) y **deben usarse**, puesto que una clase definida que no esté usada puede ser el reflejo de una referencia omitida **(CM_{3} / CM_{8})**.

- El **número de clases** existentes en un sistema influye en la simplicidad, ya que cuanto menor sea, será más fácil controlar el sistema **(-SI_{66})**.

- Las **clases aisladas** (sin ninguna relación de herencia con el resto) implican un crecimiento de la complejidad **(-SI_{77})**.

- El **número de objetos de cada clase** incrementa la complejidad del sistema **(-SI$_{67}$)**.

- Debe seguirse una **estructura uniforme y estándar en la forma y en el orden de definir los miembros** de las clases **(CS$_8$ / DC$_7$)**.

- Es preferible repartir **cada clase en un fichero**, puesto que esto proporciona un modo estándar de representarlas y favorece la expansibilidad, ya que está siempre identificado el lugar en el que se encuentra el código a modificar. Además, dota de mayor modularidad y simplicidad al sistema **(CS$_{26}$ / EX$_{11}$ / MO$_{30}$ / SI$_{42}$)**.

- Las clases deben **tener un único objetivo**. El concepto de modularidad se basa en que cada módulo represente una única funcionalidad **(MO$_5$)**.

- Deben **definirse distintos tipos de constructores para cada clase**, con el fin de que ésta sea lo más general posible, es decir, pueda inicializarse del mayor número de formas posibles **(GE$_{32}$)**.

- Las clases deben **tener definidos atributos y métodos**, pues la ausencia de atributos puede indicar una incompletitud de la clase y la falta de métodos puede reflejar la ausencia de servicios de la misma. Además, se consigue una mayor consistencia y estructuración de las clases **(CM$_{23}$ / CM$_{17}$ / CS$_{15}$ / CS$_{18}$ / DC$_8$ / SR$_4$)**.

- Las clases deben **permitir el acceso a sus atributos únicamente a través de sus métodos**, ocultando sus atributos. La presencia de atributos públicos y, en menor medida, de atributos protegidos (C++) va en contra de la idea estándar de las clases como estructuras de datos dotadas de servicios, ya que se podría acceder a su información sin utilizar la interfaz prevista, con lo que se reduciría la modularidad. Hay que evitar principalmente el uso de atributos públicos para impedir accesos peligrosos. Así mismo, hay que reducir al máximo los atributos protegidos, ya que puede llegarse a permitir el acceso a su información tras heredar de su clase. El uso de un atributo sin usar la interfaz puede provocar que se modifique de tal forma que afecte al resto de los clientes de la clase reduciéndose la estabilidad **(DC$_5$ / ST$_4$ / MO$_{10}$)**.

- Las clases deben **facilitar las especializaciones**, permitiendo que se herede de ella todo lo necesario para construir clases más específicas (esto no se cumpliría si se tienen atributos privados sin una interfaz para acceder a ellos), de lo contrario se reduce la expansibilidad del sistema **(EX$_2$)**.

- Las clases deben tener el **número apropiado de atributos**. El número medio según [Lorenz, 93] es 6, puesto que una media superior podría indicar que la clase hace más de lo que debe, mientras que una media inferior podría reflejar una clase demasiado elemental **(MO$_4$)**. Además, se ha de tener en cuenta que para favorecer la sencillez de las clases el **número de atributos no debe ser elevado**. Las clases grandes pueden resultar más difíciles de entender y, por tanto, de reutilizar (principalmente a través de la herencia, requiriendo modificaciones en la clase) **(-SI$_{39}$)**.

- Las clases deben tener el **número apropiado de métodos**. El número medio adecuado según [Lorenz, 93] es 20. Una media superior indica demasiadas responsabilidades en demasiadas pocas clases. Una media inferior refleja demasiadas clases para pocos servicios, esto es, una modularización excesiva **(MO$_3$)**. Para favorecer la sencillez de

las clases hay que intentar minimizar el número de métodos. Cuantos más métodos tenga una clase (tanto definidos como heredados), más compleja será su utilización debido a que su interfaz será más complicada. Hay que tener en cuenta, además, que a mayor número de métodos en una clase, mayor será el posible impacto en sus hijos, debido a que éstos heredarán dichos métodos **(SI_{46})**. Sin embargo, una clase debe ofrecer un buen número de métodos para dar un amplio servicio cumpliendo las normas de la modularidad **(MO_{17})**. Además, una clase será más general si su interfaz proporciona un amplio número de elementos accesibles, aunque su interfaz será más complicada de manejar **(GE_4 / $-SI_{78}$)**.

- Las clases deben **ser autosuficientes** y necesitar el menor número posible de servicios de otras clases, de lo contrario se produce un excesivo acoplamiento entre las clases, lo que provoca un deterioro de la modularidad del sistema **(MO_{18})**.

- El **uso abusivo de las clases** hace que crezca la posibilidad de un impacto en su modo de funcionamiento por una modificación, ya que una revisión en el código de una clase obligaría a revisar todo el código desde el que se utilice dicha clase **($-ST_{19}$ / $-ST_{20}$ / $-ST_{30}$)**. Sin embargo, esto indica también una mayor generalidad de las clases **(GE_{11} / GE_{29})**.

- El número de **objetos definidos como atributos en el seno de cada clase** incrementa la influencia de un posible cambio en la definición de dichos objetos, aumentando, además, la complejidad **($-ST_{22}$ / $-SI_{71}$)**.

- La **cantidad de mensajes enviados desde una clase** influye negativamente en su eficiencia de ejecución y su simplicidad. También afecta a su estabilidad, ya que una modificación en cualquiera de las clases de las que requiere servicios podría provocar una modificación en el código de la clase **($-EE_1$ / $-ST_{33}$ / $-SI_{62}$ / $-SI_{91}$ / $-SI_{92}$)**.

- El **proceso lógico de los métodos debe ser independiente de constantes** (valores límite, tamaños de vectores o de *buffers*...) o debe estar parametrizado. Esto favorece la expansibilidad y generalidad del sistema **(EX_1 / GE_{20})**.

- Las **clases** deben ser **independientes** de la aplicación para ser generales. Esto se consigue pensando de forma global a la hora de diseñar las clases y sin centrarse en el caso específico que se trata **(GE_3)**.

- Si alguno de los **atributos** de una clase necesita una **gestión dinámica de memoria** o interviene en una **jerarquía**, entonces dicha clase debe contener un **constructor por omisión**, un **constructor de copia**, un **destructor** y el **operador de asignación definidos explícitamente**, aunque el lenguaje o la clase no lo exija. De esta forma se puede definir completamente su comportamiento sin dejar que el compilador les asigne un funcionamiento por omisión. Todo esto, además, proporciona mayor consistencia y ayuda a mantener un patrón que da mayor estructuración y modularidad **(CM_{15} / CS_{25} / SR_{10} / MO_9)**.

- Debe **evitarse el uso de la amistad** (propia de C++) puesto que no es consistente con la idea de la ocultación de la información de la orientación a objetos. Esto provoca pérdida de consistencia y estabilidad, puesto que una modificación en la clase afectará también a sus clases o funciones amigas. Tampoco es consistente con los

patrones estándar de programación actual, por lo que se perjudica la estructuración. Además, la modularidad y la simplicidad también se ven afectadas negativamente **(CS_{27} / DC_{10} / ST_{25} / SR_{9} / MO_{8} / SI_{30})**.

II.2.2. ATRIBUTOS

- Las **declaraciones** de los atributos deben estar **comentadas**, explicando su utilización, propiedades, unidades, etc., ya que esto permite que el sistema sea auto-descriptivo **(SD_{10})**.

- Todos los atributos deben **definirse satisfactoriamente** (en C++ esto está asegurado pues es un lenguaje fuertemente tipado) y **deben usarse**, puesto que un atributo definido que no esté usado puede ser el reflejo de una referencia omitida. La existencia de atributos que no se usan influye negativamente en la concisión y la eficiencia de almacenamiento del sistema, pues suponen código inútil y una reserva de memoria innecesaria. Además, el hecho de que un atributo no se use puede ser el reflejo de alguna inconsistencia del sistema y complica el código **(CM_{1} / CM_{6} / CN_{14} / CS_{14} / SE_{15} / SI_{36})**.

- Los atributos se deben **utilizar tanto para modificar su valor como para consultarlo**. No realizar alguna de las dos operaciones implica algún tipo de omisión o defecto en la utilidad de dicho atributo **(CM_{10})**.

- El **uso directo de los atributos desde fuera de su propia clase** incide negativamente en la consistencia y la estructuración, al ir en contra de los estándares de la orientación a objetos **(-CS_{19} / -CS_{20} / -SR_{1} / -SR_{2} / -SR_{6})**. También afecta a la estabilidad, ya que cualquier modificación en la clase podría influir en los métodos o clases que usen sus atributos **(-ST_{1} / -ST_{2} / -ST_{3} / -ST_{18})**. Además, influye en la expansibilidad, puesto que al violarse la encapsulación se dificulta la ampliación de la clase **(-EX_{13} / -EX_{14} / -EX_{18})**. Al producirse así un fuerte acoplamiento se perjudica la modularidad **(-MO_{13} / -MO_{14} / -MO_{15} / -MO_{19})**. La simplicidad se ve afectada al resultar más difícil comprender el uso de esos atributos **(-SI_{35} / -SI_{37} / -SI_{38} / -SI_{59})**.

- La forma estándar adecuada de compartir información entre todos los objetos de una clase es utilizar un **atributo de clase** (un atributo estático en C++), que es el mecanismo que proporciona la orientación a objetos y que, además, permite ahorrar espacio de almacenamiento **(DC_{11} / SE_{16})**.

- Un excesivo **uso de los atributos** (por herencia o composición) aumentará la influencia de los cambios en el sistema, así como su complejidad. Además, habrá una mayor dificultad en comprender su utilidad **(-ST_{6} / -ST_{7} / -SI_{43} / -SI_{44})**.

- La **cohesión entre los atributos** de una clase beneficia a la modularidad. La falta de cohesión puede indicar que la clase debería ser dividida en dos o más subclases. Además, una baja cohesión incrementa la complejidad pues aumenta la probabilidad de aparición de errores en el diseño **(MO_{24} / SI_{99})**.

- Para aumentar la generalidad de los atributos, éstos deben ser **independientes de la aplicación**. Es decir, no deben ser resultado de la presencia de otros componentes **(GE_{1})**.

II.2.3. MÉTODOS

- Los métodos deben disponer de **comentarios descriptivos con estructura estándar** (explicando su utilización, propiedades, funcionamiento...). Este tipo de información es especialmente valorada por el personal nuevo que tiene que trabajar con un *software* desarrollado previamente **(SD_3 / SD_{11})**.

- Todos los métodos deben **definirse satisfactoriamente** (en C++ está asegurado pues es un lenguaje fuertemente tipado) y **deben usarse**, puesto que un método definido que no esté usado puede ser el reflejo de una referencia omitida, lo que implicaría que el sistema está incompleto o podría reflejar una inconsistencia **(CM_2 / CM_7 / CN_{15} / GE_{24} / SI_{47})**.

- El uso de **objetos globales** (en los lenguajes que lo permitan) hace al sistema menos consistente. Es decir, debe cumplirse la **Ley de Demeter**: Para todas las clases *C*, y para todos sus métodos *M*, todos los objetos a los que *M* envía un mensaje deben ser: (1) un objeto enviado como parámetro a *M*, incluido el objeto indicado por `this` (en C++), `Current` (en Eiffel) o `self` (en Smalltalk); o (2) un objeto que sea atributo de la clase *C* [Lieberherr, 88] **(-CS_4)**.

- Cada método solamente debe **estar definido en una clase**. Hay que exceptuar a los métodos operador y aquellos métodos que son sobrecargas o redefiniciones de un método heredado. De lo contrario, se viola la modularidad y consistencia del sistema, al tiempo que se complica la comprensión del código **(CS_{21} / MO_{28} / SI_{61})**.

- Los métodos deben tener un **tamaño reducido**, de lo contrario, probablemente, se esté dando poca modularidad al sistema. [Lorenz, 93] entiende por tamaño reducido aquel método que no supera las 25 líneas de código C++ (ó 9 en Smalltalk) con sentencias ejecutables o declaraciones **(MO_1)**. No obstante, los métodos con un **tamaño excesivamente pequeño** penalizan la eficiencia de la ejecución debido a que una parte importante del tiempo de proceso se gasta durante la secuencia de llamada y la secuencia de retorno. Se entiende por tamaño excesivamente pequeño aquél que es inferior a la quinta parte de los tamaños reducidos estimados por [Lorenz, 93], es decir, 5 líneas de código C++ (a menos que sea `inline`) ó 2 de Smalltalk. En ese caso, se alcanzaría una modularidad exagerada, es decir, se podría llegar a una atomización de los componentes **(-EE_{21} / -MO_2)**.

- Los métodos deben **ser independientes del sistema desarrollado**, puesto que un cambio en éste podría afectar al funcionamiento de los mismos **(ST_5 / GE_2)**.

- El excesivo **uso de los métodos** hace crecer la influencia de los cambios en el sistema, con la consiguiente reducción de su estabilidad y simplicidad **(-ST_{14} / -ST_{15} / -SI_{56} / -SI_{57})**. Sin embargo, los **métodos no públicos** de una clase deben ser utilizados por el mayor número posible de métodos de dicha clase para que resulten más generales **(GE_{10})**. Por otro lado, conviene que los métodos sean **utilizados varias veces** (al menos una vez). El que un método definido no se use (o se use poco) refleja que el sistema no está completo, o que se han olvidado ciertas referencias. No obstante, si los métodos se usan mucho, se hace menos estable al sistema **(CM_{29} / -ST_9)**.

- Las operaciones que **producen una excesiva cantidad de procesamiento** afectan a distintos criterios de calidad. Gran parte del tiempo de procesamiento necesario para ejecutar una función viene dado por la cantidad de mensajes que se envían. Si un método realiza muchas llamadas a otros métodos, será demasiado complejo y resultará poco eficiente y estable, pues depende de muchos otros métodos **(-EE_1 / -EE_{23} / -ST_{17} / -ST_{26} / -SI_{58} / -SI_{62} / -SI_{94})**.

- **La entrada, el proceso y la salida no deben estar mezclados** en un mismo método. Cada método se debe encargar de una única tarea. Un método que realiza la entrada/salida y el proceso no es tan general como un método que solo lleva a cabo el proceso **(GE_{12})**.

- Los **argumentos** presentes en los métodos de las clases deben estar cohesionados para favorecer la modularidad y la simplicidad **(MO_{27} / MO_{29} / SI_{98} / SI_{100})**.

- Deben existir **conexiones entre los métodos** de una misma clase (a través de sus atributos) para favorecer la cohesión de la clase [Bieman, 95a], que beneficia a la modularidad y a la simplicidad **(MO_{25} / MO_{26} / SI_{101} / SI_{102})**.

- Todo método que no modifique el valor de los atributos del objeto desde el que ha sido llamado debe ser de **tipo constante** (en C++) para conservar la consistencia y mantener una correcta estructuración del sistema **(CS_{17} / SR_5)**.

- El uso de **funciones no miembro** (permitidas en lenguajes como C++) no es consistente con la idea de una programación orientada a objetos, basada en clases que proporcionan una serie de servicios, con lo que se reduce la modularidad. Hay que exceptuar las funciones obligatorias de ciertos lenguajes (como la función `main` de C++). Esto también puede provocar un descenso en la estabilidad del sistema, al tiempo que no es consistente con los patrones que se deben definir para la estructura del sistema **(-CS_{22} / -SR_7 / -MO_{23} / -SI_{63})**.

- El uso abusivo de las **funciones que se expanden** en la línea de la llamada (`inline`, en C++) hace que el tamaño final del código resultante pueda crecer desmesuradamente, aunque aumenta la eficiencia de ejecución **(-SE_{12} / EE_{10})**.

II.2.3.1. Parámetros

- Los parámetros formales de los métodos deben ir **comentados**, explicando su utilización, propiedades, unidades... **(SD_{12})**.

- Los **parámetros actuales** de los mensajes enviados a los objetos **deben coincidir** en número y tipo con los parámetros formales del método que recibirá el mensaje **(CM_{16})**.

- Tienen que **utilizarse todos** los parámetros formales de los métodos o funciones, ya que la aparición de un parámetro formal que no se use indica algún fallo en la completitud del sistema y complica el mismo **(CM_{28} / SI_{34})**.

- El **tamaño de los parámetros** (tanto formales como actuales) de los métodos no debe ser demasiado elevado, para no consumir una cantidad excesiva de espacio de memoria en la pila, lo que repercutiría negativamente en la eficiencia de

almacenamiento y ejecución (el paso de argumentos a los métodos consume un tiempo considerable debido a que es necesario copiar toda la información de los parámetros actuales a los parámetros formales), así como en la complejidad del sistema **(SE_{18} / EE_{18} / EE_{19} / SI_{32} / SI_{33})**. En cualquier caso, una solución sería pasar por referencia (o con punteros) aquellos parámetros de gran tamaño, aunque asegurándose entonces de que no se permite modificar dicho parámetro desde el cuerpo del método.

- En los métodos, no se deben usar **datos con distintos alcances** (locales, parámetros, globales y atributos), ya que esto complica el código **(SI_{23})**.

II.2.3.2. Sobrecarga

- Conviene **sobrecargar los métodos** para permitir manejar distintos tipos de datos **(DC_{12} / DC_{13})**. Cuantas más veces se sobrecargue un método, más general resultará el mismo, aunque perjudicará su estabilidad y simplicidad **(-ST_{37} / GE_6 / GE_9 / GE_{25} / -SI_{48})**.

- El abuso en la **utilización de los métodos sobrecargados** complica el código, ya que una modificación en el modo de funcionamiento del servicio implantado en estos métodos podrá afectar a sus usuarios, haciendo que el código sea menos estable. Si algún método sobrecargado sufre alguna modificación, habrá que tenerlo en cuenta cada vez que sea llamado. No obstante, la generalidad se ve favorecida **(GE_{17} / -ST_{10} / -SI_{49}).**

II.2.3.3. Operadores

- La **redefinición de operadores** (propia de lenguajes como C++) permite que el código generado que los utilice sea más conciso. También dan mayor consistencia al sistema al permitir realizar la llamada de igual forma que cuando se utiliza un operador normal, permitiendo, además, ampliar el código intuitivamente y que éste sea más general pues se utiliza como con los tipos predefinidos. Además, se mejora la modularidad al dar a las clases más servicios **(CN_2 / CS_{16} / EX_{16} / GE_{27} / MO_{22})**.

- La **redefinición de los operadores** debe hacerse **de manera uniforme**, sin modificar su semántica para conservar la consistencia del código **(CS_{11} / SI_{52})**.

- El **uso de métodos operador** simplifica el código, al no necesitar emplear nombres de métodos para realizar ciertas operaciones habituales sobre tipos definidos por el usuario **(SI_{52})**.

II.2.3.4. Variables, Tipos, etc.

- Las declaraciones de todas las variables, tipos de datos, etiquetas, identificadores constantes, nombres simbólicos, macros y directivas de compilación deben estar **comentadas**, explicando su utilización, propiedades, unidades... **(SD_9)**.

- Todas las variables deben **definirse satisfactoriamente** (en C++ está asegurado) y **deben usarse**, puesto que una variable definida que no esté usada puede ser el reflejo de una referencia que ha sido omitida **(CM_4 / CM_9 / CN_{16})**.

- Cada variable debe utilizarse con **una única finalidad** para hacer más simple el código **(SI$_{16}$)**.

- El **tamaño de las variables locales** condiciona el espacio de memoria (pila) necesario para almacenarlas, lo que incide negativamente en la eficiencia de almacenamiento y en la simplicidad del sistema **(-SE$_{17}$ / -SI$_{60}$)**.

- La **densidad de variables usadas** en los métodos no debe ser excesiva, puesto que cuantas más variables (locales, parámetros y globales) se usen, mayor complejidad tendrá dicho método **(-SI$_{24}$)**.

- No resulta adecuado usar el **mismo nombre para un identificador en distintos ámbitos** ya que esto puede impedir una adecuada estructuración del programa y complicar la comprensión del sistema **(CS$_{32}$ / SR$_{3}$ / EX$_{6}$ / SI$_{15}$)**.

- Se debe utilizar la **elipsis** de C++ (un método es capaz de admitir un número variable de parámetros) en la medida de lo posible, ya que de este modo se podrá utilizar en distintas situaciones evitando tener que definir varios métodos similares, cuya única diferencia sea el número de parámetros formales, lo cual permite escribir un código más conciso aunque más complejo **(CN$_{6}$)**.

- El uso de alias o **referencias** (con el signo & en C++) entre nombres de variables complican la comprensión del programa **(-SI$_{90}$)**.

- La complejidad se ve aumentada por la presencia de **variables volátiles** de C++, ya que éstas pueden ser modificadas desde el exterior del código, dificultando la comprensión del mismo al no poder saber fácilmente el valor de las mismas en cada momento **(-SI$_{89}$)**.

II.2.4. GENERICIDAD

La genericidad se implementa de distinta forma en los diferentes lenguajes orientados a objetos. Por ejemplo, C++ usa la palabra `template`, Eiffel usa `generic` y Java carece de esta propiedad.

- Deben definirse **clases genéricas** en la medida de lo posible. La definición de clases genéricas evita tener que definir varias clases con las mismas funcionalidades y estructura para ser utilizadas con distintos tipos de datos, por lo que permiten que el código sea más conciso y consistente, se mantiene una estructura estándar y se mejoran la expansibilidad y la generalidad, si bien complican el sistema **(CN$_{7}$ / CS$_{24}$ / DC$_{15}$ / SR$_{8}$ / EX$_{10}$ / GE$_{7}$ / -SI$_{68}$)**.

- Deben definirse **métodos genéricos** en la medida de lo posible, ya que la definición de métodos genéricos evita tener que definir varios métodos con el mismo comportamiento para distintos tipos de datos, por lo que permiten que el código sea más conciso y consistente, se mantiene una estructura estándar y se mejoran la expansibilidad y la generalidad. Sin embargo, el sistema se complica **(CN$_{5}$ / CS$_{23}$ / DC$_{14}$ / EX$_{19}$ / GE$_{8}$ / -SI$_{65}$)**.

- Las **clases y métodos** que puedan ser aplicados sobre distintos tipos de datos deben estar **parametrizados** o bien sobrecargados, para que el sistema sea mas completo y más general **(CM_{22} / GE_{18})**.

- Las **clases genéricas** permiten trabajar sobre distintos tipos de datos. Sin embargo, **puede ser necesario sobrecargarlas** para manejar ciertos tipos de datos que necesitan un control diferente. Las clases genéricas sobrecargadas mantienen los mismos beneficios que las clases genéricas simples (estructuración, expansibilidad y generalidad) **(DC_{16} / SR_{11} / EX_{22} / GE_{31})**.

- Debe usarse con cuidado la genericidad, ya que el uso de las **clases genéricas** puede causar que una **modificación** afecte al modo de funcionamiento de una buena parte del programa, con la consiguiente pérdida de estabilidad del sistema **(-ST_{21})**.

II.2.5. TIPOS ABSTRACTOS DE DATOS

- Los tipos abstractos de datos creados deberán estar dotados de un **completo conjunto de operadores** redefiniendo y adaptando (por medio de la sobrecarga de operadores) el comportamiento habitual de los operadores necesarios, consiguiendo que sean lo más completo y estándar posible **(CM_{21} / DC_4)**.

- Debe seguirse una **representación estándar** de los tipos abstractos de datos para que resulten consistentes **(CS_6 / DC_6)**.

- Las traducciones o **transformaciones entre tipos abstractos de datos** deben realizarse dentro de cada tipo abstracto de datos **(DC_3)**.

II.3. HERENCIA

- Las **clases se deben reutilizar**, bien por medio de la herencia, bien por una relación de uso, lo que da mayor consistencia al sistema, permite conservar la estructura de las clases y aumenta la expansibilidad y la generalidad. Hay que tener en cuenta que cuanto más se reutilice una clase, más inestabilidad dará al sistema **(CS_{31} / -ST_8 / EX_{12} / GE_{30})**.

- La **reutilización de las clases** por medio del uso o la herencia es beneficiosa pues evita el tener que definir nuevas clases iguales para realizar las mismas tareas o similares, por lo que resultará un código más conciso y expansible **(CN_9 / CN_{10} / EX_{23})**. También será más consistente y estructurado, ya que esto coincide con los estándares de la orientación a objetos, lo que, además, dota al sistema de mayor modularidad **(CS_{29} / CS_{30} / SR_{12} / SR_{13} / MO_{16} / MO_{20})**. Hay que tener en cuenta que cuanto más se reutilice una clase mas inestabilidad dará al sistema y se complicará el código, pero indicará que ésta es más general **(-ST_{23} / -ST_{24} / GE_{33} / GE_{34} / -SI_{72} / -SI_{73})**.

- La **reutilización de métodos** por herencia disminuye el código de las clases al no tener que redefinir los métodos heredados, facilitando la ampliación del sistema y la generalidad, si bien, se penaliza la estabilidad y la simplicidad **(CN_4 / -ST_{13} / GE_{28} / -SI_{53})**.

- La **herencia** por parte de las diversas clases **de un atributo** indica que dicho atributo es general, aunque complica la comprensión del código **(GE_{22} / -SI_{41})**.

- La cantidad de **atributos que se heredan** proporciona una idea de lo general que es la clase, pese a que dificulta la comprensión del código **(GE_{23} / -SI_{45})**.

- Las **clases abstractas** favorecen la modularidad debido a que permiten definir esquemas de comportamiento comunes a otras clases **(MO_{21})**.

- La **herencia múltiple** debe **evitarse en lo posible**. Una clase que pueda heredar de muchas otras clases evita tener que implementar de nuevo la mayoría de sus funciones (es beneficiosa para la concisión del sistema). Sin embargo, aumenta la inestabilidad y complejidad del sistema, puesto que un cambio en cualquier padre puede afectar al futuro funcionamiento de sus hijos **(CN_8 / -ST_{28} / -ST_{34} / -SI_{28} / -SI_{69})**.

- El **número de descendientes** de una clase afecta a su complejidad y al impacto que sufrirán si dicha clase sufre una alteración. Por otro lado, la clase resultará mas general **(-ST_{35} / GE_{35} / -SI_{76})**.

- El número de **hijos directos** de una clase influye en su complejidad. Cuanto mayor sea, a más clases afectará un posible cambio en el padre y más complicado resultará expandir la clase (habrá que modificar a sus hijos) **(ST_{36} / EX_{21} / SI_{70})**.

- El número de **clases de las que hereda** una clase (directa o indirectamente) influye en su simplicidad **(-SI_{75})**.

II.3.1. MÉTODOS VIRTUALES

Los métodos virtuales se denominan `virtual` en C++, `deferred` en Eiffel o métodos genéricos en CLOS. Algunos lenguajes orientados a objetos no tienen métodos virtuales, mientras que en otros, todos los métodos son virtuales.

- La presencia de **métodos virtuales** permite que el código generado sea más conciso y fácilmente expansible, al no tener que preocuparse de seleccionar el método apropiado durante el desarrollo, ya que se enlaza dinámicamente en la ejecución. Sin embargo, penaliza la eficiencia de ejecución y la simplicidad, ya que en tiempo de ejecución se debe estimar cuál de los métodos es el que se debe invocar **(CN_3 / -EE_{20} / EX_{17} / -SI_{29}).** Además, la existencia de métodos virtuales necesita internamente de una estructura de datos para almacenar diferente información durante la ejecución, con lo que se utiliza un espacio de memoria extra para cada método, que no sería necesario si no fueran virtuales **(-SE_{14})**.

- Las llamadas a métodos que **se resuelven en tiempo de ejecución** (enlace dinámico) hacen el código más conciso, pero tardan más en procesar su invocación. La llamada a un método virtual emplea más tiempo que la llamada a un método no virtual porque en tiempo de ejecución debe determinarse a qué función hay que invocar dependiendo del tipo del objeto desde el que se realiza la llamada. También se ve perjudicada la sencillez del sistema **(CN_{13} / -EE_9 / -SI_{54})**.

- Las clases que hereden de una clase abstracta deben tener **redefinidos los métodos virtuales puros heredados** para que el sistema resulte completo **(CM_{24})**.

- Es conveniente que los **destructores de las clases raíz de una jerarquía sean virtuales** [Ellis, 91], principalmente cuando la jerarquía tiene o puede llegar a tener clases abstractas. Esto permite un correcto funcionamiento del enlace dinámico, al tiempo que hace al sistema más consistente y estable. También se favorece la expansión del código, al no ser necesario modificarlo si se va a utilizar el enlace dinámico para destruir los objetos **(CS_{35} / ST_{12} / EX_{15})**.

II.3.2. SOBRE-ESCRITURA

- La **sobre-escritura de atributos y métodos** hace el código más complejo **(-SI_{26} / -SI_{40})**.

- Si algún **método sobre-escrito sufre alguna modificación**, habrá que tenerlo en cuenta en cada llamada, para verificar que la modificación no ha alterado su funcionamiento, con la consiguiente disminución de estabilidad y sencillez del sistema **(-ST_{11} / -SI_{51})**.

- Un método es **menos general si hay que sobre-escribirlo**, puesto que esto indica que tal como estaba diseñado no cumplía plenamente la función para la que se pensó **(-GE_{26} / -SI_{50})**.

II.3.3. AMBIGÜEDAD

- Las **referencias** a atributos, métodos, clases y variables **deben quedar resueltas**, es decir, no deben ser ambiguas **(CM_{11} / CM_{12} / CM_{13} / CM_{14})**.

- La **repetición de nombres** en identificadores diferentes puede producir confusiones o ambigüedades, obteniendo un código menos consistente y simple, además de permitir que se pueda expandir con menor facilidad **(-CS_{32} / -SR_{3} / -EX_{6} / -SI_{15})**.

II.4. REQUISITOS

- Debe realizarse y presupuestarse la realización de un **análisis de errores**, para asegurarse que se cumplen los requisitos de precisión del sistema **(AU_{1})**.

- Deben existir requisitos de **precisión** en los datos que se manejan **(AU_{3})**.

- En la especificación de requisitos deben indicarse los requerimientos de **recuperación de errores** en los datos de entrada **(ET_{4})**.

- Deben definirse requisitos para la **recuperación ante fallos computacionales o del *hardware*** **(ET_{9} / ET_{16})**.

- Deben definirse requisitos para la **recuperación ante fallos de los dispositivos de entrada/salida**, como un fin de fichero inesperado, errores de lectura o escritura... **(ET_{18})**.

- Deben estar definidos los **requisitos de comunicación** con otros sistemas o máquinas **(CC_1)**.
- Los **requisitos de ejecución** deben reflejarse en las siguientes fases del ciclo de vida del desarrollo del *software* **(EE_2)**.

II.5. ANÁLISIS

- Los **requerimientos del *software*** deben estar reflejados e identificados en la documentación del análisis **(TR_1)**.
- Para que el sistema sea completo, el análisis tiene que **satisfacer** todos los requerimientos descritos en los **requisitos *software***. Hay que tener en cuenta las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual implica modificaciones a realizar en el análisis **(CM_{25})**.
- Debe realizarse una **representación estándar del análisis** utilizando, además, alguna metodología orientada a objetos. Debe definirse y seguirse un estándar de representación de los elementos del sistema. Esto proveerá de consistencia al sistema **(CS_1)**.
- El análisis debe reflejar los **requerimientos de almacenamiento**, que se tendrán en cuenta al evaluar la eficiencia de almacenamiento **(SE_1)**.

II.6. DISEÑO

- El diseño tiene que **satisfacer todos los aspectos contemplados en el análisis**. Hay que tener en cuenta las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual hace cambiar el análisis y, por tanto, supone cambios en el diseño **(CM_{26})**.
- Debe realizarse una **representación estándar del diseño** empleando, además, alguna metodología orientada a objetos. Debe definirse y seguirse un estándar de representación de los elementos del sistema **(CS_1)**.
- Las **interfaces** de las clases deben estar en concordancia durante todas las fases del desarrollo para facilitar la expansión de las mismas **(EX_7)**.
- Todos los elementos referenciados (atributos, métodos, variables, tipos...) deben haber sido definidos previamente. Un sistema no está completo si, en cualquiera de sus fases de desarrollo, se pretenden **utilizar elementos no definidos (-CM_{18})**.
- Los **requerimientos de la especificación de requisitos *software*** deben estar reflejados e identificados en los productos obtenidos durante el diseño **(TR_2)**.
- Con un **diseño *top-down*** resulta un sistema más sencillo, ya que ésta es la forma natural de razonar de una persona **(SI_1)**.
- La existencia de **métodos o clases duplicadas** complican la comprensión del diseño del sistema **(-SI_2)**.

- El diseño debe **reflejar los requerimientos de almacenamiento (SE_1)**.
- **La capacidad de disponer de un proceso paralelo debe estar controlada**, para permitir una mayor tolerancia a errores. Las funciones que puedan ser utilizadas concurrentemente deben ser controladas de forma centralizada para proporcionar el control de la concurrencia, bloqueos de entrada/salida, etc. **(ET_1)**.
- Deben establecerse **protocolos estándar de comunicaciones** con otros sistemas **(CC_2)**.
- Deben establecerse **interfaces sencillas** para facilitar la **entrada** y la **salida** desde y hacia otros sistemas **(CC_3 / CC_4)**.
- Deben establecerse una serie de **convenciones** uniformes para el manejo consistente de las **operaciones de entrada/salida (CS_2)**.
- Debe definirse una **representación de datos estándar para la comunicación** con otros sistemas **(DC_1)**.
- Deben establecerse **estándares de traducción entre distintas representaciones de información (DC_2)**.

II.7. IMPLEMENTACIÓN

- El **uso de constantes** dificulta la comprensión del código y su modificación, si pueden ser sustituidas por identificadores constantes o nombres simbólicos **(-SD_{22})**.
- Todos los elementos referenciados (atributos, métodos, variables, tipos...) deben haber sido definidos previamente. Un sistema no está completo si, en cualquiera de sus fases de desarrollo, se pretenden **utilizar elementos no definidos (-CM_{18})**.
- Todas las referencias para consultar el valor de un identificador deben realizarse tras haber sido definido con datos en la inicialización, calculados en una expresión u obtenidos en una llamada a una función (que los podría conseguir del exterior). Es decir, **cada dato debe tener un origen específico**. En estos identificadores también se incluyen los punteros **(CM_{19})**.
- La implementación tiene que **satisfacer todos los aspectos contemplados en el diseño**. Hay que tener en cuenta las sucesivas modificaciones que suelen sufrir las especificaciones de requisitos *software*, lo cual hace cambiar el análisis y el diseño, por lo que hay que mantener actualizada la implementación **(CM_{27})**.
- Deben utilizarse, en la medida de lo posible, **librerías estándar de clases**, proporcionadas por muchos de los compiladores de lenguajes orientados a objetos, ya que esto favorece la utilización de representaciones de datos estándar, la generalidad y la concisión, puesto que muchos componentes no tendrán que ser implementados por el programador **(CN_{12} / DC_9 / GE_5)**.
- Si se ha realizado un diseño orientado a objetos, para mantener la consistencia entre el diseño y la implementación, deberá utilizarse un **lenguaje orientado a objetos**, lo cual permitirá escribir también un código más sencillo. Aunque los lenguajes *basados*

en objetos poseen muchas de las características de los orientados a objetos, carecen de herencia, por lo que, si es posible, también habrá que evitarlos **(CS_{33} / SI_7)**.

- En un desarrollo orientado a objetos, para mantener la consistencia, deberá utilizarse un **lenguaje que soporte la herencia múltiple** (por ejemplo, C++, CLOS o Eiffel). Si el lenguaje elegido no lo soporta (como ocurre con las implementaciones estándar de Smalltalk o Java), habrá que realizar algún tipo de traducción del diseño para su implementación. De todas formas, ya se ha mencionado que el uso de herencia múltiple puede generar algún problema, aunque muchas veces permite una transición más natural del diseño a la implementación **(CS_{34})**.

- Para aumentar la estabilidad del sistema es preferible usar uno de los **lenguajes orientados a objetos fuertemente tipados** que ayudan al programador a detectar posibles inconsistencias de tipos en las expresiones y sentencias y, por tanto, a evitar posibles errores **(ST_{32})**.

- Se han de **prever posibles ampliaciones** incluyendo código, que debe estar oculto y bien documentado **(EX_4)**.

- Los **efectos colaterales** hacen que el código sea menos general y expansible. Por lo que, en caso de existir, deben estar muy bien documentados **(-EX_5 / -GE_{21})**.

- Las **interfaces de las clases** deben estar en concordancia durante todas las fases del desarrollo **(EX_7)**.

- Deben **quedar recursos disponibles para facilitar las ampliaciones**. Por ejemplo, en los programas en DOS, existe la limitación de los segmentos de memoria de 64Kb de tamaño máximo, límite que no debe ser alcanzado **(EX_8)**.

- La ampliación del sistema se ve dificultada si las clases disponen de métodos en los que el tiempo de ejecución sea crítico, es decir, las clases no deben pertenecer a un **sistema de tiempo real (-EX_9)**.

- Deben de usarse **herramientas para la edición o lectura del código fuente**, la compilación, la depuración, la integración del sistema y las pruebas, ya que estas facilitan la labor de realizar ampliaciones al sistema. Estas herramientas son especialmente importantes para los lenguajes orientados a objetos por la dificultad de manejar la herencia y el enlace dinámico en grandes sistemas **(EX_{20})**.

- La **librería matemática** usada debe proporcionar la precisión suficiente, por lo que debe ser chequeada teniendo en cuenta los objetivos generales de precisión **(AU_2)**.

- Los **algoritmos numéricos** utilizados deben proporcionar la precisión requerida **(AU_4)**.

- Los resultados alcanzados deben quedar dentro de los **límites de tolerancia (AU_5)**.

- No debe haber pérdida de **precisión en las asignaciones** o expresiones **(AU_6)**.

- Los **requisitos del *software*** deben estar reflejados e identificados en la implementación. Debe usarse alguna notación uniforme, prólogos de comentarios o comentarios embebidos para proporcionar esta referencia cruzada **(TR_3)**.

- **Depender del origen de la entrada de los datos o del destino de la salida** para procesar la información dificulta su comprensión **(-SI_3)**.
- Si el procesamiento **tiene memoria de las ejecuciones anteriores**, es decir, depende del conocimiento o resultados de procesamientos previos, afecta a la simplicidad **(-SI_4)**.
- La **entrada y salida de información al sistema deben estar localizadas en pocas clases**, puesto que cuanto mayor es el número de clases que actúan como interfaz con otro sistema, más complicado resulta esta comunicación, así como la implementación de los protocolos estándar **(CC_5 / CC_6 / MI_2 / MI_3)**.
- La **ejecución de cada método debe producirse de forma secuencial**, es decir, no deben producirse saltos en su interior, debiendo comenzar siempre por el principio (sin puntos de entrada alternativos) y sin poder regresar a distintos lugares tras su ejecución **(SI_8)**.
- Si un módulo tiene la capacidad de modificar su lógica de proceso, sería muy difícil reconocer su estado si ocurre un error. Por tanto, el **código de los módulos no debe ser modificado desde la ejecución del propio programa**. Además, el estudio estático de la lógica del módulo se complica **(-SI_{12})**.
- Todos **los datos que necesite un método** tienen que ser pasados como parámetros o bien ser atributos de la clase a la que pertenece, evitándose los problemas potenciales que pueden aparecer si se usan variables globales, constantes... **(SI_{13})**.
- La presencia de **diferentes tipos de datos** en una misma expresión complica el código **(-SI_{17})**.
- El **volumen y la dificultad del programa** afecta a su complejidad **(-SI_{96} / -SI_{97})**.

II.7.1. GENERALIDAD

- La existencia de métodos con referencias a **funciones dependientes de la máquina** (tanto *software* como *hardware*) disminuye la generalidad **(-GE_{13})**.
- La limitación del **volumen de datos que pueda procesar un método** afecta a la generalidad. Un método que ha sido diseñado e implementado para no aceptar más de 100 entradas para su procesamiento ciertamente no es tan general como un método que acepta cualquier volumen de datos de entrada **(-GE_{14})**.
- La limitación de los **valores de los datos a procesar por los métodos** penaliza a la generalidad. Cuanto más pequeño sea el subconjunto de todas las posibles entradas válidas, menos general es **(-GE_{15})**.
- Cada **constante y nombre simbólico** deben estar definidos una y solo una vez **(GE_{16})**.

II.7.2. INDEPENDENCIA

- Para la implementación del sistema debe utilizarse un **lenguaje de programación** que se encuentre disponible en otras máquinas (la misma versión y dialecto del lenguaje) **(MI_1)**.
- El **código y los datos** deben ser **independientes de los tamaños de las palabras**, caracteres, etc. de la máquina **(MI_4 / MI_5)**.
- Los métodos y atributos han de ser **independientes de la máquina** y **del sistema *software*** **(MI_{12} / MI_{13} / SS_5 / SS_6)**.
- La utilización de **código máquina o ensamblador** produciría una dependencia de la máquina en la que se ejecute el sistema **(-MI_6)**.
- El uso de **código dependiente del compilador** afecta a la independencia **(-SS_8)**.
- Se deben utilizar **librerías de rutinas estándar**, puesto que una función proporcionada por el compilador o por el sistema operativo puede no ser exactamente la misma que en otro **(MI_7 / SS_2)**.
- La **salida gráfica** debe ser independiente del tipo de monitor y de la resolución y número de colores de la tarjeta de vídeo **(MI_8)**.
- La **salida impresa** ha de ser independiente del tipo de impresora, trazador... **(MI_9)**.
- Las **comunicaciones por los puertos** tienen que ser independientes de su tipo y de su configuración **(MI_{10})**.
- Las **llamadas al sistema operativo** relacionadas con un *hardware* determinado afectan a la independencia **(-MI_{11} / -SS_4)**.
- Las partes del **código** que tengan algún tipo de **dependencia de la máquina** afectan a la simplicidad **(-MI_{14} / -SI_{25})**.
- Cuanta más **dependencia de utilidades *software* externas** se tengan, el programa será más dependiente del sistema *software* **(-SS_1)**.
- La utilización de ciertas **construcciones no estándar del lenguaje** permitidas por ciertos compiladores puede causar problemas de conversión cuando el *software* se migra a un nuevo entorno **(-SS_3)**.
- El sistema debe ser independiente del **sistema de gestión de bases de datos** empleado **(SS_9)**.
- El uso de **directivas de compilación** propias del compilador (como ejemplo, la directiva `pragma` del preprocesador de C++) no favorece la independencia **(-SS_7)**.

II.7.3. ERRORES

- **Cuando se detecte un error**, debe informarse al módulo que ha realizado la llamada, para que éste pueda tomar la decisión de cómo procesar el error **(ET_3)**.

- Se debe **comprobar toda la información de entrada antes de iniciar su procesamiento**, con el fin de identificar todos los posibles datos erróneos. El procesamiento no debe comenzar hasta que los errores hayan sido notificados y las correcciones se hayan realizado. Además, se debe comprobar que toda la información de entrada necesaria para el proceso se encuentra **disponible**, para evitar avanzar a lo largo de varios pasos del proceso antes de descubrir que los datos son incompletos **(ET_5 / ET_6 / ET_8)**.

- Debe verificarse que los **índices de los vectores** (o matrices), de los **bucles** o **transferencias** de datos se encuentran dentro del rango permitido antes de su utilización **(ET_{10} / ET_{11})**.

- Todos los errores posibles deberán ir asociados a **mensajes de error y avisos al usuario claros y sin ambigüedad** y estar descritos en los manuales **(CO_{11} / CO_{15} / DO_{13} / TA_5 / IN_{12} / IN_{13} / OP_2)**.

- Deben **verificarse que los resultados** de los procesos son adecuados **(ET_{15})**.

- El sistema permitirá **medir y resumir los errores** producidos, para estudiar los que se comenten con mayor frecuencia para intentar subsanarlos **(IN_{11})**.

- Deben establecerse una serie de convenciones para el **manejo consistente de los errores** que pudieran producirse **(CS_3)**.

- Deben preverse mecanismos para la **recuperación ante errores o fallos** del *hardware* o de los dispositivos **(ET_{17} / ET_{18} / ET_{19})**.

- Para poder afrontar los **errores** que puedan producirse, deben **interceptarse** para poder manejarlos de la forma apropiada, para lo cual debe utilizarse el sistema estándar de **gestión de excepciones** del lenguaje **(IN_8 / ET_{20} / ET_{21} / ET_{22})**, como el sistema de instrucciones `try` y `catch` de C++ y Java.

II.7.4. ESTRUCTURAS DE CONTROL

- Todos los cambios en el **flujo del programa deben estar comentados** para poder seguir la lógica del programa fácilmente **(SD_5)**.

- El **número de predicados** (en sentencias condicionales o repetitivas) de los métodos, así como la complejidad de éstos, dificultan la comprensión del código del programa **(-SD_{14} / -SI_{27}).**

- Los **puntos de decisión deben tener sus condiciones definidas**, así como disponer de código alternativo para resultar completos (por ejemplo, disponer de la opción `default` en la sentencia de selección múltiple) **(CM_{20})**.

- Se deben **excluir de los bucles las operaciones independientes** al bucle (como por ejemplo, la evaluación de expresiones constantes). Con esto se conseguirá una mayor eficiencia de ejecución al no ejecutarse sentencias innecesarias **(EE_3)**.

- La repetición innecesaria de la **evaluación de las expresiones compuestas** penaliza a la eficiencia **(-EE_5)**.

- Las **expresiones lógicas excesivamente complejas o negativas** incrementan la complejidad. Las expresiones compuestas con dos o más operadores lógicos o negaciones a menudo pueden ser eludidas **(-SI$_9$)**.

- El sistema se complica con los **saltos hacia el interior de los bucles** y los **saltos desde el interior de un bucle hacia el exterior**. Los bucles deben tener un único punto de entrada y un único punto de salida **(-SI$_{10}$)**.

- La **modificación de un índice de un bucle tipo** `for` dentro de su cuerpo complica la lógica del bucle y puede causar serios problemas a la hora de la depuración **(-SI$_{11}$)**.

- Las **etiquetas** no son necesarias a menos que se realice algún salto, por lo que su presencia, en términos generales, complicará el código haciéndolo más difícil de comprender **(-SI$_{14}$)**.

- La cantidad de posibles **ramas o caminos de ejecución** disminuye la simplicidad **(-SI$_{19}$)**.

- La existencia de **bloques anidados** aumenta la complejidad de la lógica del programa **(-SI$_{18}$)**.

- La presencia de **saltos** (sentencias `goto`) dentro del programa atenta contra las normas elementales de la programación estructurada **(-SI$_{20}$)**.

II.7.5. MEMORIA DINÁMICA

- Cuando se manejen variables o atributos de tipo **puntero**, es necesario **obtener antes memoria** dinámica para almacenar sus correspondientes valores. Es decir, se debe realizar la petición de memoria lo antes posible con el fin de evitar inconsistencias y que el sistema sea incompleto **(CM$_5$ / CS$_{10}$)**.

- Cuando en una aplicación se usa memoria dinámica, debe emplearse un **único gestor de memoria**, puesto que la utilización de más de uno puede llevar a un mal funcionamiento y a inconsistencias de la información almacenada **(CS$_9$)**.

- Para favorecer la eficiencia del almacenamiento, **es preferible usar memoria dinámica en lugar de memoria estática**, puesto que se minimiza la cantidad de memoria requerida durante toda la ejecución, aunque resulte un poco más cara su gestión **(SE$_5$)**.

- El **uso de punteros** complica considerablemente el código **(-SI$_{21}$)**.

- Debe **verificarse la cantidad de memoria dinámica proporcionada** por el sistema operativo tras una petición de memoria, para evitar posteriores errores **(ET$_{12}$)**.

- Debe **liberarse toda la memoria dinámica proporcionada** por el sistema operativo con una petición de memoria **(ET$_{13}$)**.

- Debe evitarse a toda costa el **acceso a punteros con un valor nulo (ET$_{14}$)**.

II.8. COMPRENSIÓN

- Los identificadores deben tener **nombres claros y descriptivos**, que indiquen la propiedad física o funcional representada, por lo que se necesita que tengan una determinada longitud en cuanto al número de caracteres que los forman **(SD_{16} / SD_{20})**.
- Los **nombres que se dan a todos los identificadores deben tener una estructura** y **un formato uniforme**, siguiendo alguna convención estándar dependiendo de su tipo, como por ejemplo, la notación húngara **(SD_{19} / CS_7 / SI_{55})**.
- Debe intentarse escribir el código fuente de tal forma que **las sentencias no ocupen más de una línea** y que **cada línea contenga una sentencia (SD_{18})**.
- La comprensión del código no tiene que requerir de un gran **esfuerzo mental (SD_{23})**.
- La escritura del **código debe seguir una organización estándar y uniforme** (estructura de los comentarios, declaraciones, sentencias...) siguiendo alguna normativa o guía de estilo, con el fin de facilitar su lectura e interpretación **(SD_{15})**.
- Las clases, métodos y demás elementos del lenguaje deben tener un **sangrado del código de forma consistente**, siguiendo una serie de reglas uniformes predefinidas, lo cual favorece la lectura del código **(SD_{17} / CS_5)**.

II.9. COMENTARIOS

- El código debe estar clara y **adecuadamente comentado** puesto que esto ayuda a su estudio. Además, se favorecen la consistencia y la expansibilidad del código. Las líneas con comentarios y en blanco (que facilitan la legibilidad) deben repartirse por todo el código **(SD_1 / SD_{21} / CS_{28} / DO_{17} / EX_{25} / SI_{79})**.
- Los comentarios han de tener una **estructura uniforme** y predefinida para las clases, métodos, tipos de datos, ficheros... **(CS_{12})**.
- Los **comentarios deben diferenciarse bien del código** de forma clara y uniforme (existen multitud de técnicas, como líneas en blanco, líneas de asteriscos, estilo de letra diferente, palabras específicas en mayúsculas...) **(SD_4)**.
- Todas las **sentencias que dependan del** ***hardware*** (por ejemplo, el manejo de interrupciones) **o del entorno** ***software*** (por ejemplo, las llamadas al sistema) **han de ir comentadas**. Estos comentarios no sólo son importantes para explicar qué se está haciendo, sino también para identificar claramente las partes del programa dependientes de la máquina **(SD_6)**.
- Todas las **rutinas e instrucciones escritas en código máquina o ensamblador deben llevar comentarios** para facilitar su legibilidad sin necesidad de estudiar dichas instrucciones **(SD_7)**.
- Las sentencias o **construcciones no estándar del lenguaje deben estar bien comentadas**. Se entiende por construcciones no estándar aquéllas que son propias del compilador y no han sido incorporadas dentro del estándar del lenguaje **(SD_8)**.

- Los **comentarios deben estar escritos de tal forma que amplíen el significado del código**, aportando alguna información adicional útil. Los comentarios deben decir el porqué se hace algo o qué hace un fragmento de código, pero no limitarse a traducir el código fuente a lenguaje natural **(SD_{13})**.

II.10. DOCUMENTACIÓN

- Todo desarrollo de *software* debe ir acompañado de una serie de documentación y manuales (de usuario, técnico...). **Todos los manuales deben seguir un esquema uniforme (CS_{13})**.
- La cantidad de **documentación técnica** debe ir en correspondencia con la cantidad de código generado **(DO_1)**.
- Cada **atributo, método, clase, jerarquía de clases, tipo de datos**, etc. **debe ir acompañado de documentación técnica** completa que justifique su uso y su utilidad, explicando bien su empleo y funcionamiento **(DO_2 / DO_3 / DO_4 / DO_5 / DO_6 / MO_{12})**.
- Cada **atributo no privado, método no privado, clase, jerarquía de clases, tipo de datos**, etc. **debe ir acompañado de documentación** para sus usuarios **(DO_8 / DO_9 / DO_{10} / DO_{11} / DO_{12} / GE_{19})**.
- La documentación debe escribirse con un lenguaje sencillo, de forma que tenga una buena **legibilidad**, controlando el nivel de dificultad de lectura del texto **(DO_7 / DO_{18} / TA_6 / OP_{12} / OP_{13})**.
- Todas las posibles **ampliaciones o modificaciones deben ser comentadas** y documentadas **(EX_3)**.
- Deben incorporarse una serie de **lecciones y material de enseñanza, ejemplos, ejercicios y tutorial** para los usuarios, así como para los encargados del mantenimiento **(TA_1 / TA_2 / TA_8 / DO_{16})**.
- Debe existir **ayuda sensible al contexto, manuales y diagnósticos en línea (TA_3 / DO_{15} / DO_{19} / OP_{16} / OP_{18})**.

II.11. EFICIENCIA

- La **segmentación del código** para que utilice la menor cantidad de memoria posible, mediante mecanismos como *overlays* o librerías de enlace dinámico, favorece las necesidades de almacenamiento. Sin embargo, esto causa una disminución de la eficiencia de ejecución al tener que cargarlas y descargarlas de memoria durante la ejecución **(SE_3 / -EE_6)**.
- Todos los **datos que se almacenen** deben ser utilizados **(SE_4)**.
- El **código inútil y duplicado** consume recursos (espacio de memoria y tiempo) sin mejorar el proceso **(-SE_6 / -SE_7 / -EE_4 / -EE_7 / -SI_{22})**.
- No deben existir **datos redundantes (-SE_9)**.

- Deben utilizarse las **opciones de compilación adecuadas** para optimizar el tamaño de almacenamiento y la eficiencia de ejecución del programa **(SE_{13} / EE_{12})**.

- Los **datos u objetos globales** consumen memoria durante toda la ejecución del programa, sin que pueda ser utilizada para almacenar otra información **(-SE_8 / -SI_6)**.

- Las **expresiones deben implementarse de forma eficiente**, empleando para ello los operadores que ofrece el lenguaje adecuadamente **(EE_{11})**.

- Deben **agruparse los datos relacionados** para que su procesamiento sea más eficiente, utilizando, por ejemplo, vectores o registros **(EE_{13})**.

- Cuando hay que enviar como argumentos una **gran cantidad de datos**, resulta más eficiente pasarlos **por referencia** que por valor, ya que de esta forma se evita tener que copiar toda la información **(EE_{15})**.

- El uso abusivo de **objetos temporales** o copias intermedias de objetos penaliza la eficiencia **(-EE_{16})**.

- Las **operaciones con números reales innecesarias** afectan a la eficiencia **(-EE_{17})**.

- El uso de **métodos que devuelven un objeto** de una clase como valor de retorno implica que habrá que copiar toda la información de dicho objeto, lo que puede consumir bastante tiempo **(-EE_{22} / -ST_{31} / -SI_{64})**; conviene devolverlos como referencia.

- El sistema deberá ser desarrollado sobre un **lenguaje para el que exista compilador**. La utilización de un intérprete afectará a su eficiencia. Los lenguajes orientados a objetos modernos disponen de compilador, aunque originalmente muchos de los lenguajes empleaban un intérprete (Smalltalk, CLOS, Java...) **(EE_{14})**.

- La eficiencia de ejecución puede verse afectada por la resolución de métodos en tiempo de ejecución (**enlace dinámico**) para implementar las operaciones polimórficas, que consisten en enlazar una operación sobre un objeto con un método específico. Esto podría requerir una búsqueda, en tiempo de ejecución, por el árbol de herencia para encontrar la clase que implementa la operación para dicho objeto. Sin embargo, la mayoría de los lenguajes optimizan este mecanismo de búsqueda para mejorar su eficiencia. Como la estructura de las clases permanece invariable durante la ejecución, el método correcto para cada operación puede almacenarse localmente en una subclase. Con esta técnica, conocida como ***method caching***, el enlace dinámico se reduce a una búsqueda en una tabla *hash*, que se ejecuta en un tiempo siempre fijo, independientemente de la profundidad del árbol de herencia o del número de métodos de la clase **(-EE_8)**.

- Las **uniones** de C++ pueden disminuir las necesidades de almacenamiento **(SE_2)**.

- La utilización de **operaciones con bits** reduce el espacio de almacenamiento necesario para albergar los operandos y el resultado a costa de complicar el código **(SE_{11} / -SI_{31})**.

- Las **estructuras de bits** permiten disminuir el espacio de almacenamiento necesario **(SE_{10})**.

II.12. PRUEBAS

- Deben cubrirse el máximo número de **posibilidades de ejecución** durante las pruebas modulares **(IN_1)**.
- Deben probarse los **valores límites** de los parámetros de entrada **(IN_2)**.
- Debe incorporarse **código específico** para ayudar a la realización de pruebas **(IN_3)**.
- Una de las pruebas de integración consiste en **probar todas las interfaces** de las clases **(IN_4)**.
- Uno de los aspectos de las pruebas de integración comprende verificar que **los requisitos de ejecución** se han cumplido **(IN_5)**.
- Deben utilizarse **escenarios** (o un mecanismo equivalente) para probar los distintos módulos del sistema **(IN_6)**.
- Debe incorporarse la posibilidad de **resumir las entradas y las salidas** proporcionadas al realizar las pruebas. Los resultados de las pruebas y la manera en que se presentan estos resultados son muy importantes para la efectividad de las pruebas. Esto es especialmente útil durante las pruebas del sistema por el potencialmente gran volumen de datos de entrada y salida **(IN_7)**.
- El sistema ha de posibilitar la realización de un **registro de las operaciones realizadas** por el usuario **(IN_9)**, así como permitir **medirlas y resumirlas**, almacenándolas en un fichero o en un listado, con el fin de analizar las más frecuentes para intentar optimizarlas o mejorarlas en la medida de lo posible **(IN_{10} / OP_6)**.

II.13. INTERACCIÓN CON EL USUARIO

- Deben estar **descritos detalladamente** todos **los pasos necesarios para poner en marcha** y hacer funcionar adecuadamente al sistema, ya que esto permite que el usuario lo utilice con mayor facilidad, con lo que se mejora la operatividad **(OP_1)**.
- Debe existir la **posibilidad de que el usuario pueda interrumpir el proceso** para continuar más tarde **(OP_3)**.
- El número de **intervenciones requeridas del usuario** debe estar limitado **(OP_4)**.
- Debe existir una **descripción exhaustiva de cómo empezar y terminar cada una de las operaciones (OP_5)**.
- El esfuerzo requerido para **aprender a manejar el sistema** y a interpretar o analizar los resultados no debe ser muy elevado **(OP_{10})**.
- La **entrada y salida de datos deben ser sencillas**, generales, independientes del dispositivo y estar bien documentadas **(OP_{11})**.
- **Influencia de los errores en el entorno**. Si un error causa una caída del sistema, siendo necesario un reinicio, se dificultará el uso del *software* **(OP_{15} / ET_7)**.

- El sistema debe permitir que el **usuario pueda configurar distintas opciones** de funcionamiento y de interfaz según sus gustos y necesidades **(OP_{17})**.
- Debe incluirse la posibilidad de que al producirse un error, el usuario **pueda solucionar sobre la marcha el error producido** y, a continuación, proseguir con la ejecución del proceso, si lo desea **(ET_2)**.
- Deben preverse mecanismos para la **recuperación ante errores o fallos** del *hardware* o de los dispositivos **(ET_{17} / ET_{18} / ET_{19})**.
- Las clases deben **haber previsto la posibilidad de que ocurran errores** y deben ser capaces de manejarlos adecuadamente **(ET_{22})**.
- Deben realizarse **previsiones para almacenar e informar de los accesos al sistema**: el *software* debe disponer de mecanismos que permitan conocer quién, cuándo, dónde y cómo se accede al sistema **(AA_1 / AA_2)**.
- Deben realizarse **previsiones para avisar inmediatamente de los accesos no autorizados**: el sistema debe disponer de mecanismos que permitan informar quién, cuándo, dónde y cómo se accede al sistema sin disponer de autorización **(AA_3)**.
- Debe realizarse un **control de acceso a los usuarios** para evitar accesos no autorizados al sistema o a las bases de datos **(AC_1 / AC_2)**. Ante **accesos no autorizados** (o intentos reiterados) el sistema debe **responder con acciones** como alertas visuales o sonoras, bloqueos del terminal, avisos a los administradores... **(AC_3)**.
- Dependiendo de los **privilegios de acceso** del usuario el sistema limitará su actividad, restringiendo su acceso sólo a los lugares o a las acciones autorizadas **(AC_4)**.
- Deben definirse **valores por omisión en los datos de entrada** del usuario, puesto que es una forma de minimizar la cantidad de información requerida para su introducción **(CO_1)**.
- Deben definirse **formatos uniformes para la introducción de los mismos tipos de datos**, puesto que cuanto mayor sea el número de formatos diferentes, será más difícil de utilizar el sistema **(CO_2)**.
- Los **registros de entrada deben estar claramente identificados (CO_3)**.
- Las **entradas de datos tienen que poder ser verificadas por el usuario** antes de ser procesadas (por ejemplo, mostrando un eco de la entrada o presentándolas en pantalla bajo petición) **(CO_4)**.
- Los **registros de entrada** deben finalizar con una instrucción de fin de entrada explícita **(CO_5)**.
- Debe existir una **previsión para realizar las entradas de información** al sistema desde diferentes dispositivos, lo cual proporciona una gran flexibilidad en la entrada de datos **(CO_6)**.

- Deben proporcionarse **controles selectivos** (salidas específicas, formatos, precisión...) para la salida de información **(CO_7)**.
- Deben poderse **identificar perfectamente cada una de las salidas** o resultados proporcionados por el sistema **(CO_8)**.
- Deben definirse **formatos uniformes para las salidas** de datos del mismo tipo **(CO_9)**.
- Las **salidas deben estar agrupadas**, permitiendo diferenciar distintos tipos de información (por líneas en blanco, líneas de asteriscos, distintos colores...) **(CO_{10})**.
- Debe existir una previsión para poder **enviar las salidas del sistema a diferentes dispositivos (CO_{12})**.
- La **interfaz de usuario** debe permitir a la persona **operar con el sistema de forma ergonómica y confortable**, siguiendo una serie de normas de uso estándar. Debe ser sencilla, atractiva y fácil de utilizar **(CO_{13} / CO_{14} / TA_7 / OP_8)**.
- La **documentación de usuario** debe incluir una lista de todos los **mensajes y errores con sus causas y sus posibles soluciones** debidamente expuestas **(DO_{13} / TA_5)**.
- Uno de los componentes de la documentación debe ser un **completo manual de usuario (DO_{14} / TA_4 / OP_9)**. Además, es conveniente incorporar una serie de **lecciones y material de enseñanza** para los usuarios, así como para los encargados del mantenimiento **(TA_1)**.
- Es conveniente que el *software* vaya acompañado de un **tutorial de manejo, ejercicios y ejemplos de funcionamiento** realistas **(DO_{16} / TA_2 / TA_8 / OP_{14})**.
- El *software* debe ir acompañado de un sistema de **ayuda interactivo** sensible al contexto, **manuales** e **información de diagnóstico en línea (DO_{15} / DO_{19} / TA_3 / TA_9 / TA_{10} / OP_{16} / OP_{18})**.
- Cada módulo debe incluir una **descripción de las entradas, las salidas, el proceso y sus limitaciones (SI_5)**.
- Los **mensajes de error y avisos al usuario deben ser claros y no ambiguos**. Su redacción debe cumplir una serie de normas generales para facilitar la comunicación usuario-máquina: relacionados con el error detectado, expresados en términos que el usuario pueda comprender (no en terminología del programador), específicos, localizados, completos, legibles, amables... **(CO_{11} / IN_{12} / OP_2)**.
- El **diálogo con el usuario** debe realizarse de manera estándar y uniforme, siguiendo las normas proporcionadas bien por el Sistema Operativo (Macintosh, MS-Windows, X-Windows...), bien por las organizaciones que desarrollan o usarán el sistema **(OP_7)**.
- Todos los errores que se puedan producir deberán ir asociados a un **mensaje de error**, así como disponer de una explicación adicional para aumentar la información y aclarar al máximo la situación al usuario **(CO_{15} / IN_{13})**.

Teoría de De Marco de la Emigración de los Costos:
Los costos emigrarán de la actividad que sea medida más cuidadosamente que sus actividades vecinas.

— Tom De Marco

[De Marco, 82]

ANEXO III. HERRAMIENTAS

III. HERRAMIENTAS

III.1. INTRODUCCIÓN

El modelo de calidad descrito en los capítulos anteriores tendría simplemente una utilidad teórica si no pudiera aplicarse en la práctica. Debido a la gran cantidad de medidas existentes en el modelo, es necesario usar utilidades informáticas para facilitar su empleo. En el presente anexo, en primer lugar, se resume una herramienta para evaluar la calidad de programas escritos en C++, en segundo lugar, se describe concisamente un programa para obtener los pesos según el Método Analítico Jerárquico y, en último lugar, se muestra el funcionamiento de una pequeña utilidad que permite consultar y buscar las medidas de calidad según unos determinados patrones.

III.2. HERRAMIENTA PARA LA EVALUACIÓN DE LA CALIDAD

Dado el gran número de medidas que posee el modelo de calidad, resulta imprescindible realizar su automatización mediante alguna herramienta informática. En este apartado se expone sucintamente el diseño y funcionamiento de un prototipo de dicha herramienta que permite evaluar la calidad de un programa con una considerable rapidez. Para simplificar, la herramienta es capaz de analizar únicamente programas escritos en C++, aunque su ampliación para manejar otros lenguajes de programación orientados a objetos no sería complicada si bien llevaría un tiempo considerable para su implementación. Evidentemente, en el mercado existen algunas herramientas de este tipo, como las descritas en [Lee, 01] [Littlefair, 01] [Parasoft, 01] [McCabe, 01] [Meyers, 00] [Cain, 00] [Kemerer, 99] [Dumke, 99] y [Bansiya, 97], pero es necesario desarrollar una nueva para incluir las nuevas medidas diseñadas en este modelo de calidad.

III.2.1. DISEÑO E IMPLEMENTACIÓN DE LA HERRAMIENTA

La herramienta realizada se basa en las habituales técnicas de construcción de compiladores [Aho, 90]. Concretamente, solamente ha sido necesario incorporar la fase de análisis, que incluye el Análisis Léxico, Análisis Sintáctico y Análisis Semántico, además del módulo de Tabla de Símbolos, pudiendo prescindir de la fase de síntesis propiamente dicha. La Figura III.1 muestra un diagrama modular de esta fase de la herramienta.

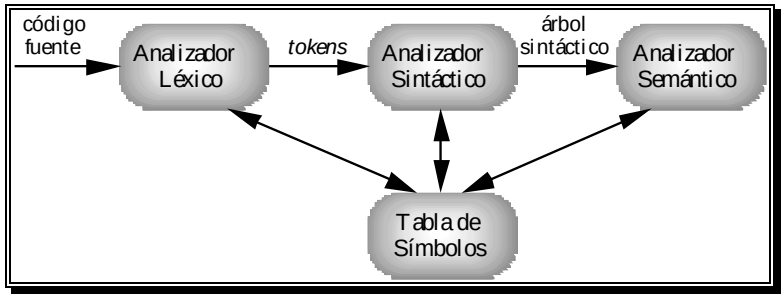


Figura III.1: Módulos de la fase de análisis

Resumidamente, a continuación se muestra la función y el proceso seguido para la construcción de estos módulos [Crespo, 03]:

- Análisis Léxico. Su principal tarea consiste en leer de izquierda a derecha la cadena de caracteres que forma el programa fuente, agrupándolos en *tokens* (secuencias elementales de caracteres con significado propio). Para la construcción del analizador léxico, ha sido necesario diseñar una gramática regular que represente todos los elementos del lenguaje C++, construir un autómata finito determinista e incorporarle una serie de acciones semánticas que complementen su funcionamiento. En este módulo se han incorporado también todas las rutinas necesarias para manejar las directivas del preprocesador de C++.

- Análisis Sintáctico. Este módulo toma como entrada los *tokens* enviados por el analizador léxico y los agrupa formando una serie de sentencias gramaticales, que pueden representarse en forma de árbol (árbol de análisis sintáctico). Para la construcción de este módulo es necesario partir de una gramática de contexto libre para, a continuación, llevar a cabo el analizador propiamente dicho con alguna de las numerosas técnicas de análisis sintáctico existentes. En primer lugar, se intentó construir una gramática LALR(1) con el fin de emplear la herramienta YACC, pero la complejidad inherente a las estructuras de C++ hizo imposible encontrar una gramática completa de este tipo que estuviera libre de conflictos. Por ello, como camino alternativo, se optó por la construcción de una analizador sintáctico descendente predictivo recursivo, para lo cual tuvo que obtenerse una gramática LL(1). Una vez diseñada la gramática, se implementó el analizador sintáctico sin problemas.

- Análisis Semántico. Este módulo toma como entrada el árbol de análisis sintáctico para revisarlo con el fin de comprobar que la semántica resulta adecuada, así como para reunir información acerca de los tipos de datos que intervienen en el programa y que se almacena en una tabla de símbolos. La realización de este módulo ha consistido en el diseño de un esquema de traducción y su incorporación sobre el código del analizador sintáctico.

- Tabla de Símbolos. El presente módulo consta de una estructura de datos que contiene un registro por cada identificador, estando compuesto cada registro por una serie de campos que almacenan los atributos de cada uno de los identificadores que aparecen en el programa fuente. En la herramienta construida se ha tenido que prestar una especial atención a la estructura y organización de la tabla de símbolos para que reflejara a la perfección todos los identificadores presentes, con el fin de poder extraer, posteriormente, la información necesaria para evaluar las medidas.

Una vez que estuvo diseñada y construida la parte del análisis del código fuente, el siguiente paso fue diseñar e implementar los mecanismos necesarios para incluir el cálculo de las medidas y la evaluación de la calidad. El proceso seguido fue el siguiente [Río, 03]:

- Se clasificaron las medidas en automatizables (aquéllas que se podían calcular automáticamente a partir del programa fuente) y en no automatizables (aquéllas que no se podían extraer fácilmente del código fuente). Algunos ejemplos de las medidas automatizables son: MO_3 (hay que contar el número de métodos de cada clase) y SI_{18} (hay que contar el nivel de anidamiento de los bloques en un método). Algunos ejemplos de medidas no automatizables son: EE_2 (hay que decir si los requisitos de ejecución han sido tenidos en cuenta) y SD_{16} (hay que indicar qué proporción de identificadores del programa fuente tienen nombres descriptivos).

- Las medidas automatizables se clasificaron a su vez según el módulo donde debían ser calculadas. De esta forma, algunas medidas se podían calcular en el analizador léxico (SI_{20}: número de sentencias `goto`), otras en el sintáctico (SD_{24}: número de predicados), otras en el semántico (CM_9: número de variables no usadas) y otras en la tabla de símbolos (MO_4: número de atributos por clase). No obstante, muchas medidas necesitan ser calculadas a partir de información proveniente de más de uno de los módulos.

- Las medidas no automatizables se implementaron como preguntas directas al usuario.

- Para almacenar todos los datos de las medidas, se construyó una estructura de datos que representa el árbol de calidad completo, y cuyos nodos se van completando conforme se va obteniendo la información. También se utiliza la estructura de la tabla de símbolos para ir anotando la información parcial que se va logrando.

- Por último, se ha diseñado una interfaz de usuario atractiva, que le permite utilizar la herramienta para calcular la calidad total de los programas, posibilitando también el análisis de los valores de calidad obtenidos para cada uno de los factores, para cada uno de los criterios dentro de cada factor e, incluso, para cada una de las medidas dentro de cada criterio. Además, si se desea profundizar en los datos obtenidos para las medidas, el sistema es capaz de proporcionar información detallada acerca de cada una de las medidas para cada uno de los elementos para los que se puede aplicar. De esta forma, se puede realizar una búsqueda dirigida de los defectos del programa.

Para terminar este apartado, solamente queda comentar que el prototipo de la herramienta se ha implementado en C++ sobre MS-Windows, empleando el Borland C++ Builder como entorno de desarrollo.

III.2.2. EJEMPLO DE USO

Seguidamente, para clarificar el funcionamiento de la herramienta ante un programa, se van a resumir los pasos que hay que llevar a cabo.

El primer paso consiste en seleccionar la fase del ciclo de vida a estudiar. Si se escoge la fase de implementación (las fases anteriores no usan ficheros fuente), hay que indicar el directorio y el fichero o ficheros que forman parte del programa (Figura III.2).

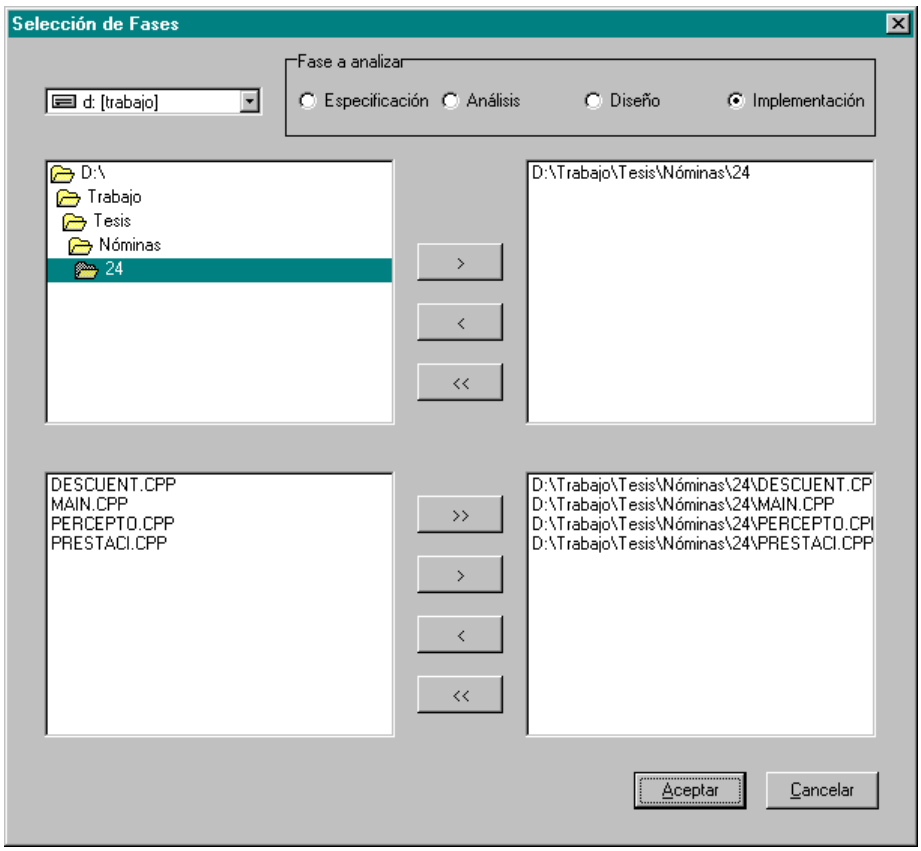


Figura III.2: Ventana de selección de los ficheros fuente del sistema a analizar

Seguidamente, la herramienta planteará una sucesión de preguntas para calcular las medidas no automatizables (Figura III.3). El usuario podrá contestar las que desee y avanzar y retroceder por la lista de preguntas.

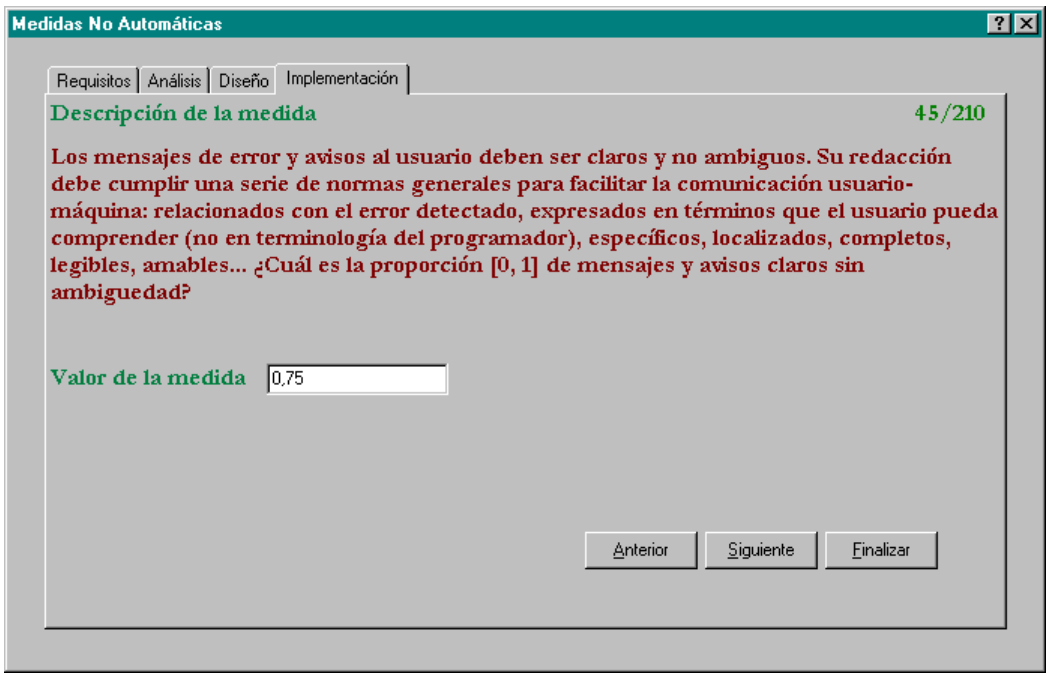


Figura III.3: Ventana con las preguntas para las medidas no automatizables

Una vez que se ha dado por finalizada la fase de preguntas y tras unos minutos (dependiendo del volumen del programa a evaluar), la herramienta proporciona los resultados obtenidos. La interfaz presenta dos zonas claramente diferenciadas (Figura III.4):

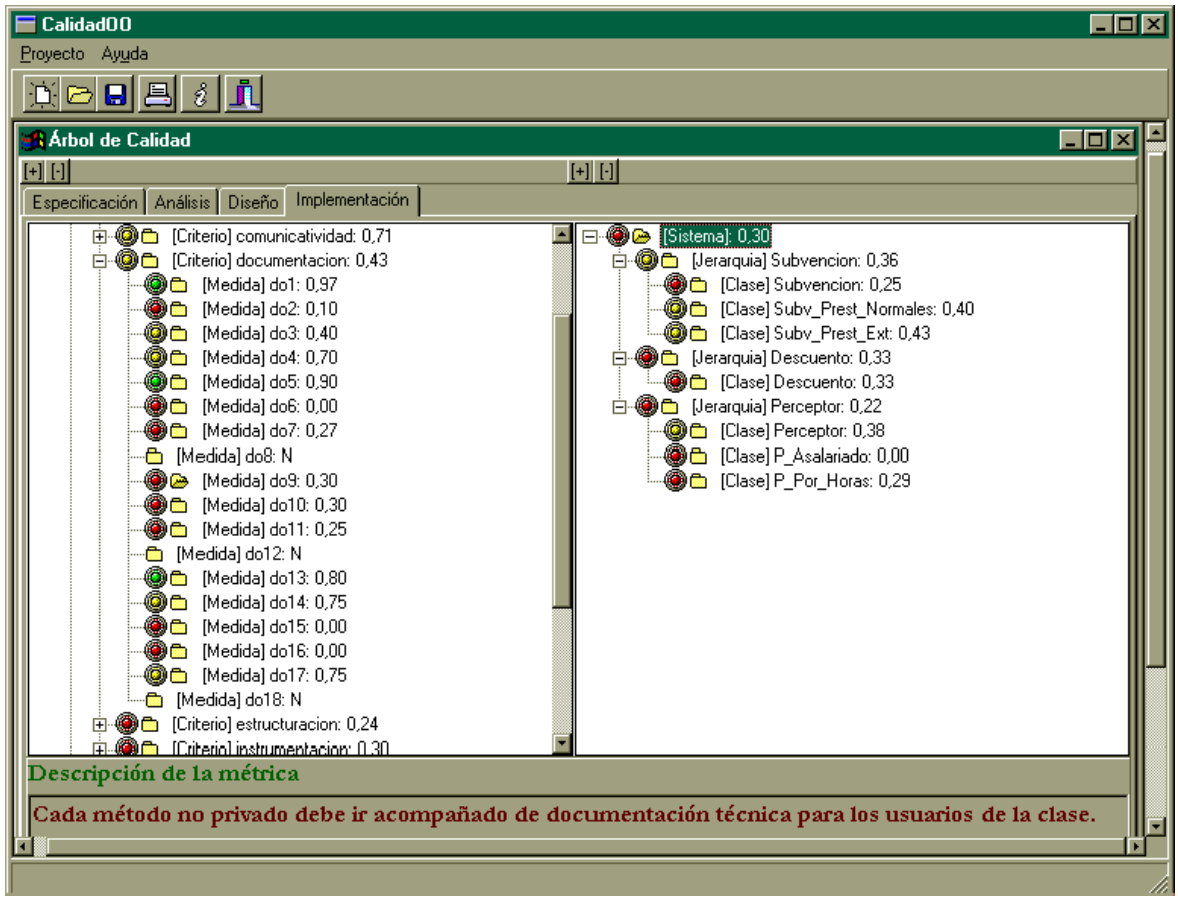


Figura III.4: Ventana con los resultados de la evaluación

- A la izquierda se muestra el árbol de calidad. En un principio se ve únicamente la calidad total, pero el usuario puede expandir los nodos con el fin de ver el vector de resultados obtenidos por los factores, por los criterios o, incluso, por las medidas. En el caso de la figura, se muestra expandido el nodo correspondiente a la documentación.

- A la derecha se muestra el árbol de las medidas. Cuando en el árbol de calidad se tiene seleccionada una medida, en esta zona se muestra el árbol correspondiente al cálculo de dicha medida, indicando los valores que se obtienen para el sistema, por cada jerarquía, por cada clase, por cada método o por cada atributo (cuando sean aplicables dependiendo de la medida). En el caso de la figura, se muestra el árbol correspondiente a la medida DO_9.

En ambos árboles, los nodos se muestran con unos indicadores de colores con el fin de guiar visualmente al usuario en la labor de análisis de los datos. Los colores tienen el siguiente significado: verde indica un buen valor, amarillo indica un valor sospechoso y rojo indica un valor malo. Por tanto, el usuario debería buscar los nodos rojos (e, incluso,

los amarillos) para localizar finalmente en el árbol de medidas las clases, métodos, etc. susceptibles de ser mejorados, tal como se ha hecho en el ejemplo de la figura.

III.2.3. FUTURAS AMPLIACIONES

Está previsto incluir una serie de comandos para ayudar a realizar el análisis de los datos. Uno de los comandos permitirá buscar aquellos valores malos de calidad, con el fin de poderlos localizar rápidamente y buscar posibles soluciones. Otro de los comandos que se han pensado posibilitará comparar los resultados obtenidos en las distintas fases del ciclo de vida por el modelo, con el fin de determinar qué factores o qué criterios ha visto descender significativamente su valor.

Uno de los aspectos interesantes del modelo de calidad descrito en este trabajo lo constituye el hecho de que el ingeniero del *software* pueda adaptarlo a las necesidades del sistema en desarrollo.

La herramienta se ha diseñado de tal forma que la modificación de las fórmulas de las medidas puede realizarse fácilmente sin necesidad de modificar el código fuente y, por tanto, sin volver a compilar. Para ello, las fórmulas de las medidas se han introducido en un fichero de configuración que es interpretado por la herramienta cada vez que se evalúa la calidad de un sistema. Además, no sólo se pueden modificar las fórmulas de las medidas, sino que también es posible cambiar las ramas del árbol de calidad. Es decir, el ingeniero del *software* puede eliminar o incluir medidas en un criterio, puede eliminar o incluir criterios en un factor y puede eliminar o incluir factores de calidad. Siempre que se utilicen las medidas, criterios y factores definidos en el modelo, estos cambios también se podrán realizar sin necesidad de recompilar el código de la herramienta, puesto que el árbol de calidad se ha incorporado en otro fichero de configuración fácilmente legible. La Figura III.5 muestra sendos fragmentos de dichos ficheros de configuración que, como puede verse, resultan totalmente legibles y fáciles de modificar.

Fichero Formulas.ini

```
[cm16]
NumeroOperandos=2
Op1=cm16n
Op2=cm16t
Formula=log ( cm16t – cm16n ) cm16t

[cm17]
NumeroOperandos=1
Op1=cm17n
Formula=1 / ( cm17n )
```

Fichero Arbol.ini

```
[correccion]
NumeroCriterios=4
Criterio1=completitud
Criterio2=consistencia
Criterio3=documentacion
Criterio4=seguimiento

[eficiencia]
NumeroCriterios=3
Criterio1=concision
Criterio2=eficienciaAlmacenamiento
Criterio3=eficienciaEjecucion
```

Figura III.5: Listado parcial de dos de los ficheros de configuración con las medidas y el árbol

También está previsto incluir un mecanismo que permita la incorporación de nuevas medidas, de nuevo sin recompilar la herramienta. Para ello, las nuevas medidas deberán estar implementadas en una librería dinámica (DLL) con una interfaz determinada. Después, solo será necesario incluir el nombre de la librería y los nombres de las funciones que realizan las medidas en uno de los ficheros de configuración para que la herramienta realice la llamada cuando se necesite el valor de las nuevas medidas.

De hecho, este es el mecanismo que se va a usar para incorporar las medidas del modelo obtenidas a partir de las medidas de Halstead y que fueron implementadas aparte por mayor simplicidad [Ramírez, 03].

Se está realizando la adaptación de esta herramienta para que sea capaz de evaluar programas realizados en Java [Canorea, 03], dada la importancia que está adquiriendo este lenguaje para el desarrollo de sistemas independientes de la plataforma. Una vez finalizado, con los dos sistemas se podría realizar un interesante estudio comparativo sobre sistemas desarrollados sobre los dos lenguajes, al estilo del análisis realizado en [Mayrand, 00].

Por último, y como complemento a esta herramienta, se está construyendo otra herramienta para el desarrollo, de manera gráfica, de sistemas orientados a objetos en Java [Rovira, 03]. El objetivo consiste en que el usuario pueda realizar gráficamente el diseño del sistema, dibujando las clases, los distintos tipos de relaciones entre clases y definiendo sus atributos y métodos. La aplicación transformará esta entrada en código Java que el usuario podrá modificar y completar con el código de los métodos, reflejándose los cambios en la interfaz gráfica. La herramienta incluirá un compilador de Java que permitirá compilar directamente el código generado. El modelo de calidad estará incorporado en la herramienta, de tal forma que el usuario podrá consultar y analizar la calidad del sistema en construcción durante las fases de diseño e implementación.

III.3. HERRAMIENTA PARA LA OBTENCIÓN DE PESOS

El cálculo de los pesos para un conjunto de elementos puede resultar un trabajo bastante largo si no se dispone de alguna herramienta informática que ayude en el proceso.

Una de las técnicas que se puede utilizar es el Método de Análisis Jerárquico [Saaty, 90], como ya se ha explicado en el apartado 4.4.1.

Con este fin, se ha desarrollado un pequeño programa en C++ Builder (de Borland) que realiza una encuesta al usuario, comparando dos a dos los elementos dados. Una vez finalizado el proceso, el programa proporciona la matriz de relevancia, junto con el autovector que define los pesos de cada uno de estos elementos.

La Figura III.6 muestra una de las preguntas realizadas al usuario para el ejemplo descrito en el apartado 4.4.1. En este caso, se están intentando obtener los pesos de los criterios correspondientes al factor de corrección.

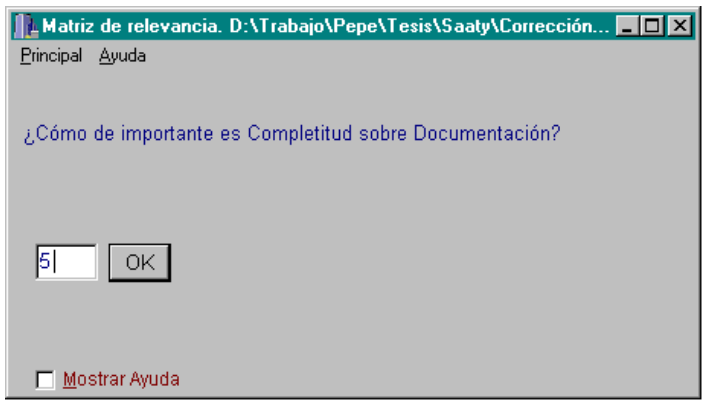


Figura III.6: Ventana con la pregunta que compara la completitud con la documentación

La Figura III.7 presenta la matriz de relevancia obtenida una vez que se han finalizado todas las preguntas. A la derecha de la matriz, se presentan los pesos (el autovector) de cada uno de los criterios de acuerdo a las respuestas proporcionadas por el usuario.

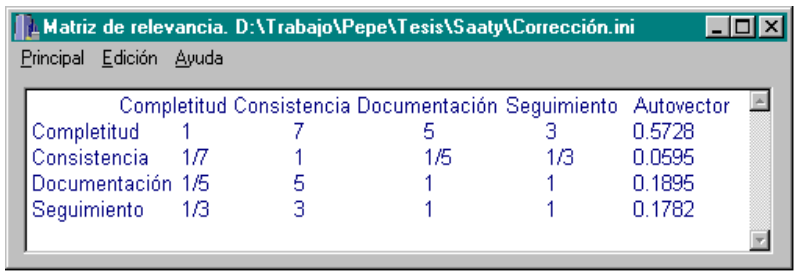


Figura III.7: Ventana con la matriz de relevancia y los pesos definidos por el autovector

Esta matriz y el autovector se pueden almacenar en un fichero de texto para un uso posterior. Así mismo, el sistema puede proporcionar, en todo momento, una pequeña ayuda para indicar al usuario cómo debe responder a las preguntas planteadas y el significado de los valores a introducir.

III.4. HERRAMIENTA PARA LA CONSULTA DE LAS MEDIDAS DE CALIDAD

Como el número de medidas del modelo de calidad es considerablemente elevado, se ha diseñado una pequeña herramienta que incorpora una base de datos con las medidas y permite realizar consultas y búsquedas de determinadas medidas.

Su funcionamiento se ha simplificado al máximo para facilitar su uso. Nada más ejecutar la herramienta, se muestra la ventana principal (Figura III.8). Esta ventana está compuesta por dos grupos de opciones. El primero presenta las cuatro fases del ciclo de vida (requisitos, análisis, diseño e implementación). El segundo grupo presenta los elementos sobre los que se pueden aplicar medidas (atributo, método, clase, jerarquía y sistema). El usuario puede actuar de tres formas con cada componente:

Figura III.8: Ventana principal: selección de los criterios de búsqueda de medidas

1. Seleccionarlo, de tal forma que se muestre una marca en la casilla de selección. En este caso, se buscarán todas aquellas medidas que presenten dicho componente.
2. No seleccionarlo, de tal forma que se muestre la casilla de selección en blanco. En este caso, se buscarán todas aquellas medidas que no presenten dicho componente.
3. Indiferente, de tal forma que se muestre la casilla de selección en gris. En este caso, el componente no influye en la búsqueda, por lo que se buscarán todas aquellas medidas que presenten o no presenten dicho componente.

En el ejemplo de la Figura III.8, se buscarán aquellas medidas que no se usen en la fase de requisitos ni en la de análisis, pero que se usen en la de implementación y que se apliquen sobre los métodos y sobre el sistema, pero no sobre los atributos. Como se ve, la respuesta será independiente de si la medida se usa durante el diseño o de si es aplicable sobre clases o jerarquías.

Seguidamente, una vez seleccionadas las fases y los elementos deseados, se podrá pulsar el botón "Buscar", con lo que se pasará a la ventana de consulta de la base de datos. Evidentemente, para finalizar la ejecución de la herramienta, deberá pulsarse el botón "Salir".

La ventana de consulta de la base de datos, presenta toda la información acerca de las medidas que cumplan los criterios de búsqueda seleccionados. La Figura III.9 muestra un ejemplo de este tipo de ventana, donde cada vez se muestra toda la información necesaria acerca de una medida, y cuyos componentes se describen a continuación.

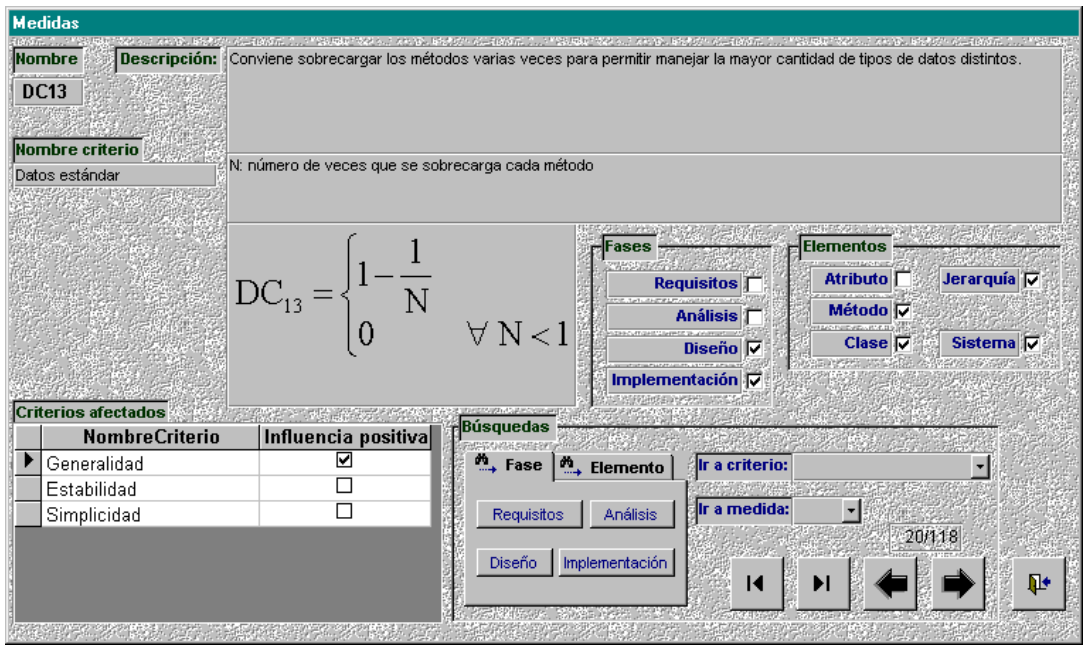


Figura III.9: Ventana con la información de cada medida

1. En primer lugar, se muestra el nombre de la medida, de acuerdo a la codificación utilizada en el capítulo 4, junto al nombre del criterio al que pertenece.
2. Seguidamente se muestra un texto con la descripción de la medida y la información necesaria para poder evaluarla, es decir, la fórmula con sus variables, o, si la medida carece de fórmula, el significado de sus posibles valores.
3. A continuación, se muestran las fases en las que se debe evaluar la medida y los elementos sobre los que se debe aplicar.
4. Si la medida afecta a otros criterios, se muestra una lista de éstos, indicando si la influencia es positiva o negativa.
5. En la parte inferior derecha, se dispone de una nueva zona de búsquedas que permite obtener la siguiente medida de una fase o de un elemento, ir a la primera medida de un criterio determinado o, incluso, ir a una medida concreta por su nombre. También se encuentran botones para avanzar a la siguiente medida o retroceder a la anterior, así como para ir a la primera o a la última. Es necesario señalar, que estas opciones de búsqueda se limitan a las medidas que cumplan los criterios de búsqueda seleccionados anteriormente.
6. Por último, se encuentra el botón de salida que permite regresar a la ventana principal.