ELABORACIÓN DEL SISTEMA DE INFORMACIÓN PARA LA MATRICULA, REGISTRO Y CONTROL DE NOTAS EN EL CENTRO EDUCATIVO DEL QUIROGA.

OSCAR ANDRÉS CALDERÓN ALONSO
DIEGO FERNANDO SÁNCHEZ MEDINA

UNIVERSIDAD LIBRE
FACULTAD DE INGENIERÍA
BOGOTÁ
2005

ELABORACIÓN DEL SISTEMA DE INFORMACIÓN PARA LA MATRICULA, REGISTRO Y CONTROL DE NOTAS EN EL CENTRO EDUCATIVO DEL QUIROGA.

NÚMERO DEL PROYECTO: 339-466

OSCAR ANDRÉS CALDERÓN ALONSO
DIEGO FERNANDO SÁNCHEZ MEDINA

PROYECTO DE GRADO PRESENTADO COMO PRERREQUISITO PARA OBTENER EL TITULO DE INGENIERO DE SISTEMAS E INFORMATICA

DIRECTOR:

SONIA SANMIGUEL

UNIVERSIDAD LIBRE
FACULTAD DE INGENIERÍA
BOGOTÁ - 2005

AGRADECIMIENTOS

Los autores expresan sus agradecimientos:

A la ingeniera Sonia Sanmiguel, que muy amablemente acepto ser la directora de nuestro proyecto y nos colaboro de manera desinteresada y desmedida, brindándonos asesoría y compartiendo todo su conocimiento con nosotros.

A los Ingenieros Jorge Ojeda y Juan Fernando Velásquez, evaluadores de nuestro proyecto, que hicieron las correcciones pertinentes y fueron pieza fundamental en la construcción del mismo.

Por ultimo, a nuestras familias que fueron nuestra fuente de apoyo y depositaron toda su confianza en nosotros.

TABLA DE CONTENIDO

1 TITULO ....- 9 -
2 PROBLEMA ....- 10 -
3 OBJETIVOS ....- 11 -
3.1 Objetivo general: ....- 11 -
3.2 Objetivos específicos: ....- 11 -
4 JUSTIFICACIÓN ....- 12 -
4.1 Para el centro educativo: ....- 12 -
4.2 Para los padres: ....- 13 -
4.3 Para el cuerpo docente y directivas: ....- 13 -
5 DELIMITACIÓN ....- 14 -
5.1 Delimitación en relación con el espacio: ....- 14 -
5.2 Delimitación en relación con el tiempo ....- 14 -
5.3 Delimitación en relación a la temática ....- 14 -
6 MARCO TEÓRICO ....- 15 -
6.1 PROGRAMACIÓN ORIENTADA A OBJETOS ....- 15 -
6.1.1 Componentes de un Objeto ....- 15 -
6.1.1.1 Relaciones ....- 15 -
6.1.1.2 Propiedades ....- 15 -
6.1.1.3 Métodos ....- 16 -
6.1.2 Organización de los objetos ....- 16 -
6.1.2.1 La raíz de la jerarquía ....- 16 -
6.1.2.2 Los objetos intermedios ....- 16 -
6.1.2.3 Los objetos terminales ....- 16 -
6.1.3 Estructura De Un Objeto ....- 17 -
6.1.3.1 Relaciones ....- 17 -
6.1.3.1.1 Relaciones jerárquicas ....- 17 -
6.1.3.1.2 Relaciones semánticas ....- 17 -
6.1.3.2 Propiedades ....- 17 -
6.1.3.3 Métodos ....- 18 -
6.2 BASE DE DATOS ....- 18 -
6.2.1 Primera forma normal (1FN): ....- 19 -
6.2.2 Segunda forma normal (2FN): ....- 19 -
6.2.3 Tercera Forma Normal (3FN): ....- 19 -
6.3 UML (Unified Modeling Language) ....- 20 -
7 MARCO CONCEPTUAL ....- 22 -
7.1 Concepto de CALIDAD ....- 22 -
7.2 Diagrama Caso de Uso ....- 22 -
7.2.1 Elementos ....- 22 -
7.2.1.1 Actor: ....- 22 -
7.2.1.2 Caso de Uso: ....- 23 -
7.2.1.3 Relaciones: ....- 23 -
7.2.1.3.1 Asociación → ....- 23 -

7.2.1.3.2 Dependencia o Instanciación .... - 23 -
7.2.1.3.3 Generalización .... - 23 -
7.3 Modelo de clases .... - 23 -
7.3.1 Elementos .... - 23 -
7.3.1.1 Clase .... - 23 -
7.3.1.2 Atributos: .... - 24 -
7.3.1.2.1 public (+,): .... - 24 -
7.3.1.2.2 private (-,): .... - 24 -
7.3.1.2.3 protected (#,): .... - 24 -
7.3.1.3 Métodos: .... - 25 -
7.3.1.3.1 public (+,): .... - 25 -
7.3.1.3.2 private (-,): .... - 25 -
7.3.1.3.3 protected (#,): .... - 25 -
7.3.1.4 Relaciones entre Clases .... - 25 -
7.3.1.5 Herencia (Especialización/Generalización): .... - 25 -
7.3.1.6 Agregación: .... - 26 -
7.3.1.7 Asociación: .... - 26 -
7.3.1.8 Dependencia o Instanciación (uso): .... - 26 -
8 MARCO METODOLOGICO .... - 28 -
8.1 conceptualización .... - 28 -
8.1.1 Visión General .... - 29 -
8.2 Fases metodológicas .... - 29 -
8.2.1 Planificación y Especificación de Requisitos: .... - 29 -
8.2.1.1 Identificación de Conceptos .... - 29 -
8.2.1.2 Fase de Planificación y Especificación de Requisitos .... - 29 -
8.2.1.2.1 Actividades .... - 29 -
8.2.1.3 Identificación de los Límites del Sistema .... - 30 -
8.2.1.4 Modelo Conceptual .... - 30 -
8.2.1.5 Creación del Modelo Conceptual .... - 30 -
8.2.2 Construcción: .... - 31 -
8.2.2.1 Diseño de Alto Nivel: .... - 31 -
8.2.2.1.1 Actividades .... - 31 -
8.2.2.2 Diseño de Bajo Nivel: .... - 31 -
8.2.2.2.1 Actividades .... - 32 -
8.2.3 Fase de Implementación y pruebas: .... - 32 -
8.2.3.1 Pruebas: .... - 32 -
8.2.3.2 Implantación: .... - 32 -
9 MARCO LEGAL .... - 33 -
9.1 Norma ISO 9000 .... - 33 -

9.2 NORMA ISO 9000-3 - 34 -
9.2.1 Generalidades - 34 -
9.2.1.1 Título - 34 -
9.2.1.2 Naturaleza - 34 -
9.2.1.3 Ámbito - 35 -
9.2.1.4 Campo de aplicación y alcance - 35 -
9.2.1.5 Estructura - 35 -
9.2.2 Análisis - 36 -
9.2.3 Normas aplicables dentro del proyecto - 36 -
9.2.3.1 Identificación y trazabilidad del producto - 36 -
9.2.3.2 Inspección y pruebas - 36 -
9.2.3.3 Estado de Inspección y pruebas: - 37 -
9.2.3.4 Control de registros de calidad - 37 -
9.2.3.5 Capacitación - 37 -
10 CONCEPTOS Y PRINCIPIOS DEL ANALISIS - 38 -
10.1 Fase de Planificación y Especificación de Requisitos - 38 -
10.1.1 Análisis de requerimientos - 38 -
10.1.2 IDENTIFICACIÓN DE REQUISITOS PARA EL SOFTWARE - 40 -
10.1.2.1 Comienzo del proceso - 40 -
10.1.2.2 Técnicas para facilitar las especificaciones de una aplicación - 41 -
10.1.2.3 Despliegue de la función de calidad - 43 -
10.1.2.4 Casos de uso - 44 -
10.1.3 PRINCIPIOS DEL ANALISIS - 45 -
10.1.3.1 El dominio de la información - 47 -
10.1.3.2 Modelado - 48 -
10.1.3.3 Partición - 49 -
10.1.3.4 Visiones esenciales y de implementación - 50 -
10.1.4 CREACIÓN DE PROTOTIPOS DEL SOFTWARE - 50 -
10.1.4.1 Selección del enfoque de creación de prototipos - 50 -
10.1.4.2 Métodos y herramientas para el desarrollo de prototipos - 51 -
10.1.5 ESPECIFICACIÓN - 52 -
10.1.5.1 Principios de la especificación - 52 -
10.1.5.2 Representación - 52 -
10.1.5.3 La especificación de los requisitos del software - 53 -
10.1.6 REVISIÓN DE LA ESPECIFICACIÓN - 54 -
11 CONCEPTOS Y PRINCIPIOS DE DISEÑO - 55 -
11.1 EL PROCESO DEL DISEÑO - 57 -
11.1.1 Diseño y calidad del software - 57 -
11.2 Principios del diseño - 58 -
11.3 CONCEPTOS DEL DISEÑO - 60 -
11.3.1 Abstracción - 60 -
11.3.2 Refinamiento - 61 -
11.3.3 Modularidad - 61 -
11.3.4 Arquitectura del software - 62 -
11.3.5 Jerarquía de control - 64 -
11.3.6 Estructura de datos - 65 -

11.4 DISEÑO MODULAR EFECTIVO .... - 66 -
11.4.1 Independencia funcional .... - 66 -
11.4.2 Cohesión .... - 67 -
11.4.3 Acoplamiento .... - 68 -
11.5 DOCUMENTACIÓN DEL DISEÑO .... - 68 -
12 CONCEPTOS Y PRINCIPIOS ORIENTADOS A OBJETOS .... - 70 -
12.1 CONCEPTOS DE ORIENTACIÓN A OBJETOS .... - 70 -
12.1.1 CLASES Y OBJETOS .... - 72 -
12.1.2 PROPIEDADES .... - 72 -
12.1.3 MÉTODOS .... - 72 -
12.1.4 MENSAJES .... - 73 -
12.1.5 Encapsulamiento, herencia y polimorfismo .... - 73 -
12.2 IDENTIFICACIÓN DE LOS ELEMENTOS DE UN MODELO DE OBJETOS .... - 75 -
12.2.1 Identificación de clases y objetos .... - 75 -
12.2.2 Especificación de atributos .... - 77 -
12.2.3 Definición de operaciones .... - 78 -
12.2.4 Fin de la definición del objeto .... - 78 -
12.3 GESTION DE PROYECTOS DE SOFTWARE ORIENTADO A OBJETOS .... - 78 -
12.3.1 Marco de proceso común para la Orientación a Objetos .... - 79 -
12.3.2 Métricas y estimación en proyectos orientados a objetos .... - 81 -
12.3.3 Enfoque Y Consejos Para Estimaciones Y Planificación De Proyectos Orientados A Objetos .... - 83 -
12.3.4 Seguimiento del progreso en un proyecto orientado a objetos .... - 84 -
13 ANÁLISIS ORIENTADO A OBJETOS .... - 86 -
13.1 ANÁLISIS ORIENTADO A OBJETOS .... - 86 -
13.1.1 Un enfoque unificado para el Análisis Orientado A Objetos .... - 87 -
13.2 ANÁLISIS DEL DOMINIO .... - 88 -
13.2.1 El proceso de análisis del dominio .... - 88 -
13.3 COMPONENTES GENERICOS DEL MODELO DE ANÁLISIS ORIENTADOS A OBJETOS .... - 90 -
13.4 EL PROCESO DE ANÁLISIS ORIENTADO A OBJETOS .... - 91 -
13.4.1 Casos de uso .... - 91 -
13.4.2 Modelado de clases-responsabilidades-colaboraciones .... - 92 -
13.4.2.1 Clases .... - 92 -
13.4.2.2 Responsabilidades .... - 93 -
13.4.2.3 Colaboradores .... - 94 -
13.4.3 Definición de estructuras y jerarquías .... - 95 -
13.4.4 Definición de subsistemas .... - 95 -
13.5 EL MODELO OBJETO-RELACIÓN .... - 95 -
14 DIAGRAMA DE CONTEXTO .... - 96 -
15 DIAGRAMAS UML DE LA FASE DE ANALISIS OO .... - 97 -
15.1 DIAGRAMA DE CASO DE USO .... - 97 -
15.2 DIAGRAMAS DE SECUENCIA .... - 101 -
15.3 DIAGRAMAS DE COLABORACIÓN .... - 103 -
15.4 ESTADOS DE UN OBJETO .... - 105 -

16 DISEÑO ORIENTADO A OBJETOS .... - 106 -
16.1 Enfoque convencional vs. 00 .... - 107 -
16.2 Aspectos del diseño .... - 108 -
16.3 El panorama de DOO .... - 109 -
16.4 Un enfoque unificado para el DOO .... - 110 -
16.4.1 PROCESO DE DISEÑO DE SISTEMA .... - 110 -
16.4.2 PROCESO DE DISEÑO DE OBJETOS .... - 111 -
16.4.2.1 Descripción de objetos .... - 111 -
16.4.2.2 Diseño de algoritmos y estructuras de datos .... - 111 -
17 DIAGRAMAS UML DE LA FASE DE DISEÑO OO 113
17.1 DIAGRAMAS DE CLASES 113
17.2 DIAGRAMA DE CASO DE USO 114
17.3 DIAGRAMAS DE SECUENCIA 120
17.4 DIAGRAMAS DE COLABORACIÓN 123
17.5 DIAGRAMA INTERACCION ENTRE TIPOS DE OBJETOS 127
17.6 CICLO VITAL DEL OBJETO 128
17.7 ESTADOS DE UN OBJETO 129
17.8 DIAGRAMAS DE ESTADOS 130
17.9 DIAGRAMAS DE COMPONENTES 133
17.10 DIAGRAMAS DE DESPLIEGUE 134
17.11 DIAGRAMA DE DISTRIBUCION 135
18 DIAGRAMA ENTIDAD – RELACION 136
18.1 DICCIONARIO DE DATOS 137
19 PRUEBAS ORIENTADAS A OBJETOS 140
19.1 Las pruebas de unidad en el contexto de la 00 140
19.2 Las pruebas de integración en el contexto 00 140
19.3 Pruebas de validación en un contexto 00 141
20 CONCLUSIONES 142
21 BIBLIOGRAFÍA 144
22 INFOGRAFÍA 145
CONCLUSIONES..130
BIBLIOGRAFÍA...131
INFOGRAFÍA..132

LISTA DE FIGURAS

FIGURA1. Análisis como puente entre la ingeniería y el diseño del software.-28-
FIGURA 2. Áreas fundamentales en el análisis de requerimientos…………....-28-
FIGURA 3. Flujo y transformación de la información .... - 48 -
FIGURA 4. Conversión del modelo de análisis en un diseño de software. ...... - 56 -
FIGURA 5. Terminologías de estructura para un estilo arquitectónico de llamada y retorno .... - 64 -
FIGURA 6. Herencia operaciones de clase a objeto .... - 71 -
FIGURA 7. Paso de mensajes entre objetos .... - 73 -
FIGURA 8. Cómo se manifiestan los objetos. .... - 76 -
FIGURA 9. Secuencia típica de un proceso para un proyecto OO .... - 81 -
FIGURA 10 La pirámide del diseño OO .... - 106 -
FIGURA 11. Transformación de un modelo de análisis OO a un modelo de diseño OO.. - 107 -

ELABORACIÓN DEL SISTEMA DE INFORMACIÓN PARA LA MATRICULA, REGISTRO Y CONTROL DE NOTAS EN EL CENTRO EDUCATIVO DEL QUIROGA.

OSCAR ANDRES CALDERON ALONSO
DIEGO FERNANDO SANCHEZ MEDINA

RESUMEN

Este proyecto logro fortalecer la comunicación entre los diferentes estamentos del Centro educativo del Quiroga, mediante la sistematización de los procesos de registro y control de notas, matriculas, así como el soporte a la toma de decisiones, que venían desempeñándose de manera manual y que por lo tanto no permitían al centro educativo contar con un buen nivel competitivo y ofrecer un valor agregado.

PALABRAS CLAVES

Análisis, diseño, programación, pruebas, calidad, sistema, usuario.

ABSTRACT

This Project strengthened the communication among the different participants of the Centro Educativo del Quiroga through the systematization of registry and control of grades processes, registry of students processes as well as making decisions support, which were performed manually and therefore didn't allow the school to have a good competitive level and offer an added value.

KEYWORDS

Analysis, design, programming, proofs, quality, system, user.

1 TITULO

Elaboración del sistema de información para la matricula, registro y control de notas en el Centro Educativo Del Quiroga.

2 PROBLEMA

El sector educativo necesita mejorar y fortalecer sus sistemas de información de tal manera que se disponga de estadísticas confiables y oportunas sobre el desempeño en todos sus niveles, soportado en la idea que nos enfrentamos a un gran desarrollo tecnológico.

Por ello se hace necesario buscar un mecanismo capaz de garantizar y facilitar una estrecha comunicación entre los diferentes estamentos del sector educativo: Padres, estudiantes, profesores y directivos.

El Centro Educativo Del Quiroga, no cuenta con un sistema apropiado que le permita llevar un registro exacto de sus alumnos, en lo que respecta a su proceso de matricula y su historial académico, aspecto que se manifiesta en el registro y control de notas; razón por la cuál se hace muy difícil llevar un seguimiento y control a los estudiantes dificultando el establecimiento de lazos de comunicación entre la comunidad educativa.

Se busca, entonces, diseñar un Sistema de Información orientado a la parte académica que sea capaz de mejorar y optimizar dichos procesos en la comunidad educativa, además de proporcionar una herramienta que permita soportar y tomar decisiones en lo que respecta al entorno del Centro Educativo Del Quiroga.

3 OBJETIVOS

3.1 OBJETIVO GENERAL:

Desarrollar un Sistema de Información que permita llevar un registro detallado de los estudiantes del Centro Educativo Del Quiroga, contemplando a su vez la gestión y el soporte a la toma de decisiones.

3.2 OBJETIVOS ESPECÍFICOS:

- Suministrar información que permita la toma de decisiones y el soporte a la parte de gestión.

- Facilitar el intercambio de información entre las diferentes dependencias del colegio.

- Ofrecer un valor agregado al colegio y elevar su nivel de competencia y desarrollo tecnológico.

4 JUSTIFICACIÓN

Hoy en día existe una necesidad que a la vez se ha convertido en un objetivo bien claro en la ingeniería de sistemas: Permitir que el desarrollo tecnológico destruya los paradigmas existentes y facilite la masificación de nuevas tecnologías. Como respuesta a este planteamiento se han creado soluciones e infraestructuras tecnológicas llamados Sistemas de Información; medio por el cual los datos fluyen de una persona o departamento hacia otros y puede ser cualquier cosa, desde la comunicación interna entre los diferentes componentes de la organización hasta sistemas de cómputo que generan reportes periódicos para varios usuarios.

Actualmente en un sector de tanta importancia social como lo es el componente educativo, y tradicionalmente olvidado de las grandes innovaciones tecnológicas, se hace imprescindible para los centros de formación y para las familias, un canal de comunicación que les permita compartir de manera sencilla y rápida, toda la información que en el día a día se genera: notas, asistencia, incidencias, horarios.

Se busca crear un sistema capaz de mejorar y optimizar todos los procesos de gestión académica del centro, y que permita a los padres acceder a la información para su seguimiento diario. No se pretende sustituir el necesario contacto personal entre familia y profesores, sino facilitar el que estos encuentros, cada día más difíciles por incompatibilidad de horarios, sean lo más fructíferos posible, de manera que cuando se produzcan, ambas partes manejen el máximo de información diariamente actualizada.

Ahora bien, el resolver este punto neurálgico podemos derivar varios beneficios:

4.1 PARA EL CENTRO EDUCATIVO:

• La seguridad de mejorar y facilitar la comunicación entre los distintos integrantes de la comunidad escolar: padres, alumnos, profesores y directivos, mediante el uso de nuevas tecnologías.

• La forma óptima de mejorar los procesos de gestión interna, y la coordinación entre profesores y cargos directivos, con la utilización de una única y completa aplicación de gestión académica.

• La ventaja competitiva que supone la implantación de un servicio de valor añadido, como es la comunicación, de cara a la captación de

clientes y a la fidelización de los ya existentes, hecho que redunda directamente en la mejor imagen del colegio.

4.2 PARA LOS PADRES:

• Ofrecer la posibilidad a los padres del seguimiento continuo del proceso educativo de sus hijos, en cualquier momento y de una manera muy sencilla.

• Manejar el máximo de información, permite que los padres se integren activamente al proceso de educación de sus hijos.

4.3 PARA EL CUERPO DOCENTE Y DIRECTIVAS:

• Gran sencillez en la introducción de datos y el tratamiento de la información. Menos trabajo en papel.

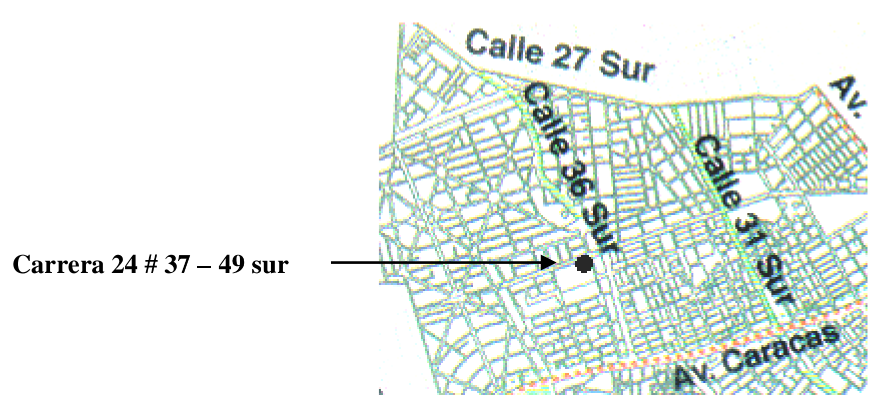
• Ayuda para preparar las entrevistas personales con la familia, y facilitar el cumplimiento con la responsabilidad de mantener informados a los padres.

• Ayuda para preparar las reuniones de profesores.

5 DELIMITACIÓN

5.1 DELIMITACIÓN EN RELACIÓN CON EL ESPACIO:

El presente proyecto se llevará a cabo en la ciudad de Bogotá D.C., en la zona 18 (Rafael Uribe Uribe), específicamente en el Centro Educativo Del Quiroga ubicado en carrera 24 # 37 – 49 sur



5.2 DELIMITACIÓN EN RELACIÓN CON EL TIEMPO

El proyecto se iniciara el día 15 de octubre de 2004 y concluirá el día 12 de febrero de 2005

5.3 DELIMITACIÓN EN RELACIÓN A LA TEMÁTICA

Desarrollo de un sistema de Información encargado de la matricula, registro y control de notas de los estudiantes en el Centro Educativo Del Quiroga, lo que involucra su análisis, diseño, programación y pruebas.

6 MARCO TEÓRICO

6.1 PROGRAMACIÓN ORIENTADA A OBJETOS

Actualmente una de las áreas más importantes en la industria y en el ámbito académico es la orientación a objetos. La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a los problemas y preocupaciones que siempre han existido en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas.

Un lenguaje orientado a objetos ataca estos problemas. Tiene tres características básicas: debe estar basado en objetos, basado en clases y capaz de tener herencia de clases. Muchos lenguajes cumplen uno o dos de estos puntos; muchos menos cumplen los tres. La barrera más difícil de sortear es usualmente la herencia.

El elemento fundamental de la OOP es, como su nombre lo indica, el objeto. Podemos definir un objeto como un conjunto complejo de datos y programas que poseen estructura y forman parte de una organización. Esta definición especifica varias propiedades importantes de los objetos. En primer lugar, un objeto no es un dato simple, sino que contiene en su interior cierto número de componentes bien estructurados. En segundo lugar, cada objeto no es un ente aislado, sino que forma parte de una organización jerárquica o de otro tipo.

6.1.1 Componentes de un Objeto

Un objeto puede considerarse como una especie de cápsula dividida en tres partes:

6.1.1.1 Relaciones

Permiten que el objeto se inserte en la organización y están formadas esencialmente por punteros a otros objetos.

6.1.1.2 Propiedades

Distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de que se

trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización.

6.1.1.3 Métodos

Son las operaciones que pueden realizarse sobre el objeto, que normalmente estarán incorporados en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia.

6.1.2 Organización de los objetos

En principio, los objetos forman siempre una organización jerárquica, en el sentido de que ciertos objetos son superiores a otros de cierto modo. Existen varios tipos de jerarquías: serán simples cuando su estructura pueda ser representada por medio de un "árbol". En otros casos puede ser más compleja.

En cualquier caso, sea la estructura simple o compleja, podrán distinguirse en ella tres niveles de objetos:

6.1.2.1 La raíz de la jerarquía

Se trata de un objeto único y especial. Este se caracteriza por estar en el nivel más alto de la estructura y suele recibir un nombre muy genérico, que indica su categoría especial.

6.1.2.2 Los objetos intermedios

Son aquellos que descienden directamente de la raíz y que a su vez tienen descendientes. Representan conjuntos o clases de objetos, que pueden ser muy generales o muy especializados, según la aplicación. Normalmente reciben nombres genéricos que denotan al conjunto de objetos que representan.

6.1.2.3 Los objetos terminales

Son todos aquellos que descienden de una clase o subclase y no tienen descendientes. Suelen llamarse casos particulares, **instancias** o **ítems**

porque representan los elementos del conjunto representado por la clase o subclase a la que pertenecen.

6.1.3 Estructura De Un Objeto

6.1.3.1 Relaciones

Las relaciones entre objetos son enlaces que permiten a un objeto relacionarse con aquellos que forman parte de la misma organización. Las hay de dos tipos fundamentales:

6.1.3.1.1 Relaciones jerárquicas

Son esenciales para la existencia misma de la aplicación porque la construyen. Son bidireccionales, es decir, un objeto es padre de otro cuando el primer objeto se encuentra situado inmediatamente encima del segundo en la organización en la que ambos forman parte; asimismo, si un objeto es padre de otro, el segundo es hijo del primero. Una organización jerárquica simple puede definirse como aquella en la que un objeto puede tener un solo padre, mientras que en una organización jerárquica compleja un hijo puede tener varios padres.

6.1.3.1.2 Relaciones semánticas

Se refieren a las relaciones que no tienen nada que ver con la organización de la que forman parte los objetos que las establecen. Sus propiedades y consecuencia solo dependen de los objetos en sí mismos (de su significado) y no de su posición en la organización.

6.1.3.2 Propiedades

Todo objeto puede tener cierto número de propiedades, cada una de las cuales tendrá, a su vez, uno o varios valores. En OOP, las propiedades corresponden a las clásicas "variables" de la programación estructurada. Son, por lo tanto, datos encapsulados dentro del objeto, junto con los métodos (programas) y las relaciones. Un objeto puede tener una propiedad de maneras diferentes: *Propiedades propias:* Están formadas dentro de la cápsula del objeto. *Propiedades heredadas:* Están definidas en un objeto diferente, antepasado de éste.

6.1.3.3 Métodos

Una operación que realiza acceso a los datos. Podemos definir método como un programa procedimental o procedural escrito en cualquier lenguaje, que está asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.

Si los métodos son programas, se deduce que podrían tener argumentos, o parámetros. Puesto que los métodos pueden heredarse de unos objetos a otros, un objeto puede disponer de un método de dos maneras diferentes: Métodos propios. Están incluidos dentro de la cápsula del objeto. Métodos heredados. Están definidos en un objeto diferente, antepasado de éste (padre,"abuelo", etc.).

6.2 BASE DE DATOS

Enfrentándonos al desarrollo de una aplicación nos encontramos con la tarea de realizar una base de datos que sea lo suficientemente buena y permita de esta manera asegurar un almacenamiento correcto, seguro y lógico de la información; por ello para elaborar de una manera consistente y segura una base de datos se debe aplicar una técnica conocida como normalización.

Por normalización se entiende el conjunto de reglas que sirven para ayudar a los desarrolladores a encontrar un esquema que minimice los problemas de lógica. La normalización se adoptó porque el viejo estilo de poner todos los datos en un solo lugar, como un archivo o una tabla de la base de datos, era ineficiente y conducía a errores de lógica cuando se trataba de manipular los datos, además, este tipo de almacenamiento dificultaba el tratamiento y extracción de la información.

La normalización también hace las cosas fáciles de entender, además la normalización de una base de datos tiende a ocupar menos espacio en disco que una no normalizada. Hay menos repetición de datos, lo que tiene como consecuencia un menor uso de espacio en disco.

El proceso de normalización tiene unas reglas y una serie de pasos a seguir, estos pasos son conocidos como grados de normalización, existen básicamente tres niveles de normalización: Primera Forma Normal (1FN), Segunda Forma Normal (2FN) y Tercera Forma Normal (3FN).

6.2.1 Primera forma normal (1FN):

La regla de la Primera Forma Normal establece que las columnas repetidas deben eliminarse y colocarse en tablas separadas y en cada tabla se debe escoger un único identificador, o clave principal que identifique a dicha tabla.
Es lógico que al realizar este paso surjan nuevas tablas que deben relacionarse, es decir, relacionar los datos de la tabla original con los de la nueva tabla. Para hacerlo, se debe añadir un campo clave similar a la clave principal a la segunda tabla de forma que se establezca la relación, este campo de relación es llamado llave foránea.

Esta forma normal ayuda a clarificar la base de datos y a organizarla en partes más pequeñas y más fáciles de entender. En lugar de tener que entender una tabla gigantesca que tiene muchos diferentes aspectos, se tienen objetos pequeños y más tangibles, así como las relaciones que guardan con otros objetos también pequeños.

6.2.2 Segunda forma normal (2FN):

La regla de la Segunda Forma Normal establece que todas las dependencias parciales se deben eliminar y separar dentro de sus propias tablas. Una dependencia parcial es un término que describe a aquellos datos que no dependen de la clave principal de la tabla para identificarlos.

Al haber alcanzado la Segunda Forma Normal, se puede disfrutar de algunas de las ventajas de las bases de datos relacionales. Por ejemplo, puede añadir nuevas columnas a las tablas sin afectar a otras tablas que estén relacionadas con está. Lo mismo aplica para las otras tablas. Alcanzar este nivel de normalización permite que los datos se acomoden de una manera natural dentro de los límites esperados.

6.2.3 Tercera Forma Normal (3FN):

La regla de la Tercera Forma Normal señala que hay que eliminar y separar cualquier dato que no sea clave. El valor de esta columna debe depender de la clave. Todos los valores deben identificarse únicamente por la clave. Al llegar a esta forma se logra más flexibilidad y se previene errores de lógica cuando inserta o borra registros. Cada columna en la tabla está identificada de manera única por la clave, y no hay datos repetidos. Esto provee un esquema limpio y elegante, que es fácil de trabajar y expandir.

Existen otros niveles de normalización que no se han mencionado. Ellos son Forma Normal Boyce-Codd, Cuarta Forma Normal (4NF), Quinta Forma Normal (5NF). Estas formas de normalización pueden llevar las cosas más allá de lo que se necesita. Éstas existen para hacer una base de datos realmente relacional. Tienen que ver principalmente con dependencias múltiples y claves relacionales, pero para este proyecto en particular no son realmente relevantes.

En resumen la normalización es una técnica que se utiliza para crear relaciones lógicas apropiadas entre tablas de una base de datos, que ayuda a prevenir errores lógicos en la manipulación de datos. La normalización facilita también agregar nuevas columnas sin romper el esquema actual ni las relaciones.

Ahora bien, el tener una base de datos normalizada – entendible – debe ir de la mano con un mecanismo para poder visualizar lo que ocurre dentro del aplicativo y además lograr documentar todo el proceso y desarrollo de un software.

6.3 UML (UNIFIED MODELING LANGUAGE)

El Lenguaje de Modelamiento Unificado (UML - Unified Modeling Language) es un lenguaje gráfico para visualizar, especificar y documentar cada una de las partes que comprende el desarrollo de software. UML entrega una forma de modelar cosas conceptuales como lo son procesos de negocio y funciones de sistema, además de cosas concretas como lo son escribir clases en un lenguaje determinado, esquemas de base de datos y componentes de software reusables.

UML es una especificación de notación orientada a objetos que divide cada proyecto en un número de diagramas que representan las diferentes vistas del proyecto. Estos diagramas juntos son los que representa la arquitectura del proyecto.

UML introduce nuevos diagramas que representa una visión dinámica del sistema. Es decir, gracias al diseño de la parte dinámica del sistema podemos darnos cuenta de problemas de la estructura al propagar errores o de las partes que necesitan ser sincronizadas, así como del estado de cada una de las instancias en cada momento.

UML también intenta solucionar el problema de propiedad de código que se da con los desarrolladores, al implementar un lenguaje de modelado común para todos los desarrollos se crea una documentación también común, que

cualquier desarrollador con conocimientos de UML será capaz de entender, independientemente del lenguaje utilizado para el desarrollo.

UML es ahora un estándar, puesto que su utilización es independiente del lenguaje de programación y de las características de los proyectos, ya que UML ha sido diseñado para modelar cualquier tipo de proyectos, tanto informáticos como de arquitectura, o de cualquier otro ramo.

7 MARCO CONCEPTUAL

7.1 CONCEPTO DE CALIDAD

La calidad se puede definir como "una característica o atributo de una cosa". De esta forma se podría decir que la calidad de los productos puede medirse como una comparación de sus características y atributos. Así, este concepto puede aplicarse a cualquier producto.

Una de las formas de realizar una medida de calidad es observar las diferencias ocurridas en la producción de dos productos iguales. La producción de artículos de cualquier especie no asegura que dos de ellos sean totalmente iguales. Quizás sea preciso realizar observaciones detalladas para lograr distinguir las variaciones entre uno y otro, ya que estas pueden no ser obvias. Es más, quizás sea necesario disponer de instrumentos adecuados y de precisión para poder observar dichos cambios de la producción. Uno de los principales objetivos de dar calidad a los productos es minimizar las diferencias entre unidades producidas. Estas diferencias tienen diversos orígenes y, por tanto, distintas y amplias formas de corregirlos, dependiendo de la naturaleza del producto. Lo primordial es tener en cuenta el concepto de brindar calidad a lo que se está realizando.

Por ello podemos concluir, el brindar calidad es una actividad esencial para un negocio que produce productos que serán utilizados por otras personas.

7.2 DIAGRAMA CASO DE USO

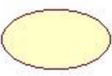
El diagrama de casos de uso representa la forma en como un Cliente (Actor) opera con el sistema en desarrollo, además de la forma, tipo y orden en como los elementos interactúan (operaciones o casos de uso).

7.2.1 Elementos

7.2.1.1 Actor:

Es un rol que un usuario juega con respecto al sistema. Es importante destacar el uso de la palabra rol, pues con esto se especifica que un Actor no necesariamente representa a una persona en particular, sino más bien la labor que realiza frente al sistema.

7.2.1.2 Caso de Uso:



Es una operación/tarea específica que se realiza tras una orden de algún agente externo, sea desde una petición de un actor o bien desde la invocación desde otro caso de uso.

7.2.1.3 Relaciones:

7.2.1.3.1 Asociación

Es el tipo de relación más básica que indica la invocación desde un actor o caso de uso a otra operación (caso de uso). Dicha relación se denota con una flecha simple.

7.2.1.3.2 Dependencia o Instanciación

Es una forma muy particular de relación entre clases, en la cual una clase depende de otra, es decir, se instancia (se crea). Dicha relación se denota con una flecha punteada.

7.2.1.3.3 Generalización

Este tipo de relación es uno de los más utilizados, cumple una doble función dependiendo de su estereotipo, que puede ser de **Uso** (<<uses>>) o de **Herencia** (<<extends>>).

7.3 Modelo de clases

Un diagrama de clases sirve para visualizar las relaciones entre las clases que involucran el sistema, las cuales pueden ser asociativas, de herencia, de uso y de contenimiento.

7.3.1 Elementos

7.3.1.1 Clase

Es la unidad básica que encapsula toda la información de un Objeto (un objeto es una instancia de una clase). A través de ella podemos modelar el entorno en estudio (una Casa, un Auto, una Cuenta Corriente, etc.).

En UML, una clase es representada por un rectángulo que posee tres divisiones:

<Nombre Clase>
<Atributos>
<Operaciones o Métodos>

En donde:

- **Superior**: Contiene el nombre de la Clase
- **Intermedio**: Contiene los atributos (o variables de instancia) que caracterizan a la Clase (pueden ser private, protected o public).

- **Inferior**: Contiene los métodos u operaciones, los cuales son la forma como interactúa el objeto con su entorno (dependiendo de la visibilidad: private, protected o public).

7.3.1.2 Atributos:

Los atributos o características de una Clase pueden ser de tres tipos, los que definen el grado de comunicación y visibilidad de ellos con el entorno, estos son:

7.3.1.2.1 public (+,):

Indica que el atributo será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

7.3.1.2.2 private (-,):

Indica que el atributo sólo será accesible desde dentro de la clase (sólo sus métodos lo pueden accesar).

7.3.1.2.3 protected (#,):

Indica que el atributo no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de las subclases que se deriven.

7.3.1.3 Métodos:

Los métodos u operaciones de una clase son la forma en como ésta interactúa con su entorno, éstos pueden tener las características:

7.3.1.3.1 public (+,):

Indica que el método será visible tanto dentro como fuera de la clase, es decir, es accesible desde todos lados.

7.3.1.3.2 private (-,):

Indica que el método sólo será accesible desde dentro de la clase (sólo otros métodos de la clase lo pueden accesar).

7.3.1.3.3 protected (#,):

Indica que el método no será accesible desde fuera de la clase, pero si podrá ser accesado por métodos de la clase además de métodos de las subclases que se deriven.

7.3.1.4 Relaciones entre Clases

Ahora ya definido el concepto de Clase, es necesario explicar como se pueden interrelacionar dos o más clases. Antes es necesario explicar el concepto de cardinalidad de relaciones: En UML, la cardinalidad de las relaciones indica el grado y nivel de dependencia, se anotan en cada extremo de la relación y éstas pueden ser:

- **uno o muchos**: 1..* (1..n)
- **0 o muchos**: 0..* (0..n)
- **número fijo**: m (m denota el número).

7.3.1.5 Herencia (Especialización/Generalización):

Indica que una subclase hereda los métodos y atributos especificados por una Súper Clase, por ende la Subclase además de poseer sus propios métodos y atributos, poseerá las características y atributos visibles de la Súper Clase (public y protected).

7.3.1.6 Agregación:

Para modelar objetos complejos, n bastan los tipos de datos básicos que proveen los lenguajes: enteros, reales y secuencias de caracteres. Cuando se requiere componer objetos que son instancias de clases definidas por el desarrollador de la aplicación, tenemos dos posibilidades:

- **Por Valor**: Es un tipo de relación estática, en donde el tiempo de vida del objeto incluido esta condicionado por el tiempo de vida del que lo incluye. Este tipo de relación es comúnmente llamada **Composición** (el Objeto base se construye a partir del objeto incluido, es decir, es "parte/todo").
- **Por Referencia**: Es un tipo de relación dinámica, en donde el tiempo de vida del objeto incluido es independiente del que lo incluye. Este tipo de relación es comúnmente llamada **Agregación** (el objeto base utiliza al incluido para su funcionamiento).

7.3.1.7 Asociación:

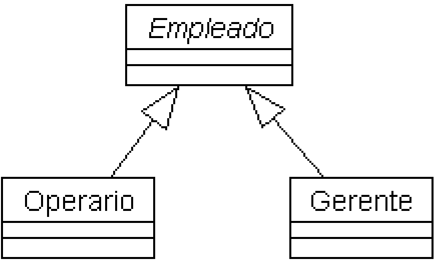
La relación entre clases conocida como Asociación, permite asociar objetos que colaboran entre si. Cabe destacar que no es una relación fuerte, es decir, el tiempo de vida de un objeto no depende del otro.

7.3.1.8 Dependencia o Instanciación (uso):

Representa un tipo de relación muy particular, en la que una clase es instanciada (su instanciación es dependiente de otro objeto/clase). Se denota por una flecha punteada.

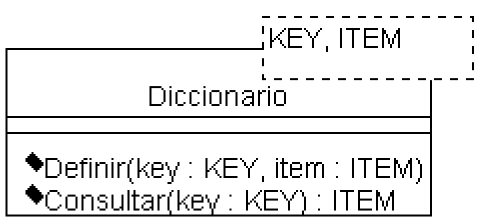
- **Casos Particulares**:

  Clase Abstracta:



Una clase abstracta se denota con el nombre de la clase y de los métodos con letra "itálica". Esto indica que la clase definida no puede ser instanciada pues posee métodos abstractos (aún no han sido definidos, es decir, sin implementación). La única forma de utilizarla es definiendo subclases, que implementan los métodos abstractos definidos.

- **Clase parametrizada**:



Una clase parametrizada se denota con un subcuadro en el extremo superior de la clase, en donde se especifican los parámetros que deben ser pasados a la clase para que esta pueda ser instanciada.

8 MARCO METODOLOGICO

8.1 CONCEPTUALIZACIÓN

Cuando se va a construir un software es necesario conocer un lenguaje de programación, pero con eso no basta. Si se quiere que el sistema sea robusto y mantenible es necesario que el problema sea analizado y la solución sea cuidadosamente diseñada. Se debe seguir un proceso robusto, que incluya las actividades principales. Si se sigue un proceso de desarrollo que se ocupa de plantear cómo se realiza el análisis y el diseño, y cómo se relacionan los productos de ambos, entonces la construcción de software va a poder ser planificable y repetible, y la probabilidad de obtener un sistema de mejor calidad al final del proceso aumenta considerablemente, especialmente cuando se trata de un equipo de desarrollo formado por varias personas.

Para este proyecto se va a seguir el método de desarrollo orientado a objetos que propone Craig Larman. Este proceso no fija una metodología estricta, sino que define una serie de actividades que pueden realizarse en cada fase, las cuales deben adaptarse según las condiciones del proyecto que se esté llevando a cabo. Se ha escogido seguir este proceso debido a que aplica los últimos avances en Ingeniería del Software, y a que adopta un enfoque eminentemente práctico y sistémico, aportando soluciones a las principales dudas y/o problemas con los que se enfrenta el desarrollador.

Su mayor aporte consiste en atar los cabos sueltos que métodos anteriores dejan. La notación que se usa para los distintos modelos, tal y como se ha dicho anteriormente, es la proporcionada por UML, que se ha convertido en el estándar de facto en cuanto a notación orientada a objetos. El uso de UML permite integrar con mayor facilidad en el equipo de desarrollo a nuevos miembros y compartir con otros equipos la documentación, pues es de esperar que cualquier desarrollador versado en orientación a objetos conozca y use UML (o se esté planteando su uso).

Se va a abarcar todo el ciclo de vida, empezando por los requisitos y terminando con el sistema implantado, proporcionando así una visión completa y coherente de la producción de software. El enfoque que toma es el de un ciclo de vida iterativo incremental, el cual permite una gran flexibilidad a la hora de adaptarlo a un proyecto y a un equipo de desarrollo específicos. El ciclo de vida está dirigido por casos de uso, es decir, por la funcionalidad que ofrece el sistema a los futuros usuarios del mismo. Así no se pierde de vista la motivación principal que debería estar en cualquier

proceso de construcción de software: el resolver una necesidad del usuario/cliente.

A propósito de la plataforma en la que desarrollaremos el proyecto decidimos emplear visual Basic 6.0 en compañía de Microsoft Access para el manejo de la Base de Datos, teniendo en cuenta que el colegio no cuenta ni con el presupuesto para adquirir licencias ni con la infraestructura computacional adecuada.

8.1.1 Visión General

El proceso a seguir para realizar desarrollo orientado a objetos es difícil, debido a la complejidad que nos vamos a encontrar al intentar desarrollar cualquier sistema software de tamaño medio-alto. El proceso está formado por una serie de actividades y subactividades, cuya realización se va repitiendo en el tiempo aplicado a distintos elementos.
En este apartado se va a presentar una visión general para poder tener una idea del proceso a alto nivel, y más adelante se verán los pasos que componen cada fase.

8.2 FASES METODOLÓGICAS

8.2.1 Planificación y Especificación de Requisitos:

Planificación, definición de requisitos, construcción de prototipos, etc.

8.2.1.1 Identificación de Conceptos

Para identificar conceptos se ha de basar en el documento de Especificación de Requisitos y en el conocimiento general acerca del dominio del problema.

8.2.1.2 Fase de Planificación y Especificación de Requisitos

Esta fase se corresponde con la Especificación de Requisitos tradicional ampliada con un Borrador de Modelo Conceptual y con una definición de Casos de Uso de alto nivel. En esta fase se decidiría si se aborda la construcción del sistema mediante desarrollo orientado a objetos o no, por lo que, en principio, es independiente del paradigma empleado posteriormente.

8.2.1.2.1 Actividades

Las actividades de esta fase son las siguientes:

1. Definir el Plan-Borrador.
2. Crear el Informe de Investigación Preliminar.
3. Definir los Requisitos.
4. Registrar Términos en el Glosario.
5. Implementar un Prototipo. (opcional)
6. Definir Casos de Uso (de alto nivel y esenciales).
7. Definir el Modelo Conceptual-Borrador.
8. Definir la Arquitectura del Sistema-Borrador.
9. Refinar el Plan.

El orden no es estricto, lo normal es que las distintas actividades se solapen en el tiempo.

8.2.1.3 Identificación de los Límites del Sistema

Al definir los límites del sistema se establece una diferenciación entre lo que es interno y lo que es externo al sistema.

8.2.1.4 Modelo Conceptual

Una parte de la investigación sobre el dominio del problema consiste en identificar los conceptos que lo conforman. Para representar estos conceptos se va a usar un Diagramas de UML.

En el Modelo Conceptual se tiene una representación de conceptos del mundo real, no de componentes software. El objetivo de la creación de un Modelo Conceptual es aumentar la comprensión del problema. Por tanto, a la hora de incluir conceptos en el modelo, es mejor crear un modelo con muchos conceptos que quedarse corto y olvidar algún concepto importante.

8.2.1.5 Creación del Modelo Conceptual

Para crear el Modelo Conceptual se siguen los siguientes pasos:

1. Hacer una lista de conceptos
2. Representarlos en un diagrama
3. Añadir las asociaciones necesarias para ilustrar las relaciones entre conceptos que es necesario conocer
4. Añadir los atributos necesarios para contener toda la información que se necesite conocer de cada concepto.

8.2.2 Construcción:

La construcción del sistema. Las fases dentro de esta etapa son las siguientes:

8.2.2.1 Diseño de Alto Nivel:

En la fase de Diseño de Alto Nivel de un ciclo de desarrollo se investiga sobre el problema, sobre los conceptos relacionados con el subconjunto de casos de uso que se esté tratando. Se intenta llegar a una buena comprensión del problema por parte del equipo de desarrollo, sin entrar en cómo va a ser la solución en cuanto a detalles de implementación.

8.2.2.1.1 Actividades

Las actividades de la fase de Diseño de Alto Nivel son las siguientes:

1. Definir Casos de Uso Esenciales en formato expandido. (si no están definidos)
2. Refinar los Diagramas de Casos de Uso.
3. Refinar el Modelo Conceptual.
4. Refinar el Glosario. (continuado en posteriores fases)
5. Definir los Diagramas de Secuencia del Sistema.
6. Definir Contratos de Operación.
7. Definir Diagramas de Estados. (opcional)

8.2.2.2 Diseño de Bajo Nivel:

El sistema se especifica en detalle, describiendo cómo va a funcionar internamente para satisfacer lo especificado en el Diseño de Alto Nivel.

En la fase de Diseño de Bajo Nivel se crea una solución a nivel lógico para satisfacer los requisitos, basándose en el conocimiento reunido en la fase de Diseño de Alto Nivel. Los modelos más importantes en esta fase son el Diagrama de Clases de Diseño y los Diagramas de Interacción, que se realizan en paralelo y que definen los elementos que forman parte del sistema orientado a objetos que se va a construir (clases y objetos) y cómo colaboran entre sí para realizar las funciones que se piden al sistema, según éstas se definieron en los contratos de operaciones del sistema.

8.2.2.2.1 Actividades

Las actividades que se realizan en la etapa de Diseño de Bajo Nivel son las siguientes:

1. Definir los Casos de Uso Reales.
2. Definir Informes e Interfaz de Usuario.
3. Refinar la Arquitectura del Sistema.
4. Definir los Diagramas de Interacción.
5. Definir el Diagrama de Clases de Diseño. (en paralelo con los Diagramas de Interacción)
6. Definir el Esquema de Base de Datos.

8.2.3 Fase de Implementación y pruebas:

Se lleva lo especificado en el Diseño de Bajo Nivel a un lenguaje de programación.

8.2.3.1 Pruebas:

Se llevan a cabo una serie de pruebas para corroborar que el software funciona correctamente y que satisface lo especificado en la etapa de Planificación y Especificación de Requisitos.

De todas las fases propuestas la fase de Construir es la que va a consumir la mayor parte del esfuerzo y del tiempo en un proyecto de desarrollo. Para llevarla a cabo se va adoptar un enfoque iterativo, tomando en cada iteración un subconjunto de los requisitos (agrupados según casos de uso) y llevándolo a través del diseño de alto y bajo nivel hasta la implementación y pruebas.

Con esta aproximación se consigue disminuir el grado de complejidad que se trata en cada ciclo, y se tiene pronto en el proceso una parte del sistema funcionando que se puede contrastar con el usuario/cliente.

8.2.3.2 Implantación:

Se tendrá en cuenta la capacitación tanto a usuarios como a operadores del sistema, así como la revisión del sistema mediante los métodos de registro de eventos, evaluación del impacto y encuestas de actitud.

9 MARCO LEGAL

9.1 NORMA ISO 9000

Las series de ISO 9000 son un grupo de 5 individuales, pero relacionadas, estándares internacionales de administración de la calidad y aseguramiento de calidad.

Estas normas son genéricas, no específicas para cualquier producto. Pueden usarse igualmente para manufactura y servicios industriales. Estos estándares fueron desarrollados para documentar efectivamente los elementos de sistemas de calidad que son instrumentados para mantener un sistema eficiente de calidad en la empresa. La serie ISO 9000 no especifica la tecnología que debe ser aplicada para la instrumentación de los elementos del sistema de calidad.

Al aplicar esta normatividad se pueden alcanzar beneficios, tales como lo son:

1. La posibilidad de darle calidad al producto o servicio.
2. Evitar costos de inspecciones finales, costos de garantías y reprocesos.
3. Mayor aceptación por parte de los clientes y acogida en los mercados tanto nacionales como internacionales

Ahora bien, hoy en día la satisfacción hacia el uso de un producto puede marcar una gran diferencia en el mercado de productos similares. Es así como el desarrollo de artículos que satisfacen las expectativas de los clientes y usuarios harán la diferencia entre las organizaciones que desarrollan productos que compiten en el mercado.

El desarrollo de productos software no está ausente de ofrecer calidad, es por esto que se buscan estrategias orientadas a la satisfacción del cliente y además que pueda garantizar:

1. La producción de código mas fiable
2. Mayor nivel de calidad en todo el ciclo de la producción del software
3. Y por ende, la satisfacción de sus clientes y mejora de su ventaja competitiva

Una estrategia mundialmente adoptada, ha sido la implementación del modelo de evaluación y mejora del proceso de software diseñado por ISO

9000, específicamente la guía ISO 9000-3, en la cual se define como aplicar la ISO 9001[1] al proceso de software.

9.2 NORMA ISO 9000-3

En estos días "calidad" es la palabra de más relevancia, los consumidores esperan productos de calidad para satisfacer sus necesidades, solucionar sus problemas y obtener beneficios. Sin embargo dentro de la industria del software, "calidad" no ha sido el fuerte de la rama.

Hoy en día la industria del software está implementando modelos para mejorar sus operaciones y corregir sus fallas. La expectativa es colocar el desarrollo de software bajo un control estadístico para verificar cuáles son las actividades repetitivas que continuamente se tienen que programar, y que producen exactamente el mismo resultado. Así, los procesos exitosos utilizados anteriormente pueden ser modelos base para la planeación de proyectos futuros, optimizando costos, incrementando la eficiencia y la productividad, desarrollando mejores productos de calidad y por consecuencia, generando más beneficios para la empresa.

Uno de estos modelos base son las normas estándares de calidad ISO 9000 que en especial han creado un interés masivo para la industria de software a causa de su aceptación a nivel internacional de muchas compañías importantes.

9.2.1 Generalidades

9.2.1.1 Título

Normas de gestión de la calidad y garantía de la calidad. Parte 3: Orientaciones para la aplicación de la Norma ISO 9001 al desarrollo, suministro y mantenimiento del software.

9.2.1.2 Naturaleza

Norma internacional

---

[1] ISO 9000 – 1: Norma que se aplica cuando la empresa debe responsabilizarse por todas las etapas de un ciclo productivo, es decir: diseño, desarrollo y elaboración.

9.2.1.3 Ámbito

Desarrollo de Sistemas de Información, procesos del ciclo de vida y calidad del software.

9.2.1.4 Campo de aplicación y alcance

Esta parte de la ISO 9000 contiene orientaciones que facilitan la aplicación de la Norma ISO 9001 a las organizaciones dedicadas al desarrollo, suministro y mantenimiento del software.

Se pretende con ella dar orientaciones en relación con situaciones en las que un contrato entre dos partes exija la demostración de la capacidad de determinado proveedor para desarrollar, suministrar y mantener productos de software.

Tales orientaciones describen las clases de control y los métodos sugeridos para la producción del software, que satisfagan los requisitos establecidos. Esto será posible principalmente a través de la prevención de "no-conforme" a lo largo de todas las fases del proceso, desde el desarrollo hasta el mantenimiento.

9.2.1.5 Estructura

1. Sistema de la calidad – estructura.
2. Responsabilidad de la gestión.
3. Sistema de la calidad.
4. Auditorias internas al sistema de la calidad.
5. Acciones correctivas.
6. Sistema de la calidad - actividades a lo largo del ciclo de vida.
7. General.
8. Análisis del contrato
9. Especificación de los requisitos del comprador
10. Planificación del desarrollo
11. Planificación de la calidad
12. Proyecto e implementación
13. Pruebas y validaciones
14. Aceptación
15. Reproducción, entrega e instalación
16. Mantenimiento
17. Sistema de la calidad - actividades de apoyo (independientes de cualquier fase)
18. Gestión de la configuración
19. Control de documentos

20. Registros de la calidad
21. Medición
22. Reglas, prácticas y convenciones
23. Herramientas y técnicas
24. Aprovisionamiento
25. Productos de software incluidos

9.2.2 Análisis

Desde que la ISO 9001 fue escrita para ser utilizada por toda clase de industrias, es regularmente difícil interpretarla para el desarrollo de software, por lo cual se publicó la ISO 9000-3 "Guía para la aplicación de ISO 9001 para el desarrollo, implementación y mantenimiento de software".

El objetivo de la ISO 9001 es construir un sistema de calidad el cual contenga la estructura de la organización, responsabilidades, procedimientos, procesos y recursos para implementar una dirección de calidad. Mientras que el de la ISO 9000-3 es proveer las especificaciones de cómo aplicar la ISO 9001 al desarrollo del software, implementación y mantenimiento. Se incluyen algunos temas que no se encuentran en las normas ISO 9000 genéricas, tales como Administración de la Configuración o Planeación de Proyectos. Sería poco probable lograr resultados de calidad en un proyecto de desarrollo software de tamaño mediano, sin haber tomado las provisiones necesarias para el control de configuración. Esto implica que para ciertos productos o servicios, la especificación de requerimientos contenida en las normas genéricas ISO 9000 no es suficiente para asegurar la calidad, y esto justifica la necesidad de otras normas o guías más específicas.

9.2.3 Normas aplicables dentro del proyecto

9.2.3.1 Identificación y trazabilidad del producto

Donde sea apropiado se establecerá y mantendrá procedimientos para identificar el producto desde la etapa de diseño hasta la entrega e instalación, pasando por todas las etapas de producción. Cuando la trazabilidad del producto sea un requisito especificado, los productos individuales o los, lotes deben tener una identificación única. Este identificador debe ser registrado.

9.2.3.2 Inspección y pruebas

Aseguramiento de que el producto desarrollado no se utilice o procese hasta que sea inspeccionado o verificado que cumple con los requerimientos específicos.

9.2.3.3 Estado de Inspección y pruebas:

El estado de inspección y pruebas del producto debe ser identificado mediante marcas, etiquetas autorizadas, sellos, rótulos, registros de inspección, programas computacionales de pruebas, localizaciones físicas, etc.
Estos elementos deben indicar la conformidad o no-conformidad del producto con respecto a las pruebas e inspecciones efectuadas. La identificación del estado y pruebas debe ser mantenida en el proceso de producción e instalación del producto para asegurar que sólo los que hayan pasado las pruebas e inspecciones requeridas sean entregados al cliente.

9.2.3.4 Control de registros de calidad

Establecer y mantener procedimientos para identificar, recolectar, indexar, llenar, archivar y desechar los registros de calidad. Todos los registros deben ser legibles e identificables con el producto del que se trate. El tiempo que deberán mantenerse esos registros debe ser definido y registrado.

9.2.3.5 Capacitación

Establecer y mantener procedimientos para identificar las necesidades de capacitación y proveer entrenamiento a todo el personal que realice tareas específicas debe ser calificado con base en su educación, entrenamiento y/o experiencia, se deben mantener registros apropiados de capacitación.

10 CONCEPTOS Y PRINCIPIOS DEL ANALISIS

10.1 FASE DE PLANIFICACIÓN Y ESPECIFICACIÓN DE REQUISITOS

El análisis y la especificación de los requisitos puede parecer una tarea relativamente sencilla, pero las apariencias engañan. El contenido de comunicación es muy denso. Abundan las ocasiones para las malas interpretaciones o falta de información. Es muy probable que haya ambigüedad. El dilema al que se enfrenta en este tipo de desarrollo de software puede **entenderse** muy bien repitiendo esta famosa frase: “Sé que cree que entendió lo que piensa que dije, pero no estoy seguro de que se dé cuenta de que lo que escuchó no es lo que yo quise decir”.

FIGURA 1. Análisis como puente entre la ingeniería y el diseño del software

10.1.1 Análisis de requerimientos

Esta fase del proyecto es la que ha permitido poner al descubierto las características de operación del software, es decir, que funciones debe poseer, que datos debe procesar y que se espera de dicho software, estableciendo de esta manara las restricciones que debe poseer la aplicación.

Para esta parte del proyecto se dividió en cinco áreas fundamentales de esfuerzo (Fig. 2), que son: Reconocimiento del problema, evaluación y

síntesis, modelado, especificación y revisión.

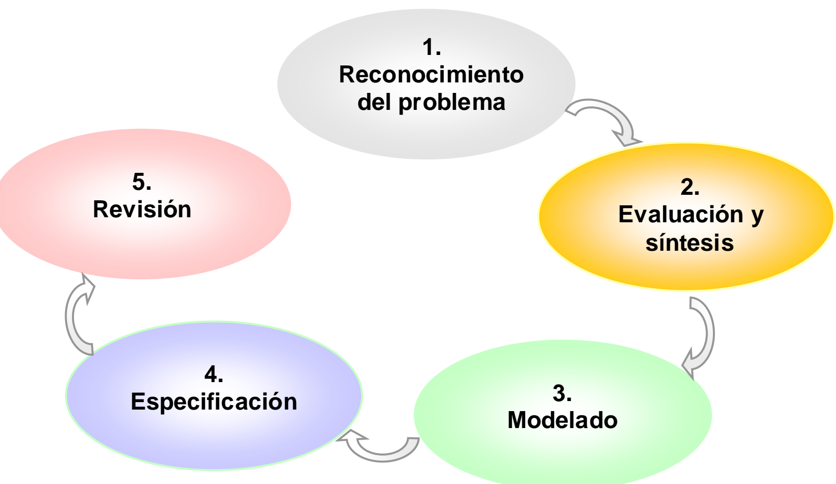


FIGURA 2. Áreas fundamentales en el análisis de requerimientos.

El objetivo de esta etapa es el reconocimiento de los elementos básicos del problema tal y como los percibe el cliente/usuario, sumado a ello la evaluación del problema y la síntesis de la solución es la siguiente área principal de esfuerzo en el análisis de requisitos. En este momento es donde se empiezan a definir todos los objetos de datos observables externamente, se evalúa el flujo y contenido de la información, adicionalmente se define y elabora las funciones del software entendiendo el comportamiento del mismo en el contexto de acontecimientos que afectan al sistema, estableciendo las características de la interfaz del sistema y descubriendo las restricciones adicionales del diseño; cada una de estas tareas sirve para describir el problema de manera que se pueda sintetizar un enfoque o solución global.

Una vez que se han identificado los problemas o falencias actuales y la información deseada el grupo de trabajo determina qué información va a producir el nuevo sistema y qué información deseada (entradas y salidas). Para empezar, se define en detalle los datos, las funciones de tratamiento y el comportamiento del sistema. Una vez que se ha establecido esta información, se consideran las arquitecturas básicas para la implementación. El proceso de evaluación y síntesis continúa hasta que exista la seguridad de que se puede especificar adecuadamente el software para posteriores fases

de desarrollo.

Durante la actividad de evaluación y síntesis de la solución, se realizaron modelos del sistema con el propósito de entender a cabalidad el flujo de datos y de control. Este modelo sirve como fundamento para el diseño del software y como base para la creación de una especificación del software.

10.1.2 IDENTIFICACIÓN DE REQUISITOS PARA EL SOFTWARE

Antes que los requisitos puedan ser analizados, modelados o especificados, deben ser recogidos a través de un proceso de obtención.

10.1.2.1 Comienzo del proceso

La técnica de obtención de requisitos que fue utilizada en la ejecución del proyecto fue llevar a cabo una reunión o entrevista preliminar. Se realizo una primera reunión que intervienen el grupo de trabajo y el cliente. En esta reunión se realizaron preguntan de cuestiones de contexto libre, es decir, un conjunto de preguntas que llevan a un entendimiento básico del problema, qué solución busca, la naturaleza de la solución que se desea y la efectividad del primer encuentro. En estas preguntas se encuentran a su vez divididas en tres grandes conjuntos:

- El primer conjunto de cuestiones de contexto libre se enfoca sobre el cliente, los objetivos generales y los beneficios esperados. Estas preguntas ayudan a identificar todos los ***participantes*** que tienen un interés en el software a construir. Además, las preguntas identifican los beneficios medibles en una implementación correcta de posibles alternativas para un desarrollo normal del software.

- El siguiente conjunto de preguntas permite al grupo de trabajo obtener un mejor entendimiento del problema y al cliente comentar sus opiniones sobre la solución

- El último conjunto de preguntas se concentra en la eficacia de la reunión. Gauge y Weinberg las denominan “meta-preguntas” donde se corrobora si la persona entrevistada tiene el suficiente peso y conocimiento de la organización, para valorar sus condicionamientos y respuestas.

10.1.2.2 Técnicas para facilitar las especificaciones de una aplicación

Los usuarios y los desarrolladores a menudo tienen una mentalidad inconsciente de nosotros y ellos. En vez de trabajar como un equipo para identificar y refinar los requisitos, cada uno define por derecho su propio territorio y se comunica a través de una serie de memorandos, papeles de posiciones formales, documentos y sesiones de preguntas y respuestas. La historia ha demostrado que este método no funciona muy bien. Abundan los malentendidos, se omite información importante y nunca se establece una buena relación de trabajo.

Con estos problemas en mente es por lo que algunos investigadores independientes han desarrollado un enfoque orientado al equipo para la reunión de requisitos que se aplica durante las primeras fases del análisis y especificación. Denominadas Técnicas para facilitar las especificaciones de la aplicación (TFEA), este enfoque es partidario de la creación de un equipo conjunto de clientes y desarrolladores que trabajan juntos para identificar el problema, proponer soluciones, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución. Hoy en día, las técnicas para facilitar las especificaciones de la aplicación son empleadas de forma general por los sistemas de información, pero la técnica ofrece un potencial de mejora en aplicaciones de todo tipo.

Se han propuesto muchos enfoques diferentes de técnicas para facilitar las especificaciones de la aplicación. Cada uno utiliza un escenario ligeramente diferente, pero todos aplican alguna variación en las siguientes directrices básicas:

1. La reunión se celebra en un lugar neutral y acuden tanto los clientes como los desarrolladores.
2. Se establecen normas de preparación y de participación.
3. Se sugiere una agenda lo suficientemente formal como para cubrir todos los puntos importantes, pero lo suficientemente informal como para animar el libre flujo de ideas.
4. Un coordinador que controle la reunión.
5. Se usa un mecanismo de definición (que puede ser hojas de trabajo, gráficos, carteles o pizarras).
6. El objetivo es identificar el problema, proponer elementos de solución, negociar diferentes enfoques y especificar un conjunto preliminar de requisitos de la solución en una atmósfera que permita alcanzar el objetivo.

Para comprender mejor el flujo de acontecimientos tal y como ocurren en

una reunión técnicas para facilitar las especificaciones de la aplicación, presentamos un pequeño escenario que esboza la secuencia de acontecimientos que llevan a la reunión, los que ocurren en la reunión y los que la siguen.

En las reuniones iniciales entre el desarrollador y el cliente se dan preguntas y respuestas básicas que ayudan a establecer el ámbito del problema y la percepción global de una solución. Fuera de estas reuniones iniciales, el cliente y el desarrollador escriben una solicitud de producto de una o dos páginas. Se selecciona lugar, fecha y hora de reunión y se elige un coordinador. Se invita a participar a representantes de ambas organizaciones; la del cliente y la de desarrollo. Se distribuye la solicitud de producto a los asistentes antes de la fecha de reunión[2].

Mientras se revisa la solicitud días antes de la reunión, se pide a todos los asistentes que hagan una lista de objetos que formen parte del entorno del sistema, otros objetos que debe producir el sistema y objetos que usa el sistema para desarrollar sus funciones. Además, se pide a todos los asistentes que hagan otra lista de servicios (procesos o funciones) que manipulan o interactúan con los objetos. Finalmente, se desarrollan también listas de restricciones (por ejemplo, costes, tamaño, peso) y criterios de rendimiento (por ejemplo, velocidad, precisión). Se informa a los asistentes que no se espera que las listas sean exhaustivas, pero que si que reflejen su punto de vista del sistema.

Cuando empieza la reunión se busca que el primer tema de estudio es la necesidad y justificación del nuevo producto, pues todo el mundo debería estar de acuerdo en que el desarrollo del producto está justificado. Una vez que se ha conseguido el acuerdo, cada participante presenta sus listas para su estudio. Las listas pueden exponerse en las paredes de la habitación usando largas hojas de papel, pinchadas o pegadas, o escritas en una pizarra. Idealmente debería poderse manejar separadamente cada entrada de las listas para poder combinarlas, borrarlas o añadir otras En esta fase, está estrictamente prohibido el debate o las críticas.

Una vez que se han presentado las listas individuales sobre un tema, el grupo crea una lista combinada. La lista combinada elimina las entradas redundantes y añade las ideas nuevas que se les ocurran durante la presentación, pero no se elimina nada más Cuando se han creado las listas combinadas de todos los temas, sigue el estudio —moderado por el coordinador— La lista combinada es ordenada, ampliada o redactada de nuevo para reflejar apropiadamente el producto o sistema que se va a

---

[2] Tomado de Ingeniería del Software. Roger Pressman.

desarrollar. El objetivo es desarrollar una *lista de consenso* por cada tema (objetos, servicios, restricciones y rendimiento). Después estas listas se ponen aparte para utilizarlas posteriormente.

Una vez que se han completado las listas de consenso, el equipo se divide en subequipos, cada uno trabaja para desarrollar una mini especificación de una o más entradas de cada lista La mini especificación es una elaboración de la palabra o frase contenida en una lista. Es aquí que cada subequipo presenta entonces sus mini especificaciones a todos los asistentes para estudiarlas. Se realizan nuevos añadidos, eliminaciones y se sigue elaborando. En algunos casos, el desarrollo de algunas mini especificaciones descubrirá nuevos objetos, servicios, restricciones o requisitos de rendimiento que se añadirán a las listas originales. Durante todos los estudios, el equipo sacará a relucir aspectos que no podrán resolverse durante la reunión. Se hará una lista de estas ideas para tratarlas más adelante.

Una vez que se han completado las mini especificaciones, cada asistente hace una lista de criterios de validación del producto/sistema y presenta su lista al equipo. Se crea así una lista de consenso de criterios de validación. Finalmente, uno o más participantes es asignado para escribir el borrador entero de especificación con todo lo expuesto en la reunión.

10.1.2.3 Despliegue de la función de calidad

El despliegue de la función calidad (DFC) es una técnica de gestión de calidad que traduce las necesidades del cliente en requisitos técnicos del software. Esta técnica se concentra en maximizar la satisfacción del cliente. Para conseguirlo, se hace énfasis en entender lo que resulta valioso para el cliente y después desplegar estos valores a lo largo del proceso de ingeniería. Despliegue De La Función Calidad identifica tres tipos de requisitos:

1. Requisitos normales.

   Se declaran objetivos y metas para un producto o sistema durante las reuniones con el cliente. Si estos requisitos están presentes, el cliente quedará satisfecho. Ejemplos de requisitos normales podrían ser peticiones de tipos de presentación gráfica, funciones específicas del sistema y niveles definidos de rendimiento.

2. Requisitos esperados.

   Estos requisitos son implícitos al producto o sistema y pueden ser tan

fundamentales que el cliente no los declara explícitamente. Su ausencia sería motivo de una insatisfacción significativa. Ejemplos de requisitos esperados son: facilidad de interacción hombre-máquina, buen funcionamiento y fiabilidad general, y facilidad de instalación del software.

3. Requisitos innovadores.

Estas características van más allá de las expectativas del cliente y suelen ser muy satisfactorias.

En realidad, el despliegue de la función calidad se extiende a todo el proceso de ingeniería. Sin embargo, muchos conceptos de esta técnica son aplicables al problema de la comunicación con el cliente a que se enfrenta un ingeniero durante las primeras fases del análisis de requisitos.

En las reuniones con el cliente el despliegue defunción se emplea para determinar el valor de cada función requerida para el sistema. El despliegue de información identifica tanto los objetos de datos como los acontecimientos que el sistema debe producir y consumir. Estos están unidos a las funciones. Finalmente, el despliegue de turcas examina el comportamiento del sistema o producto dentro del contexto de su entorno. El análisis de valor es llevado a cabo para determinar la prioridad relativa de requisitos determinada durante cada uno de los tres despliegues mencionados anteriormente.

El despliegue de la función calidad utiliza observaciones y entrevistas con el cliente, emplea encuestas y examina datos históricos (por ejemplo, informes de problemas) como datos de base para la actividad de recogida de requisitos. Estos datos se traducen después en una tabla de requisitos (denominada tabla de opinión del cliente) que se revisa con el cliente. Entonces se usa una variedad de diagramas, matrices y métodos de evaluación para extraer los requisitos esperados e intentar obtener requisitos innovadores.

10.1.2.4 Casos de uso

Una vez recopilados los requisitos, se procede a crear un conjunto de escenarios que identifiquen una línea de utilización para el sistema que va a ser construido. Los escenarios, algunas veces llamados casos de uso, facilitan una descripción de cómo el sistema se usará.

Para crear un caso de uso, se debe primero identificar los diferentes tipos de personas (o dispositivos) que utiliza el sistema o producto. Estos actores

actualmente representan papeles que la gente (o dispositivos) juegan como impulsores del sistema. Definido más formalmente, un actor es algo que comunica con el sistema o producto y que es externo al sistema en sí mismo.

Es importante indicar que un actor y un usuario no son la misma cosa. Un usuario normal puede jugar un número de papeles diferentes cuando utiliza un sistema, por lo tanto un actor representa una clase de entidades externas que lleva a cabo un papel.

Puesto que la obtención de requisitos es una actividad de evolución, no todos los actores se identifican durante la primera iteración. Es posible identificar actores iniciales durante la primera iteración y actores secundarios cuando más se aprende del sistema. Los actores iniciales interactúan para conseguir funciones del sistema y derivar el beneficio deseado del sistema. Estos trabajan directa y frecuentemente con el software. Los actores secundarios existen para dar soporte al sistema que los actores iniciales han dado forma con su trabajo.

Una vez que se han identificado los JLIOICS, se pueden desarrollar los casos de uso. El caso de uso describe la manera en que los actores interactúan con el sistema. En general, un caso de uso es, simplemente, un texto escrito que describe el papel de un actor que interactúa con el acontecer del sistema.

Cuando un modelo de proceso iterativo es usado en la creación de software orientado a objetos, las prioridades pueden influir en que funcionalidad del sistema será entregada primero.

10.1.3 PRINCIPIOS DEL ANALISIS

Los investigadores han identificado los problemas del análisis y sus causas y han desarrollado vanas notaciones de modelado y sus correspondientes conjuntos de heurísticas para solucionarlos. Cada método de análisis tiene su punto de vista. Sin embargo, todos los métodos de análisis se relacionan por un conjunto de principios operativos:

1. Debe representarse y entenderse el dominio de información de un problema.
2. Deben definirse las funciones que debe realizar el software.
3. Debe representarse el comportamiento del software (como consecuencia de acontecimientos externos)
4. Deben dividirse los modelos que representan información, función y

comportamiento de manera que se descubran los detalles por capas (o jerárquicamente).

5. El proceso de análisis debería ir desde la información esencial hasta el detalle de la implementación.

Aplicando estos principios, nos aproximamos al problema sistemáticamente. Se examina el dominio de información de manera que pueda entenderse completamente la función. Se emplean modelos para poder comunicar de forma compacta las características de la función y su comportamiento. Se aplica la partición para reducir la complejidad. Son necesarias las visiones esenciales y de implementación del software para acomodar las restricciones lógicas impuestas por los requisitos del procesamiento y las restricciones físicas impuestas por otros elementos del sistema.

Además de los principios operativos de análisis mencionados, se siguen un conjunto de principios directrices para la ingeniería de requisitos:

1. Entendimiento del problema antes de empezar a crear el modelo de análisis. Hay tendencia a precipitarse en busca de una solución, incluso antes de entender el problema. ¡Esto lleva a menudo a un elegante software para el problema equivocado'

2. Desarrollar prototipos que permitan al usuario entender como será la interacción hombre – maquina. Como el concepto de calidad del software se basa a menudo en la opinión sobre la amigabilidad de la interfaz

3. Registrar el origen y la razón de cada requisito. Este es el primer paso para establecer un seguimiento hasta el cliente.

4. Usar múltiples planteamientos de requisitos. La construcción de modelos de datos, funcionales y de comportamiento, proporcionan puntos de vista diferentes. Esto reduce la probabilidad de que se olvide algo y aumenta la probabilidad de reconocer la falta de consistencia.

5. Dar prioridad a los requisitos. Las fechas ajustadas de entrega pueden impedir la implementación de todos los requisitos del software.

6. Trabajar para eliminar la ambigüedad. Como la mayoría de los requisitos están descritos en un lenguaje natural, abunda la oportunidad de ambigüedades. El empleo de revisiones técnicas formales es la manera de descubrir y eliminar la ambigüedad.

10.1.3.1 El dominio de la información

Todas las aplicaciones software pueden denominarse colectivamente procesamiento de datos, Es interesante que este término contenga una clave para nuestro entendimiento de los requisitos del software. El software se construye para procesar datos, para transformar datos de una forma a otra, es decir, para aceptar uncí entrada de información, manipularla de alguna manera y producir una salida de información. Esta declaración fundamental de objetivos es cierta, tanto si construimos software por lotes para un sistema de nóminas, como si construimos software dedicado de tiempo real para controlar el flujo de combustible de un motor de automóvil.

Es importante recalcar, sin embargo, que el software también procesa sucesos. Un suceso representa algún aspecto de control del sistema y no es más que un dato binario (es encendido o apagado, verdadero o falso, está allí o no). Por tanto, los datos (números, caracteres, imágenes, sonidos, etc.) y el control (acontecimientos) residen dentro del dominio de la información de un problema.

El primer principio operativo de análisis requiere el examen del dominio de la información y la creación de un **modelo de datos**. Este dominio contiene tres visiones diferentes de los datos y del control a medida que se procesa cada uno en un programa de computadora:

1. contenido de la información y relaciones
2. Flujo de la información
3. Estructura de la información.

El **contenido de la información** representa los objetos individuales de datos y de control que componen alguna colección mayor de información a la que transforma el software. El contenido de un objeto de control denominado estado del sistema podría definirse mediante una cadena de bits. Cada bit representa un elemento diferente de información que indica si un dispositivo en particular esta en línea o fuera de línea. Los objetos de datos y de control pueden relacionarse con otros objetos de datos o de control.

El **flujo de la información** representa cómo cambian los datos y el control a medida que se mueven dentro de un sistema Como se muestra en la Figura 3. Los objetos de entrada se transforman para intercambiar información (datos y/o control), hasta que se transforman en información de salida. A lo largo de este camino de transformación (o caminos), se puede introducir información adicional de un almacén de datos (por ejemplo, un archivo en disco o memoria intermedia). Las transformaciones que se aplican a los datos son funciones o subfunciones que debe realizar un programa. Los

datos y control que se mueven entre dos funciones definen la interfaz de cada función.

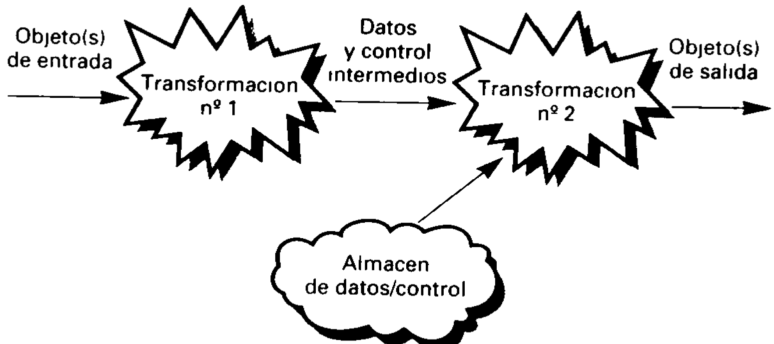


FIGURA 3. Flujo y transformación de la información[3].

La **estructura de la información** representa la organización interna de los elementos de datos o de control.

10.1.3.2 Modelado

Los modelos se crean para entender mejor la entidad que se va a construir. Cuando la entidad es algo físico (un edificio, un avión, una máquina), podemos construir un modelo idéntico en forma, pero más pequeño Sin embargo, cuando la entidad que se va a construir es software, nuestro modelo debe tomar una forma diferente. Debe ser capaz de modelar la información que transforma el software, las funciones (y subfunciones) que permiten que ocurran las transformaciones y el comportamiento del sistema cuando ocurren estas transformaciones. El segundo y tercer principios operativos del análisis requieren la construcción de modelos de función y comportamiento

1. **Modelos funcionales**.

   El software transforma la información y para hacerlo debe realizar al menos tres funciones genéricas: entrada, procesamiento y salida Cuando se crean modelos funcionales de una aplicación, se concentra en funciones específicas del problema. El modelo funcional empieza con un sencillo modelo lo a nivel de contexto Después de una serie de iteraciones se consiguen mas y mas detalles funcionales hasta que se consigue representar una minuciosa definición de toda la funcionalidad del sistema

---

[3] Tomado de Ingeniería del Software. Roger Pressman.

2. **Modelos de comportamiento.**

La mayoría del software responde a los acontecimientos del mundo exterior. Esta característica estimulo-respuesta forma la base del modelo del comportamiento Un programa de computadora siempre está en un estado, un modo de comportamiento observable exteriormente que cambia solo cuando ocurre algún suceso. Un modelo de comportamiento crea una representación de los estados del software y los sucesos que causan que cambie el estado.

Los modelos creados durante el análisis de requisitos desempeñan unos papeles muy importantes:

- El modelo ayuda al analista a entender la informa cien, la función y el comportamiento del sistema haciendo por tanto más fácil y sistemática la tarea de análisis de requisitos

- El modelo se convierte en el punto de mira para la revisión y por tanto la clave para determinar si se ha completado, su consistencia y la precisión de la especificación.

- El modelo se convierte en el fundamento para el diseño, proporcionando al diseñador una representación esencial del software que pueda trasladarse al contexto de la implementación

10.1.3.3 Partición

A menudo los problemas son demasiado grandes o complejos para entenderlos globalmente. Por este motivo, tendemos a hacer una partición (dividir) de estos problemas en partes que puedan entenderse fácilmente y establecer las interacciones entre las partes de manera que se pueda conseguir la función global. El cuarto principio operativo del análisis sugiere que se pueden partir los dominios de la información, funcional y de comportamiento.

En esencia, la partición descompone un problema en sus partes constitutivas. Conceptualmente, establecemos una representación jerárquica de la información o de la función y después partimos el elemento de orden superior (1) exponiendo más detalles cada vez al movernos verticalmente en la jerarquía o (2) descomponiendo el problema si nos movemos horizontalmente en la jerarquía.

10.1.3.4 Visiones esenciales y de implementación

Una visión esencial de los requisitos del software presenta las funciones a conseguir y la información a procesar sin tener en cuenta los detalles de la implementación. Enfocando nuestra atención en la esencia del problema en las primeras fases del análisis de requisitos, dejamos abiertas nuestras opciones para especificar los detalles de implementación durante fases posteriores de especificación de requisitos y diseño del software.
La visión de implementación de los requisitos del software introduce la manifestación en el mundo real de las funciones de procesamiento y las estructuras de la información. En algunos casos, se desarrolla una representación física en la primera fase del diseño del software. Sin embargo, la mayoría de los sistemas basados en computadora se especifican de manera que se acomode a ciertos detalles de implementación.

Ya hemos comentado anteriormente que el análisis de los requisitos del software debería concentrarse en qué es lo que debe hacer el software, en vez de cómo se implementará el procesamiento. Sin embargo, la visión de implementación no debería interpretarse necesariamente como una representación del cómo. Más bien, un modelo de implementación representa el modo actual de operación, es decir, la asignación existente o propuesta de todos los elementos del sistema. El modelo esencial (de función o datos) es genérico en el sentido de que la realización de la función no se indica explícitamente.

10.1.4 CREACIÓN DE PROTOTIPOS DEL SOFTWARE

Hay circunstancias que requieren la construcción de un prototipo al comienzo del análisis, ya que el modelo es el único medio a través del cual se pueden obtener eficazmente los requisitos. El modelo evoluciona después hacia la producción del software.

10.1.4.1 Selección del enfoque de creación de prototipos

El paradigma de creación de prototipos puede ser cerrado o abierto. El enfoque cerrado se denomina a menudo prototipo desechable. Este prototipo sirve únicamente como una basta demostración de los requisitos. Después se desecha y se hace una ingeniería del software con un paradigma diferente. Un enfoque abierto, denominado prototipo evolutivo, emplea el prototipo como primera parte de una actividad de análisis a la que seguirá el

diseño y la construcción. El prototipo del software es la primera evolución del sistema terminado
Antes de poder elegir un enfoque abierto o cerrado, es necesario determinar si se puede crear un prototipo del sistema a construir Se pueden definir vanos factores candidatos a la creación de prototipos: área de aplicación, complejidad, características del cliente y de proyectos.

En general, cualquier aplicación que cree pantallas visuales dinámicas, interactúe intensamente con el ser humano, o demande algoritmos o procesamiento cíe combinaciones que deban crearse de manera progresiva, es un buen candidato para la creación de un prototipo. Sin embargo, estas áreas de aplicación deben ponderarse con la complejidad de la aplicación.
Como el cliente debe interactuar con el prototipo en fases posteriores, es esencial que (1) se destinen recursos del cliente a la evaluación y refinamiento del prototipo, y (2) el cliente sea capaz de tomar decisiones inmediatas sobre los requisitos. Finalmente, la naturaleza del proyecto de desarrollo tendrá una gran influencia en la capacidad de crear un prototipo.

10.1.4.2 Métodos y herramientas para el desarrollo de prototipos

Para que la creación del prototipo de software sea efectiva, debe desarrollarse rápidamente para que el cliente pueda valorar los resultados y recomendar los cambios oportunos. Para poder crear prototipos rápidos, hay disponibles dos clases genéricas de métodos y herramientas:

1. **Técnicas de Cuarta Generación.**
   Las técnicas de cuarta generación comprenden una amplia gama de lenguajes de consulta e informes de bases de datos, generadores de programas y aplicaciones y de otros lenguajes no procedimentales de muy alto nivel. Como las técnicas T4G permiten generar código ejecutable rápidamente, son ideales para la creación rápida de prototipos.

2. **Componentes de software reutilizables.**
   Otro enfoque para crear prototipos rápidos es ensamblar, más que construir el prototipo mediante un conjunto de componentes software existente. La combinación de prototipos con la reutilización de componentes de programa sólo funcionará si se desarrolla un sistema bibliotecario de manera que los componentes existentes estén catalogados y puedan recogerse Debería resaltarse que se puede usar un producto software existente como prototipo de un «nuevo y mejorado» producto competitivo. En cierta manera, ésta es una forma de reutilización en la creación de prototipos software.

10.1.5 ESPECIFICACIÓN

10.1.5.1 Principios de la especificación

1. Separar la funcionalidad de la implementación.
2. Desarrollar un modelo del comportamiento deseado de un sistema que comprenda datos y las respuestas funcionales de un sistema a varios estímulos del entorno.
3. Establecer el contexto en que opera el software especificando la manera en que otros componentes del sistema interactúan con él.
4. Definir el entorno en que va a operar el sistema e indicar como una colección de agentes altamente entrelazados reaccionan a estímulos del entorno (cambios de objetos) producidos por esos agentes.
5. Crear un modelo intuitivo en vez de un diseño o modelo de implementación.
6. Reconocer que «la especificación debe ser tolerante a un posible crecimiento si no es completa». Una especificación es siempre un modelo —una abstracción— de alguna situación real (o prevista) que normalmente suele ser compleja. De ahí que será incompleta y existirá a muchos niveles de detalle.
7. Establecer el contenido y la estructura de una especificación de manera que acepte cambios. Esta lista de principios básicos proporciona la base para representar los requisitos del software. Sin embargo, los principios deben traducirse en realidad. En la siguiente sección examinamos un conjunto de directrices para la creación de una especificación de requisitos.

10.1.5.2 Representación

Ya hemos visto que los requisitos del software pueden especificarse de varias maneras. Sin embargo, si los requisitos se muestran en papel o en un medio electrónico de representación (¡y debería ser casi siempre así!), merece la pena seguir este sencillo grupo de directrices:

- **El formato de la representación y el contenido deberían editar relacionados con el problema**. Se puede desarrollar un esbozo general del contenido de la estancación de los requisitos del software. Sin embargo, las temías de representación contenidas en la especificación es probable que varíen con el área de aplicación.

- **La información contenida dentro de la especificación debería estar escalonada.** Las representaciones deberían revelar capas de información de manera que el lector se pueda mover en el nivel de detalle requerido. La numeración de párrafos y diagramas debería indicar el nivel de detalle que se muestra. A veces, merece la pena presentar la misma información con diferentes niveles de abstracción para ayudar a su comprensión

- **Los diagramas y otras formas de notación deberían restringirse en número y ser consistentes en su empleo.** Las notaciones confusas o inconsistentes, tanto graneas como simbólicas, degradan la comprensión y fomentan errores.

- **Las representaciones deben permitir revisiones.** Seguramente el contenido de una especificación cambiará. Idealmente, debería haber herramientas CASE disponibles para actualizar todas las representaciones afectadas por cada cambio.

10.1.5.3 La especificación de los requisitos del software

La **especificación de los requisitos del software** se produce en la culminación de la tarea de análisis. La función y rendimiento asignados al software como parte cíe la ingeniería de sistemas se retinan estableciendo una completa descripción de la información, una descripción detallada de la función y del comportamiento, una indicación de los requisitos del rendimiento y restricciones del diseño, criterios de validación apropiados y otros datos pertinentes a los requisitos.

La Introducción a la especificación de los requisitos del software establece las metas y objetivos del software describiéndolo en el contexto del sistema basado en computadora. De hecho, la Introducción puede no ser más que el alcance del software en el documento de planificación.

La Descripción de la información proporciona una detallada descripción del problema que el software va a resolver. Se documentan el contenido de la información y sus relaciones, flujo y estructura. Se describen las interfaces hardware, software y humanas para los elementos externos del sistema y para las funciones internas del software.

En la **Descripción funcional** se describen todas las funciones requeridas para solucionar el problema. Se proporciona una descripción del proceso de cada función; se establecen y justifican las restricciones del diseño; se establecen las características del rendimiento; y se incluyen uno o más

diagramas para representar gráficamente la estructura global del software y las interacciones entre las funciones software y otros elementos del sistema. La sección de las especificaciones Descripción del comportamiento examina la operativa del software como consecuencia de acontecimientos externos y características de control generadas internamente.
Probablemente la más importante, e irónicamente, la sección más a menudo ignorada de una especificación de requisitos del software son los criterios de validación.

En muchos casos la Especificación de requisitos del software puede estar acompañada de un prototipo ejecutable (que en algunos casos puede sustituir a la especificación), un prototipo en papel o un Manual de usuario preliminar. El Manual de usuario preliminar presenta al software como una caja negra. Es decir, se pone gran énfasis en las entradas del usuario y las salidas (resultados) obtenidas. El manual puede servir como una valiosa herramienta para descubrir problemas en la interfaz hombre-maquina.

10.1.6 REVISIÓN DE LA ESPECIFICACIÓN

La revisión de la Especificación de requisitos del software es llevada a cabo tanto por el desarrollador del software como por el cliente. Como la especificación forma el fundamento para el diseño y las subsiguientes actividades de la ingeniería del software, se debería poner extremo cuidado al realizar la revisión.

Inicialmente la revisión se lleva a cabo a nivel macroscópico. A este nivel, los revisores intentan asegurarse de que la especificación sea completa, consistente y correcta cuando la información general, funcional y de los dominios del comportamiento son considerados. Asimismo, una exploración completa de cada uno de estos dominios, la revisión profundiza en el detalle, examinando no solo las descripciones superficiales, sino la vía en la que los requisitos son expresados. Por ejemplo, cuando una especificación contiene un «término vago» (por ejemplo, algo, algunas veces, a veces, normalmente, corrientemente, mucho, o principalmente), el revisor señalará la sentencia para su clarificación.

Una vez que se ha completado la revisión, se firma la especificación de requisitos del software por el cliente y el desarrollador. La especificación se conviene en un «contrato» para el desarrollo del software Las peticiones de cambios en los requisitos una vez que se ha finalizado la especificación no se eliminarán, pero el cliente debe saber que cada cambio a posteriori significa una ampliación del ámbito del software, y por tanto pueden aumentar los costes y prolongar los plazos de la planificación temporal del proyecto.

11 CONCEPTOS Y PRINCIPIOS DE DISEÑO

El diseño del software se encuentra en el núcleo técnico de la ingeniería del software y se aplica independientemente del modelo de diseño de software que se utilice. Una vez que se analizan y especifican los requisitos del software, el diseño del software es la primera de las tres actividades técnicas (diseño, generación de código y pruebas) que se requieren para construir y verificar el software. Cada actividad transforma la información de manera que dé lugar por último a un software de computadora validado.

Cada uno de los elementos del modelo de análisis proporciona la información necesaria para crear los cuatro modelos de diseño que se requieren para una especificación completa de diseño. El flujo de información durante el diseño del software se muestra en la Figura 4.

Los requisitos del software, manifestados por los modelos de datos funcionales y de comportamiento, alimentan la tarea del diseño, que produce un diseño de datos, un diseño arquitectónico, un diseño de interfaz y un diseño de componentes.

El diseño de datos transforma el modelo del dominio de información que se crea durante el análisis en las estructuras de datos que se necesitarán para implementar el software. Los objetos de datos y las relaciones definidas en el diagrama relación entidad y el contenido de datos detallado que se representa en el diccionario de datos proporcionan la base de la actividad del diseño de datos. Es posible que parte del diseño de datos tenga lugar junto con el diseño de la arquitectura del software. A medida que se van diseñando cada uno de los componentes del software, van apareciendo más detalles de diseño.

El diseño arquitectónico define la relación entre los elementos estructurales principales del software, los patrones de diseño que se pueden utilizar para lograr los requisitos que se han definido para el sistema, y las restricciones que afectan a la manera en que se pueden aplicar los patrones de diseño arquitectónicos. La representación del diseño arquitectónico puede derivarse de la especificación del sistema, del modelo de análisis y de la interacción del subsistema definido dentro del modelo de análisis.

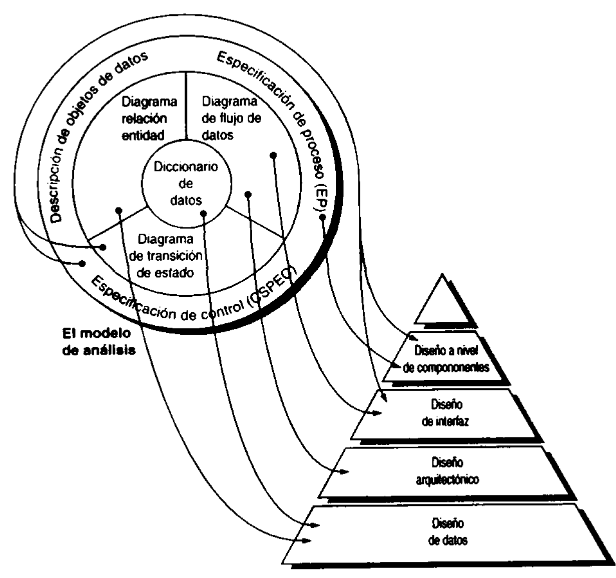


FIGURA 4. Conversión del modelo de análisis en un diseño de software[4].

El diseño de la interfaz describe la manera de comunicarse el software dentro de sí mismo, con sistemas que interoperan dentro de él y con las personas que lo utilizan. Una interfaz implica un flujo de información y un tipo específico de comportamiento. Por tanto, los diagramas de flujo de control y de datos proporcionan gran parte de la información que se requiere para el diseño de la interfaz.

El diseño a nivel de componentes transforma los elementos estructurales de la arquitectura del software en una descripción procedimental de los componentes del software.

La importancia del diseño del software se puede describir con una sola palabra —calidad—. El diseño es el lugar en donde se fomentará la calidad en la ingeniería del software. El diseño proporciona las representaciones del software que se pueden evaluar en cuanto a calidad. El diseño es la única forma de convertir exactamente los requisitos de un cliente en un producto o sistema de software finalizado. El diseño del software sirve como fundamento para todos los pasos siguientes del soporte del software y de la ingeniería del software. Sin un diseño, corremos el riesgo de construir un sistema inestable —un sistema que fallará cuando se lleven a cabo cambios; un sistema que

[4] Tomado de Ingeniería del Software. Roger Pressman.

puede resultar difícil de comprobar; y un sistema cuya calidad no puede evaluarse hasta muy avanzado el proceso, sin tiempo suficiente y con mucho dinero gastado en él—.

11.1 EL PROCESO DEL DISEÑO

El diseño del software es un proceso iterativo mediante el cual los requisitos se traducen en un «plano» para construir el software. Inicialmente, el plano representa una visión holística del software. Esto es, el diseño se representa a un nivel alto de abstracción —un nivel que puede rastrearse directamente hasta conseguir el objetivo del sistema específico y según unos requisitos más detallados de comportamiento, funcionales y de datos—. A medida que ocurren las iteraciones del diseño, el refinamiento subsiguiente conduce a representaciones de diseño a niveles de abstracción mucho más bajos. Estos niveles se podrán rastrear aún según los requisitos, pero la conexión es más sutil.

11.1.1 Diseño y calidad del software

A lo largo de todo el proceso del diseño, la calidad de la evolución del diseño se evalúa con una serie de revisiones técnicas formales. Se sugiere tres características que sirven como guía para la evaluación de un buen diseño:

1. El diseño deberá implementar todos los requisitos explícitos del modelo de análisis, y deberán ajustarse a todos los requisitos implícitos que desea el cliente.
2. El diseño deberá ser una guía legible y comprensible para aquellos que generan código y para aquellos que comprueban y consecuentemente, dan soporte al software.
3. El diseño deberá proporcionar una imagen completa del software, enfrentándose a los dominios de comportamiento, funcionales y de datos desde una perspectiva de implementación.

Con el fin de evaluar la calidad de una representación de diseño, deberán establecerse los criterios técnicos para un buen diseño. Así que se presentan las siguientes directrices:

1. Un diseño deberá presentar una estructura arquitectónica que (1) se haya creado mediante patrones de diseño reconocibles, (2) que esté formada por componentes que exhiban características de buen diseño (aquellas que se abordarán más adelante en este mismo capítulo), y (3) que se puedan implementar de manera evolutiva, facilitando así la

implementación y la comprobación.

2. Un diseño deberá ser modular; esto es, el software deberá dividirse lógicamente en elementos que realicen funciones y subfunciones específicas.
3. Un diseño deberá contener distintas representaciones de datos, arquitectura, interfaces y componentes (módulos).
4. Un diseño deberá conducir a estructuras de datos adecuadas para los objetos que se van a implementar y que procedan de patrones de datos reconocibles.
5. Un diseño deberá conducir a componentes que presenten características funcionales independientes.
6. Un diseño deberá conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo.
7. Un diseño deberá derivarse mediante un método repetitivo y controlado por la información obtenida durante el análisis de los requisitos del software. Estos criterios no se consiguen por casualidad. El proceso de diseño del software fomenta el buen diseño a través de la aplicación de principios de diseño fundamentales, de metodología sistemática y de una revisión cuidadosa.

11.2 PRINCIPIOS DEL DISEÑO

El diseño de software es tanto un proceso como un modelo. El proceso de diseño es una secuencia de pasos que hacen posible que el diseñador describa todos los aspectos del software que se va construir. Sin embargo, es importante destacar que el proceso de diseño simplemente no es un recetario. Un conocimiento creativo, experiencia en el tema, un sentido de lo que hace que un software sea bueno, y un compromiso general con la calidad son factores críticos de éxito para un diseño competente.

Los principios básicos de diseño hacen posible que se navegue por el proceso de diseño. Davis sugiere un conjunto de principios para el diseño del software, los cuales han sido adaptados y ampliados en la lista siguiente:

- **En el proceso de diseño no deberá utilizarse orejeras.** Un buen diseñador deberá tener en cuenta enfoques alternativos, juzgando todos los que se basan en los requisitos del problema, los recursos disponibles para realizar el trabajo y los conceptos de diseño presentados. El diseño deberá poderse rastrear hasta el modelo de análisis. Dado que un solo elemento del modelo de diseño suele hacer un seguimiento de los múltiples requisitos, es necesario tener un

medio de rastrear cómo se han satisfecho los requisitos por el modelo de diseño.

- **El diseño no deberá inventar nada que ya esté inventado.** Los sistemas se construyen utilizando un conjunto de patrones de diseño, muchos de los cuales probablemente ya se han encontrado antes. Estos patrones deberán elegirse siempre como una alternativa para reinventar. Hay poco tiempo y los recursos son limitados. El tiempo de diseño se deberá invertir en la representación verdadera de ideas nuevas y en la integración de esos patrones que ya existen.

- **El diseño deberá minimizar el camino entre el software y el problema** como si de la misma vida real se tratara. Es decir, la estructura del diseño del software (siempre que sea posible) imita la estructura del dominio del problema.

- **El diseño deberá presentar uniformidad e integración.** Un diseño es uniforme si parece que fue una persona la que lo desarrolló por completo. Las reglas de estilo y de formato deberán definirse para un equipo de diseño antes de comenzar el trabajo sobre el diseño. Un diseño se integra si se tiene cuidado a la hora de definir interfaces entre los componentes del diseño. El diseño deberá estructurarse para admitir cambios. Los conceptos de diseño estudiados en la sección siguiente hacen posible un diseño que logra este principio.

- **El diseño deberá estructurarse para degradarse poco a poco. incluso cuando se enfrenta con datos, sucesos o condiciones de operación aberrantes.** Un software bien diseñado no deberá nunca explotar como una «bomba». Deberá diseñarse para adaptarse a circunstancias inusuales, y si debe terminar de funcionar, que lo haga de forma suave.

- **El diseño no es escribir código y escribir código no es diseñar**. Incluso cuando se crean diseños procedimentales para componentes de programas, el nivel de abstracción del modelo de diseño es mayor que el código fuente. Las únicas decisiones de diseño realizadas a nivel de codificación se enfrentan con pequeños datos de implementación que posibilitan codificar el diseño procedimental.

- **El diseño deberá evaluarse en función de la calidad mientras se va creando, no después de terminarlo.**

- **El diseño deberá revisarse para minimizar los errores conceptuales** (semánticos). A veces existe la tendencia de centrarse

en minucias cuando se revisa el diseño, olvidándose del bosque por culpa de los árboles. Un equipo de diseñadores deberá asegurarse de haber afrontado los elementos conceptuales principales antes de preocuparse por la sintaxis del modelo del diseño.

Cuando los principios de diseño descritos anteriormente se aplican adecuadamente, el ingeniero del software crea un diseño que muestra los factores de calidad tanto internos como externos. Los factores de calidad son esas propiedades del software que pueden ser observadas fácilmente por los usuarios (por ejemplo, velocidad, fiabilidad, grado de corrección, usabilidad).

11.3 CONCEPTOS DEL DISEÑO

11.3.1 Abstracción

Cuando se tiene en consideración una solución modular a cualquier problema, se pueden exponer muchos niveles de abstracción. En el nivel más alto de abstracción, la solución se pone como una medida extensa empleando el lenguaje del entorno del problema. En niveles inferiores de abstracción, se toma una orientación más procedimental. La terminología orientada a problemas va emparejada con la terminología orientada a la implementación en un esfuerzo por solucionar el problema. Finalmente, en el nivel más bajo de abstracción, se establece la solución para poder implantarse directamente. Wasserman proporciona una definición útil: “La noción psicológica de «abstracción» permite concentrarse en un problema a algún nivel de generalización sin tener en consideración los datos irrelevantes de bajo nivel, la utilización de la abstracción también permite trabajar con conceptos y términos que son familiares en el entorno del problema sin tener que transformarlos en una estructura no familiar”

Cada paso del proceso del software es un refinamiento en el nivel de abstracción de la solución del software. Durante la ingeniería del sistema, el software se asigna como un elemento de un sistema basado en computadora. Durante el análisis de los requisitos del software, la solución del software se establece en estos términos- «aquellos que son familiares en el entorno del problema». A medida que nos adentramos en el proceso de diseño, se reduce el nivel de abstracción. Finalmente el nivel de abstracción más bajo se alcanza cuando se genera el código fuente.

A medida que vamos entrando en diferentes niveles de abstracción, trabajamos para crear abstracciones procedimentales y de datos. Una abstracción procedimental es una secuencia nombrada de instrucciones que

tiene una función específica y limitada. Una abstracción de datos es una colección nombrada de datos que describe un objeto de datos). La abstracción de control es la tercera forma de abstracción que se utiliza en el diseño del software. Al igual que las abstracciones procedimentales y de datos, este tipo de abstracción implica un mecanismo de control de programa sin especificar los datos internos.

11.3.2 Refinamiento

El refinamiento paso a paso es una estrategia de diseño descendente propuesta originalmente por Niklaus Wirth. El desarrollo de un programa se realiza refinando sucesivamente los niveles de detalle procedimentales. Una jerarquía se desarrolla descomponiendo una sentencia macroscópica de función (una abstracción procedimental) paso a paso hasta alcanzar las sentencias del lenguaje de programación. Wirth proporciona una visión general de este concepto:

"En cada paso, se descompone una o varias instrucciones del programa dado en instrucciones más detalla das. Esta descomposición sucesiva o refinamiento de especificaciones termina cuando todas las instrucciones se expresan en función de cualquier computadora subyacente o de cualquier lenguaje de programación. De la misma manera que refinan las tareas, los datos también se tienen que refina, descomponer o estructurar, y es natural refinar el programa y las especificaciones de los datos en paralelo.

El proceso de refinamiento de programas propuesto por Wirth es análogo al proceso de refinamiento y de partición que se utiliza durante el análisis de requisitos. La diferencia se encuentra en el nivel de detalle de implementación que se haya tomado en consideración, no en el enfoque.

El refinamiento verdaderamente es un proceso de elaboración. Se comienza con una sentencia de función (o descripción de información) que se define a un nivel alto de abstracción. Esto es, la sentencia describe la función o información conceptualmente, pero no proporciona información sobre el funcionamiento interno de la información. El refinamiento hace que el diseñador trabaje sobre la sentencia original, proporcionando cada vez más detalles a medida que van teniendo lugar sucesivamente todos y cada uno de los refinamientos (elaboración).

11.3.3 Modularidad

El concepto de modularidad se ha ido exponiendo desde hace casi cinco

décadas en el software de computadora. La arquitectura de computadora expresa la modularidad; es decir, el software se divide en componentes nombrados y abordados por separado, llamados frecuentemente módulos, que se integran para satisfacer los requisitos del problema.

Se ha afirmado que la modularidad es el único atributo del software que permite gestionar un programa intelectualmente. El software monolítico (es decir, un programa grande formado por un único módulo) no puede ser entendido fácilmente por el lector La cantidad de rutas de control, la amplitud de referencias, la cantidad de variables y la complejidad global hará que el entendimiento esté muy cerca de ser imposible.

Ahora, otra consideración que se toma en cuanta es definir el tamaño de un modulo, para esta tarea podemos valernos de los métodos utilizados para definir los módulos dentro de un sistema. Meyer define cinco criterios que nos permitirán evaluar un método de diseño en relación con la habilidad de definir un sistema modular efectivo:

1. **Capacidad de descomposición modular**. Si un método de diseño proporciona un mecanismo sistemático para descomponer el problema en subproblemas, reducirá la complejidad de todo el problema, consiguiendo de esta manera una solución modular efectiva
2. **Capacidad de empleo de componentes modulares**. Si un método de diseño permite ensamblar los componentes de diseño (reusables) existentes en un sistema nuevo, producirá una solución modular que no inventa nada ya inventado
3. **Capacidad de comprensión modular**. Si un módulo se puede comprender como una unidad autónoma (sin referencias a otros módulos) será más fácil de construir y de cambiar
4. **Continuidad modular.** Si pequeños cambios en los requisitos del sistema provocan cambios en los módulos individuales, en vez de cambios generalizados en el sistema, se minimizará el impacto de los efectos secundarios de los cambios.
5. **Protección modular**. Si dentro de un módulo se produce una condición aberrante y sus efectos se limitan a ese módulo, se minimizará el impacto de los efectos secúndanos inducidos por los errores.

11.3.4 Arquitectura del software

La arquitectura del software alude a la «estructura global del software y a las formas en que la estructura proporciona la integridad conceptual de un sistema». En su forma más simple, la arquitectura es la estructura jerárquica de los componentes del programa (módulos), la manera en que los

componentes interactúan y la estructura de datos que van a utilizar los componentes. Sin embargo, en un sentido más amplio, los componentes se pueden generalizar para representar los elementos principales del sistema y sus interacciones.

Un objetivo del diseño del software es derivar una representación arquitectónica de un sistema. Esta representación sirve como marco de trabajo desde donde se llevan a cabo actividades de diseño más detalladas. Un conjunto de patrones arquitectónicos permiten que el ingeniero del software reutilice los conceptos a nivel de diseño.

Shaw y Garlan describen un conjunto de propiedades que deberán especificarse como parte de un diseño arquitectónico:

- **Propiedades estructurales.** Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (por ejemplo, módulos, objetos, filtros) y la manera en que esos componentes se empaquetan e interactúan unos con otros. Por ejemplo, los objetos se empaquetan para encapsular tanto los datos como el procesamiento que manipula los datos e interactúan mediante la invocación de métodos.

- **Propiedades extra-funcionales.** La descripción del diseño arquitectónico deberá ocuparse de cómo la arquitectura de diseño consigue los requisitos para el rendimiento, capacidad, fiabilidad, seguridad, capacidad de adaptación y otras características del sistema.

- **Familias de sistemas relacionados.** El diseño arquitectónico deberá dibujarse sobre patrones repetibles que se basen comúnmente en el diseño de familias de sistemas similares. En esencia, el diseño deberá tener la habilidad de volver a utilizar los bloques de construcción arquitectónicos.

Dada la especificación de estas propiedades, el diseño arquitectónico se puede representar mediante uno o más modelos diferentes. Los modelos estructurales representan la arquitectura como una colección organizada de componentes de programa. Los modelos del marco de trabajo aumentan el nivel de abstracción del diseño en un intento de identificar los marcos de trabajo (patrones) repetibles del diseño arquitectónico que se encuentran en tipos similares de aplicaciones. Los modelos dinámicos tratan los aspectos de comportamiento de la arquitectura del programa, indicando cómo puede cambiar la estructura o la configuración del sistema en función de los acontecimientos externos. Los modelos de proceso se centran en el diseño del proceso técnico de negocios que tiene que adaptar el sistema.

Finalmente los modelos funcionales se pueden utilizar para representar la jerarquía funcional de un sistema.

Se ha desarrollado un conjunto de lenguajes de descripción arquitectónica para representar los modelos destacados anteriormente. Aunque se han propuesto muchos diferentes, la mayoría proporcionan mecanismos para describir los componentes del sistema y la manera en que se conectan unos con otros.

11.3.5 Jerarquía de control

La jerarquía de control, denominada también estructura de programa, representa la organización de los componentes de programa (módulos) e implica una jerarquía de control. No representa los aspectos procedimentales del software, ni se puede aplicar necesariamente a todos los estilos arquitectónicos.

Para representar la jerarquía control de aquellos estilos arquitectónicos que se avienen a la representación se utiliza un conjunto de notaciones diferentes. El diagrama más común es el de forma de árbol (Fig. 5) que representa el control jerárquico para las arquitecturas de llamada y de retorno. Según la Figura 5, la profundidad y la anchura proporcionan una indicación de la cantidad de niveles de control y el ámbito de control global, respectivamente. El grado de salida es una medida del número de módulos que se controlan directamente con otro módulo. El grado de entrada indica la cantidad de módulos que controlan directamente un módulo dado.

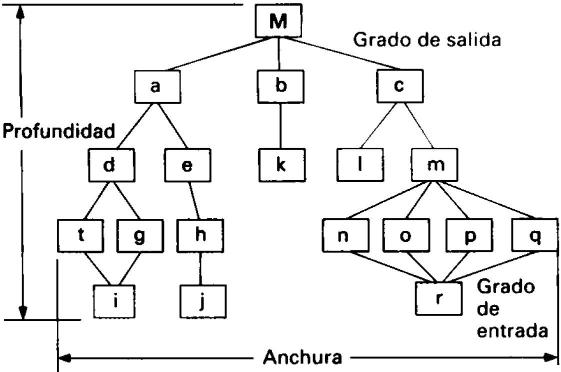


FIGURA 5**. Terminologías de estructura para un estilo arquitectónico de llamada y retorno**[5]**.**

La relación de control entre los módulos se expresa de la manera siguiente:

[5] Tomado de Ingeniería del Software. Roger Pressman.

se dice que un módulo que controla otro módulo es superior a él, e inversamente, se dice que un módulo controlado por otro módulo es subordinado del controlador

La jerarquía de control también representa dos características sutiles diferentes de la arquitectura del software: visibilidad y conectividad. La visibilidad indica el conjunto de componentes de programa que un componente dado puede invocar o utilizar como datos, incluso cuando se lleva a cabo indirectamente.

11.3.6 Estructura de datos

La estructura de datos es una representación de la relación lógica entre elementos individuales de datos. Como la estructura de la información afectará invariablemente al diseño procedimental final, la estructura de datos es tan importante como la estructura de programa para la representación de la arquitectura del software.

La estructura dicta las alternativas de organización, métodos de acceso, grado de capacidad de asociación y procesamiento de la información. Se han dedicado libros enteros a estos temas, y un estudio más amplio sobre este tema queda fuera del ámbito de este libro. Sin embargo, es importante entender la disponibilidad de métodos clásicos para organizar la información y los conceptos que subyacen a las jerarquías de información.

La organización y complejidad de una estructura de datos están limitadas únicamente por la ingenuidad del diseñador. Sin embargo, existe un número limitado de estructuras de datos clásicas que componen los bloques de construcción para estructuras más sofisticadas.

Un elemento escalar es la estructura de dalos mas simple. Como su nombre indica, un elemento escalar representa un solo elemento de información que puede ser tratado por un identificador; es decir, se puede lograr acceso especificando un sola dirección en memoria. El tamaño y formato de un elemento escalar puede variar dentro de los límites que dicta el lenguaje de programación. Por ejemplo, un elemento escalar puede ser: una entidad lógica de un bit de tamaño; un entero o número de coma flotante con un tamaño de 8 a 64 bits; una cadena de caracteres de cientos o miles de bytes.

Cuando los elementos escalares se organizan como una lista o grupo contiguo, se forma un vector secuencial. Los vectores son las estructuras de datos más comunes y abren la puerta a la indexación variable de la información. Cuando el vector secuencial se amplía a dos, tres y por último a

un número arbitrario de dimensiones, se crea un espacio n-dimensional. El espacio n-dimensional más común es la matriz bidimensional. En muchos lenguajes de programación, un espacio n-dimensional se llama array.

Es importante destacar que las estructuras de datos, al igual que las estructuras de programas, se pueden representar d diferentes niveles de abstracción. Por ejemplo, una pila es un modelo conceptual de una estructura de datos que se puede implementar como un vector o una lista enlazada Dependiendo del nivel de detalle del diseño, los procesos internos de la pila pueden especificarse o no.

11.4 DISEÑO MODULAR EFECTIVO

Los conceptos fundamentales de diseño descritos sirven para incentivar diseños modulares De hecho, la modularidad se ha convertido en un enfoque aceptado en todas las disciplinas de ingeniería. Un diseño modular reduce la complejidad, facilita los cambios (un aspecto crítico de la capacidad de mantenimiento del software), y da como resultado una implementación más fácil al fomentar el desarrollo paralelo de las diferentes partes de un sistema.

11.4.1 Independencia funcional

El concepto de independencia funcional es la suma de la modularidad y de los conceptos de abstracción y ocultación de información En referencias obligadas sobre el diseño del software. Parnas y Wirth aluden a las técnicas de refinamiento que mejoran la independencia de módulos.

La independencia funcional se consigue desarrollando módulos con una función «determinante» y una «aversión» a una interacción excesiva con otros módulos Dicho de otra manera, queremos diseñar el software de manera que cada módulo trate una subfunción de requisitos y tenga una interfaz sencilla cuando se observa desde otras partes de la estructura del programa. Es justo preguntarse por qué es importante la independencia.

El software con una modularidad efectiva, es decir, módulos independientes, es más fácil de desarrollar porque la función se puede compartimentar y las interfaces se simplifican (tengamos en consideración las ramificaciones cuando el desarrollo se hace en equipo). Los módulos independientes son más fáciles de mantener (y probar) porque se limitan los efectos secundarios originados por modificaciones de diseño/código; porque se reduce la propagación de errores; y porque es posible utilizar módulos usables. En resumen, la independencia funcional es la clave para un buen diseño y el diseño es la clave para la calidad del software.

La independencia se mide mediante dos criterios cualitativos: la cohesión y el acoplamiento. La cohesión es una medida de la fuerza relativa funcional de un módulo. El acoplamiento es una medida de la independencia relativa entre los módulos.

11.4.2 Cohesión

La cohesión es una extensión natural del concepto de ocultación de información. Un módulo cohesivo lleva a cabo una sola tarea dentro de un procedimiento de software, lo cual requiere poca interacción con los procedimientos que se llevan a cabo en otras partes de un programa. Dicho de manera sencilla, un módulo cohesivo deberá (idealmente) hacer una sola cosa.

La cohesión se puede representar como un «espectro». Siempre debemos buscar la cohesión más alta, aunque la parte media del espectro suele ser aceptable. La escala de cohesión no es lineal. Es decir, la parte baja de la cohesión es mucho «peor» que el rango medio, que es casi tan bueno como la parte alta de la escala. En la práctica, un diseñador no tiene que preocuparse de categorizar la cohesión en un módulo específico. Más bien, se deberá entender el concepto global, y así se deberán evitar los niveles bajos de cohesión al diseñar los códigos.

En la parte inferior (y no deseable) del espectro, encontraremos un módulo que lleva a cabo un conjunto de tareas que se relacionan con otras débilmente, si es que tienen algo que ver. Tales módulos se denominan coincidencialmente cohesivos. Cuando un módulo con tiene tareas que están relacionadas entre sí por el hecho de que todas deben ser ejecutadas en el mismo intervalo de tiempo, el módulo muestra cohesión temporal.

Los niveles moderados de cohesión están relativamente cerca unos de otros en la escala de dependencia modular. Cuando los elementos de procesamiento de un módulo están relacionados, y deben ejecutarse en un orden específico, existe cohesión procedimental. Cuando todos los elementos de procesamiento se centran en un área de una estructura de datos, tenemos presente una cohesión de comunicación Una cohesión alta se caracteriza por un módulo que realiza una única tarea.

Como ya se ha mencionado anteriormente, no es necesario determinar el nivel preciso de cohesión. Más bien, es importante intentar conseguir una cohesión alta y reconocer cuando hay poca cohesión para modificar el diseño del software y conseguir una mayor independencia funcional.

11.4.3 Acoplamiento

El acoplamiento es una medida de interconexión entre módulos dentro de una estructura de software. El acoplamiento depende de la complejidad de interconexión entre los módulos, el punto donde se realiza una entrada o referencia a un módulo, y los datos que pasan a través de la interfaz.

En el diseño del software, intentamos conseguir el acoplamiento más bajo posible. Una conectividad sencilla entre los módulos da como resultado un software más fácil de entender y menos propenso a tener un «efecto ola» causado cuando ocurren errores en un lugar y se propagan por el sistema.

El diagnóstico de problemas en estructuras con acoplamiento común es costoso en tiempo y es difícil. Sin embargo, esto no significa necesariamente que el uso de datos globales sea «malo». Significa que el diseñador del software deberá ser consciente de las consecuencias posibles del acoplamiento común y tener especial cuidado de prevenirse de ellos.

El grado más alto de acoplamiento, acoplamiento de contenido, se da cuando un módulo hace uso de datos o de información de control mantenidos dentro de los límites de otro módulo. En segundo lugar, el acoplamiento de contenido ocurre cuando se realizan bifurcaciones a mitad de módulo. Este modo de acoplamiento puede y deberá evitarse.

11.5 DOCUMENTACIÓN DEL DISEÑO

La Especificación del diseño aborda diferentes aspectos del modelo de diseño y se completa a medida que el diseñador refina su propia representación del software. En primer lugar, se describe el ámbito global del esfuerzo realizado en el diseño. La mayor parte de la información que se presenta aquí se deriva de la Especificación del sistema y del modelo de análisis (Especificación de los requisitos del software).

A continuación, se especifica el diseño de datos. Se definen también las estructuras de las bases de datos, cualquier estructura externa de archivos, estructuras internas de datos y una referencia cruzada que conecta objetos de datos con archivos específicos.

El diseño arquitectónico indica cómo se ha derivado la arquitectura del programa del modelo de análisis. Además, para representar la jerarquía del módulo se utilizan gráficos de estructuras.

Se representa el diseño de interfaces internas y externas de programas y se describe un diseño detallado de la interfaz hombre-máquina. En algunos casos, se podrá representar un prototipo detallado del IGU.

Los componentes —elementos de software que se pueden tratar por separado tales como subrutinas, funciones o procedimientos— se describen inicialmente con una narrativa de procesamiento en cualquier idioma (castellano, Inglés). La narrativa de procesamiento explica la función procedimental de un componente (módulo). Posteriormente, se utiliza una herramienta de diseño procedimental para convertir esa estructura en una descripción estructural.

La Especificación del diseño contiene una referencia cruzada de requisitos. El propósito de esta referencia cruzada (normalmente representada como una matriz simple) es: (1) establecer que todos los requisitos se satisfagan mediante el diseño del software, e (2) indicar cuales son los componentes críticos para la implementación de requisitos específicos.

El primer paso en el desarrollo de la documentación de pruebas también se encuentra dentro del documento del diseño. Una vez que se han establecido las interfaces y la estructura de programa podremos desarrollar las líneas generales para comprobar los módulos individuales y la integración de todo el paquete. En algunos casos, esta sección se podrá borrar de la Especificación del diseño.

Las restricciones de diseño, tales como limitaciones físicas de memoria o la necesidad de una interfaz externa especializada, podrán dictar requisitos especiales para ensamblar o empaquetar el software. Consideraciones especiales originadas por la necesidad de superposición de programas, gestión de memoria virtual, procesamiento de alta velocidad u otros factores podrán originar modificaciones en diseño derivadas del flujo o estructura de la información. Además, esta sección describe el enfoque que se utilizará para transferir software al cliente.

La última sección de la Especificación del diseño contiene datos complementarios. También se presentan descripciones de algoritmos, procedimientos alternativos, datos tabulares, extractos de otros documentos y otro tipo de información relevante, todos mediante notas especiales o apéndices separados. Será aconsejable desarrollar un Manual preliminar de Operaciones/Instalación e incluirlo como apéndice para la documentación del diseño.

12 CONCEPTOS Y PRINCIPIOS ORIENTADOS A OBJETOS

Actualmente una de las áreas más importantes en la industria y en el ámbito académico es la orientación a objetos. La orientación a objetos promete mejoras de amplio alcance en la forma de diseño, desarrollo y mantenimiento del software ofreciendo una solución a los problemas y preocupaciones que siempre han existido en el desarrollo de software: la falta de portabilidad del código y reusabilidad, código que es difícil de modificar, ciclos de desarrollo largos y técnicas de codificación no intuitivas.

Las tecnologías de objetos llevan a reutilizar, y la reutilización (de componente de software) lleva a un desarrollo de software más rápido y a programas de mejor calidad. El software orientado a objetos es más fácil de mantener debido a que su estructura es inherentemente poco acoplada. Esto lleva a menores efectos colaterales cuando se deben hacer cambios y provoca menos frustración en el ingeniero del software y en el cliente. Además, los sistemas orientados a objetos son más fáciles de adaptar y más fácilmente escalables (por ejemplo: pueden crearse grandes sistemas ensamblando subsistemas reutilizables).

12.1 CONCEPTOS DE ORIENTACIÓN A OBJETOS

Para entender un desarrollo de software orientado a objetos primero debemos aclarar algunos términos y conceptos fundamentales como por ejemplo: ¿Qué es un objeto? ¿Qué es método orientado a objetos? En los siguientes párrafos se dejaran estos y otros conceptos en claro.

Para entender mejor la visión orientada a objetos, consideremos un ejemplo claro y puntual de un objeto del mundo real, una Lavadora. Lavadora es un miembro o instancia de una clase mucho más grande de objetos que llamaremos Electrodomésticos. Un conjunto de atributos genéricos puede asociarse con cada objeto, en la clase Electrodomésticos. Por ejemplo, todo Electrodoméstico tiene una marca, modelo, capacidad, número de serie y motor, entre otros muchos posibles atributos. Éstos son aplicables a cualquier elemento sobre el que se hable, una nevera, un horno o una lavadora. Como Lavadora es un miembro de la clase Electrodomésticos, hereda todos los atributos definidos para dicha clase. Este concepto se ilustra en la Figura utilizando una notación conocida como UML.

Una vez definida la clase, los atributos pueden reutilizarse al crear nuevas instancias de la clase. Por ejemplo, supongamos que debemos definir un nuevo objeto llamado Nevera que es un miembro de la clase Electrodomésticos.

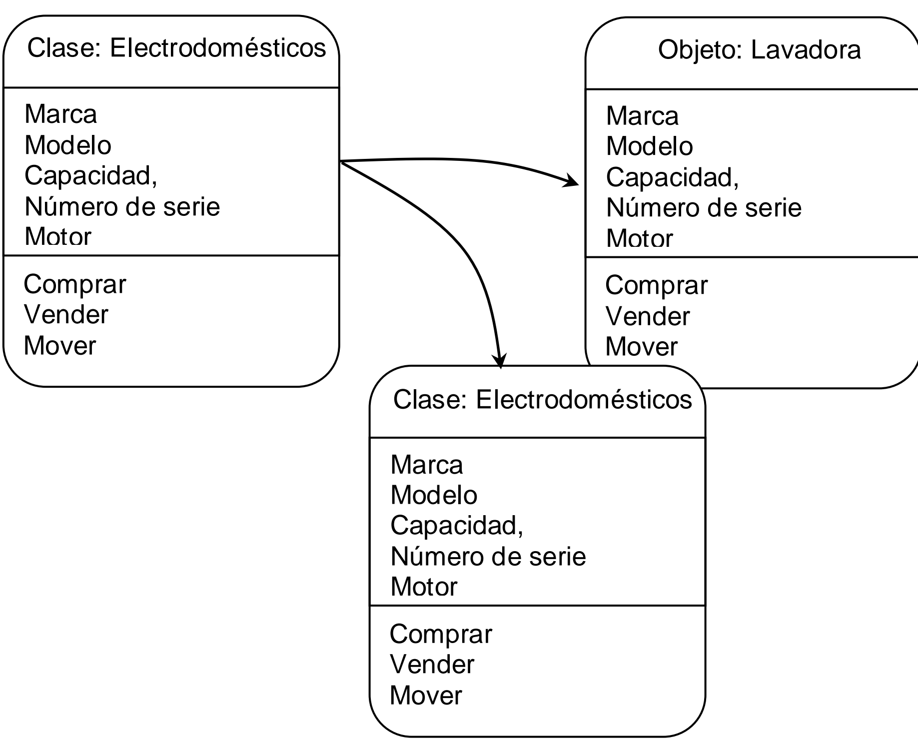


FIGURA 6**. Herencia operaciones de clase a objeto**[6]

Ahora bien, todo objeto en la clase Electrodomésticos puede manipularse de varias maneras. Puede comprarse y venderse, modificarse físicamente o moverse de un lugar a otro. Cada una de estas operaciones modificará uno o más atributos del objeto. Por ejemplo, si el atributo localización es un dato compuesto definido como:

localización = edificio + piso + habitación

Entonces una operación denominada mover modificaría uno o más de los elementos dato (edificio, piso o habitación) que conforman el atributo localización. Para hacer esto, mover debe tener «conocimiento» sobre estos elementos. La operación mover puede usarse para una Lavadora o una nevera, debido a que ambas son instancias de la clase Mobiliario. Todas las operaciones válidas de la Electrodomésticos Mobiliario están «conectadas» a la definición del objeto como se muestra en la Figura 6 y son heredadas por todas las instancias de esta clase.

[6] Tomado de Aprendiendo UML en 24 horas. Joseph Schmuller.

12.1.1 CLASES Y OBJETOS

Una clase es un concepto Orientado a Objetos que encapsula las abstracciones de datos y procedimientos que se requieren para describir el contenido y comportamiento de alguna entidad del mundo real.

Las abstracciones de datos (atributos) que describen la clase están encerradas dentro de abstracciones procedimentales llamadas operaciones que son capaces de manipular datos de alguna manera. La única forma de alcanzar los atributos es ir a través de alguno de los métodos que forman la muralla. Por lo tanto, la clase encapsula datos y el proceso que manipula los datos. Esto posibilita la ocultación de información y reduce el impacto de efectos colaterales asociados a cambios.

Puesto de otra manera, una clase es una descripción generalizada que describe una colección de objetos similares. Por definición, todos los objetos que existen dentro de una clase heredan sus atributos y las operaciones disponibles para la manipulación de los atributos. Una superclase es una colección de clases y una subclase es una instancia de una clase.

Estas definiciones implican la existencia de una jerarquía de clases en la cual los atributos y operaciones de la superclase son heredados por subclases que pueden añadir, cada una de ellas, atributos privados y métodos.

12.1.2 PROPIEDADES

Distinguen un objeto determinado de los restantes que forman parte de la misma organización y tiene valores que dependen de la propiedad de que se trate. Las propiedades de un objeto pueden ser heredadas a sus descendientes en la organización. Todo objeto puede tener cierto número de propiedades, cada una de las cuales tendrá, a su vez, uno o varios valores. Un objeto puede tener una propiedad de maneras diferentes: *Propiedades propias:* Están formadas dentro de la cápsula del objeto. *Propiedades heredadas:* Están definidas en un objeto diferente, antepasado de éste.

12.1.3 MÉTODOS

Una operación que realiza acceso a los datos. Podemos definir método como un programa procedimental o procedural escrito en cualquier lenguaje, que está asociado a un objeto determinado y cuya ejecución sólo puede desencadenarse a través de un mensaje recibido por éste o por sus descendientes.

Si los métodos son programas, se deduce que podrían tener argumentos, o parámetros. Puesto que los métodos pueden heredarse de unos objetos a otros, un objeto puede disponer de un método de dos maneras diferentes: Métodos propios. Están incluidos dentro de la cápsula del objeto. Métodos heredados. Están definidos en un objeto diferente, antepasado de éste (padre,"abuelo", etc.).

12.1.4 MENSAJES

Los mensajes son el medio a través del cual interactúan los objetos. Usando la terminología presentada en la sección precedente, un mensaje estimula la ocurrencia de cierto comportamiento en el objeto receptor. El comportamiento se realiza cuando se ejecuta una operación.

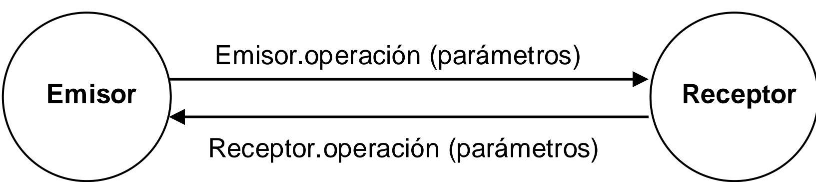


FIGURA 7. **Paso de mensajes entre objetos**[7]

En la Figura 7 se ilustra esquemáticamente la interacción entre objetos. Una operación dentro de un objeto emisor genera un mensaje de la forma:

destino.operación (parámetros)

Donde destino define el objeto receptor el cual es estimulado por el mensaje, operación se refiere al método que recibe el mensaje y parámetros proporciona información requerida para el éxito de la operación.

12.1.5 Encapsulamiento, herencia y polimorfismo

Aunque la estructura y terminología ya introducida diferencian los sistemas Orientado a Objetos a partir de sus componentes convencionales, hay tres características de los sistemas orientados a objetos que los hacen únicos. Como se ha mencionado las clases y los objetos derivados de ella encapsulan los datos y las operaciones que trabajan sobre estos en un único paquete. Esto proporciona un número importante de beneficios:

- Los detalles de implementación interna de datos y procedimientos

[7] Tomado de Ingeniería del Software. Roger Pressman.

están ocultos al mundo exterior (ocultación de la información). Esto reduce la propagación de efectos colaterales cuando ocurren cambios.

- Las estructuras de datos y las operaciones que las manipulan están mezcladas en una entidad sencilla: la clase. Esto facilita la reutilización de componentes.

- Las interfaces entre objetos encapsulados están simplificadas. Un objeto que envía un mensaje no tiene que preocuparse de los detalles de las estructuras de datos internas en el objeto receptor, lo que simplifica la interacción y hace que el acoplamiento del sistema tienda a reducirse.

La herencia es una de las diferencias clave entre sistemas convencionales y sistemas Orientado a Objetos. Una subclase Y hereda todos los atributos y operaciones asociadas con su superclase X. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para X están inmediatamente disponibles para Y.

Cualquier cambio en los datos u operaciones contenidas dentro de una superclase es heredado inmediatamente por todas las subclases que se derivan de la superclase. Debido a esto, la jerarquía de clases se convierte en un mecanismo a través del cual los cambios (a altos niveles) pueden propagarse inmediatamente a través de todo el sistema.

Es importante destacar que en cada nivel de la jerarquía de clases, pueden añadirse nuevos atributos y operaciones a aquellos que han sido heredados de niveles superiores de la jerarquía. De hecho, cada vez que se debe crear una nueva clase se tienen varias opciones vanas opciones:

- La clase puede diseñarse y construirse de la nada Esto es, no se usa la herencia.
- La jerarquía de clases puede ser rastreada para determinar si una clase ascendiente contiene la mayoría de los atributos y operaciones requeridas La nueva clase hereda de su clase ascendiente, y pueden añadirse otros elementos si hacen falta
- La jerarquía de clases puede reestructurarse de tal manera que los atributos y operaciones requeridos puedan ser heredados por la nueva clase.
- Las características de una clase existente pueden sobrescribirse y se pueden implementar versiones privadas de atributos u operaciones para la nueva clase.

En algunos casos, es tentador heredar algunos atributos y operaciones de una clase y otros de otra clase Esta acción se llama herencia la múltiple y es

controvertida. En general, la herencia múltiple complica la jerarquía de clases y crea problemas potenciales en el control de la configuración. Como las secuencias de herencia múltiple son más difíciles de seguir, los cambios en la definición de una clase que reside en la parte superior de la jerarquía pueden tener impactos no deseados originalmente en las clases definidas en zonas inferiores de la arquitectura.

El polimorfismo es una característica que reduce en gran medida el esfuerzo necesario para extender un sistema Orientado a Objetos. Esto hace referencia cuando una operación tiene el mismo nombre en diferentes clases, en esta orientación, cada clase sabe como realizar dicha operación. El polimorfismo permite mantener la terminología dentro del proyecto sin tener que crear palabras artificiales para sustentar la unicidad innecesaria de términos.

12.2 IDENTIFICACIÓN DE LOS ELEMENTOS DE UN MODELO DE OBJETOS

12.2.1 Identificación de clases y objetos

Se puede empezar a identificar objetos examinando el planteamiento del problema. Los objetos se determinan subrayando cada nombre o cláusula nominal e introduciéndola en una tabla simple. Los sinónimos deben destacarse. Si se requiere del objeto que implemento una solución, entonces éste formará parte del espacio de solución; en caso de que el objeto se necesite solamente para describir una solución, éste forma parte del espacio del problema. Seguida a esta tarea se debe buscar con que se relaciona o manifiesta dicho objeto.

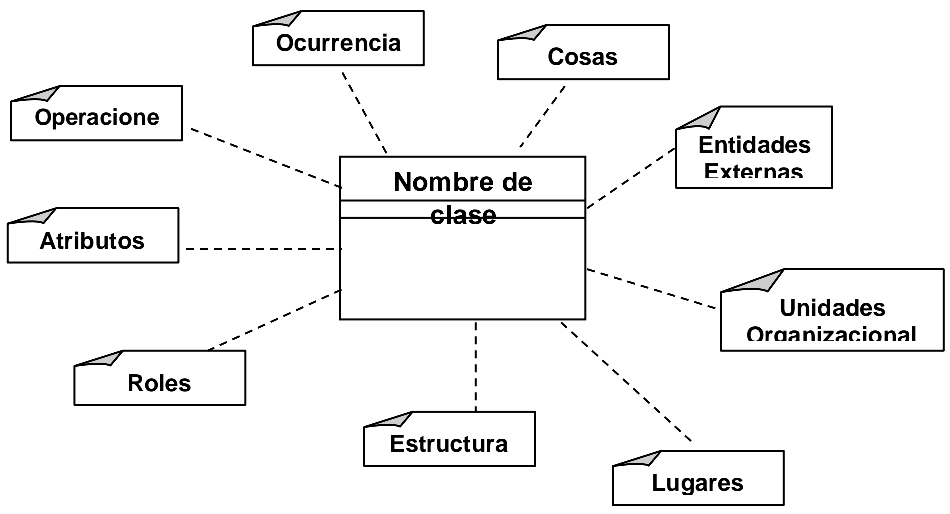


FIGURA 8**. Cómo se manifiestan los objetos[8].**

Los objetos se manifiestan de alguna de las formas mostradas en la Figura 8 y pueden ser:

a. Entidades Externas (por ejemplo: otros sistemas, dispositivos, personas) que producen o consumen información a usar por un sistemas de información.
b. Cosas (por ejemplo: informes, presentaciones, cartas, señales) que son parte del dominio de información del problema.
c. Ocurrencias o sucesos (por ejemplo: una transferencia de propiedad) que ocurren dentro del contexto de una operación del sistema.
d. Papeles o roles (por ejemplo: director, ingeniero, vendedor) desempeñados por personas que interactúan con el sistema.
e. Unidades organizacionales (por ejemplo: división, grupo, equipo) que son relevantes en una aplicación.
f. Lugares (por ejemplo: planta de producción o muelle de carga) que establecen el contexto del problema y la función general del sistema.
g. Estructuras (por ejemplo: sensores, o computadoras) que definen una clase de objetos o, en casos extremos, clases relacionadas de objetos.

Para incluir o no un objeto potencial dentro el modelo del análisis se deben tener en cuanta las siguientes seis características:

---

[8] Tomado de Ingeniería del Software. Roger Pressman.

1. *Infamación retenida:*
   El objeto potencial será de utilidad durante el análisis solamente si la información acerca de él debe recordarse para que el sistema funcione.
2. *Servicios necesarios:*
   El objeto potencial debe poseer un conjunto de operaciones identificables que pueden cambiar de alguna manera el valor de sus atributos.
3. *Atributos múltiples:*
   Durante el análisis de requisitos, se debe centrar la atención en la atención en la información principal (un objeto con un solo atributo puede, en efecto, ser útil durante el diseño).
4. *Atributos comunes:*
   Puede definirse un conjunto de atributos para el objeto potencial, los cuales son aplicables a todas las ocurrencias del objeto.
5. *Operaciones comunes*:
   Puede definirse un conjunto de operaciones para el objeto potencial, las cuales son aplicables a todas las ocurrencias del objeto.
6. *Requisitos esenciales:*
   Entidades externas que aparecen en el espacio del problema y producen o consumen información esencial para la producción de cualquier solución para el sistema, serán casi siempre definidas como objetos en el modelo de requisitos.

Para ser considerado un objeto válido a incluir en el modelo de requisitos, un objeto potencial debe satisfacer todas (o casi todas) las características anteriores La decisión de incluir objetos potenciales en el modelo de análisis es algo subjetivo, y una evaluación posterior puede llegar a descartar un objeto o reincluirlo Sin embargo, el primer paso del Análisis Orientado a Objetos debe ser la definición de objetos, y la consiguiente toma de decisiones.

12.2.2 Especificación de atributos

Los atributos describen un objeto que ha sido seleccionado para ser incluido en el modelo de análisis. En esencia, son los atributos los que definen al objeto, los que clarifican lo que se representa el objeto en el contexto del espacio del problema

Para desarrollar un conjunto significativo de atributos para un objeto, se debe estudiar de nuevo la narrativa del proceso (o descripción del ámbito del alcance) para el problema y seleccionar aquellos elementos que razonablemente pertenecen al objeto. Además, para cada objeto debe

responderse el siguiente interrogante: ¿Qué elementos (compuestos y/o simples) definen completamente al objeto en el contexto del problema actual?

12.2.3 Definición de operaciones

Las operaciones definen el comportamiento de un objeto y cambian, de alguna manera, los atributos de dicho objeto Más concretamente, una operación cambia valores de uno o más atributos contenidos en el objeto Por lo tanto, una operación debe tener conocimiento de la naturaleza de los atributos de los objetos y deben ser implementadas de manera tal que le permita manipular las estructuras de datos derivadas de dichos atributos.

Aunque existen muchos tipos diferentes de operaciones, éstas pueden clasificarse en tres grandes categorías:

1. operaciones que manipulan, de alguna manera, datos, por ejemplo: añadiendo, eliminando, seleccionando.
2. Operaciones que realizan algún cálculo
3. Operaciones que monitorizan un objeto frente a la ocurrencia de un suceso de control.

12.2.4 Fin de la definición del objeto

La definición de operaciones es el último paso para completar la especificación del objeto; la historia de la vida genérica de un objeto puede definirse reconociendo que dicho objeto debe ser creado, modificado, manipulado o leído de manera diferente, y posiblemente borrado. Para el objeto Sistema, esto puede expandirse para reflejar actividades conocidas que ocurren durante su vida. Algunas de las operaciones pueden determinarse a partir de comunicaciones semejantes entre objetos.

12.3 GESTION DE PROYECTOS DE SOFTWARE ORIENTADO A OBJETOS

Hoy en día la gestión de proyectos de software puede subdividirse en las siguientes actividades:

1. Establecimiento de un marco de proceso común para el proyecto.
2. Uso del marco y de métricas/histó^icas para desarrollar estimaciones de esfuerzo y tiempo.
3. Especificación de productos de trabajo e hitos que permitirán medir el progreso.

4. Definición de puntos de comprobación para la gestión de riesgos, aseguramiento de la calidad y control.
5. Gestión de los cambios que ocurren invariablemente al progresar el proyecto.
6. Seguimiento, monitorización y control del progreso.

Debido a la naturaleza única del software orientado a objetos, cada una de estas actividades de gestión tiene un matiz levemente diferente y debe ser enfocado usando un modelo propio.

12.3.1 Marco de proceso común para la Orientación a Objetos

Un marco de proceso común define un enfoque organizativo para el desarrollo y mantenimiento de software. De esta manera se identifica el paradigma de ingeniería del software aplicado para construir y mantener el software, así como las tareas, hitos y entregas requeridos. Establece el grado de rigor con el cual se enfocarán los diferentes tipos de proyectos. Este marco siempre es adaptable, de tal manera que siempre cumpla con las necesidades individuales del equipo del proyecto. Ésta es su característica más importante.

Muchos autores sugieren el uso de un modelo recursivo/paralelo para el desarrollo de software orientado a objetos. En esencia, el modelo recursivo/paralelo funciona de la siguiente manera:

1. Realizar los análisis suficientes para aislar las clases del problema y las conexiones más importantes.
2. Realizar un pequeño diseño para determinar si las clases y conexiones pueden ser implementadas de manera práctica.
3. Extraer objetos reutilizables de una biblioteca para construir un prototipo previo.
4. Conducir algunas pruebas para descubrir errores en el prototipo.
5. Obtener realimentación del cliente sobre el prototipo.
6. Modificar el modelo de análisis basándose en lo que se ha aprendido del prototipo, de la realización del diseño y de la realimentación obtenida del cliente.
7. Refinar el diseño para acomodar sus cambios.
8. Construir objetos especiales (no disponibles en la biblioteca).
9. Ensamblar un nuevo prototipo usando objetos de la biblioteca y los objetos que se crearon nuevos.
10. Realizar pruebas para descubrir errores en el prototipo.
11. Obtener realimentación del cliente sobre el prototipo.

Este enfoque continúa hasta que el prototipo evoluciona hacia una aplicación en producción.

El progreso se produce iterativamente. Lo que hace diferente al modelo recursivo/paralelo es el reconocimiento de que (1) el modelo de análisis y diseño para sistemas Orientado a Objetos no puede realizarse a un nivel uniforme de abstracción, y (2) el análisis y diseño pueden aplicarse a componentes independientes del sistema de manera concurrente.

Para controlar el marco de proceso recursivo/paralelo, se tiene que reconocer que el progreso está planificado y medido de manera incremental. Esto es, las tareas y la planificación del proyecto están unidas a cada una de las componentes altamente independientes, y el progreso se mide para cada una de estas componentes individualmente.

Cada iteración del proceso recursivo/paralelo requiere planificación, ingeniería (análisis, diseño, extracción de clases, prototipado y pruebas) y actividades de evaluación. Durante la planificación, las actividades asociadas con cada una de las componentes independientes del programa son incluidas en la planificación. Durante las primeras etapas del proceso de ingeniería, el análisis y el diseño se realizan iterativamente, la intención es identificar todos los elementos importantes del análisis Orientado a Objetos y de los modelos de diseño. Al continuar el trabajo de ingeniería, se producen versiones increméntales del software Durante la evaluación se realizan, para cada incremento, revisiones, evaluaciones del cliente y pruebas, las cuales producen una realimentación que afecta a la siguiente actividad de planificación y al subsiguiente incremento.

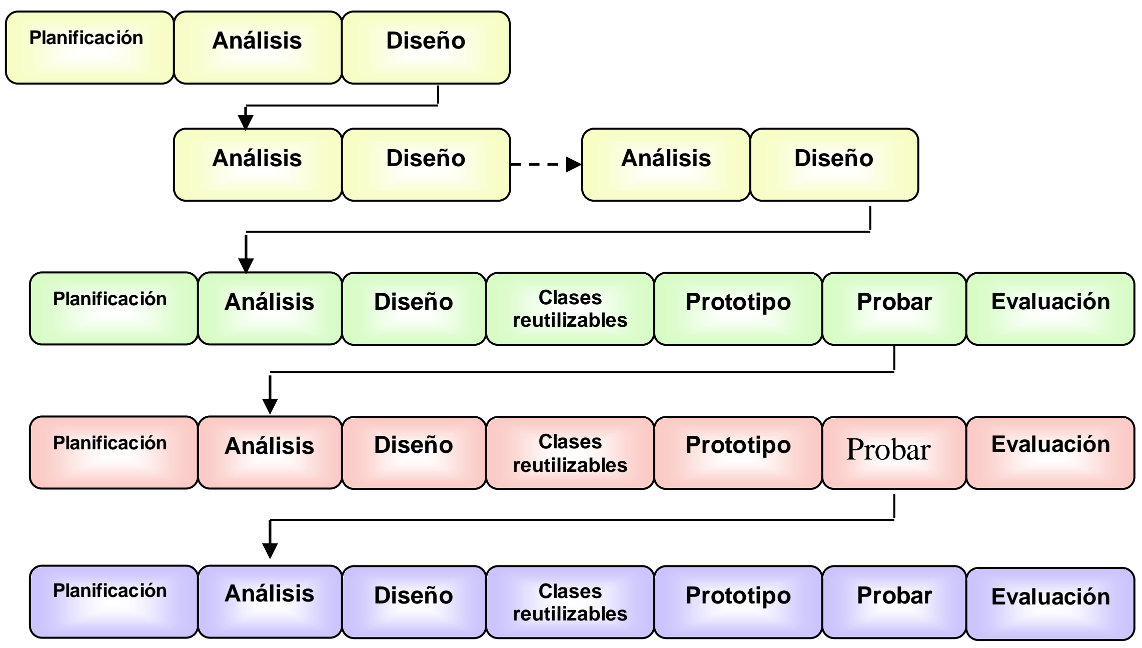


FIGURA 9. **Secuencia típica de un proceso para un proyecto Orientado a Objetos**[9]

12.3.2 Métricas y estimación en proyectos orientados a objetos

Las técnicas de estimación en proyectos de software convencionales requieren estimaciones de líneas de código o puntos de función como controlador principal de estimación. Las estimaciones realizadas a partir de líneas de código tienen poco sentido en proyectos Orientado a Objetos debido a que el objetivo principal es la reutilización. Las estimaciones a partir de puntos de función pueden usarse de manera efectiva, pues los elementos del dominio de información requeridos se pueden determinar a partir del planteamiento del problema. El análisis de puntos de función puede aportar valores para estimaciones en proyectos Orientado a Objetos, pero la medida de puntos de función no provee una granularidad suficiente para ajustes de planificación y esfuerzo a realizar, los cuales se requieren cuando iteramos a través del paradigma recursivo/paralelo.

En esta área son sugeridas los siguientes conjuntos de métricas para proyectos:

[9] Tomado de Ingeniería del Software. Roger Pressman.

1. Número de guiones de escenario:

   Un guión de escenario es una secuencia detallada de pasos que describen la interacción entre el usuario y la aplicación. Cada guión se organiza en tripletes de la forma:

   {**iniciador**, acción, **participante**}

   Donde iniciador es el objeto que solicita algún servicio (que inicia un mensaje), acción es el resultado de la solicitud, y participante es el objeto servidor que cumple la petición (solicitud). El número de guiones de actuación está directamente relacionado al tamaño de la aplicación y al número de casos de prueba que se deben desarrollar para ejercitar el sistema una vez construido.

2. Número de clases clave:

   Las clases clave son las componentes altamente independientes definidas inicialmente en el Análisis Orientado a Objetos. Debido a que estas clases son centrales al dominio del problema, el número de dichas clases es una indicación del esfuerzo necesario para desarrollar el software y de la cantidad potencial de reutilización a aplicar durante el desarrollo del sistema.

3. Número de clases de soporte:

   Las clases de soporte son necesarias para implementar el sistema, pero no están directamente relacionadas con el dominio del problema.

4. Número promedio de clases de soporte por clase clave.

   En general las clases clave son conocidas en las primeras etapas del proyecto Las clases de soporte se definen a lo largo de éste.

5. Número de subsistemas:
   Un subsistema es una agregación de clases que dan soporte a una función visible al usuario final del sistema. Una vez que se identifican los subsistemas, resulta más fácil realizar una planificación razonable en la cual el trabajo en los subsistemas está repartido entre los miembros del proyecto.

12.3.3 Enfoque Y Consejos Para Estimaciones Y Planificación De Proyectos Orientados A Objetos

Una de las facetas más difíciles de manejar dentro de un proyecto de software es la estimación y planificación del mismo. Para desarrollar estimaciones razonables es esencial el desarrollo de múltiples puntos de datos. Esto significa que las estimaciones deben derivarse usando diferentes técnicas. La realización y estimación respecto al esfuerzo y la duración usadas en el desarrollo de software convencional son aplicables dentro de la corriente Orientadas a Objetos, no obstante , por ser una técnica relativamente nueva no existe abundante información histórica para proyectos Orientados a Objetos

Por esta razón se sugiere el siguiente enfoque:

1. Desarrollo de estimaciones usando la descomposición de esfuerzos y cualquier otro método aplicable a aplicaciones convencionales.

2. Desarrollar escenarios y determinar una cuenta, usando Análisis Orientado a Objetos.

3. Determinar la cantidad de clases clave usando Análisis Orientado a Objetos.

4. Clasificar el tipo de interfaz para la aplicación y desarrollar un multiplicador para las clases de soporte: Multiplicar el número de clases clave por el multiplicador anterior para obtener una estimación del número de clases de soporte.

5. Multiplicar la cantidad total de clases (clave + soporte) por el número medio de unidades de trabajo por clase. Autores como Lorenz y Kidd sugieren entre 15 y 20 días - persona por clase.

La planificación de proyectos orientados a objetos es complicada por la naturaleza iterativa del marco de trabajo del proceso. Por ello se sugieren un conjunto de métricas que pueden ayudar durante esta planificación del proyecto:

1. Número de iteraciones principales:
   Recordando el modelo en espiral, una iteración principal corresponderá a un recorrido de 360 grados de la espiral. El modelo de proceso recursivo/paralelo engendrará un número de mini espirales que suceden durante el progreso de la iteración principal. Por ello se sugiere que las iteraciones de entre 2.5 y 4 meses de duración son

más fáciles de seguir y gestionar.

2. Número de contratos completos:
   Un contrato es “un grupo de responsabilidades públicas relacionadas que los subsistemas y las clases proporcionan a sus clientes”[10]. Se debe asociar al menos un contrato a cada iteración del proyecto.

12.3.4 Seguimiento del progreso en un proyecto orientado a objetos

Dentro de un proyecto Orientado a Objetos surgen dificultades debido a que el paralelismo de tareas dificulta el seguimiento de las mismas. En términos generales, si se cumplen los siguientes ítems o tareas dentro de cada fase, esta se puede dar por concluida.

1. **Análisis Orientado a Objetos Terminado**

   a. Todas las clases, y la jerarquía de clases, están definidas y revisadas.
   b. Se han definido y revisado los atributos de clase y las operaciones asociadas a una clase.
   c. Se han establecido y revisado las relaciones entre clases.
   d. Se ha creado y revisado un modelo de comportamiento
   e. Se han marcado clases reutilizables.

2. **Diseño Orientado a Objetos terminado**

   a. Se ha definido y revisado el conjunto de subsistemas.
   b. Las clases se han asignado a subsistemas y han sido revisadas.
   c. Se ha establecido y revisado la asignación de tareas.
   d. Se han identificado responsabilidades y colaboraciones.
   e. Se han diseñado y revisado los atributos y operaciones.
   f. Se ha creado y revisado el modelo de paso de mensajes.

3. **Programación Orientada a Objetos terminada**

   a. Cada nueva clase ha sido implementada en código a partir del modelo de diseño.
   b. Las clases extraídas se han integrado.
   c. Se ha construido un prototipo o incremento.

[10] Lorenz, M., yKidd, J., Métricas para software Orientado a Objetos

4. **Prueba Orientada a Objetos terminada**
    - **a.** Han sido revisadas la corrección e integridad del análisis Orientado a Objetos y del modelo de diseño.
    - **b.** Se ha desarrollado y revisado una red de clases responsabilidades - colaboraciones.
    - **c.** Se han diseñado casos de prueba y ejecutado pruebas al nivel de clases para cada clase.
    - **d.** Se han diseñado casos de prueba y completado pruebas de agrupamientos y las clases se han integrado.
    - **e.** Se han terminado las pruebas del sistema

13 ANÁLISIS ORIENTADO A OBJETOS

El propósito del análisis orientado a objetos es definir todas las clases que son relevantes al problema que se va a resolver, las operaciones y atributos asociados, las relaciones y comportamientos asociadas con ellas. Para cumplirlo se deben ejecutar las siguientes tareas:

1. Los requisitos básicos del usuario deben comunicarse entre el cliente y el desarrollador
2. Identificar las clases (es decir, definir atributos y métodos).
3. Se debe especificar una jerarquía de clases.
4. Representan las relaciones objeto a objeto (conexiones de objetos).
5. Modelar el comportamiento del objeto.
6. Repetir iterativamente las tareas de la 1 a la 5 hasta completar el modelo.

Es importante observar que no existe un acuerdo universal sobre los conceptos que sirven de base para el análisis orientado a objetos, pero un limitado número de ideas clave se repiten a menudo, y son éstas las que consideraremos en este capitulo.

13.1 ANÁLISIS ORIENTADO A OBJETOS

El objetivo del análisis orientado a objetos es desarrollar una serie de modelos que describan el software de computadora al trabajar para satisfacer un conjunto de requisitos definidos por el cliente. El análisis orientado a objetos, como los métodos de análisis convencionales descritos, forman un modelo de análisis multiparte para satisfacer este objetivo.

Para realizar un análisis orientado a objetos, un ingeniero del software debería ejecutar las siguientes etapas genéricas:

1. Obtener los requisitos del cliente para el sistema
2. Identificar escenarios o casos de uso
3. Seleccionar clases y objetos usando los requisitos básicos como guías.
4. Identificar atributos y operaciones para cada objeto del sistema
5. Definir estructuras y jerarquías que organicen las clases
6. Construir un modelo objeto-relación
7. Construir un modelo objeto-comportamientos
8. Revisar el modelo de análisis orientado a objetos con relación a los casos de uso/escenarios

13.1.1 Un enfoque unificado para el Análisis Orientado A Objetos

Al final de la pasada década, Grady Booch, James Rumbaugh e Ivar Jacobson empezaron a colaborar para combinar y recopilar las mejores características de cada uno de sus métodos de diseño y análisis orientado a objetos en un método unificado. El resultado, denominado Lenguaje de Modelado Unificado (UML), se ha convertido en el método más utilizado por la industria.

UML permite expresar un modelo de análisis utilizando una notación de modelado con unas reglas sintácticas, semánticas y prácticas. En UML, un sistema viene representado por cinco vistas diferentes que lo describen desde diferentes perspectivas. Cada vista se representa mediante un conjunto de diagramas. En UML están presentes las siguientes vistas:

1. **Vista del usuario**
   Representa el sistema (producto) desde la perspectiva de los usuarios (llamados actores en UML) El caso de uso es el enfoque elegido para modelar esta vista Esta importante representación del análisis, que describe un escenario de uso desde la perspectiva del usuario final.

2. **Vista estructural** los datos y la funcionalidad se muestran desde dentro del sistema, es decir, modela la estructura estática (clases, objetos y relaciones)

3. **Vista del comportamiento.** Esta parte del modelo del análisis representa los aspectos dinámicos o de comportamiento del sistema. También muestra las interacciones o colaboraciones entre los diversos elementos estructurales descritos en las vistas anteriores.

4. **Vista de implementación:** los aspectos estructurales y de comportamiento se representan aquí tal y como van a ser implementados.

5. **Vista del entorno:** aspectos estructurales y de comportamiento en el que el sistema a implementar se representa.

En general, el modelo de análisis de UML se centra en las vistas del usuario y estructural.

13.2 ANÁLISIS DEL DOMINIO

El análisis en sistemas orientados a objetos puede ocurrir a muchos niveles diferentes de abstracción. A nivel de negocios o empresas, las técnicas asociadas con el análisis orientado a objetos pueden acoplarse con un enfoque de ingeniería de la información en un esfuerzo por definir clases, objetos, relaciones y comportamientos que modelen el negocio por completo. A nivel de áreas de negocios, puede definirse un modelo de objetos que describa el trabajo de un área específica de negocios (o una categoría de productos o sistemas). A nivel de las aplicaciones, el modelo de objetos se centra en los requisitos específicos del cliente, pues éstos afectan a la aplicación que se va a construir.

A nivel de abstracción, el análisis orientado a objetos cae dentro del alcance general de la ingeniería del software orientado a objetos y es el centro de atención del resto de las secciones de este capítulo. En esta sección nos centraremos en el análisis orientado a objetos que se realiza a un nivel medio de abstracción. Esta actividad, llamada análisis del dominio, tiene lugar cuando una organización desea crear una biblioteca de clases reutilizables (componentes) ampliamente aplicables a una categoría completa de aplicaciones.

13.2.1 El proceso de análisis del dominio

Firesmith describe el análisis del dominio del software de la siguiente manera: “El análisis del dominio del software es la identificación análisis y especificación de requisitos comunes de un dominio de aplicación especifico, normalmente para su reutilización en múltiples proyectos dentro del mismo dominio de aplicación (el análisis orientado a objetos del dominio es la identificación, análisis y especificación de capacidades comunes y reutilizables dentro de un dominio de aplicación especifica, en términos de objetos, clases, sub - montajes y marcos de trabajo comunes.

Así el análisis del dominio puede verse como la actividad de cobertura para el proceso del software. En cierta forma, el papel de un análisis del dominio es similar al e un maestro tornero dentro de un entorno de fabricación fuerte El trabajo del maestro tornero es diseñar y construir herramientas que pueden usarse por anas personas que trabajan en aplicaciones similares, pero no necesariamente idénticas

En esencia el análisis del dominio es muy similar a la ingeniería del

conocimiento El ingeniero del conocimiento investiga un área de interés específica, intentando extraer hechos claves que se puedan usar para la construcción de un sistema experto o una red neuronal artificial Durante el análisis del dominio ocurre la extracción de objetos (y clases)

- **Definir el dominio a investigar**. Para llevar a cabo esta tarea, el analista debe primero aislar el área de negocio, tipo de sistema o categoría del producto de interés A continuación, se deben extraer los elementos tanto orientados a objetos como no orientados a objetos. Los elementos orientados a objetos incluyen especificaciones, diseños y código para clases de aplicaciones orientadas a objetos ya existentes, clases de soporte, bibliotecas de componentes comerciales ya desarrolladas (CYD) relevantes al dominio y casos de prueba. Los elementos no 00 abarcan políticas, procedimientos, planes, estándares y guías; partes de aplicaciones no orientados a objetos (incluyendo especificación, diseño e información de prueba), métricas y CYD del software no orientados a objetos.

- **Clasificar los elementos extraídos del dominio**. Los elementos se organizan en categorías y se establecen las características generales que definen la categoría Se propone un esquema de clasificación para las categorías y se definen convenciones para la nomenclatura de cada elemento. Se establecen jerarquías de clasificación en caso de ser apropiado

- **Recolectar una muestra representativa de aplicaciones en el dominio.** Para realizar esta tarea, el analista debe asegurar que la aplicación en cuestión tiene elementos que caen dentro de las categorías ya definidas.

- **Analizar cada aplicación dentro de la muestra**. Se debe seguir las etapas siguientes:
  - Identificar objetos candidatos reutilizables
  - Indicar las razones que hacen que el objeto haya sido identificado como reutilizable.
  - Definir adaptaciones al objeto que también pueden ser reutilizables.
  - Estimar el porcentaje de aplicaciones en el dominio que pueden reutilizar el objeto.
  - Identificar los objetos por nombre y usar técnicas de gestión de configuración para controlarlos

- **Desarrollar un modelo de análisis para los objetos**. El modelo de análisis servirá como base para el diseño y construcción de los objetos del dominio.

13.3 COMPONENTES GENERICOS DEL MODELO DE ANÁLISIS ORIENTADOS A OBJETOS

El proceso de análisis orientado a objetos se adapta a conceptos y principios básicos de análisis. Aunque la terminología, notación y actividades difieren respecto de los usados en métodos convencionales, el análisis orientado a objetos (en su núcleo) resuelve los mismos objetivos subyacentes.

Para desarrollar un «modelo preciso, conciso, comprensible y correcto del mundo real», se debe seleccionar una notación que se soporte a un conjunto de componentes genéricos de análisis orientado a objetos. Monarchi y Puhr definen un conjunto de componentes de representación genéricos que aparecen en todos los modelos de análisis oriantado a objetos. Los componentes estáticos son estructurales por naturaleza, e indican características que se mantienen durante toda la vida operativa de una aplicación. Los componentes dinámicos se centran en el control, y son sensibles al tiempo y al tratamiento de sucesos. Esto últimos definen cómo interactúa un objeto con otros a lo largo del tiempo. Pueden identificarse los siguientes componentes:

- **Vista estática de clases semánticas**. Estas clases persisten a través de todo el período de vida de la aplicación y se den van basándose en la semántica de los requisitos del cliente.

- **Vista estática de los atributos.** Toda clase debe describirse explícitamente. Los atributos asociados con la clase aportan una descripción de la clase, así como una indicación inicial de las operaciones relevantes a esta ríase.

- **Vista estática de las relaciones**. Los objetos están conectados unos a otros de varias formas. El modelo de análisis debe representar las relaciones de manera tal que puedan identificarse las operaciones (que afecten a estas conexiones) y que pueda desarrollarse un buen diseño de intercambio de mensajes.

- **Vista estática de los comportamientos.** Las relaciones indicadas anteriormente definen un conjunto de comportamientos que se adaptan al escenario utilizado (casos de uso) del sistema. Estos

comportamientos se implementan a través de la definición de una secuencia de operaciones que los ejecutan.

- **Vista dinámica de la comunicación.** Los objetos deben comunicarse unos con otros y hacerlo basándose en una serie de mensajes que provoquen transiciones de un estado a otro del sistema.

- **Vista dinámica del control y manejo del tiempo**. Debe describirse la naturaleza y duración de los sucesos que provocan transiciones de estados.

13.4 EL PROCESO DE ANÁLISIS ORIENTADO A OBJETOS

El proceso de análisis orientado a objetos no comienza con una preocupación por los objetos. Más bien comienza con una comprensión de la manera en la que se usará el sistema: por las personas, si el sistema es de interacción con el hombre; por otras máquinas, si el sistema está envuelto en un control de procesos; o por otros programas; si el sistema coordina y controla otras aplicaciones. Una vez que se ha definido el escenario, comienza el modelado del software.

Las secciones que siguen definen una serie de técnicas que pueden usarse para recopilar requisitos básicos del usuario y después definen un modelo de análisis para un sistema orientado a objetos.

13.4.1 Casos de uso

Los casos de uso modelan el sistema desde el punto de vista del usuario. Creados durante la obtención de requisitos, los casos de uso deben cumplir los siguientes objetivos:

- Definir los requisitos funcionales y operativos del sistema (producto), diseñando un escenario de uso acordado por el usuario final, y el equipo de desarrollo; proporcionar una descripción clara y sin ambigüedades de cómo el usuario final interactúa con el sistema y viceversa.

- Proporcionar una base para la validación de las pruebas.

Durante el análisis orientado a objetos los casos de uso sirven como base para los primeros elementos del modelo de análisis. Utilizando UML se puede crear una representación visual de los casos de uso llamada diagrama de casos de uso. Como otros elementos del análisis, los casos de uso pueden representarse a diferentes niveles de abstracción. Los diagramas de casos de uso contienen casos de uso y actores, siendo estos últimos las entidades que interactúan con el sistema. Pueden ser humanos u otras máquinas o sistemas que tengan definidas interfaces con nuestro sistema.

13.4.2 Modelado de clases-responsabilidades-colaboraciones

Una vez que se han desarrollado los escenarios de uso básicos para el sistema, es el momento de identificar las clases candidatas e indicar sus responsabilidades y colaboraciones. El modelado de clases – responsabilidades - colaboraciones (CRC) aporta un medio sencillo de identificar y organizar las clases que resulten relevantes al sistema o requisitos del producto.

13.4.2.1 Clases

Los objetos se manifiestan en una variedad de formas entidades externas, cosas, ocurrencias o sucesos, roles, unidades organizativas, lugares, o estructuras. Una técnica para identificarlos en el contexto de un problema del software es realizar un análisis gramatical con la narrativa de procesamiento para el sistema Todos los nombres se transforman en objetos potenciales. Sin embargo, no todo objeto potencial podrá incluirse en el modelo. Un objeto potencial debe satisfacer estas seis características para poder ser considerado como posible miembro del modelo:

1. **Retener información**: El objeto potencial será útil durante el análisis si la información sobre el mismo debe guardarse para que el sistema funcione.

2. **Servicios necesarios**: El potencial objeto debe tener un conjunto de operaciones identificables que permitan cambiar los valores de sus atributos

3. **Múltiples atributos:** durante el análisis de requisitos nos centramos más en la información más importante Un objeto con un solo atributo puede, en efecto, ser útil durante el diseño, pero probablemente será un atributo de otro objeto durante el análisis de actividades.

4. **Atributos comunes:** el conjunto de atributos definido para la clase debe ser aplicable a todas las ocurrencias del objeto.

5. **Operaciones comunes:** el objeto potencial debe definir un conjunto de operaciones aplicables, al igual que antes, a todos los objetos de la clase.

6. **Requisitos esenciales:** las entidades externas que aparecen en el espacio del problema y producen información esencial para la operación de una solución para el sistema casi siempre se definen como objetos en el modelo de requisitos.

13.4.2.2 Responsabilidades

Los atributos representan características estables de una clase, esto es, información sobre la clase que debe retenerse para llevar a cabo los objetivos del software especificados por el cliente. Los atributos pueden a menudo extraerse del planteamiento de alcance o discernirse a partir de la comprensión de la naturaleza de la clase. Las operaciones pueden extraerse desarrollando un análisis gramatical sobre la narrativa de procesamiento del sistema. Los verbos se transforman en candidatos a operaciones. Cada operación elegida para una clase exhibe un comportamiento de la clase.

Existen cinco pautas para especificar responsabilidades para las clases:

1. *La inteligencia del sistema debe distribuirse de manera igualitaria.* Toda aplicación encierra un cierto grado de inteligencia, por ejemplo, lo que sabe el sistema y lo que puede hacer. Esta inteligencia puede distribuirse ente las clases de varias maneras. Las clases «tontas» (aquellas con pocas responsabilidades) pueden modelarse de manera que actúen como sirvientes de unas pocas clases «listas» (aquellas con muchas responsabilidades). Aunque este enfoque hace que el flujo de control dentro de un sistema sea claro, posee algunas desventajas: (1) Concentra toda la inteligencia en pocas clases, haciendo los cambios más difíciles; (2) Tiende a necesitar más clases y por lo tanto el esfuerzo de desarrollo aumenta.

2. *Cada responsabilidad debe establecerse lo más general posible.* Esta directriz implica que las responsabilidades generales (tanto los atributos como las operaciones) deben residir en la parte alta de la jerarquía de clases (puesto que son genéricas, se aplicarán a todas las subclases). Adicionalmente, debe usarse el polimorfismo para definir

las operaciones que generalmente se aplica a la superclase, pero que se implementan de manera diferente en cada una de las subclases.

3. *La información y el comportamiento asociado a ella, debe encontrarse dentro de la misma clase.* Esto implementa el principio orientado a objetos de encapsulamiento. Los datos y procesos que manipulan estos datos deben empaquetarse como una unidad cohesionada.

4. *La información sobre un elemento debe estar localizada dentro de una clase, no distribuida a través de varias clases*. Una clase simple debe asumir la responsabilidad de almacenamiento y manipulación de un tipo específico de información. Esta responsabilidad no debe compartirse, de manera general, entre varias clases. Si la información está distribuida, el software se torna más difícil de mantener y probar.

5. Compartir responsabilidades entre clases relacionadas cuando sea apropiado. Existen muchos casos en los cuales una gran variedad de objetos exhibe el mismo comportamiento al mismo tiempo.

13.4.2.3 Colaboradores

Las clases cumplen con sus responsabilidades de una de las dos siguientes maneras: (1) una clase puede usar sus propias operaciones para manipular sus propios atributos, cumpliendo por lo tanto con una responsabilidad particular, o (2) puede colaborar con otras clases.
Se puede definir las colaboraciones de la siguiente manera: "Las colaboraciones representan solicitudes de un cliente a un servidor en el cumplimiento de una responsabilidad del cliente. Decimos que un objeto colabora con otro, si para ejecutar una responsabilidad necesita enviar cualquier mensaje al otro objeto. Una colaboración simple fluye en una dirección, representando una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor."

Las colaboraciones identifican relaciones entre clases. Cuando todo un conjunto de clases colabora para satisfacer algún requisito, es posible organizarías en un subsistema (un elemento del diseño). Las colaboraciones se identifican determinando si una clase puede satisfacer cada responsabilidad. Si no puede, entonces necesita interactuar con otra clase. De aquí surge lo que hemos llamado una colaboración.

13.4.3 Definición de estructuras y jerarquías

Una vez que se han identificado las clases y objetos usando todo el análisis se centra en la estructura del modelo de clases y las jerarquías resultantes que surgen al emerger clases y subclases. Usando la notación UML podemos crear gran variedad de diagramas. Debe crearse una estructura de generalización-especialización para las clases identificadas.

13.4.4 Definición de subsistemas

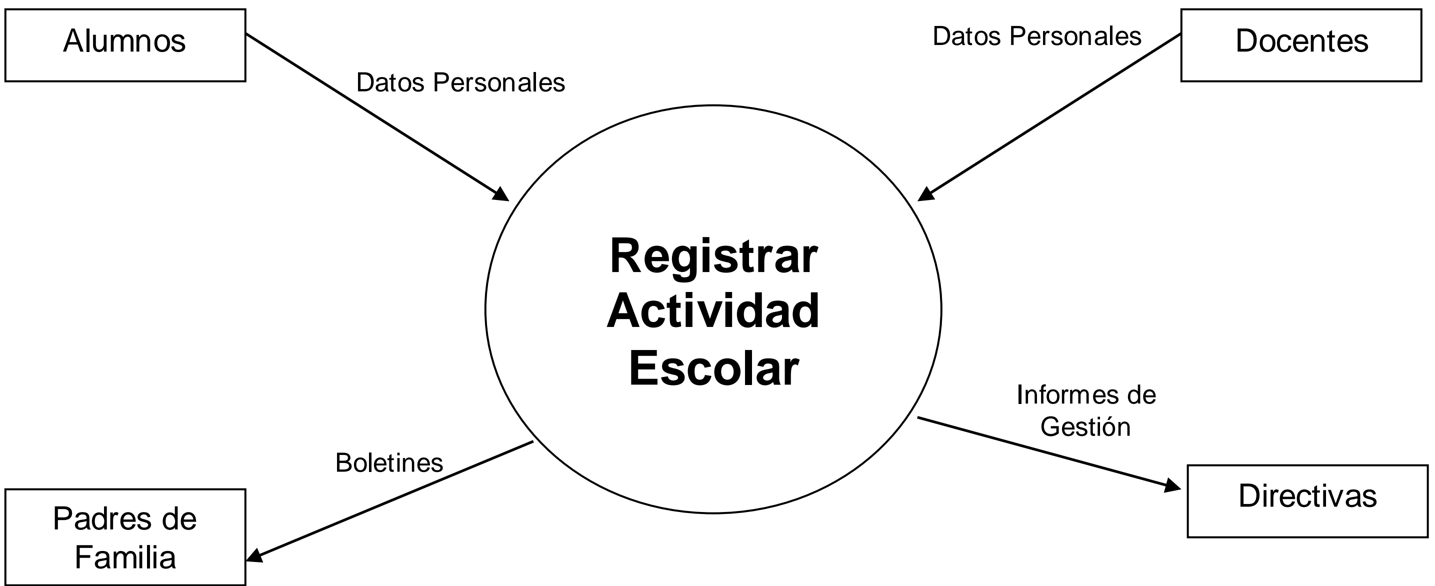
Un modelo de análisis para una aplicación compleja puede tener cientos de clases y docenas de estructuras. Los subconjuntos de clases que colaboran entre sí para llevar a cabo un conjunto de responsabilidades cohesionadas, se les llama normalmente subsistemas o paquetes (en terminología UML). Los subsistemas o paquetes son abstracciones que aportan una referencia o puntero a los detalles en el modelo de análisis. Si se observa desde el exterior, un subsistema puede tratarse como una caja negra que contiene un conjunto de responsabilidades y que posee sus propias colaboraciones (externos).

13.5 EL MODELO OBJETO-RELACIÓN

El primer paso en el establecimiento de las relaciones es comprender las responsabilidades de cada clase. El siguiente paso es definir aquellas clases colaboradoras que ayudan en la realización de cada responsabilidad. Esto establece la «conexión» entre las clases

La notación del lenguaje unificado de modelado para el modelo objeto-relación utiliza una simbología adaptada de las técnicas del modelo entidad-relación. En esencia, los objetos se conectan con otros objetos utilizando relaciones con nombres. Se especifica la cardinalidad de la conexión y se establece toda una red de relaciones.

14 DIAGRAMA DE CONTEXTO



15 DIAGRAMAS UML DE LA FASE DE ANALISIS OO

15.1 DIAGRAMA DE CASO DE USO

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.

NOMBRE: Evaluación y Aceptación de Matriculas
ACTORES: Llamado por el “Alumno” se comunica con el “Jefe de matriculas”

CONDICION INICIAL:

1. El Jefe de matriculas recibe los documentos del registro y matricula del alumno

FLUJO DE EVENTOS:

2. El Jefe de Matriculas revisa los documentos de la matricula teniendo en cuenta que se encuentren la constancia de notas, certificado medico, fotografías y recibo de pago de los derechos de matriculas, sistematización y asociación de padres de familia.
3. El Jefe de notas elabora la orden de matricula con los datos del alumno.
4. Revisión de la orden de matricula por parte del alumno.

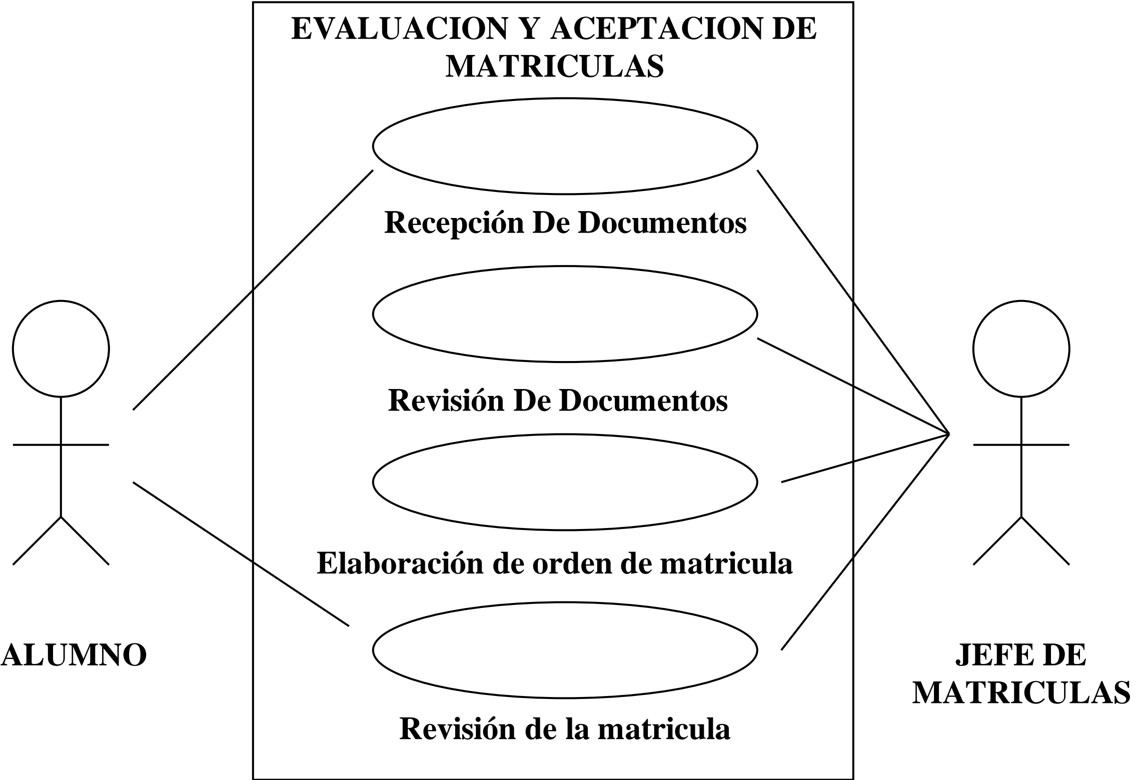
CONDICIONES DE SALIDA:

5. Después de aprobada la Orden de matricula por parte del Alumno se archiva en el fólder correspondiente a cada alumno.

REQUERIMIENTOS ESPECIALES:

- La Recepción del los documentos de la matricula es efectuado por cualquier docente de la institución

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.



PROCESO: Matricula de alumnos
SUBPROCESO: Registro y Control de notas.

NOMBRE: Registro y control de notas.
ACTORES: Llamado por el "Alumno" se comunica con el "Jefe de Matriculas" y el "Jefe de Matriculas" se comunica con los "profesores" y este se comunica con los "Empresa de Impresión"

CONDICIONES INICIALES:

1. El Jefe de Matricula archiva la Matricula.

FLUJO DE EVENTOS:

2. Se realiza una relación de logros con las notas de cada alumno.
3. La planilla de la relación es enviada a al empresa que imprime los boletines.
4. La empresa realiza la impresión de los boletines.
5. Se envía los boletines al colegio.

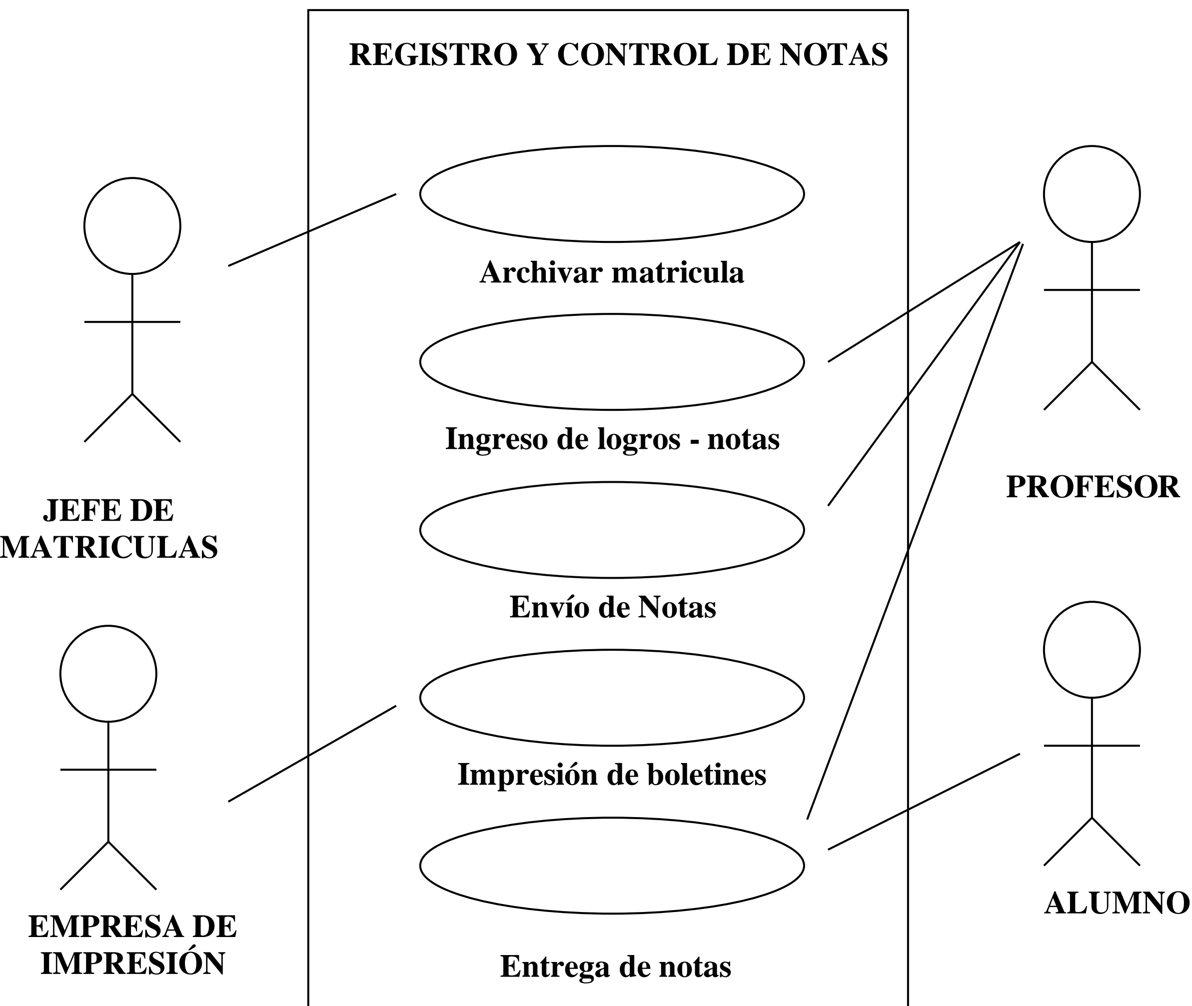
CONDICIONES DE SALIDA:

6. Se entrega los boletines a los alumnos.

REQUERIMIENTOS ESPECIALES:

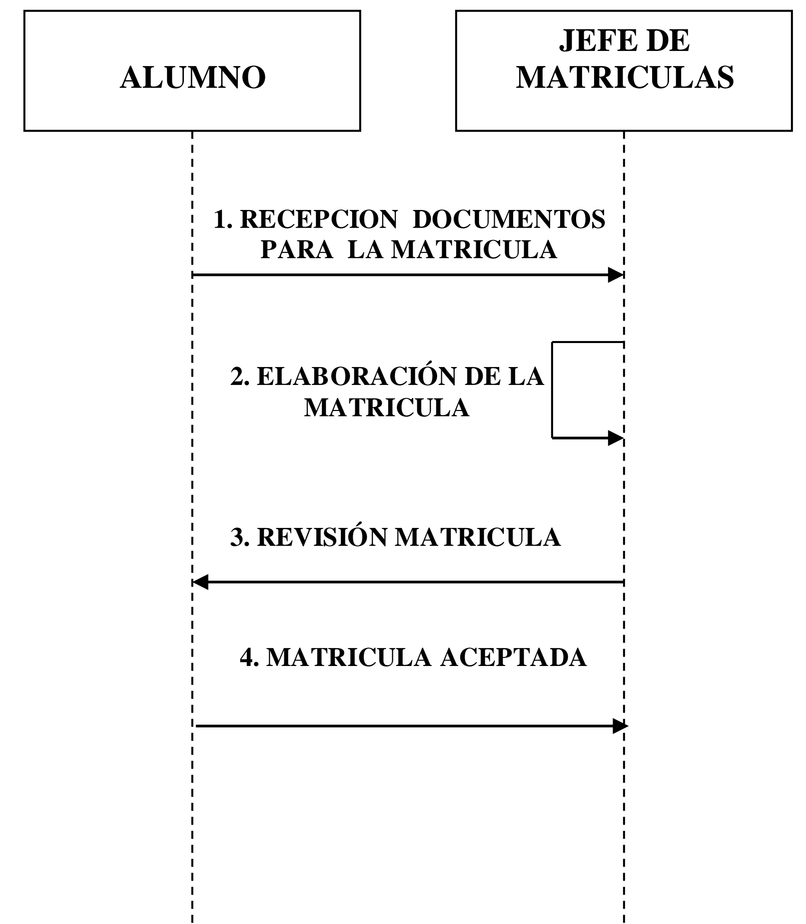
- Si en el Boletín no coinciden las notas con el alumno se hace una revisión por parte del Jefe de Notas.

PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.

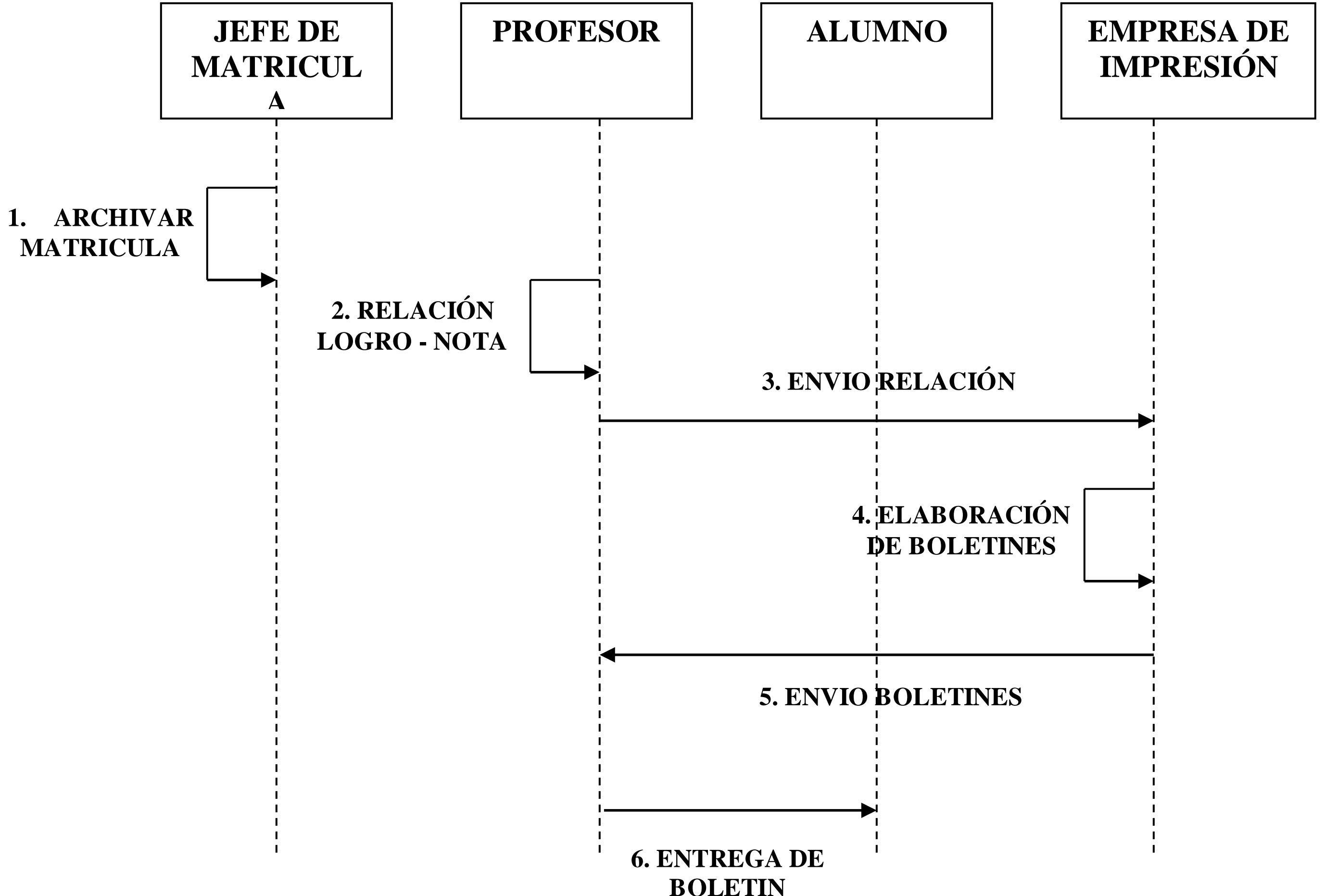


15.2 DIAGRAMAS DE SECUENCIA

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.

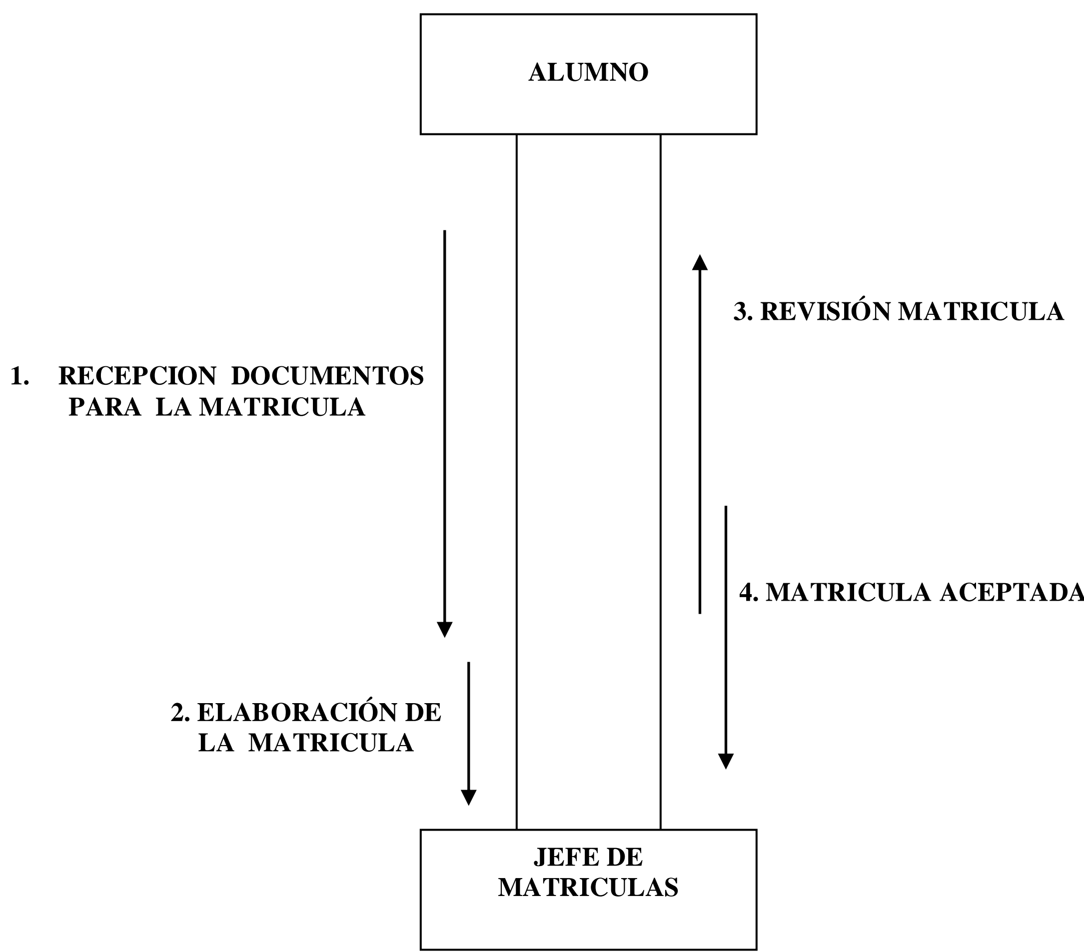


PROCESO: Matricula de alumnos
SUBPROCESO: Registro y Control de notas.

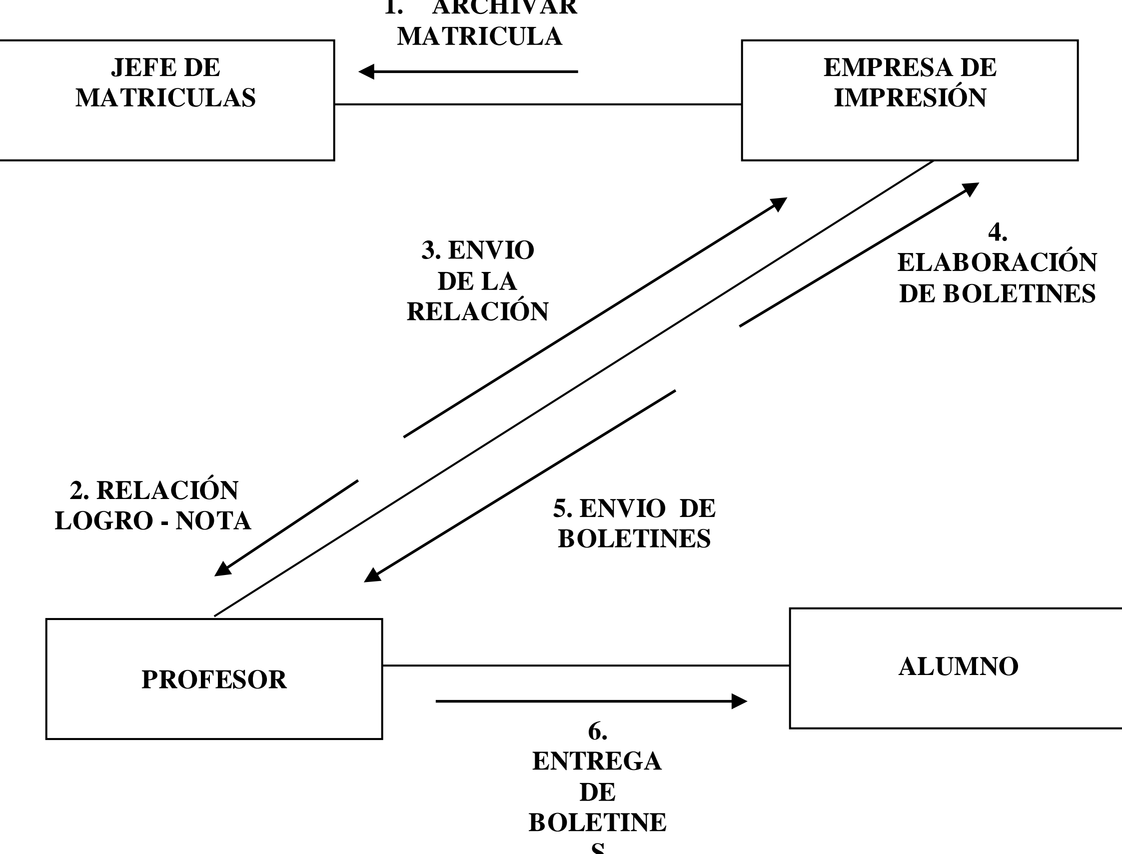


15.3 DIAGRAMAS DE COLABORACIÓN

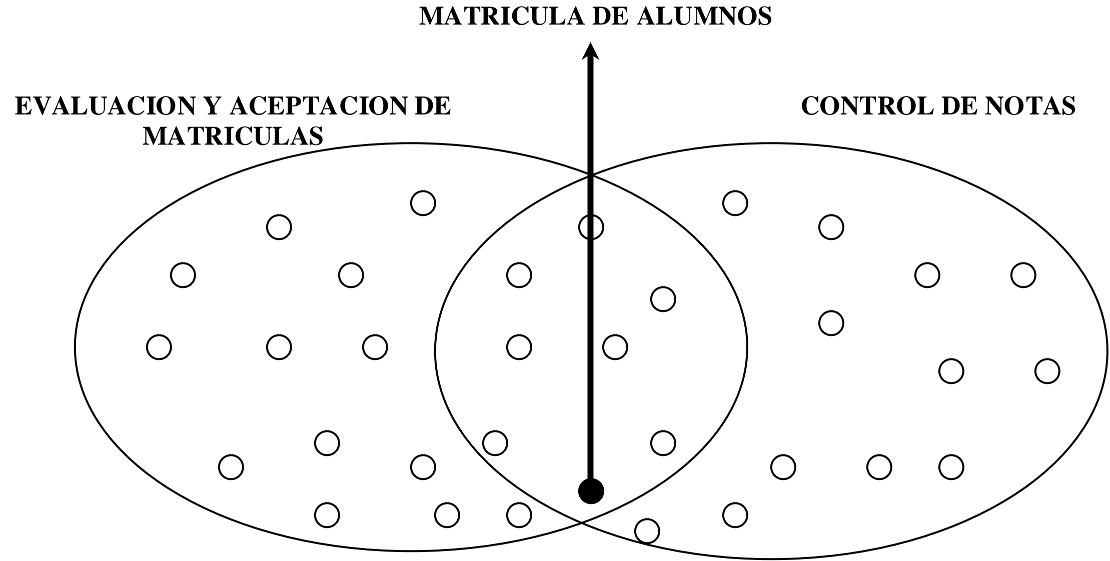
PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.



PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.



15.4 ESTADOS DE UN OBJETO



16 DISEÑO ORIENTADO A OBJETOS

La naturaleza única del diseño orientado a objetos, reside en su capacidad para construir cuatro conceptos importantes de diseño de software: abstracción, ocultamiento (ocultación) de información, independencia funcional y modularidad. Todos los métodos de diseño procuran software que exhiba estas características fundamentales, pero sólo el DOO provee un mecanismo que permite al diseñador alcanzar las cuatro, sin complejidad ni compromiso.

El diseño orientado a objetos, la programación orientada a objetos, y las pruebas orientadas a objetos son actividades de construcción para sistemas orientados a objetos.

Las cuatro capas de la pirámide de diseño orientado a objetos son:

- La capa subsistema: Contiene una representación de cada uno de los subsistemas, para permitir al software conseguir sus requisitos definidos por el cliente e implementar la infraestructura que soporte los requerimientos del cliente.
- La capa de clases y objetos: Contiene la jerarquía de clases, que permiten al sistema ser creado usando generalizaciones y cada vez especializaciones más acertadas. Esta capa también contiene representaciones.
- La capa de mensajes: Contiene detalles de diseño, que permite a cada objeto comunicarse con sus colaboradores. Esta capa establece interfaces externas e internas para el sistema.
- La capa de responsabilidades: Contiene estructuras de datos y diseños algorítmicos, para todos los atributos y operaciones de cada objeto.

FIG. 10 **La pirámide del diseño OO**[11].

[11] Tomado de Ingeniería del Software. Roger Pressman.

16.1 ENFOQUE CONVENCIONAL VS. 00

Los enfoques convencionales para el diseño de software aplican distintas notaciones y conjunto de heurísticas para trazar el modelo de análisis en un modelo de diseño. Recordando la Figura 4, cada elemento del modelo convencional de análisis se corresponde con uno o más capas del modelo de diseño. Al igual que el diseño convencional de software, el DOO aplica el diseño de datos cuando los atributos son representados, el diseño de interfaz cuando se desarrolla un modelo de mensajería, y diseño a nivel de componentes (procedimental), para operaciones de diseño. Es importante notar que la arquitectura de un diseño 00 tiene más que ver con la colaboración entre objetos que con el control de flujo entre componentes del sistema.

A pesar de que existen similitudes entre los diseño convencionales y 00, se ha optado por renombrar las capas de la pirámide de diseño, para reflejar con mayor precisión la naturaleza de un diseño 00. La Figura 11 ilustra la relación entre el modelo de análisis 00 y el modelo de diseño que se derivará de ahí.

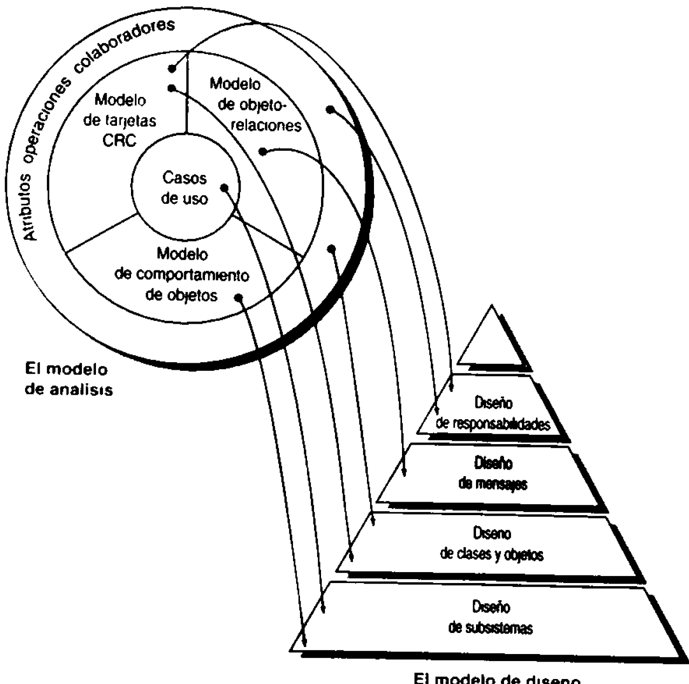


FIGURA 11**. Transformación de un modelo de análisis OO a un modelo de diseño OO.** [12]

---

[12] Tomado de Ingeniería del Software. Roger Pressman.

Fichman y Kemerer sugieren diez componentes de diseño modelado, que pueden usarse para comparar vanos métodos convencionales y orientados a objetos:

1 Representación de Ia jerarquía de módulos.
2 Especificación de las definiciones de datos
3 Especificación de la lógica procedimental
4 Indicación de secuencias de proceso final-a-final
(end-to-end)
5. Representación de estados y transiciones de los objetos.
6 Definición de clases y jerarquías.
7 Asignación de operaciones a las clases
8 Definición detallada de operaciones.
9 Especificación de conexiones de mensajes.
10 Identificación de servicios exclusivos

Ya que existen muchos enfoques de diseño convencionales y orientados a objetos, es difícil desarrollar una comparación generalizada entre los dos métodos. Sin embargo, se puede asegurar que las componentes de modelado 5 al 10 no están soportadas usando diseño estructurado o sus derivados.

16.2 ASPECTOS DEL DISEÑO

Bertrand Meyer sugiere cinco criterios para juzgar la capacidad de métodos de diseño para conseguir modularidad, y los relaciona al diseño orientado a objetos:

• Descomponibilidad, la facilidad con que un método de diseño ayuda al diseñador a descomponer un problema grande en problemas más pequeños, haciéndolos más fácil de resolver.

• Componibilidad: el grado con el que un método de diseño asegura que los componentes del programa (módulos), una vez diseñados y construidos, pueden ser reutilizados para crear otros sistemas.

• Comprensibilidad: la facilidad con la que el componente de un programa puede ser entendido, sin hacer referencia a otra información o módulos.

• Continuidad: la habilidad para hacer pequeños cambios en un programa y que se revelen haciendo los cambios pertinentes en uno o muy pocos módulos.

- Protección: una característica arquitectónica, que reduce la propagación de efectos colaterales, si ocurre un error en un módulo dado.

De estos criterios, Meyer, sugiere cinco principios básicos de diseño, que pueden ser deducidos para arquitecturas modulares (1) unidades lingüísticas modulares, (2) pocas interfaces, (3) pequeñas interfaces (acoplamiento débil), (4) interfaces explícitas y (5) ocultación de información.

Los criterios, y principios de diseño ya presentados pueden ser aplicados a cualquier método de diseño (así como al diseño estructurado). Como se verá, el método de diseño orientado a objetos logra cada uno de los criterios más eficientemente que otros enfoques, y resulta en arquitecturas modulares, que cumplen efectivamente cada uno de los criterios.

A pesar de que la terminología y etapas de proceso para cada uno de estos métodos de DOO difieren, los procesos de DOO global son bastante consistentes.

16.3 EL PANORAMA DE DOO

Una gran variedad de métodos de análisis y diseño orientados a objetos fue propuesta y utilizada durante los ochenta y los noventa. Estos métodos establecieron los fundamentos para la notación moderna de DOO, heurísticas de diseño y modelos.

Para llevar a cabo un diseño orientado a objetos, un ingeniero de software debe ejecutar las siguientes etapas generales:

1. Describir cada subsistema y asignar a procesadores y tareas.
2. Elegir una estrategia para implementar la administración de datos, soporte de interfaz y administración de tareas.
3. Diseñar un mecanismo de control, para el sistema apropiado.
4. Diseñar objetos creando una representación procedural para cada operación, y estructuras de datos para los atributos de clase.
5. Diseñar mensajes, usando la colaboración entre objetos y relaciones.
6. Crear el modelo de mensajería.
7. Revisar el modelo de diseño y renovarlo cada vez que se requiera.

Es importante hacer notar que las etapas de diseño discutidas en esta sección son iterativas. Eso significa que deben ser ejecutadas incrementalmente, junto con las actividades de AOO, hasta que se produzca el diseño completo.

16.4 UN ENFOQUE UNIFICADO PARA EL DOO

Grady Booch, James Rumbaugh e Ivar Jacobson, combinaron las mejores cualidades de sus métodos personales de análisis y diseño orientado a objetos, en un método unificado. El resultado, llamado el Lenguaje de Modelado Unificado (UML), se ha vuelto ampliamente usado en la industria. UML se organiza en dos actividades mayores: diseño del sistema y diseño de objetos. El principal objetivo de UML, diseño de sistema, es representar la arquitectura de software.

El diseño de objetos se centra en la descripción de objetos y sus interacciones con los otros. Una especificación detallada de las estructuras de datos de los atributos y diseño procedural de todas las operaciones, se crea durante el diseño de objetos. La visibilidad para todos los atributos de clase se define, y las interfaces entre objetos se elaboran para definir los detalles de un modelo completo de mensajes.

El diseño de sistemas y objetos en UML se extiende para considerar el diseño de interfaces, administración de datos con el sistema que se va a construir y administración de tareas para los subsistemas que se han especificado.

16.4.1 PROCESO DE DISEÑO DE SISTEMA

El diseño de sistema desarrolla el detalle arquitectónico requerido para construir un sistema o producto. El proceso de diseño del sistema abarca las siguientes actividades:

1. Partición del modelo de análisis en subsistemas
2. Identificar la concurrencia dictada por el problema
3. Asignar subsistemas de procesadores y tareas
4. Desarrollar un diseño para la interfaz de usuario
5. Elegir una estrategia básica para implementar la administración (gestión) de datos
6. Identificar recursos globales y los mecanismos de control requeridos para su acceso
7. Diseñar un mecanismo de control apropiado para el sistema, incluyendo administración de tareas
8. Considerar cómo deben manejarse las condiciones de frontera
9. Revisar y considerar trade –offs.

16.4.2 PROCESO DE DISEÑO DE OBJETOS

El diseño de objetos tiene que ver con el diseño detallado de los objetos y sus interacciones. Se completa dentro de la arquitectura global, definida durante el diseño del sistema y de acuerdo con las reglas y protocolos de diseño aceptados. El diseño del objeto está relacionado en particular con la especificación de los tipos de atributos, cómo funcionan las operaciones y cómo los objetos se enlazan con otros objetos.

16.4.2.1 Descripción de objetos

Una descripción del diseño de un objeto (instancia de clase o subclase) puede tomar una o dos formas: (1) Una descripción de protocolo que establece la interfaz de un objeto, definiendo cada mensaje que el objeto puede recibir y las operaciones que el objeto lleva a cabo cuando recibe un mensaje, o (2) Una descripción de implementación que muestra detalles de implementación para cada operación implicada por un mensaje pasado a un objeto. Los detalles de implementación incluyen información acerca de la parte privada del objeto; esto significa, detalles internos acerca de la estructura de datos, que describen los atributos del objeto, y detalles de procedimientos, que describen las operaciones.

16.4.2.2 Diseño de algoritmos y estructuras de datos

Una variedad de representaciones contenidas en el modelo de análisis y el diseño de sistema, proveen una especificación para todas las operaciones y atributos. Los algoritmos y estructuras de datos se diseñan utilizando un enfoque, que difiere un poco de los enfoques del diseño de datos y del diseño a nivel de componentes examinadas para la ingeniería del software convencional.

Se crea un algoritmo para implementar la especificación para cada operación. En muchas ocasiones, el algoritmo es una simple secuencia computacional o procedural, que puede ser implementada como un módulo de software autocontenido. Sin embargo, si la especificación de la operación es compleja, será necesario modularizar la operación. Las técnicas convencionales de diseño de componentes se pueden usar para resolver esta tarea.

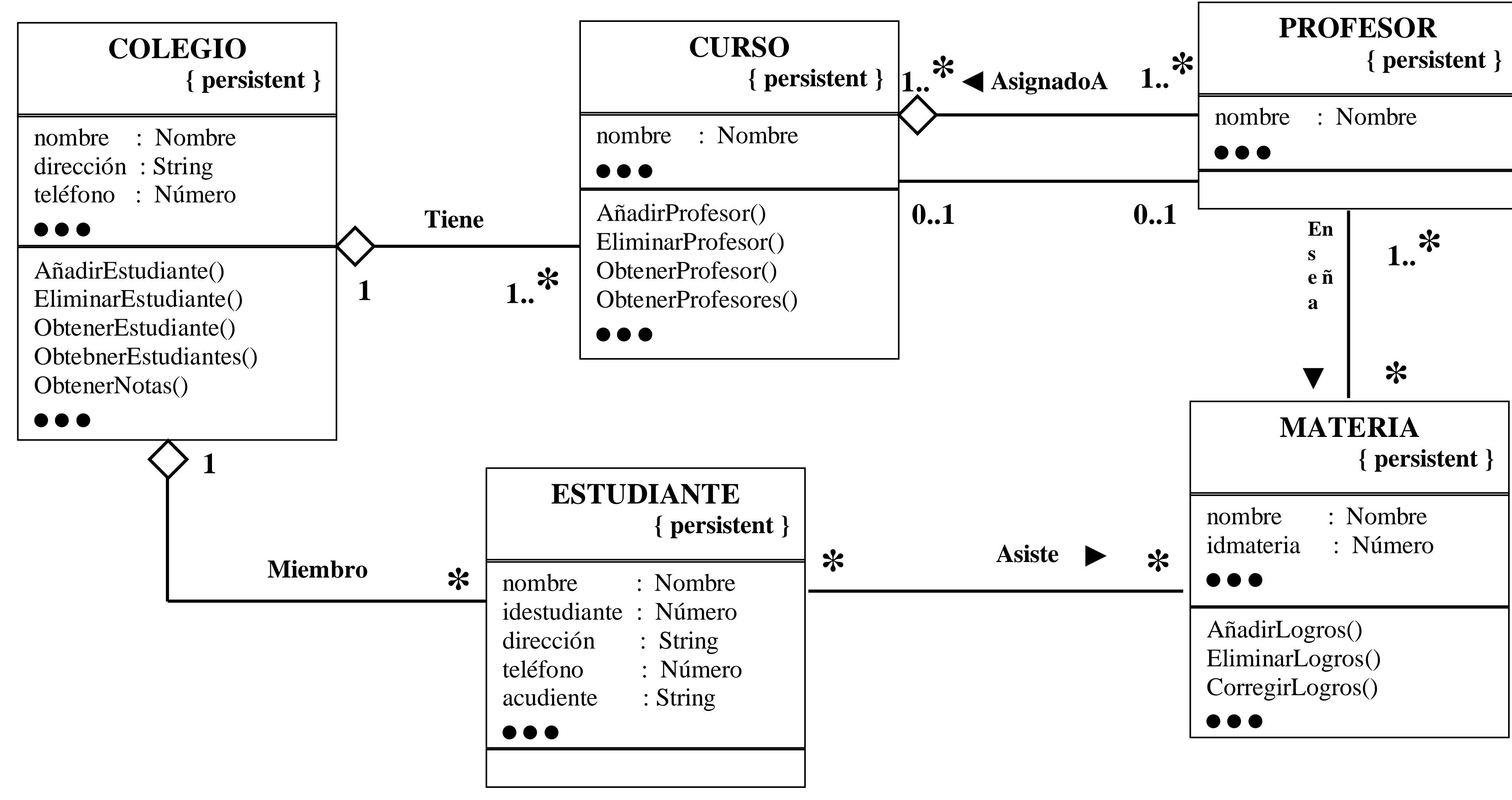
Las estructuras de datos se diseñan al mismo tiempo que los algoritmos. Ya que las operaciones manipulan los atributos de una clase, el diseño de

estructuras de datos, que reflejan mejor los atributos, tendrá un fuerte sentido en el diseño algorítmico de las operaciones correspondientes.

Aunque existen muchos tipos diferentes de operaciones, normalmente se pueden dividir en tres grandes categorías: (1) operaciones que manipulan los datos de alguna manera (por ejemplo, agregando, eliminando, reformateando, seleccionando), (2) operaciones que ejecutan cálculos, y (3) operaciones que monitorizan (supervisan) al objeto para la ocurrencia de un suceso controlado.

17 DIAGRAMAS UML DE LA FASE DE DISEÑO OO

17.1 DIAGRAMAS DE CLASES



17.2 DIAGRAMA DE CASO DE USO

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.

NOMBRE: Evaluación y Aceptación de Matriculas
ACTORES: Llamado por el "Alumno" se comunica con el "Jefe de matriculas"

CONDICION INICIAL:

6. El Jefe de matriculas recibe los documentos del registro y matricula del alumno

FLUJO DE EVENTOS:

7. El Jefe de Matriculas revisa los documentos de la matricula teniendo en cuenta que se encuentren la constancia de notas, certificado medico, fotografías y recibo de pago de los derechos de matriculas, sistematización y asociación de padres de familia.
8. El Jefe de notas elabora la orden de matricula con los datos del alumno (Curso, acudientes)
9. Después de elaborada la orden de matricula es entregada al alumno.
10. El Alumno recibe la orden de matricula la cual es sometida a revisión por parte del mismo.
11. El Alumno aprueba y firma la orden de matricula elaborada por el Jefe de Notas.

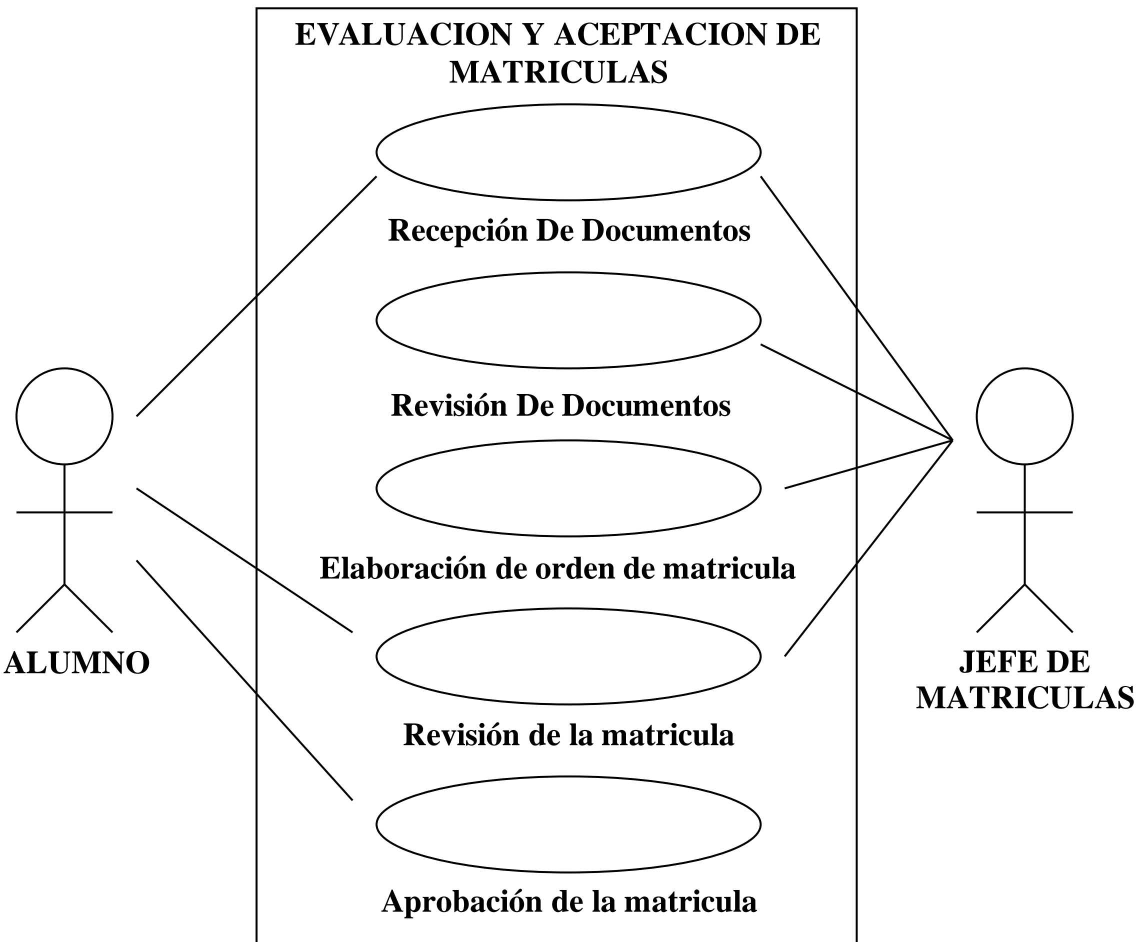
CONDICIONES DE SALIDA:

12. Después de aprobada la Orden de matricula por parte del Alumno se toma la información y se registra conectando con el subproceso de Registro y control de notas.

REQUERIMIENTOS ESPECIALES:

- La Recepción del los documentos de la matricula es efectuado por director de grado del Alumno
- En caso de ser rechazado la matricula se devuelve al Alumno para ser revisado por este.

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.



PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.

NOMBRE: Registro y control de notas.
ACTORES: Llamado por el “Alumno” se comunica con el “Jefe de Matriculas” y el “Jefe de Matriculas” se comunica con el “Jefe de notas” y este se comunica con los “Profesores”

CONDICIONES INICIALES:

7. El Jefe de Matricula recibe la Matricula aprobada por Cliente.

FLUJO DE EVENTOS:

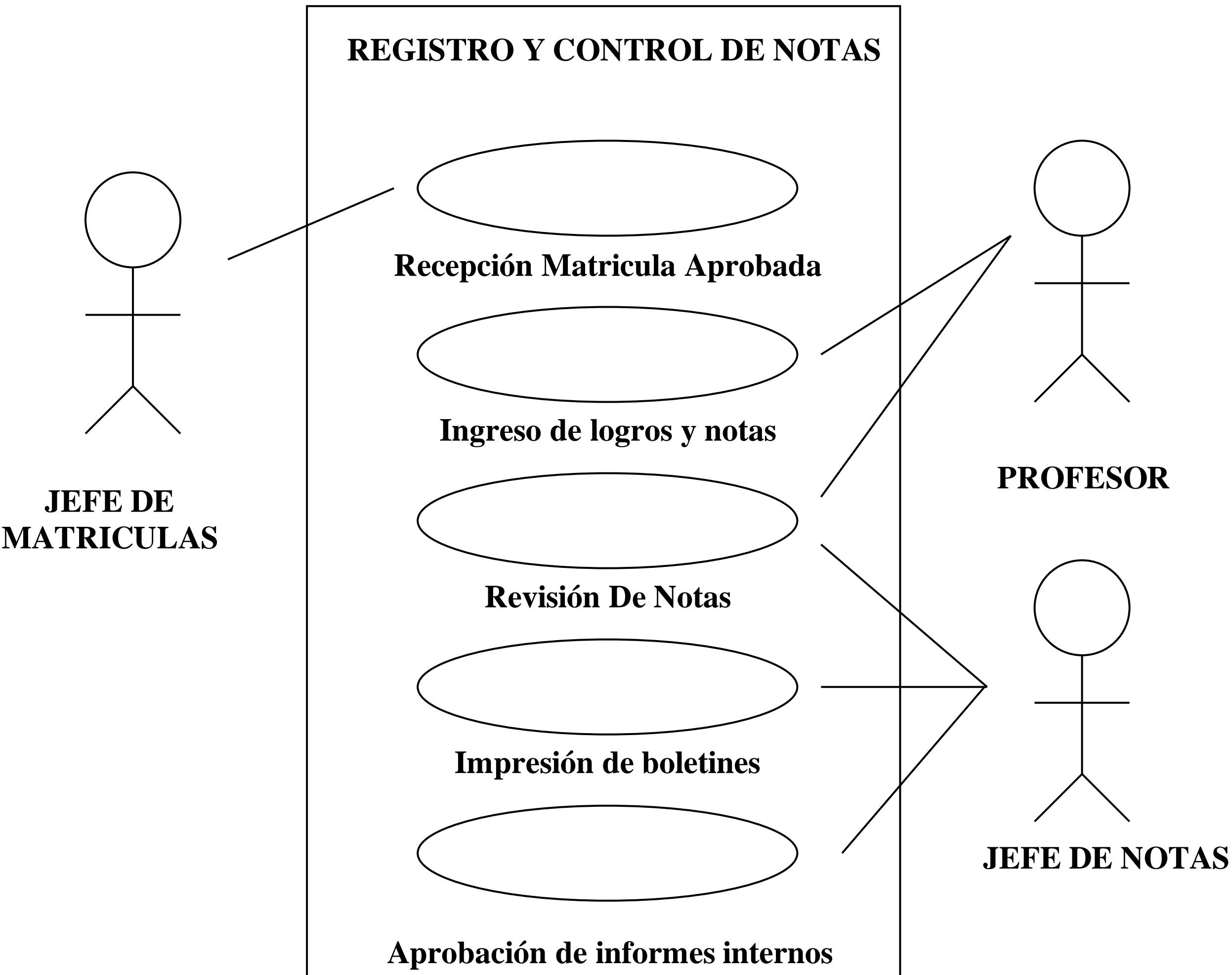
8. Se efectúa la preparación interna de ingreso de logros por parte del Jefe de Matriculas.
9. Posteriormente se revisa el ingreso de logros por parte del Jefe de Notas.
10. El jefe de notas solicita el ingreso de notas por el parte de los profesores
11. Los profesores ingresan las notas correspondientes a loa alumnos de acuerdo con los registros de matricula.
12. El Jefe de notas revisa el ingreso de las notas.
13. El jefe de notas da la orden de impresión de boletines.

CONDICIONES DE SALIDA:

14. El Jefe de notas da el visto bueno para la realización de los informes Internos.

REQUERIMIENTOS ESPECIALES:
- Si en el Boletín no coinciden las notas con el alumno se hace una revisión por parte del Jefe de Notas.
- Una vez almacenadas las notas se da el visto bueno y la probación para la generación de reportes internos (soporte de decisiones)

PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.



PROCESO: Matricula de alumnos
SUBPROCESO: Generación de documentos para el soporte de decisiones

NOMBRE: Generación de documentos para el soporte de decisiones
ACTORES: Llamado por el "Rector" se comunica con el "Director de notas".

CONDICION INICIAL:

1. El Rector solicito la situación académica de la institución al Jefe de notas

FLUJO DE EVENTOS:

2. El jefe de notas silicita a la aplicación la generación de los reportes internos que necesita la institución.
3. El Jefe de notas revisa que los reportes sean los mismos que fueron solicitados por "El Rector".
4. El Rector revisa los informes generados.

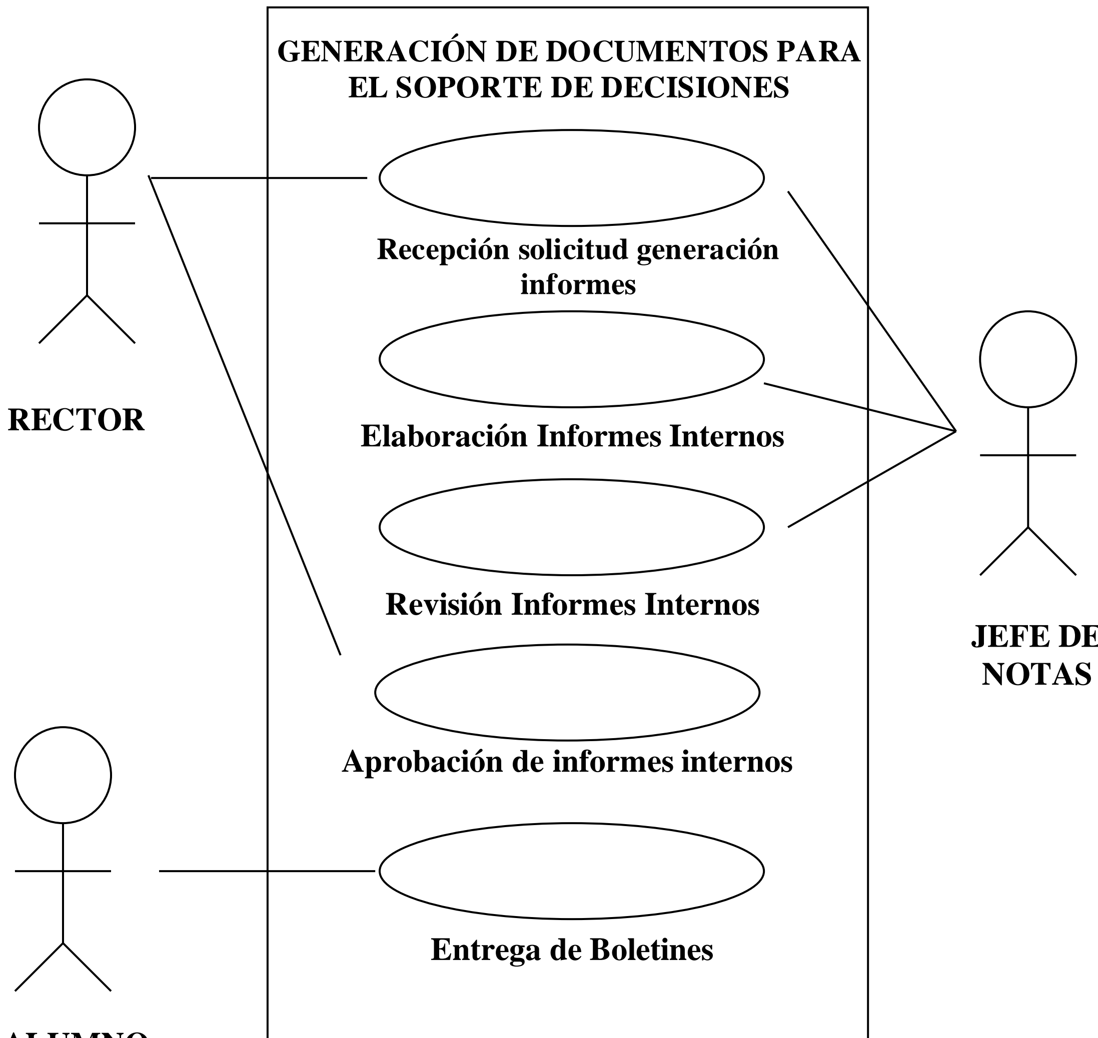
CONDICIONES DE SALIDA:

5. El Rector convoca a la entrega de boletines y seguimiento de alumnos.

REQUERIMIENTOS ESPECIALES:

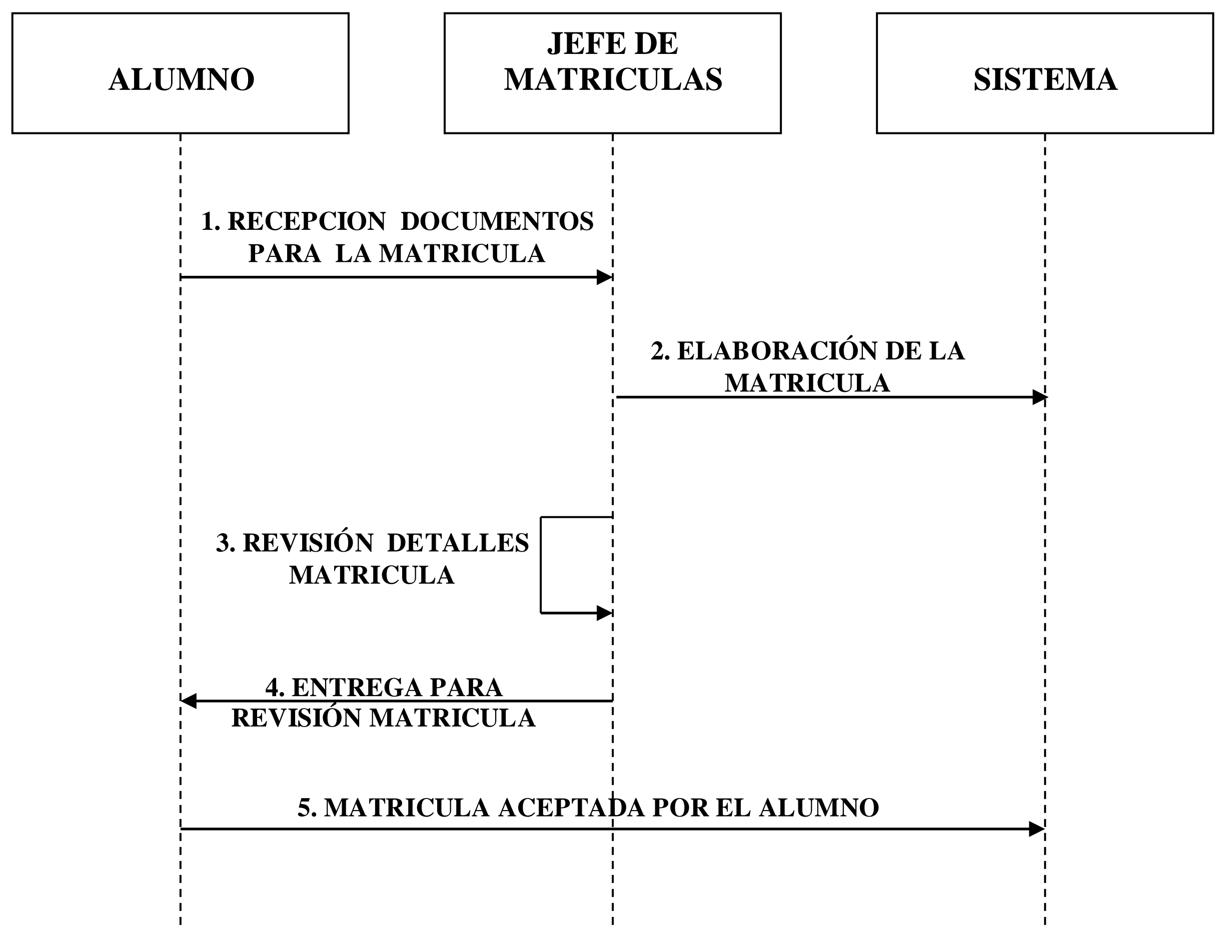
- La entrega de boletines es dirigida por cada uno de los profesores que tiene una reunión personal con cada alumno y se acudiente.
- El boletín tiene como objeto dar un soporte al desempeño del alumno en la institución

PROCESO: Matricula de alumnos
SUBPROCESO: Generación de documentos para el soporte de decisiones

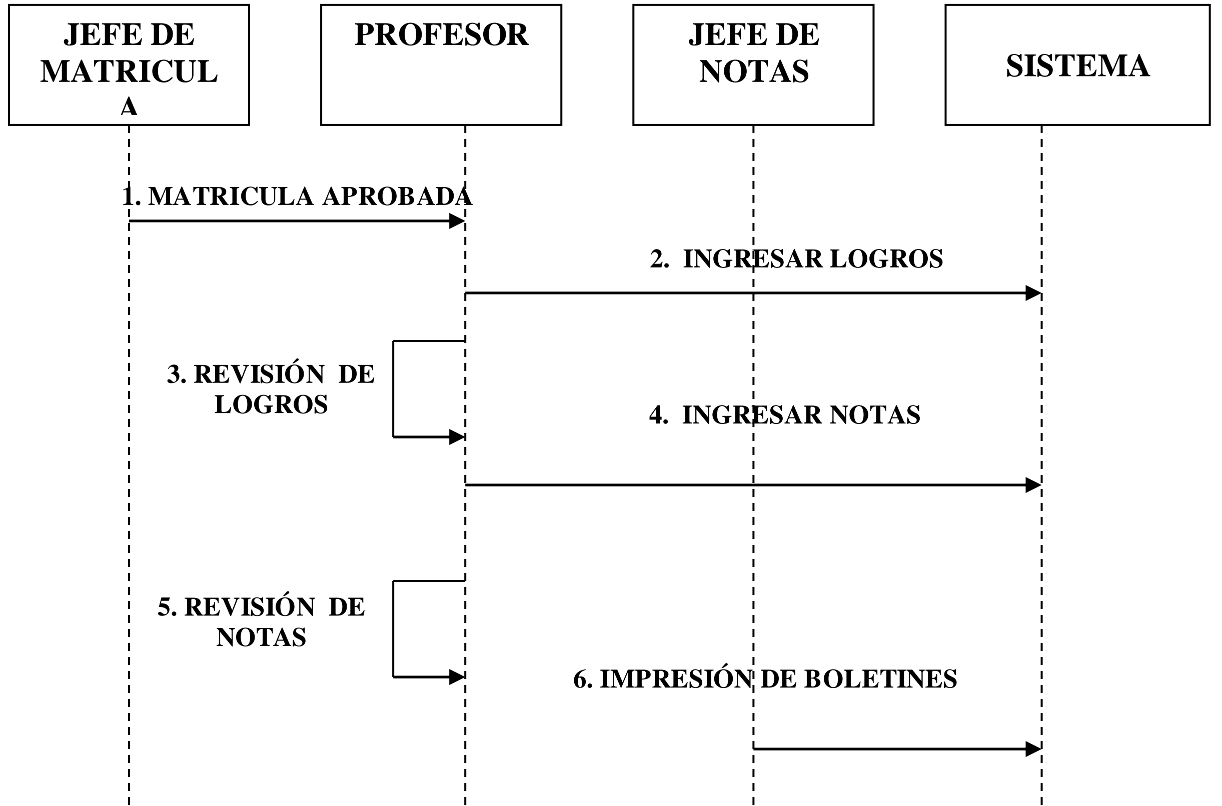


17.3 DIAGRAMAS DE SECUENCIA

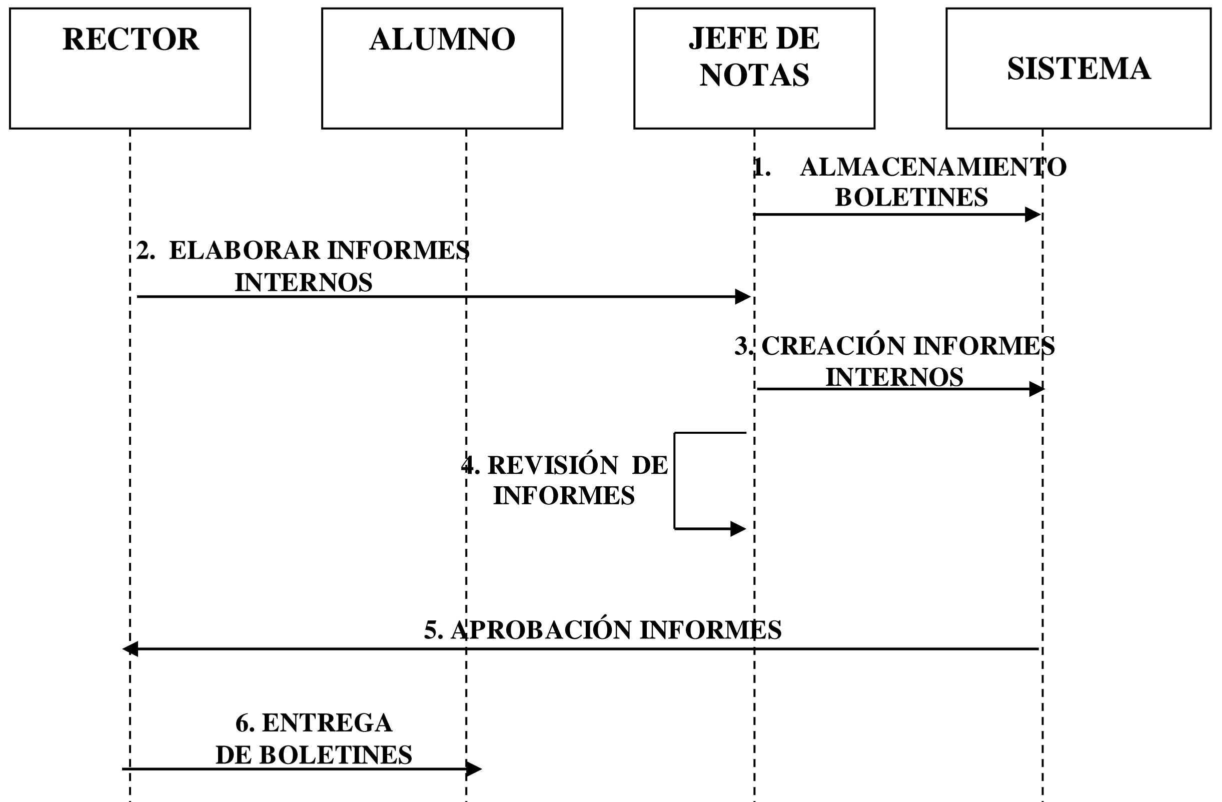
PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.



PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.



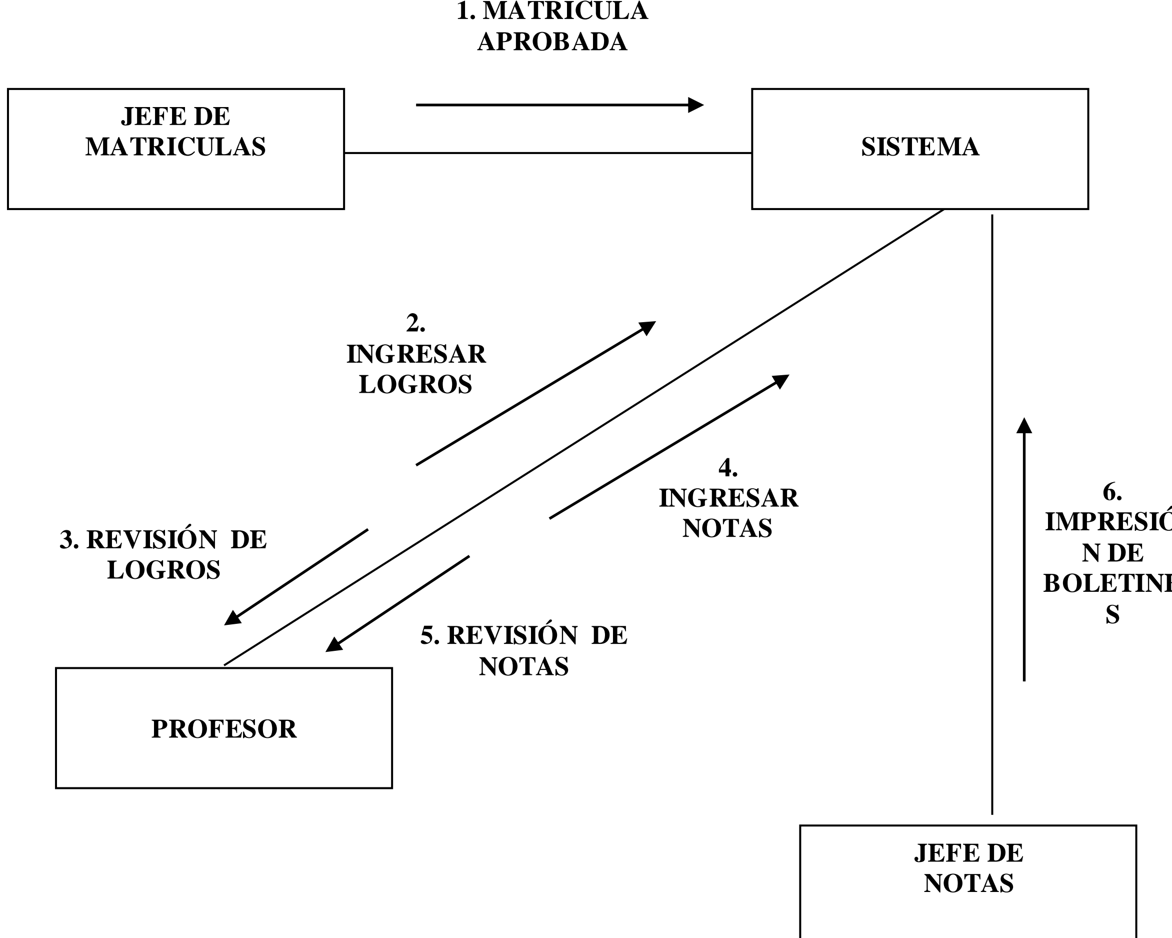
PROCESO: Matricula de alumnos
SUBPROCESO: Generación de documentos para el soporte de decisiones



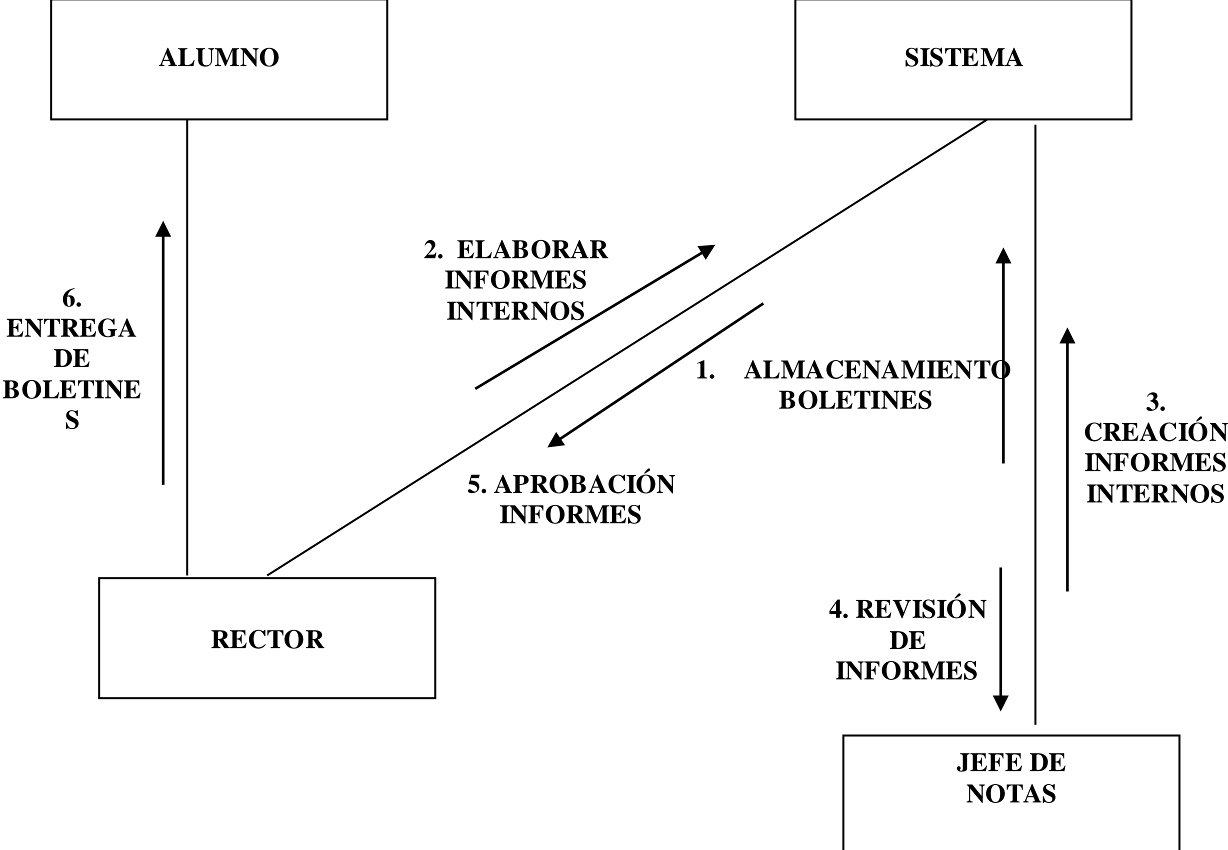
17.4 DIAGRAMAS DE COLABORACIÓN

PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.

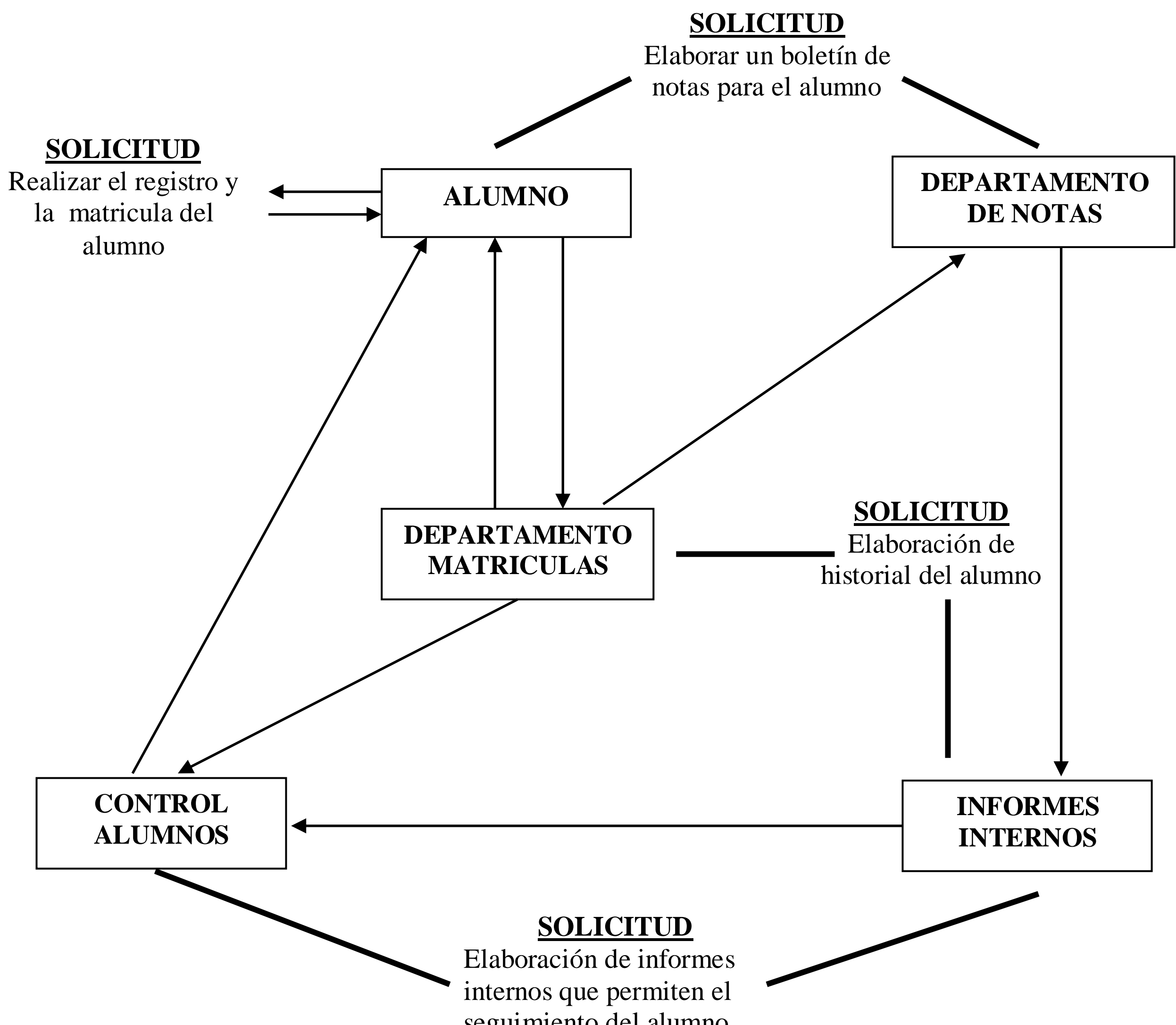
PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.



PROCESO: Matricula de alumnos
SUBPROCESO: Generación de documentos para el soporte de decisiones



17.5 DIAGRAMA INTERACCION ENTRE TIPOS DE OBJETOS



17.6 CICLO VITAL DEL OBJETO

Recepción de documentos matricula

Revisión y aprobación de documentos

Matricula aceptada

Matricula en espera

Matricula cancelada

Ingreso de logros

Aprobación por parte del docente

Logros aprobados

Logros rechazados

Ingreso de notas

Aprobación por parte del docente

Notas aprobadas

Notas rechazadas

Elaboración boletines

Preparación informes internos

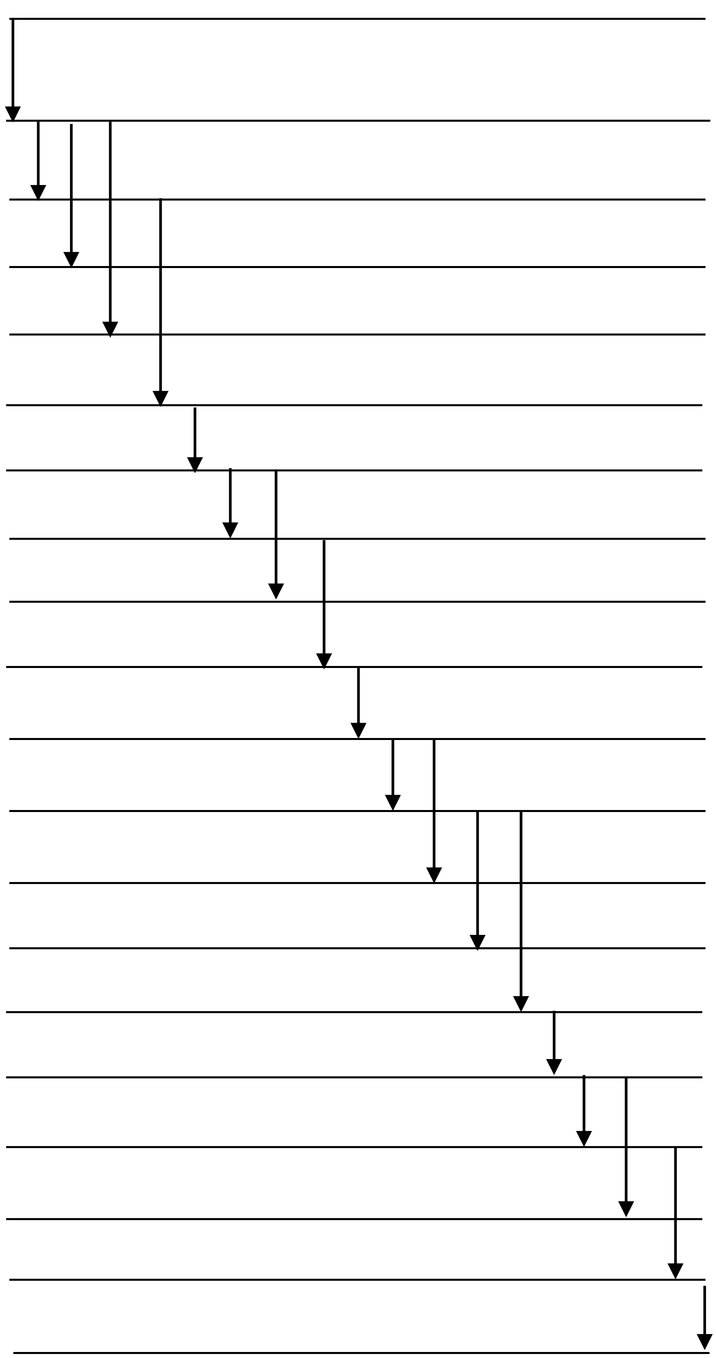
Aprobación por parte del docente

Informes internos aprobados

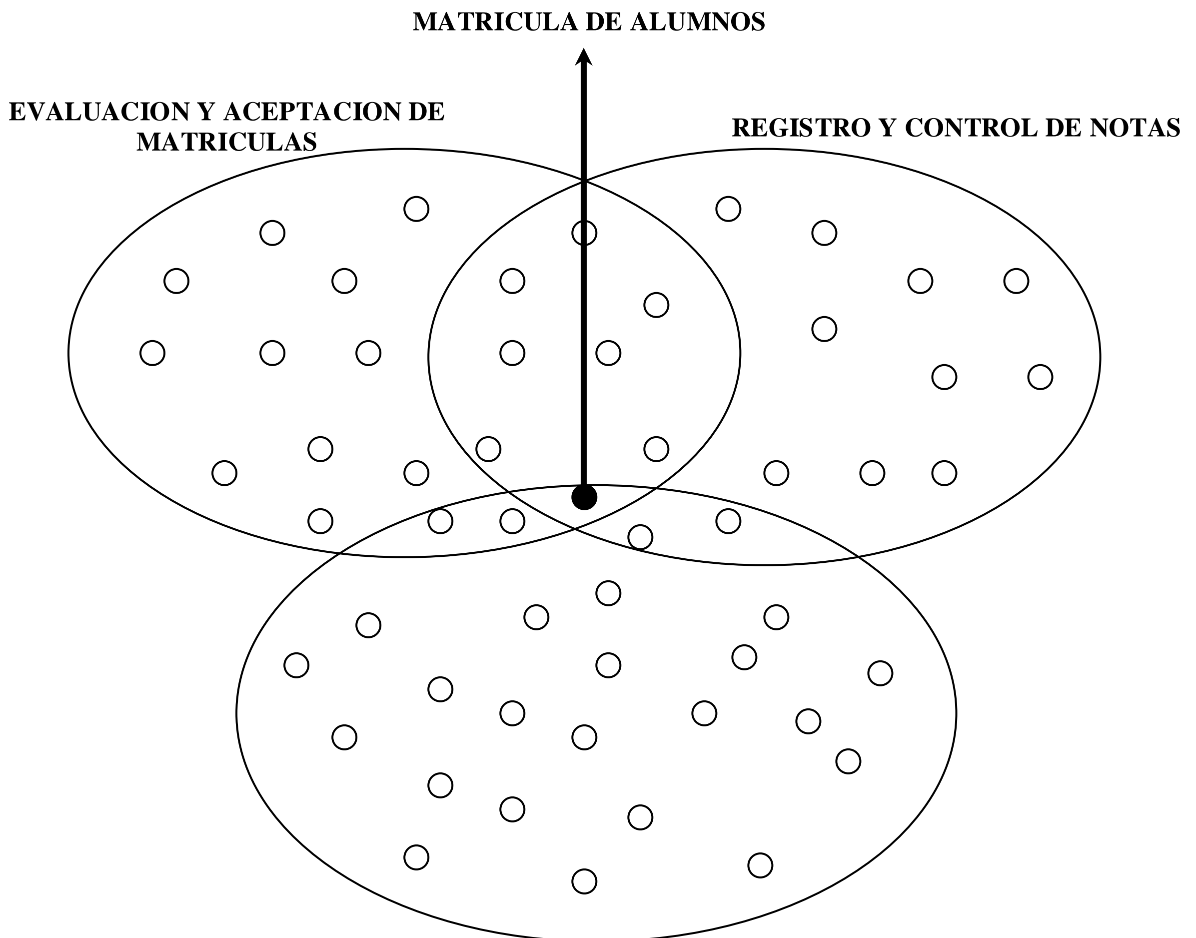
Informes internos rechazados

Elaboración de informes internos

Entrega de boletines

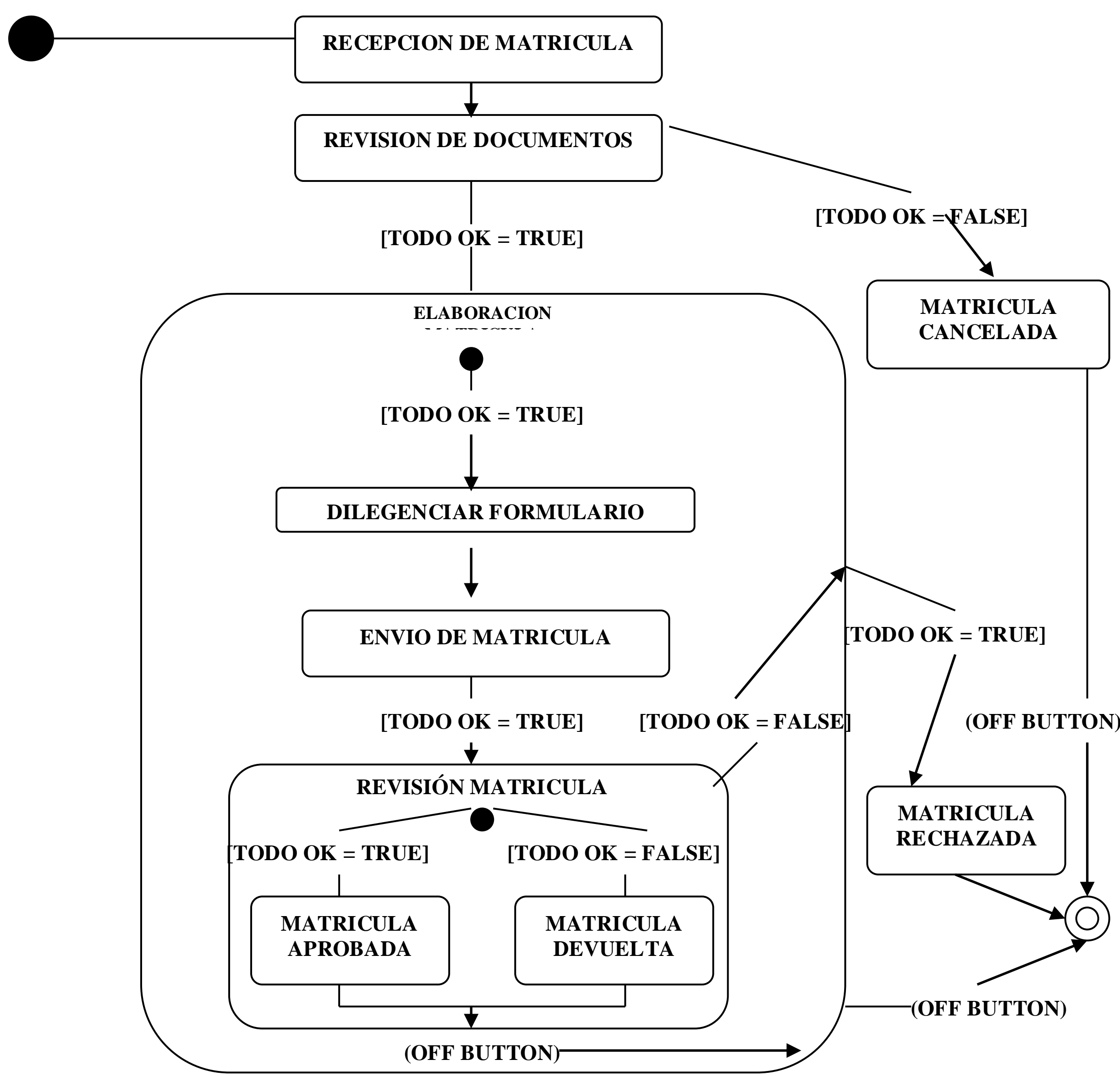


17.7 ESTADOS DE UN OBJETO

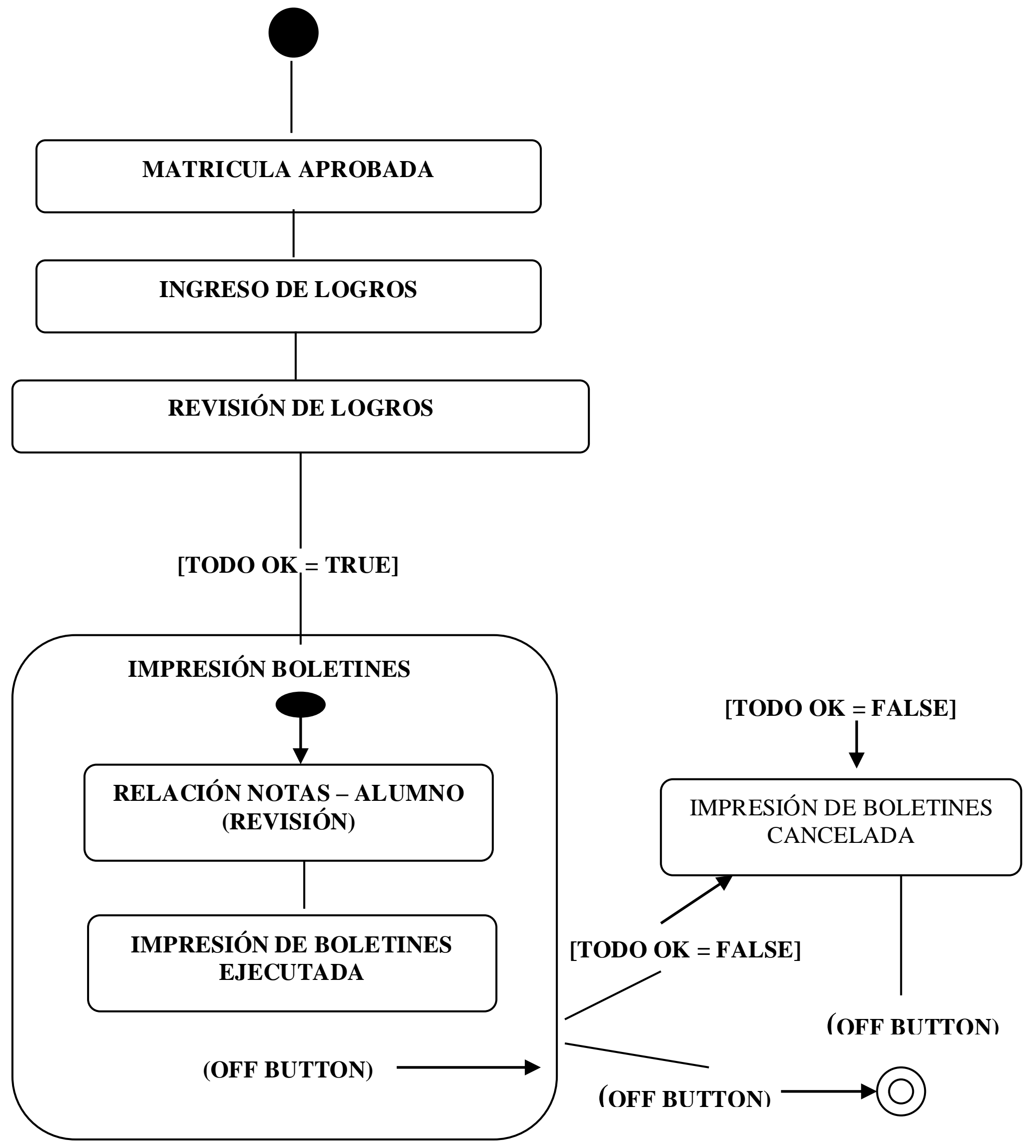


17.8 DIAGRAMAS DE ESTADOS

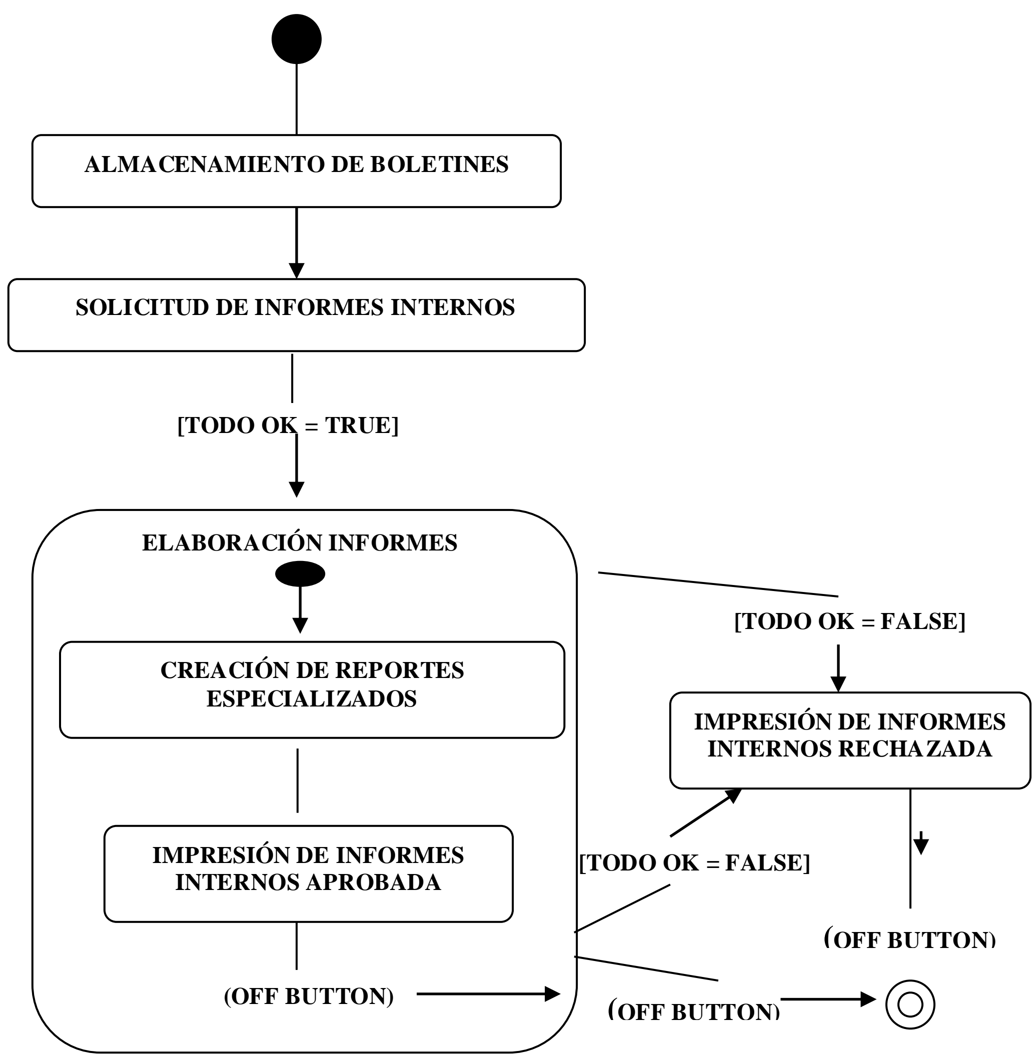
PROCESO: Matricula de alumnos
SUBPROCESO: Evaluación y Aceptación de Matriculas.



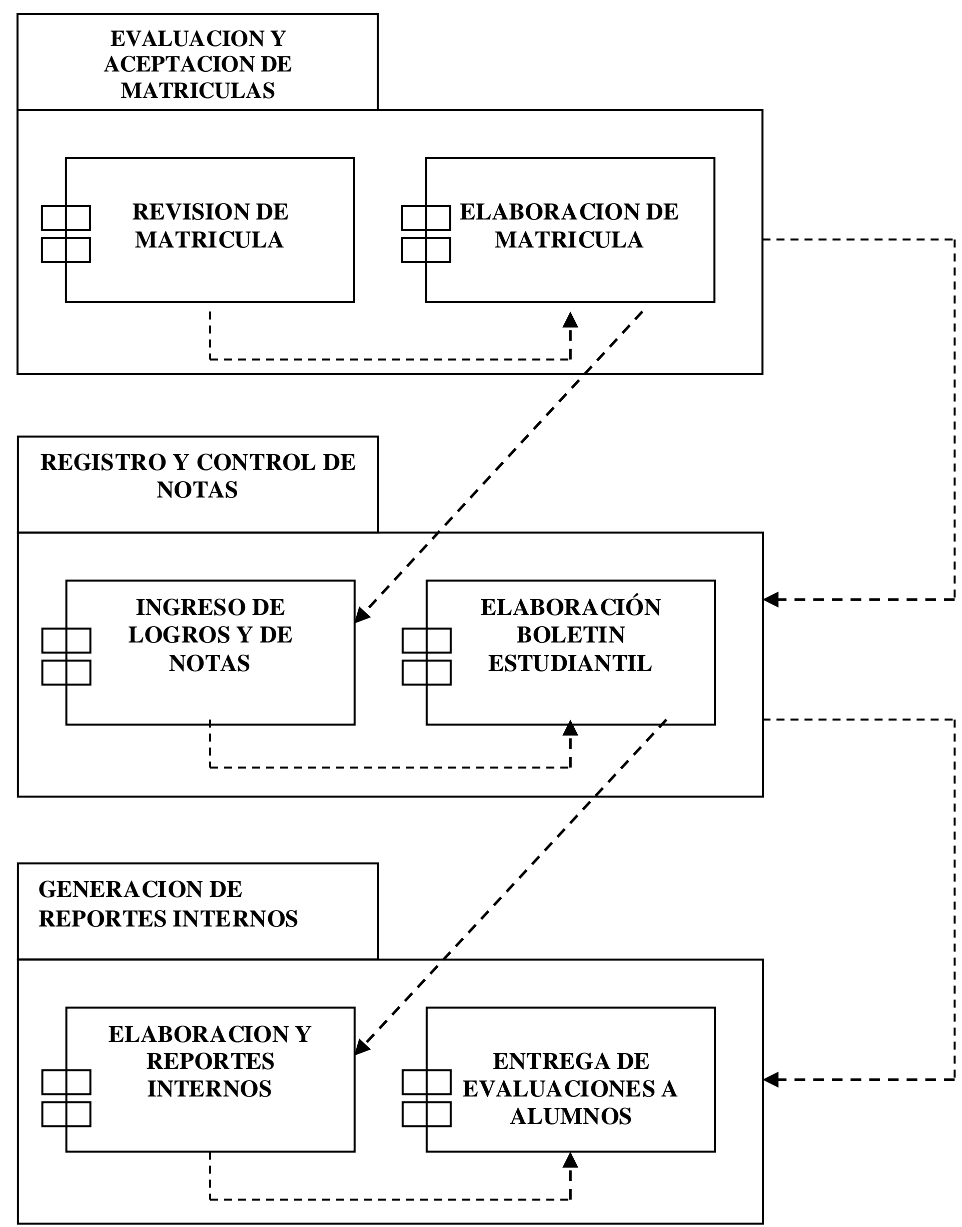
PROCESO: Matricula de alumnos
SUBPROCESO: Registro y control de notas.



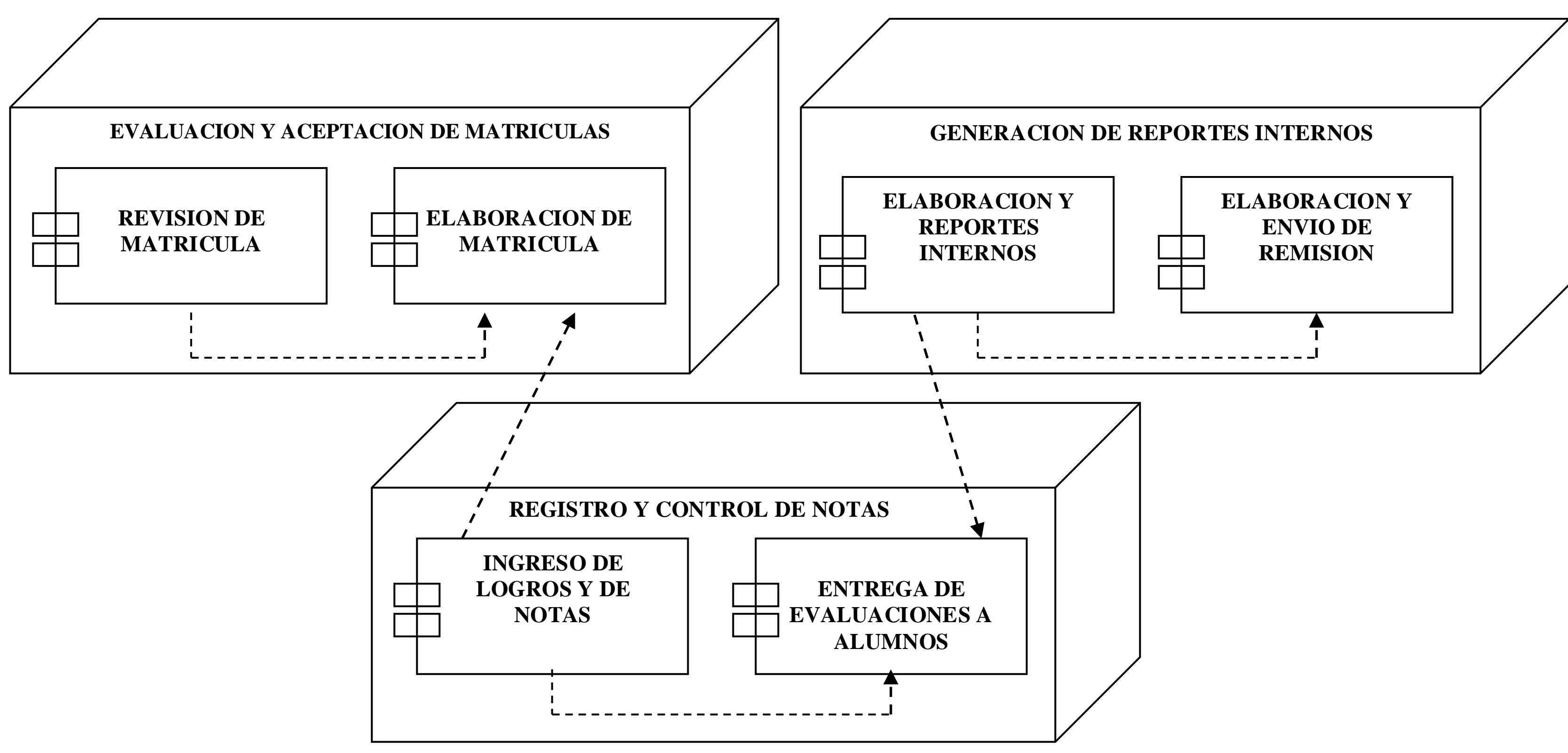
PROCESO: Matricula de alumnos
SUBPROCESO: Generación de documentos para el soporte de decisiones



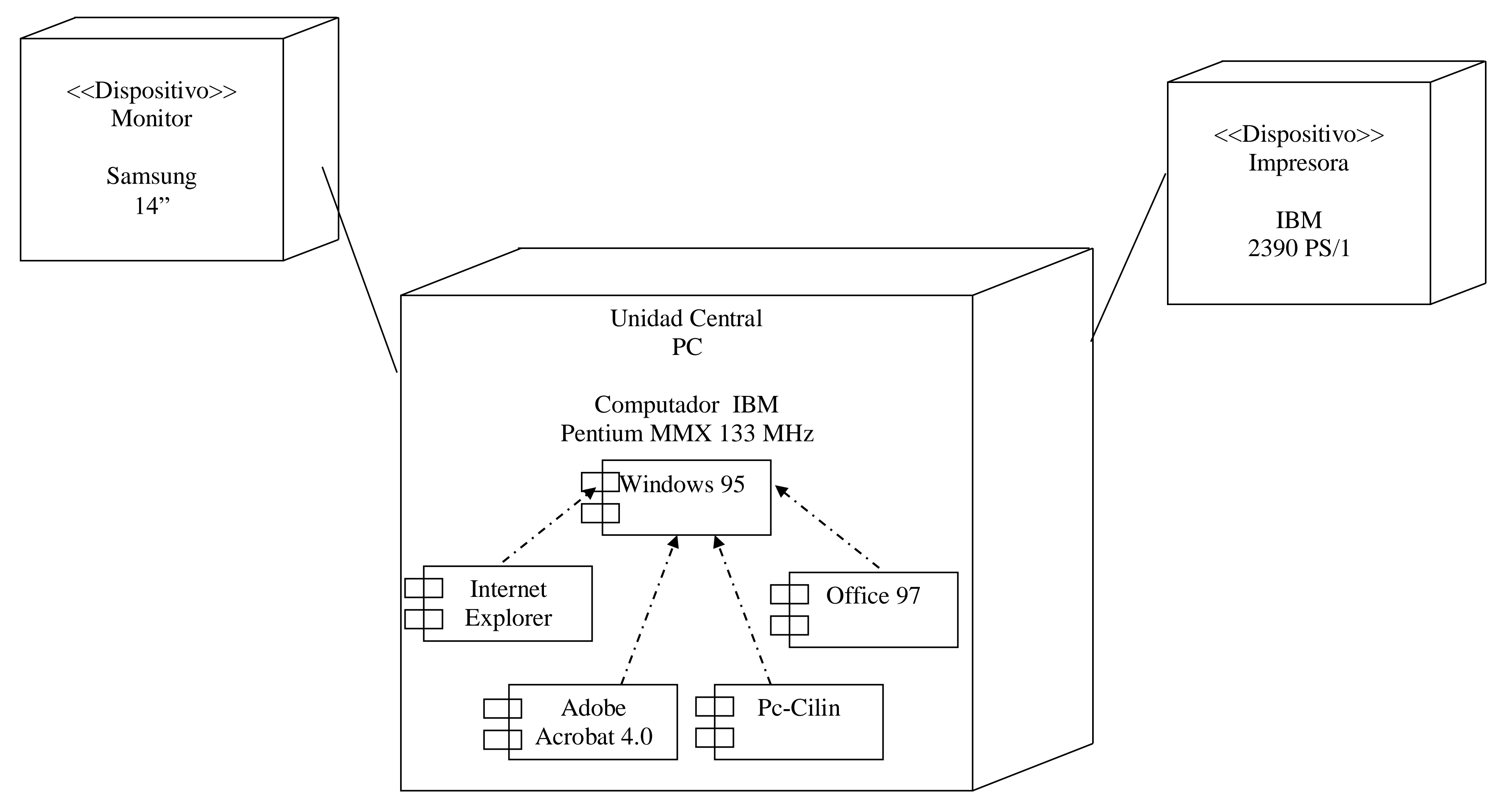
17.9 DIAGRAMAS DE COMPONENTES



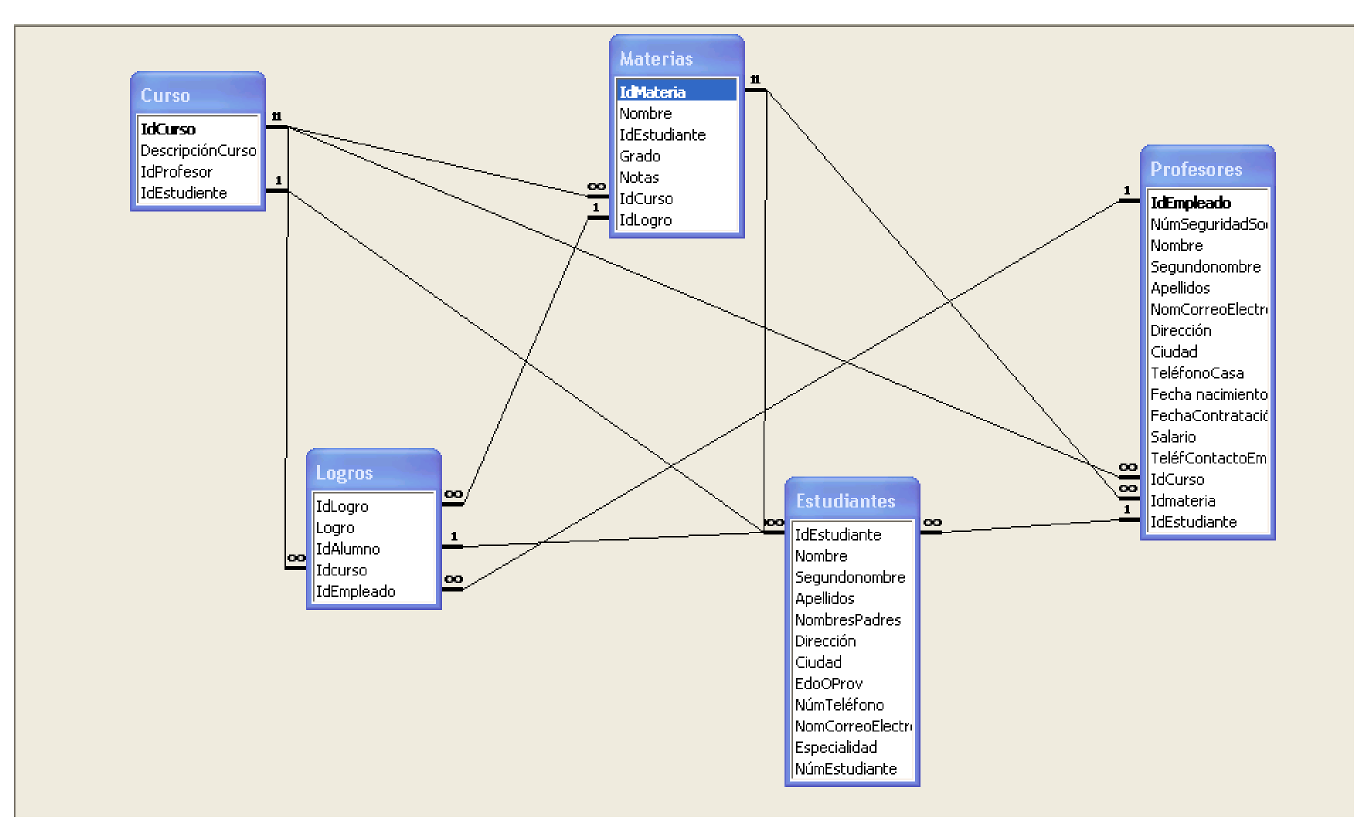
17.10 DIAGRAMAS DE DESPLIEGUE



17.11 DIAGRAMA DE DISTRIBUCION



18 DIAGRAMA ENTIDAD – RELACION



18.1 DICCIONARIO DE DATOS

- Tabla: estudiantes

  @ IdEstudiante = { Números }

  Nombre = { nombre + (segundo nombre) + apellido + segundo
  apellido }

  Acudientes = { nombre + (segundo nombre) + apellido + segundo apellido }

  Dirección = { carácter legal }

  Teléfono = { carácter legal }

  Documento de identidad = { carácter legal }

  Lugar de nacimiento = { carácter legal }

  Fecha de nacimiento = { carácter legal }
  **
  *Unidades: días a partir del 1° de enero de 1970

  Sexo = [M | F]

  Observaciones = { carácter legal }
  **
  *Indicaciones especiales que se realicen del alumno

  nombre = { carácter legal }

  segundo nombre = { carácter legal }

  apellido = { carácter legal }

  segundo apellido = { carácter legal }

  carácter legal = [A – Z | a – z | 0-9 |' | - | # | °]

- Tabla: profesores

@ IdEmpleados = { carácter legal }

Nombre = { nombre + (segundo nombre) + apellido + segundo
apellido }

Dirección = { carácter legal }

Teléfono = { carácter legal }

Documento de identidad = { carácter legal }

Lugar de nacimiento = { carácter legal }

Fecha de nacimiento = { carácter legal }
**
*Unidades: días a partir del 1° de enero de 1950

Sexo = [M | F]

Especialidad = { carácter legal }

E- mail = { carácter legal }

Fecha de contratación = { carácter legal }
**
*Unidades: días a partir del 1° de enero de 1996

Sueldo = { carácter legal }

Seguro social = { carácter legal }

Numero S.S = { carácter legal }

nombre = { carácter legal }

segundo nombre = { carácter legal }

apellido = { carácter legal }

segundo apellido = { carácter legal }

carácter legal = [A – Z | a – z | 0-9 |’ | - | # | ° | @]

- Tabla: Materia

  @ IdMateria = { carácter legal }

  Materia = { carácter legal }

  Grado = { carácter legal }

  Nota = { carácter legal }

  materia = { carácter legal }

  grado = { carácter legal }

  carácter legal = [A – Z | a – z | 0-9 | - | “ | #]

- Tabla: Logros

  @ IdLogro = { carácter legal }

  Logro = { carácter legal }

  Recuperación = {carácter legal}

  Logro = {carácter legal }

  Recuperación = {carácter legal}

  Carácter legal = [A – Z | a – z | 0-9 | - | “” | #]

- Tabla: Cursos

  @ IdCurso= {carácter legal}

  Encargado = {carácter legal}

  Salón = {carácter legal}

  Encargado = {carácter legal}

  Salón = {carácter legal}

  Carácter legal = [A – Z | a – z | 0-9 | - | “” | #]

19 PRUEBAS ORIENTADAS A OBJETOS

Todos los modelos orientados a objetos deben ser probados, para asegurar la exactitud, compleción y consistencia, dentro del contexto de la sintaxis, semántica y pragmática del modelo.

La estrategia clásica para la prueba de software de ordenador, comienza con “probar lo pequeño” y funciona hacia fuera haciendo “probar lo grande”. Se comienza con las pruebas de unidad, después se progresa hacia las pruebas de integración y se culmina con las pruebas de validación del sistema. En aplicaciones convencionales, las pruebas de unidad se centran en las unidades de programa compilables más pequeñas (el subprograma), por ejemplo: módulo, subrutina, procedimiento, componente. Una vez que cada una de estas unidades han sido probadas individualmente, se integran en una estructura de programa, mientras que se ejecutan una serie de pruebas de regresión para descubrir errores, debidos a las interfaces entre los módulos y los efectos colaterales producidos por añadir nuevas unidades. Por último, el sistema se comprueba como un todo para asegurarse de que se descubren los errores en requisitos.

19.1 LAS PRUEBAS DE UNIDAD EN EL CONTEXTO DE LA 00

Cuando se considera el software orientado a objetos, el concepto de unidad cambia. La encapsulación conduce a la definición de clases y objetos. Esto significa que cada clase y cada instancia de una clase (objeto), envuelven atributos (datos) y operaciones (también conocidos como métodos o servicios), que manipulan estos datos. En vez de probar un módulo individual, la unidad más pequeña comprobable es la clase u objeto encapsulado. Ya que una clase puede contener un número de operaciones diferentes, y una operación particular debe existir como parte de un número de clases diferentes, el significado de la unidad de prueba cambia drásticamente.

La prueba de clases para el software 00 es el equivalente de las pruebas de unidad para el software convencional. A diferencia de las pruebas de unidad del software convencional que tienden a centrarse en el detalle algorítmico de un módulo y de los datos que fluyen a través de la interfaz del módulo, la prueba de clases para el software 00 se conduce mediante las operaciones encapsuladas por la clase y el comportamiento de la clase.

19.2 LAS PRUEBAS DE INTEGRACIÓN EN EL CONTEXTO 00

Ya que el software orientado a objetos no tiene una estructura de control jerárquico, las estrategias convencionales de integración descendente (top-down) y ascendente (bottom-up) tienen muy poco significado. En suma, la integración de operaciones una por una en una clase (la aproximación de la integración incremental convencional), a menudo es imposible por la "interacción directa e indirecta de los componentes que conforman la clase".

Existen dos estrategias diferentes para las pruebas de integración de los sistemas 00. El primero, las pruebas basadas en hilos, integran el conjunto de clases requeridas, para responder una entrada o suceso al sistema. Cada hilo se integra y prueba individual mente. Las pruebas de regresión se aplican para asegurar que no ocurran efectos laterales.

La segunda aproximación de integración, la prueba basada en el uso, comienza la construcción del sistema probando aquellas clases (llamadas clases independientes), que utilizan muy pocas (o ninguna) clases servidoras. Después de que las clases independientes se prueban, esta secuencia de pruebas por capas de clases dependientes continúa hasta que se construye el sistema completo. A diferencia de la integración convencional, el uso de drivers y stubs como operaciones de reemplazo, debe evitarse siempre que sea posible.

19.3 PRUEBAS DE VALIDACIÓN EN UN CONTEXTO 00

Al nivel de sistema o de validación, los detalles de conexiones de clases desaparecen así como la validación convencional, la validación del software 00 se centra en las acciones visibles al usuario y salidas reconocibles desde el sistema. Para ayudar en la construcción de las pruebas de validación, el probador debe utilizar los casos de uso, que son parte del modelo de análisis. Los casos de uso proporcionan un escenario, que tiene una gran similitud de errores con los revelados en los requisitos de interacción del usuario.

Los métodos de prueba convencionales de caja negra pueden usarse para realizar pruebas de validación.

20 CONCLUSIONES

1. Para desarrollar un Sistema de Información que satisfaga todas las necesidades de una organización es necesario realizar un estudio exhaustivo de su sistema organizacional como un todo y comprender los canales informales, interdependencias, personas y funciones clave así como los enlaces críticos de información.
2. Los lenguajes orientados a objetos facilitan el desarrollo de sistemas más precisos, generales y robustos gracias a conceptos como identidad, clasificación, polimorfismo y herencia.
3. Existen tres estrategias para el desarrollo de sistemas, el método clásico del ciclo de vida de desarrollo de sistemas, el método de desarrollo por análisis estructurado y el método de construcción de prototipos, que tienen un uso amplio en organizaciones de todo tipo y tamaño y resultan efectivos cuando se emplean adecuadamente.
4. Dentro de los indicadores útiles para determinar el grado de calidad del software tenemos la corrección, la facilidad de mantenimiento, la integridad y la facilidad de uso.
5. El software orientado a objeto evoluciona iterativamente y debe dirigirse teniendo en cuenta que el producto final se desarrollara a partir de una serie de incrementos.
6. Los casos de uso representan una visión, a un alto nivel funcional, de un sistema en UML.
7. UML tiene dos notaciones equivalentes para definir las interacciones, el diagrama de secuencias y el diagrama de colaboración.
8. A diferencia de las pruebas de unidad del software convencional que tienen a centrarse en el detalle algorítmico de un modulo, las pruebas de clase para el software orientado a objetos se conduce mediante las operaciones encapsuladas por la clase y el comportamiento de la clase.
9. El objetivo de las normas ISO 9000-3 es proveer las especificaciones de cómo aplicar la ISO 9001 al desarrollo de software, implementación y mantenimiento.
10. Existen cuatro términos que tienen una influencia sustancial en la gestión de proyectos de software: personal, producto, proceso y proyecto.
11. El software se ha convertido en el elemento clave de la evolución de los sistemas y productos informáticos, suministrando solución a diferentes tipos de requerimientos organizacionales.
12. En programación orientada a objetos, las clases permiten la agrupación de objetos que comparten las mismas propiedades y

comportamientos.

13. El esfuerzo del programador ante una aplicación orientada a objetos se centra en la identificación de las clases, sus atributos y operaciones asociadas.

21 BIBLIOGRAFÍA

BOOCH, Grady. 1999. *El lenguaje unificado de modelado*. México, Addison Wesley.

COHOON, James. 2000. *Programación y diseño en C++ Introducción a la programación y al diseño orientado a objetos*. México. Mac Graw Hill.

DEPARTAMENTO DE INVESTIGACIÓN. 2004. *Guía para la elaboración de proyectos de investigación en Ingeniería*. Bogotá, Universidad Libre. Mc Graw Hill.

MICROSOFT PRESS. 1998. *Visual Basic 6.0 Guía de Acceso a Datos.* España, Mc Graw Hill.

MICROSOFT PRESS. 1998. *Visual Basic 6.0 Manual del Programador.* España, Mc Graw Hill.

MICROSOFT PRESS. 1998. *Visual Basic 6.0 Referencia y Control Guía de Herramientas y Componentes.* España, Mc Graw Hill.

MIRANDA, Juan José. 2001. *Gestión de proyectos*. Independiente.

PRESSMAN, Roger. 2002. *Ingeniería del Software.* España, Mc Graw Hill.

SCHMULLER, Joseph. 1999. *Aprendiendo UML en 24 horas*. México, Prentice Hall.

SENN, James. 1992. *Análisis y Diseño de Sistemas de Información.* México,

SILBERSCHATZ KORTH, Sudarshan. 2002. *Fundamentos de Base de Datos.* México, Mac Graw Hill.

VIESCAS, John. 1999. *Running Microsoft Access 2000*. España, Mac Graw Hill.

YOURDON, Edward. 1993. *Análisis Estructurado Moderno.* México, Prentice Hall.

22 INFOGRAFÍA

cursos.frcu.utn.edu.ar/extension/vb/febrero2003/

delta.cs.cinvestav.mx/~pmejia/davila-mejia.pdf

mat21.etsii.upm.es/ayudainf/ aprendainf/VisualBasic6/v**basic**60.pdf

Members.aol.com/acockburn/papers/onusecases.htm

mini.net/cetus/oo_uml.html

www.dacs.dtic.mil/techs/design/Design.TOC.html

www.dcc.uchile.cl/~psalinas/**uml**/introduccion.html

www.edicionsupc.es/virtuals/caplln/IN03303X.htm

www.fdi.ucm.es/profesor/anavarro/
4._Proceso_de_**software**_y_**metricas**_de_proyectos.pdf

www.itapizaco.edu.mx/paginas/Poo/Tutorial/Home.html

www.itba.edu.ar/capis/webcapis/cursosdisponibles/cis2-
estimaciondeproyectosde**software**yconfiguracion.pdf

www.kinetica.com/ootips/ood-principles.html

www.learnthenet.com/spanish/glossary/oop.htm

www.lsi.uned.es/gcal/contenidos.html

www.megaserv.com/Pdf/ ContenidoProgramáticoVisualBasic.pdf

www.microsoft.com/spanish/msdn/
comunidad/dce/1/entrenamiento/foxpro/1a.asp

www.omg.org/**uml**/

www.rbsc.com

www.univ-paris1.fr/CRINFO/dmrg/MEE/misop014

www.pmi.org

www.4pm.com

www.projectmanagement.com

www.sei.cmu.edu

www.qualityworld.com

www.inv.nasa.gov/SWG/resources/NASA-GB-001-94.pdf