Universitat Politècnica de Catalunya

Proyecto Final de Carrera

Métodos ágiles para el desarrollo de software

TITULAR: Raúl Úbeda González

TITULACIÓN: Ingeniero de telecomunicación

ESPECIALIDAD: Sin especialidad

TÍTULO: Métodos ágiles para el desarrollo de software

DIRECTOR DEL PROYECTO: Jaume Mussons Sellés

DEPARTAMENTO: Organización de empresas

Marzo 2009

*Dedico este proyecto a mi familia,
por soportar el arduo camino
que he recorrido para poder llegar
a iniciar este proyecto.*

*Agradezco la disponibilidad de mi director de proyecto,
Jaume Mussons, a quien espero que este trabajo
ayude en su labor docente.*

Lo que consigas, lo puedes perder, pero lo que aprendas, no lo perderás nunca.

Resumen

Un método de desarrollo de software ágil es aquel capaz de adaptarse a los cambios, especialmente de los requisitos, según el *feedback* constante y temprano del cliente. El objetivo es minimizar el tiempo y el coste de realizar un proyecto de software, pero manteniendo la calidad. Todo esto se traduce en un ímpetu por las pruebas continuas, preferentemente automatizadas, y la integración continua para crear versiones de forma seguida para poderlas mostrar al cliente y recoger su opinión.

Los métodos ágiles comparten un sustrato: el manifiesto ágil y los 12 principios ágiles, donde no se detallan trucos de programación sino recomendaciones para las personas dedicadas al proyecto. Se trata más de una manera de actuar, a alto nivel, que de prácticas específicas a bajo nivel, que se dejan a la elección de los propios programadores y su experiencia.

En primer lugar, se dará una perspectiva concisa de los métodos predecesores. Seguidamente, se analizarán los 14 métodos ágiles más comunes, detallando el proceso con sus fases, las prácticas, papeles, workproducts o artefactos que se generan en cada fase, etc. También se mostrarán herramientas software que ayudan a adoptar prácticas ágiles o el método completo.

Se verán las estadísticas sobre los puntos débiles de los métodos pesados y sus consecuencias, especialmente el Chaos Report (1994), y la mejora que ha supuesto la aplicación de prácticas o métodos ágiles desde entonces. Como conclusiones, se resumirán las muchas similitudes entre los diferentes métodos. También se discutirán ventajas e inconvenientes de las prácticas ágiles, muchas de ellas basadas en la experiencia y el sentido común, pero que pocas veces se aplican todas juntas para lograr una gestión rápida y adecuada.

Se puede encontrar un resumen de toda esta exposición en forma de 120 transparencias en los anexos.

Palabras clave

Desarrollo ágil software, Agile development, adaptive process, Extreme Programming, XP, Scrum, Crystal, Feature Driven Development FDD, Rational Unified Process RUP, Enterprise EUP, Agile AUP, Dynamic Systems Development Method DSDM, Adaptive Software Development ASD, Open Source Software OSS, Lean LSD, AM, Evolutionary Project Management Evo, Microsoft Solutions Framework MSF, Internet Speed ISD, Pragmatic programmer.

Abstract

An agile software development method adapts itself to the changes during the whole programming process, mainly during the requirements analysis phase, according to the constant and early client feedback. The goal is to minimize wastes of time and money while doing a software project, but assuring quality. To achieve this, great effort is made in continuous testing (preferably automated), and continuous integration to build a large number of releases to show the clients and ask for their feedback.

All of the agile methods share the same basis: the agile manifesto and the 12 agile principles, where there are no programming tricks but recommendations dedicated to the whole team project. They are a global high-level guideline, not specific low level practices, which are left to the developers' experience.

First of all, a brief overview on predecessor methods, like waterfall or incremental delivery, will be done. After that, we will take a closer look to the 14 most common agile methods, analyzing their processes, phases, practices, roles, workproducts, deliverables, etc. Some software tools will also be shown to help to adopt some agile practices or the complete method.

Focusing on The Chaos Report (1994), weaknesses of the "heavy" methods will be highlighted, as well as the improvements when adopting agile practices or methods. Concerning conclusions, a brief summary comparing the different methods will be done, analyzing the practices, which are not new, but founded in experience and common sense,although unfortunately, hardly ever put in practice all of them at the same time to manage and develop software quickly and efficiently.

A 120-slide summary of the complete report can be found as the last appendix.

Keywords

Agile software development, adaptive process, Extreme Programming XP, Scrum, Crystal, Feature Driven Development FDD, Rational Unified Process RUP, Enterprise EUP, Agile AUP, Dynamic Systems Development Method DSDM, Adaptive Software Development ASD, Open Source Software OSS, Lean LSD, AM, Evolutionary Project Management Evo, Microsoft Solutions Framework MSF, Internet Speed ISD, Pragmatic programmer.

Contenido

0. Presentación, objetivos y alcance ***14***

0.1. Presentación **14**
0.1.1. Situación anterior 14
0.1.2. Análisis de necesidades 15
0.1.3. Solución aportada por los métodos ágiles 15
0.1.4. El manifiesto ágil y los 12 principios ágiles 18
0.1.5. Uso de los métodos (o prácticas) ágiles en la actualidad 19

0.2. Objetivos **21**
0.2.1. Esquemas de los métodos más utilizados como base 23

0.3. Alcance **24**
0.3.1. Métodos 25
0.3.2. Software 26

1. Introducción ***28***

1.1. Marco histórico **28**

1.2. La necesidad de nuevos métodos **36**

1.3. El Manifiesto Ágil **39**

1.4. Los 12 principios de los métodos ágiles **40**

1.5. Características comunes de los métodos ágiles **41**

1.6. Manifiesto de proyecto ágil **44**

2. Métodos Ágiles de Desarrollo de Software ***48***

2.1. Extreme Programming – XP **48**
2.1.1. Los valores de XP (1999) 49
2.1.2. Los nuevos valores de XP (2004) 50
2.1.3. Proceso (1999) 50
2.1.4. Papeles y responsabilidades (1999) 53
2.1.5. Las prácticas (1999) 54
2.1.6. Principios de XP (1999) 55
2.1.7. Las nuevas prácticas (2004) 60
2.1.8. Las prácticas secundarias (2004) 61
2.1.9. Lemas de XP 64
2.1.10. Los nuevos principios de XP 65
2.1.11. Adopción y experiencias 67
2.1.12. Limitaciones 68
2.1.13. Investigación actual 68

2.2. Scrum **70**
2.2.1. Proceso 71
2.2.2. Papeles y responsabilidades 72
2.2.3. La práctica 73
2.2.4. Adopción y experiencias 76
2.2.5. Limitaciones 80

2.3. Familia de métodos Crystal **81**
2.3.1. Proceso 83
2.3.2. Papeles y responsabilidades 86
2.3.3. La práctica 88
2.3.4. Adopción y experiencias 93
2.3.5. Limitaciones 93

2.4. Feature Driven Development – FDD **94**
2.4.1. El proceso 94

2.4.2. Papeles y responsabilidades 96
2.4.3. La práctica 98
2.4.4. Adopción y experiencias 100
2.4.5. Limitaciones 100

2.5. Rational / Enterprise / Agile Unified Process – RUP, EUP y AUP 101
2.5.1. Proceso de RUP 101
2.5.2. Papeles y responsabilidades de RUP 103
2.5.3. La práctica de RUP 104
2.5.4. Adopción y experiencias de RUP 104
2.5.5. Enterprise Unified Process (EUP) 104
2.5.6. Agile Unified Process (AgileUP, AUP o dX) 105
2.5.7. Limitaciones 112

2.6. Dynamic Systems Development Method – DSDM 113
2.6.1. Proceso 113
2.6.2. Papeles y responsabilidades 115
2.6.3. La práctica 116
2.6.4. Adopción y experiencias 116
2.6.5. Limitaciones y actualidad 117

2.7. Adaptive Software Development – ASD 118
2.7.1. Proceso 118
2.7.2. Papeles y responsabilidades 120
2.7.3. La práctica 120
2.7.4. Adopción y experiencias 121
2.7.5. Limitaciones y actualidad 121

2.8. Open Source Software Development – OSS 122
2.8.1. Proceso 122
2.8.2. Papeles y responsabilidades 123
2.8.3. La práctica 124
2.8.4. Adopción y experiencias 124
2.8.5. Limitaciones y actualidad 124

2.9. Lean Software Development - LSD 126
2.9.1. Proceso 126
2.9.2. Los 7 principios y las 22 herramientas por Darrell Norton 128
2.9.3. Papeles y responsabilidades 141
2.9.4. La práctica 141
2.9.5. Adopción y experiencias 146
2.9.6. Limitaciones 146

2.10. Agile Modelling – AM, AMDD 147
2.10.1. Proceso 148
2.10.2. Papeles y responsabilidades 149
2.10.3. La práctica 149

2.11. Evolutionary Project Management - Evo 152
2.11.1. Proceso 153
2.11.2. Papeles y responsabilidades 154
2.11.3. La práctica 155
2.11.4. Adopción y experiencias 157

2.12. Internet-Speed Development - ISD 159

2.13. Microsoft Solutions Framework - MSF 161
2.13.1. Proceso 162
2.13.2. Papeles y responsabilidades 165
2.13.3. La Práctica 166
2.13.4. Adopción y experiencias 168

2.14. Pragmatic Programming 170

3. Software 176

3.1. Herramientas 176
3.1.1. Pruebas automáticas 176
3.1.2. Integración continua 177
3.1.3. Trabajo colaborativo 180
3.1.4. Refactorización 181
3.1.5. Estándares de codificación 183

3.2. XPlanner 184

3.3. Evo Task Administrator 192

3.4. Rally 194

3.5. VersionOne Agile Team y Agile Enterprise 208

3.6. TargetProcess 211

3.7. ExtremePlanner 222

3.8. Atlassian 227

3.9. xProcess 229

3.10. Microsoft Visual Studio 232

4. Conclusiones y líneas de futuro 236

4.1. Resumen 236

4.2. Crítica a las prácticas ágiles 242

4.3. Métodos ágiles y desarrollo de software libre 246

4.4. Otros factores influyentes 248
4.4.1. La programación extrema y Smalltalk 248
4.4.2. La curva de Boehm y XP 249
4.4.3. El proyecto C3 250
4.4.4. Marketing. El poder de la palabra 252

4.5. Complementariedad y similitudes 252

4.6. Estadísticas 253
4.6.1. Estadísticas del grupo Standish (1994-2004) 253
4.6.2. Estadísticas de Ambysoft (2006) 255

4.7. Problemas comunes a los métodos ágiles 258

4.8. Limitaciones de los métodos ágiles 260

4.9. Líneas de futuro 262

5. Anexos 266

Anexo A. Acrónimos y glosario 266

Anexo B. The Agile Manifesto, Principles & Agile Project Manifesto 270

Anexo C. CASE - Computer Aided Software Engineering 272

Anexo D. CMM: Capability Maturity Model 276

Anexo E. UML - Unified Modeling Language 281

Anexo F. The CHAOS Report, 1994 (Standish group) 287

Anexo G. Casos de uso 295

Anexo H. RAD – Rapid Application Development 304

Anexo I. EUP – Enterprise Unified Process **309**
Anexo J. Scrum + XP, ejemplo de aplicación **316**
Anexo K. Dilbert y los métodos ágiles **331**
Anexo L. Epigramas sobre la programación **333**
Anexo M. Resumen en transparencias **337**
6. Bibliografía e índices ***400***
6.1. Métodos ágiles en Internet **400**
6.2. Índice de ilustraciones **402**
6.3. Índice de tablas **405**
6.4. Bibliografía **406**

CAPÍTULO

0

CONTENIDO

0.1. Presentación

0.2. Objetivos

0.3. Alcance

PRESENTACIÓN, OBJETIVOS Y ALCANCE

En este capítulo veremos una breve presentación del trabajo completo, justificando la necesidad

También delimitaremos los objetivos y el alcance, ya que como se verá, el tema es muy amplio, abarcando muchos métodos y también mucho software que ayuda a aplicar esos métodos. Muchos de estos métodos y programas necesitarían un proyecto para ellos solos si se pretendiera analizarlos a fondo.

“Esto es lo que pedí, pero no es realmente lo que necesitaba”

Cliente insatisfecho al ver el proyecto acabado

0. PRESENTACIÓN, OBJETIVOS Y ALCANCE

0.1. Presentación

El abaratamiento y el uso masificado de ordenadores y otros dispositivos con software como teléfonos móviles, PDA's, consolas de videojuegos, etc. supone una gran demanda de software, no sólo en número sino también en variedad.

A su vez, el incremento constante de la velocidad de proceso y capacidad de almacenamiento, provoca que la evolución del software sea continua y rápida, reclamando métodos de desarrollo rápidos.

La imparable competencia de países como la India en el sector de la programación, obliga a que las empresas ajusten al máximo costes y tiempo, manteniendo una calidad aceptable, para poder sobrevivir. Esta es la razón para buscar métodos o *best practices* que permitan desarrollar software de calidad de forma eficaz con mínimos costes, es decir, buscando la excelencia técnica: ***"hacer lo correcto, correctamente"***.

0.1.1. Situación anterior

Veamos el método en cascada (Royce, 1970), principal motivo, junto con CMM, del nacimiento de los métodos ágiles. El método en cascada debe su nombre a que como en una cascada, una vez que ha bajado el agua, no puede volver a subir. Aquí, una vez pasada una fase, tras realizar toda la extensa documentación requerida, es muy costoso volver atrás para corregir errores. Por tanto, en vez de adaptarse a los cambios, prefiere tratarlos pero no cambiar el diseño inicial. Es una filosofía totalmente diferente a la propuesta por los métodos ágiles que se adaptan constantemente a las condiciones del proyecto y a la opinión del cliente que es constante durante todo el desarrollo. En un método rígido como el de cascada, la aprobación del cliente sólo se pide al final, con las consecuencias negativas de tiempo y dinero que conlleva modificar un proyecto una vez ya totalmente acabado.

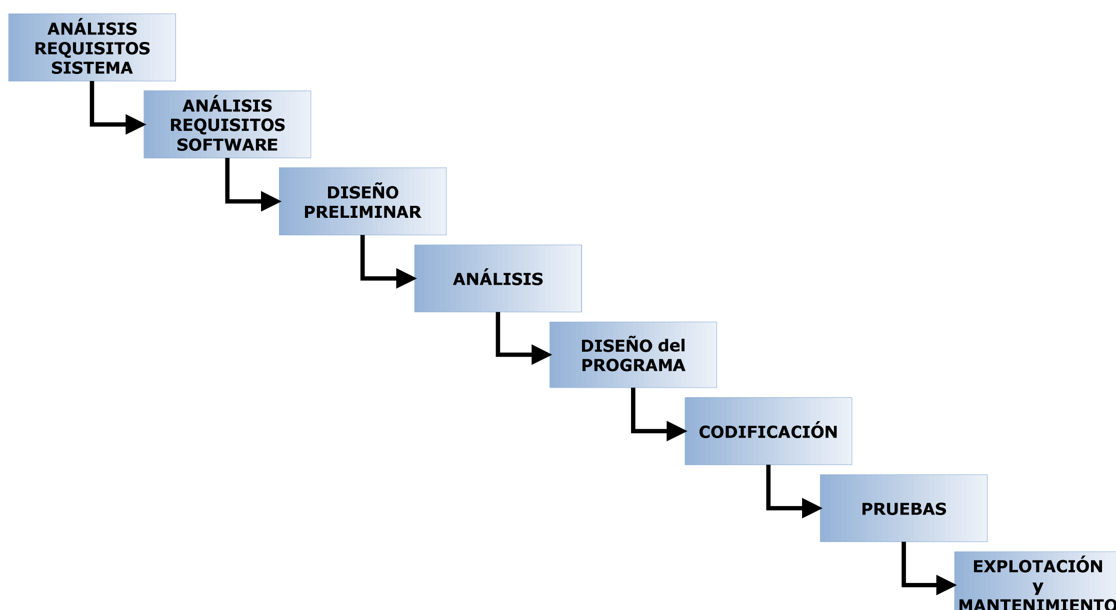


Ilustración 1. Método en cascada, contra el que luchan todos los métodos ágiles.

0.1.2. Análisis de necesidades

En 1994, el grupo Standish, analizó más de 8.000 proyectos de software y las conclusiones de *The CHAOS Report* (en los anexos se encuentra el informe completo) fueron:

- Sólo el 16.2% de los proyectos se completó a tiempo, cumpliendo el presupuesto y con las funcionalidades inicialmente propuestas.
- El 31.1% de los proyectos se canceló antes de acabar.
- El 52.7% de los proyectos no cumplió presupuesto, tiempo o funcionalidades iniciales (o varios factores). Costaron en media el 189% de su presupuesto inicial.
- Más del 25% se completó con un 25 a un 49% de las funciones y características especificadas originalmente.

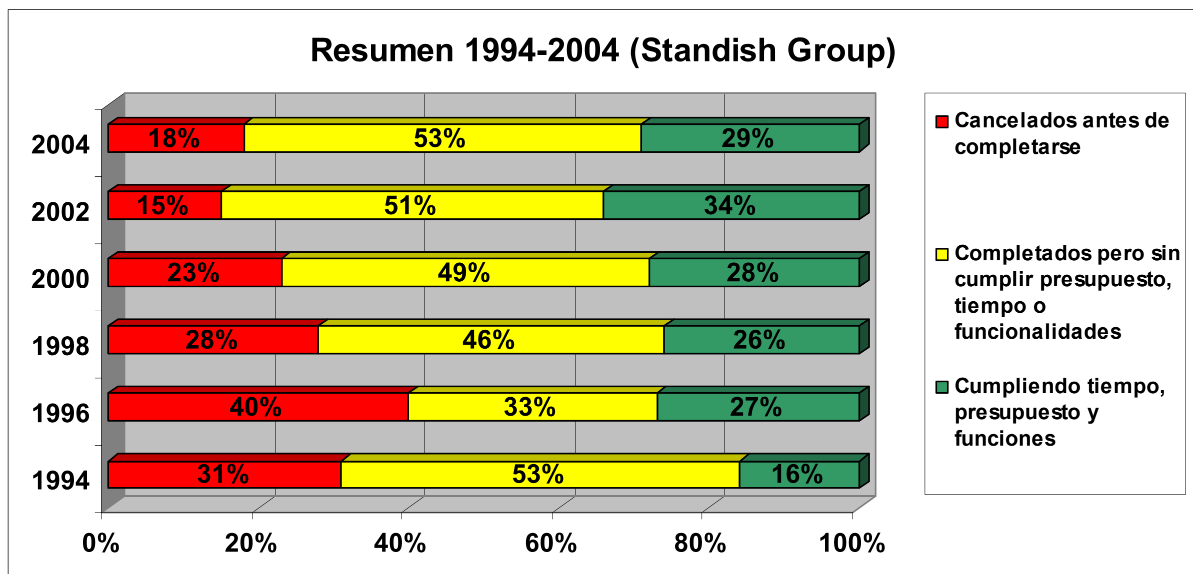


Ilustración 2. Resumen estadísticas 1994-2004 de Standish Group.

0.1.3. Solución aportada por los métodos ágiles

Veamos las diferencias entre el enfoque metodológico, las características del desarrollo de software en la práctica y la filosofía de los métodos ágiles:

	Enfoque metodológico	Desarrollo de Software real	Visión de los métodos ágiles
Actividades	Tareas independientes	Proyectos personales interrelacionados	Proyectos personales interrelacionados
	Duración predecible	Finalización impredecible	Finalización de la siguiente release predecible
	Repetible	Dependiente del contexto	A menudo dependiente del contexto
Ejecución del proceso	Fiable	Dependiente de las condiciones del entorno	Ejecución ligada a pequeñas versiones, por tanto, fiable
	Interacciones especificables	Inherentemente interactivo	Iteraciones especificadas fomentando interactividad
	Tareas en secuencia	Muchas tareas están entrelazadas	Las tareas muchas veces no se mandan; dependen del contexto

Esfuerzo de los desarrolladores	Dedicados a proyectos de software	Comunes a todas las actividades (proyecto, no proyecto, personal, rutinas...)	Los desarrolladores estiman el esfuerzo requerido
	No diferenciados	Específico a individuos	Específico a individuos
	Totalmente disponibles	Totalmente utilizados	Totalmente utilizados
Control de trabajo	Regularidad	Oportunismo, improvisación e interrupción	Sólo existe control mutuo acordado respecto al trabajo de otros (Crystal o DSDM definen controles)
	Milestones, planning, gestión de control	Preferencia individual y negociación mutua	Preferencia individual y negociación mutua

Tabla 1. Diferencias de enfoques: metodológico, el requerido en la práctica y el proporcionado por los métodos ágiles.

En resumen, las principales características a las que deben dar forma los métodos ágiles son:

- **Incremental**: versiones pequeñas de software, con ciclos rápidos.
- **Cooperativo**: desarrolladores y cliente siempre en contacto constante.
- **Directo**: el método en si es fácil de aprender y modificar, bien documentado.
- **Adaptativo**: capaz de tolerar los cambios propuestos por el cliente.

Métodos Tradicionales	**Métodos Ágiles**
Proceso controlado, gestionado	*Aleatorio, oportunista, guiado por sucesos (accident-driven)*
Proceso secuencial, lineal	*Procesos simultáneos, solapados*
Proceso replicable	*Ocurre de forma completamente única*
Proceso racional, determinado, orientado a proceso (goal-driven)	*Negociado, comprometido y caprichoso*
Procesos y herramientas	*Individuos e interacciones*
Documentación comprensible	*Software que funciona*
Negociación de contratos	*Colaboración con el cliente*
Seguir un plan	*Responder al cambio*
Se basan en normas	*Se basan en heurísticas para crear código*
Mayor o menor resistencia a los cambios	*Aceptan e incluso fomentan el cambio*
El cliente no forma parte del equipo, sólo mantiene reuniones con éste	*El cliente forma parte del equipo*
Equipos grandes (>15-20 miembros)	*Equipos de trabajo pequeños o medianos*
Procesos con muchas normas	*Procesos con pocas reglas*
Mucha documentación	*Poca documentación*
Mucho análisis y diseño	*Poco análisis y diseño*
Conceden mucha importancia a la arquitectura de los sistemas	*Conceden poca importancia a la arquitectura de los sistemas*

Tabla 2. Resumen de diferencias entre los métodos tradicionales y los ágiles.

Un acercamiento al planteamiento utilizado por los métodos ágiles: el desarrollo iterativo, donde el proceso entero se va haciendo por capas, sin esperar, por ejemplo, a que esté hecho todo el diseño completo para ejecutar los tests o mostrarlo al cliente para recibir su *feedback*.

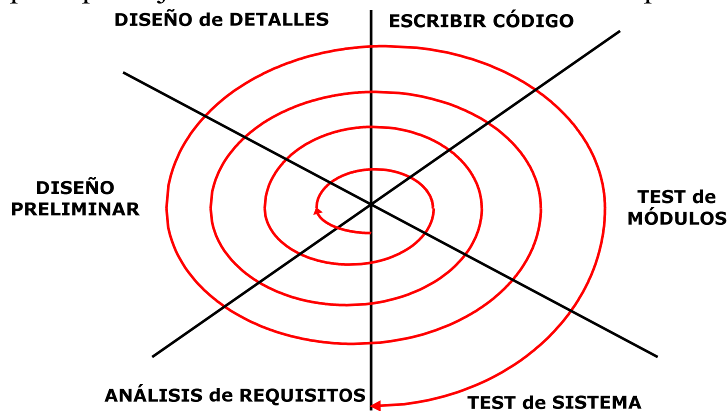


Ilustración 3. Esquema del modelo iterativo.

Otro enfoque similar al anterior es el propuesto por Steve McConnell, la entrega incremental:

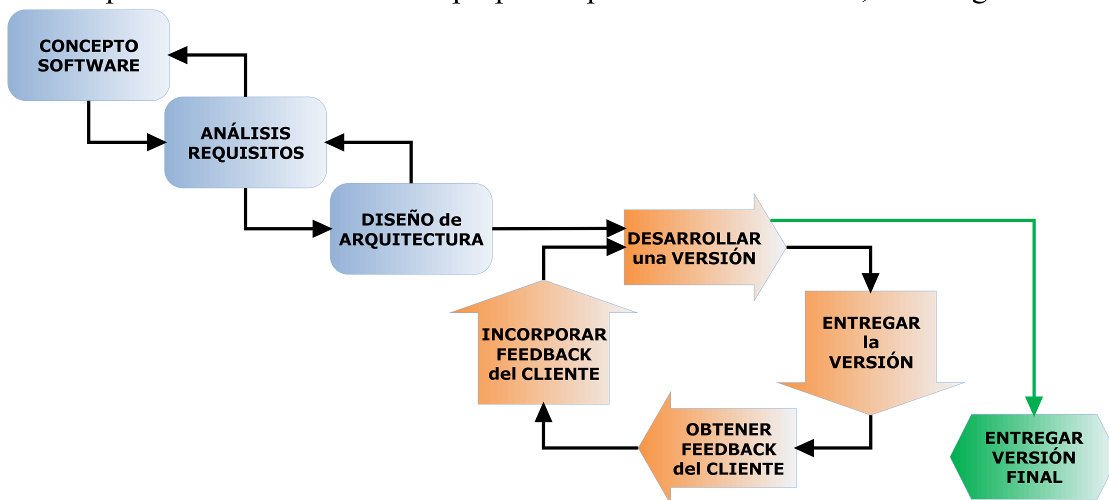


Ilustración 4. Entrega incremental de Steve McConnel (Microsoft, 1996).

La siguiente ilustración sintetiza el cambio de filosofía: en el método en cascada las fases son consecutivas; el modelo iterativo (mismas fases) añade varias entregas, y XP, todavía genera más versiones y hace las fases simultáneamente (nótese que las fases están en columna y no en fila), de ahí el énfasis de estos métodos en automatizar todas las tareas posibles.

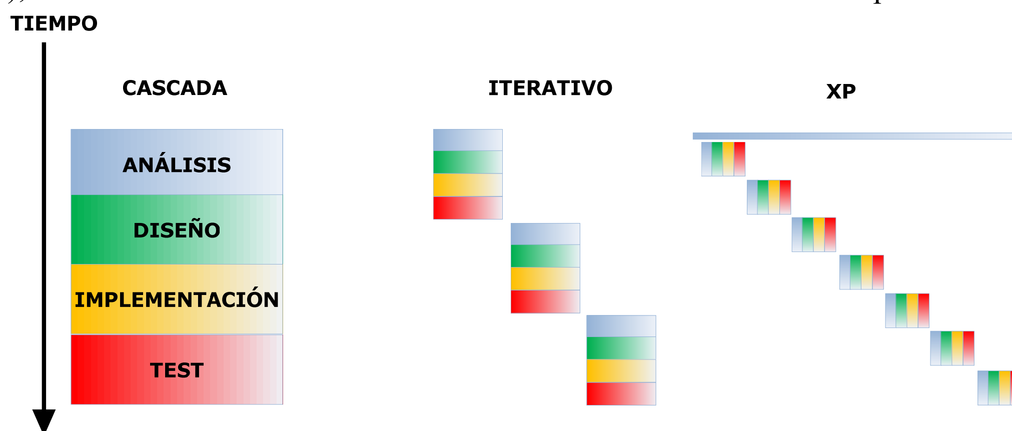


Ilustración 5. Comparación del orden de las fases entre métodos de cascada, iterativo y XP Programming.

0.1.4. El manifiesto ágil y los 12 principios ágiles

Estamos descubriendo mejores maneras de desarrollar software,
tanto por nuestra propia experiencia como ayudando a terceros.
A través de esta experiencia hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas.
Software que funciona sobre documentación exhaustiva.
Colaboración con el cliente sobre negociación de contratos.
Responder ante el cambio sobre seguimiento de un plan.

Es decir, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Nosotros seguimos estos principios:

1) Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.

2) Aceptamos requisitos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

3) Entregamos software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.

4) Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.

5) Construimos proyectos con profesionales motivados. Les damos el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.

6) El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.

7) Software que funciona es la principal medida de progreso.

8) Los procesos ágiles promueven el desarrollo sostenible. Patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

9) La atención continua a la excelencia técnica y los buenos diseños mejoran la agilidad.

10) La simplicidad, el arte de maximizar la cantidad de trabajo no realizado, es esencial.

11) Las mejores arquitecturas, requisitos y diseños surgen de equipos que se auto organizan.

12) A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo; entonces mejora y ajusta su comportamiento de acuerdo con sus conclusiones.

0.1.5. Uso de los métodos (o prácticas) ágiles en la actualidad

Es indudable que estos métodos se están utilizando, en su totalidad o en parte, por la mayoría de las empresas. Una encuesta realizada sobre 4.232 profesionales de las tecnologías de la información en marzo de 2006, llevada a cabo por Scott Ambler (Ambysoft), muestra unas impresiones positivas al utilizar estos métodos. Ambler es creador de los métodos *Enterprise Unified Process* EUP y *Agile Unified Process* AUP.

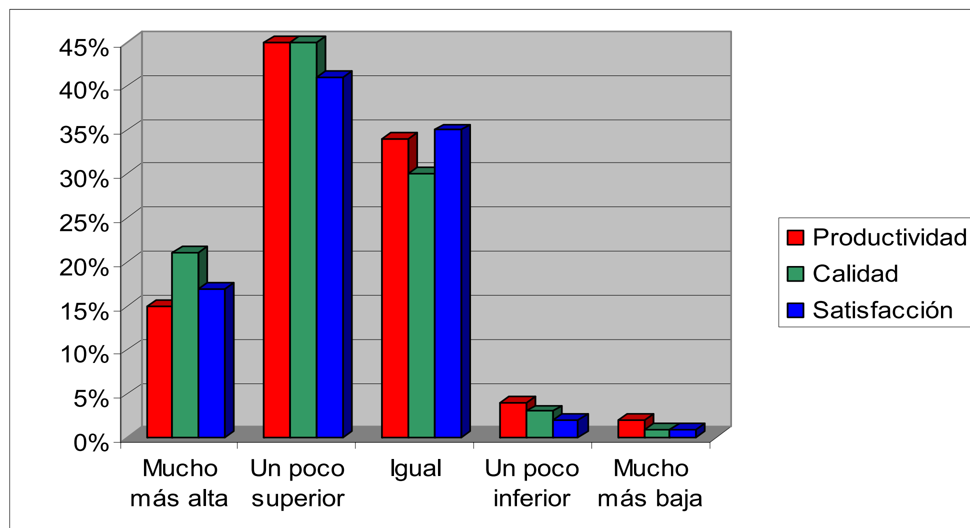


Ilustración 6. Opinión sobre el cambio en productividad, calidad y satisfacción al aplicar métodos o algunas de las prácticas ágiles.

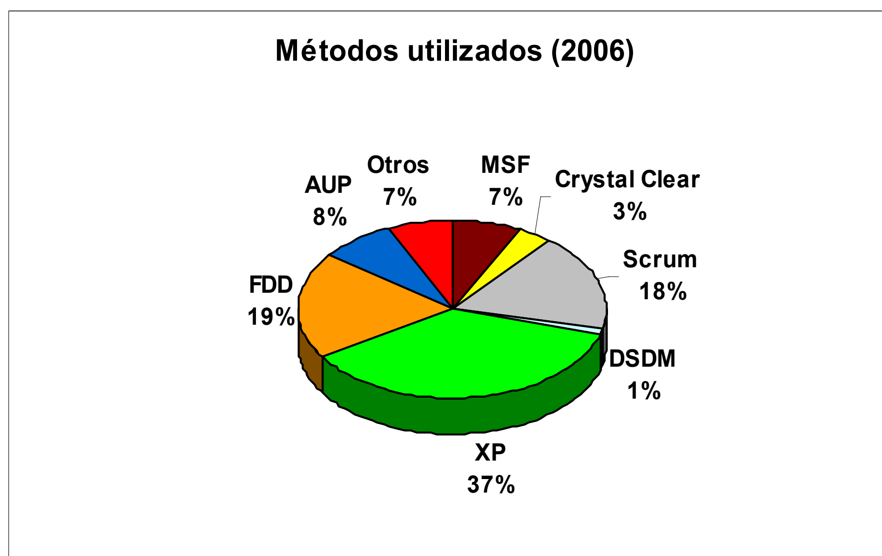


Ilustración 7. Distribución de los métodos ágiles utilizados en la encuesta de Ambysoft.

La aparición de *Extreme Programming* por parte de Kent Beck en 1999, se considera el punto de partida de los métodos ágiles. Los métodos ágiles, utilizan prácticas adaptativas (no basadas en predicciones), iterativas, centradas en la gente (cliente y programadores),

orientadas a entregas incrementales, con mucha comunicación y necesitan que el cliente esté muy involucrado en el proyecto para recibir su *feedback*. El *feedback* continuo es indispensable para evitar que el cliente, con el software acabado, diga **"*es lo que pedí, pero no es lo que necesitaba*"**, algo habitual cuando se utiliza el método en cascada.

Tras *Extreme Programming,* muchos otros métodos y prácticas se hicieron populares. Algunas prácticas, siempre se habían utilizado en determinados ámbitos, pero otras eran totalmente nuevas, demasiado para usarse en masa por empresarios reacios a tantos cambios. A continuación vemos, respecto a la encuesta de 4.232 profesionales, las prácticas utilizadas:

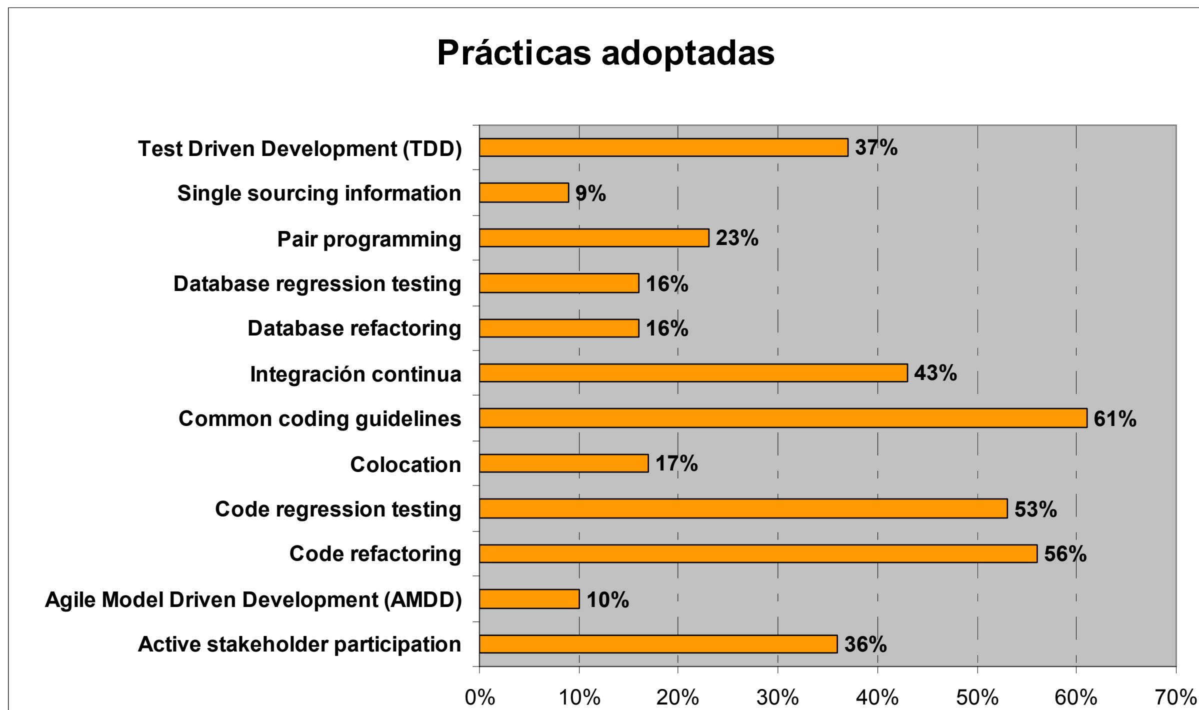


Ilustración 8. Prácticas ágiles adoptadas (Ambysoft 2006).

El siguiente diagrama muestra el resumen de la experiencia ante la incorporación de las prácticas ágiles según los usarios.

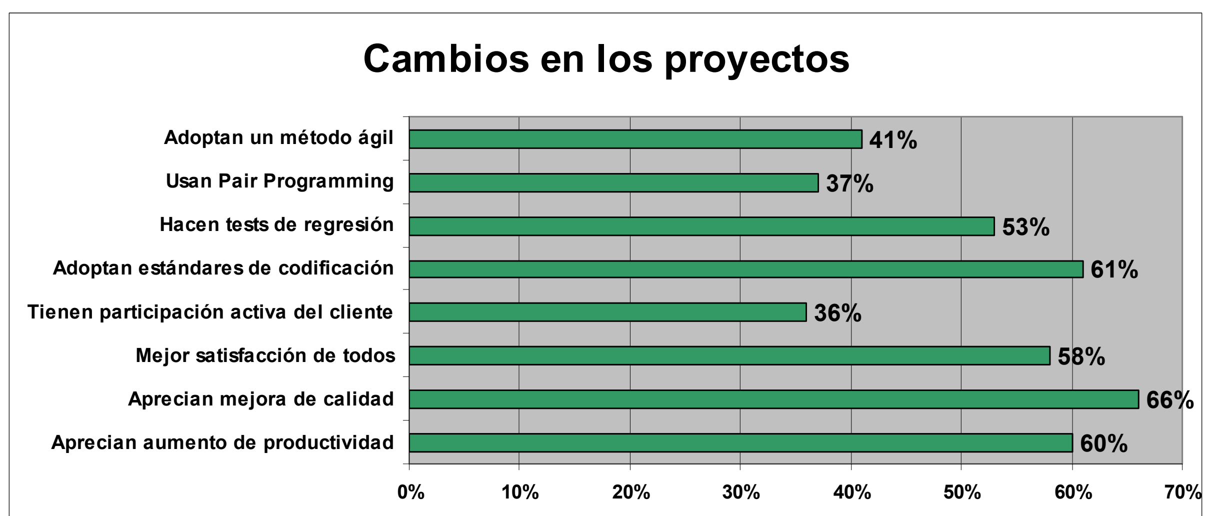


Ilustración 9. Mejoras al aplicar técnicas ágiles, según encuesta de Ambysoft (2006).

0.2. Objetivos

El objetivo principal es dar una visión compacta y objetiva de los métodos ágiles más aceptados, describiendo el proceso o fases, la manera de ponerlo en práctica, los papeles, *workproducts* que generan y sus peculiaridades. La intención, como lo fue la creación de los métodos ágiles, es que se vean de forma clara y rápida los pasos, prácticas o papeles que usa cada método con el fin de evaluar si serían beneficiosos al aplicarlos a un proyecto, nuevo o ya en marcha.

Los métodos que se analizarán son:

AÑO	MÉTODO	AUTORES
1999	Extreme Programming	Beck
1995, 2002	Scrum	Schwaber; Beedle
2002	Crystal	Cockburn
2002	Feature Driven Development	Palmer & Felsing
1996, 2000	Rational Unified Process	Kruchten
2004	Enterprise Unified Process	Ambler
2005	Agile Unified Process	Ambler
1997	Dynamic Systems Development Method	Stapleton
2000	Adaptive Software Development	Highsmith
-	Open Source Software development	-
2003	Lean Software Development	Charette; Poppendieck
2002	Agile Modeling	Ambler
Desde 1976	Evolutionary Project Management	Gilb
finales de los 90	Internet-Speed Development	-
finales de los 90	Microsoft Solutions Framework	Microsoft
2000	Pragmatic Programming	Hunt & Thomas

Tabla 3. Esquema de los métodos que se detallarán, con sus autores y año de publicación.

Se empezará por abordar las debilidades de los métodos clásicos y citar las bases del movimiento ágil: el manifiesto ágil, los 12 principios ágiles y el manifiesto de proyecto ágil (este último el menos conocido quizás por ser más reciente).

La bibliografía primaria utilizada, han sido los libros o publicaciones oficiales de los creadores de los métodos. Las pequeñas y constantes mejoras de estos métodos, suelen publicarse en sus páginas web oficiales.

Como todas las filosofías ágiles comparten el mismo sustrato, el manifiesto ágil y los 12 principios ágiles, se detallará más a fondo el más popular (y que ha sido inspiración de otros), *Extreme Programming*. También la puesta en práctica de un método en concreto bastante aceptado, *Lean Software Development*, por su validez y experiencia demostrada en entornos de producción, se ejemplificará a través de 7 principios y 22 herramientas concretas.

No es habitual que las empresas publiquen información detallada sobre sus proyectos: tiempo de desarrollo, personal involucrado, costos económicos, errores cometidos y sus consecuencias, etc. Esto hace difícil poder contrastar las mejoras teóricas de los métodos ágiles respecto a sus predecesores. Sin embargo, sí que se puede conocer mediante noticias qué métodos han usado o algún caso concreto (SirsiDynix y StarSoft Development Laboratories) que sí que se comenta.

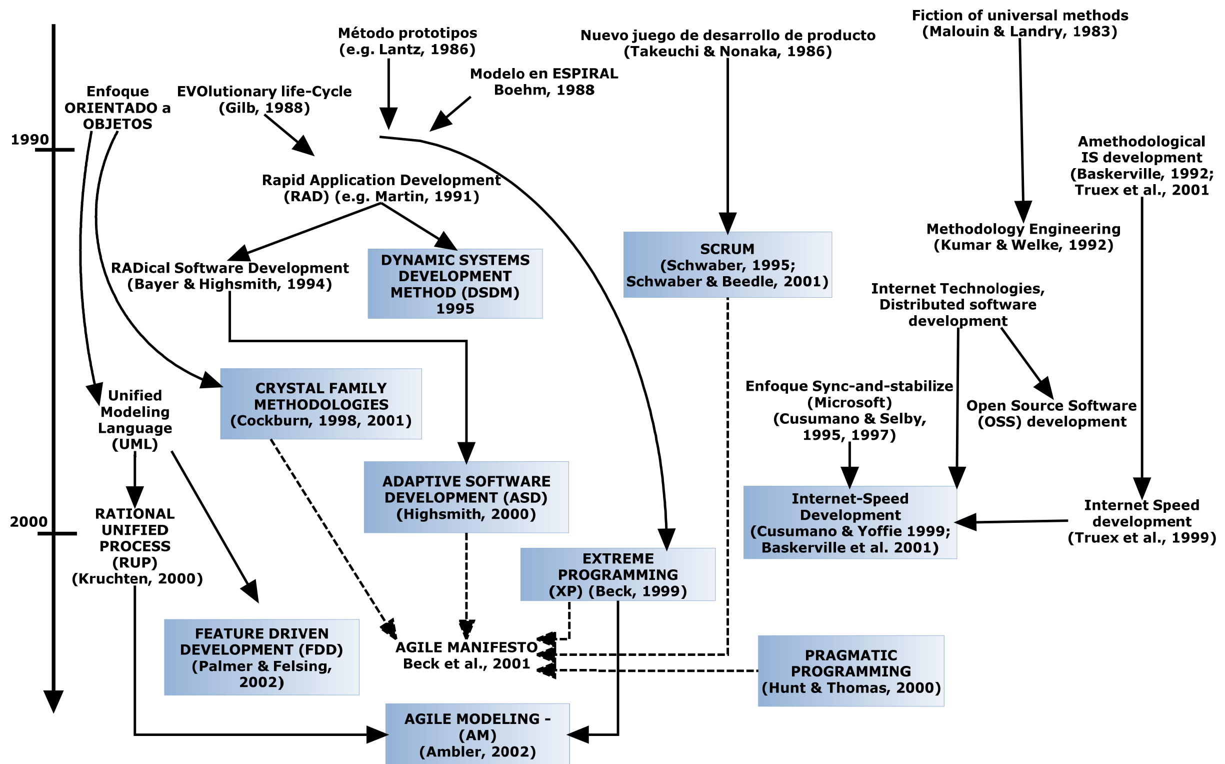


Ilustración 10. Visión cronológica general de los métodos ágiles más destacables y sus influencias.

Hay que tener en consideración la muy diferente naturaleza de los métodos, aunque todos se engloben dentro de los métodos ágiles. Mientras que Scrum se decanta por la gestión, otros como XP especifican las prácticas a seguir en el equipo, *Pragmatic Programming* da pautas para desarrollar un buen código, algunos como FDD no abarcan la totalidad del proceso de desarrollo, *AgileUP* surge como una versión ágil de RUP y propone EUP que es una ampliación para abarcar todo el ciclo de vida del software, etc.

La visión general facilitará el entendimiento e incluso el poder complementar varios métodos. Por ejemplo, se podría proponer MSF como marco general, Planguage como lenguaje de especificación de requisitos, Scrum (con sus patrones organizacionales) como método de gestión, XP (con patrones de diseño, programación guiada por pruebas y refactorización) como metodología de desarrollo, RUP como abastecedor de artefactos, ASD como cultura empresarial y quizá hasta CMM como método de evaluación de madurez.

0.2.1. Esquemas de los métodos más utilizados como base

La visión general de cómo propone realizar un proyecto XP es la siguiente:

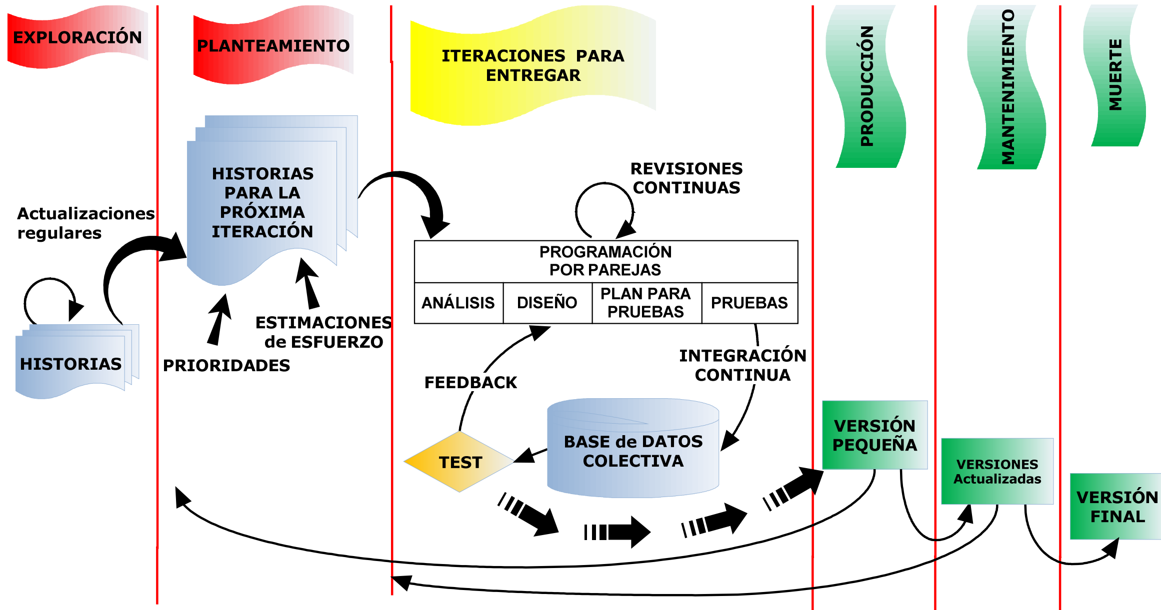


Ilustración 11. Fases y pasos en un proyecto XP.

Dentro de los métodos de gestión, Scrum se ha aceptado de forma representativa:

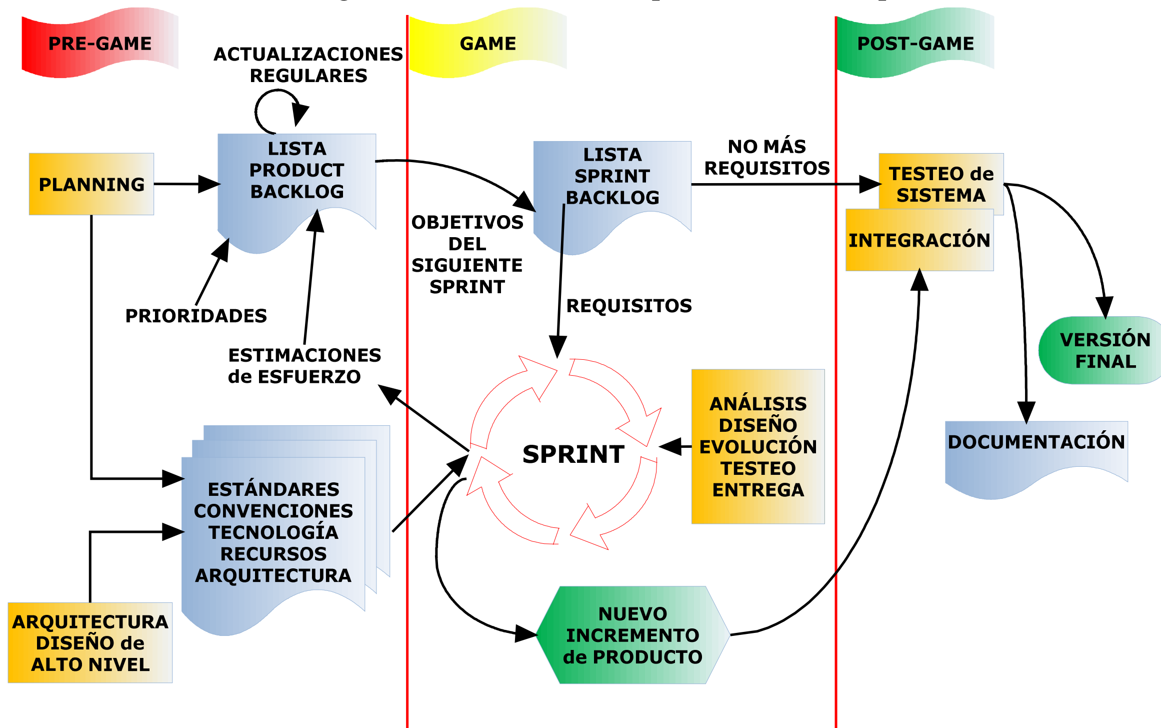


Ilustración 12. Fases y workproducts del método de gestión Scrum, punto de referencia para muchos otros.

Las prácticas de un sprint en Scrum comprenden los siguientes pasos:

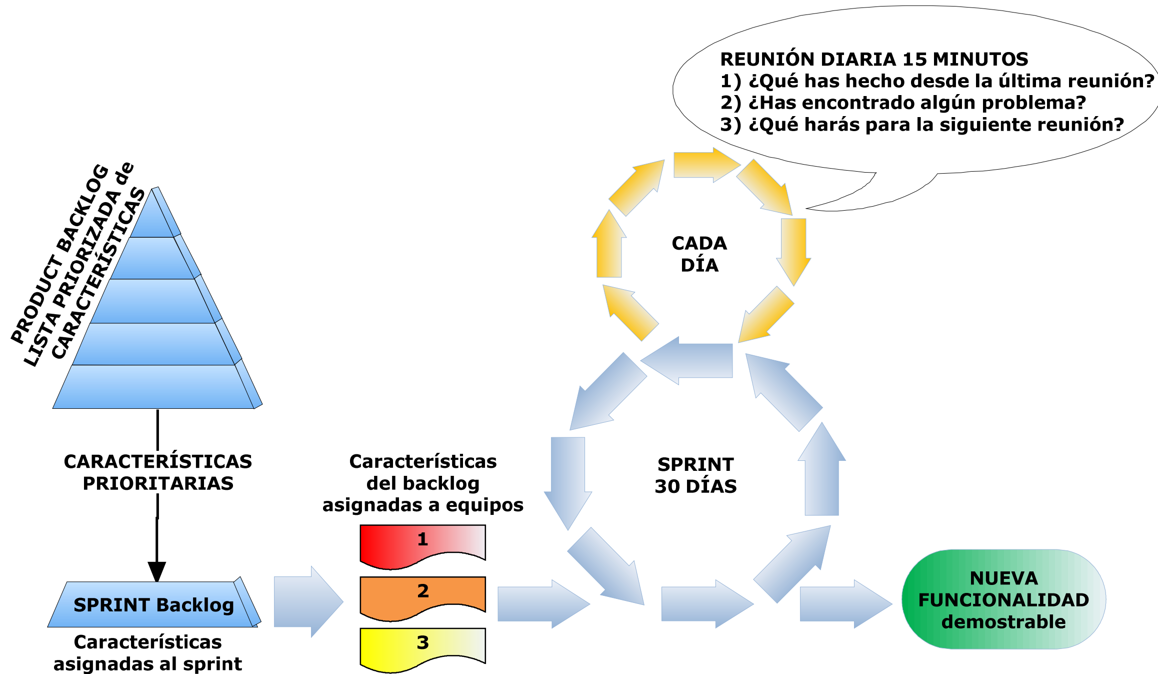


Ilustración 13. Prácticas de un sprint en Scrum.

Otra intención era sintetizar todo el contenido en transparencias.

Por tanto, no es el propósito:

- Profundizar al máximo un método en particular.
- Desarrollar métodos de gestión como PMBOK o Prince2.
- Explorar todas las capacidades de las herramientas software.
- Comparar de forma exhaustiva los papeles, fases, entregas, etc. de todos los métodos.

0.3. Alcance

Existe una variada bibliografía sobre la mayoría de los métodos, además del soporte en sus respectivas páginas web. Es habitual encontrar un libro de cada uno de ellos (o varios si ha habido versiones posteriores) hecho por el propio autor del método, así como también numerosas publicaciones críticas, a veces negativas o que minimizan los beneficios obtenidos directamente gracias a aplicar estos métodos.

En esta síntesis se ha pretendido esquematizar a alto nivel los diferentes pasos, papeles, prácticas y workproducts de cada uno. La explicación inicial de los métodos está basada en la de sus respectivos creadores, siendo por tanto, su visión quizás imparcial.

Muchos métodos como XP utilizan prácticas que son específicas de la programación como por ejemplo la refactorización. Estas prácticas, que podríamos considerar de bajo nivel corresponden estrictamente al ámbito de la programación, y por tanto no se tratarán en su máximo detalle, aunque se mostrarán herramientas software para tales fines.

También se han consultado obras que analizan en general los métodos ágiles, ya que comparten la mayoría de las ideas en las que están basados (para cumplir con el manifiesto ágil) aunque después cada uno las lleve a cabo de forma diferente.

Dentro del capítulo de conclusiones, se han tenido en cuenta las propias ideas, resultado de analizar y comparar el resumen de todos los métodos, así como también ideas de otros autores. Resulta de utilidad la opinión de ajenos a los métodos, las conclusiones basadas en experiencias reales al haberlos adoptado, etc. Se citan varias opiniones de profesionales, con las que se puede coincidir o no, pero que merecen un espacio ya que es el lado real.

0.3.1. Métodos

De cada método se detallará:

- Breve historia, entorno para el que se creó, su evolución y apuntes sobre la biografía del autor(es) del método que hayan influido en él.
- El proceso, formado por las fases de las que se compone, y el objetivo o *milestone* de cada fase, antes de pasar a la siguiente.
- Las prácticas, guías o disciplinas a seguir para hacer un buen uso del método.
- Los papeles y las responsabilidades de los participantes del proyecto.
- Consejos sobre la adopción y experiencias.
- Limitaciones del método en lo referente a número de participantes, si es adecuado para sistemas críticos, requisitos especiales, etc.

Se hará un breve repaso a los métodos predecesores a los métodos ágiles:

- Cascada, con sus modificaciones más importantes: reducción de riesgos, sahismi o fases superpuestas, cascada con subproyectos y entrega por etapas.
- Code-and-Fix
- Espiral
- Modelo en V
- Entrega incremental

Los problemas que comparten los “métodos pesados” dieron lugar a que por 17 programadores-consultores de éxito firmaran el manifiesto ágil (2001) junto con los 12 principios que forman la base o entorno de todos los métodos ágiles. Sin embargo, no son ningún método en particular sino pautas y preferencias, lo cual ha permitido que surgieran varios métodos compartiendo la filosofía ágil.

Aunque CMM no está sujeto a un modelo particular, ni constituye un modelo de proceso de ingeniería de software en el sentido estricto de la palabra sino más bien un canon de evaluación que establece criterios para calificar la madurez de un proyecto en cinco niveles que van de lo caótico a la optimización continua, es considerado el enemigo a vencer por los métodos ágiles. Las razones: su pesadez, formalismo, excesiva documentación y rigidez... características que si bien pueden funcionar (y han funcionado) en muchos proyectos, pueden resultar lentas y caras para el desarrollo de software.

Una primera voz de alarma la dio el matemático Barry Boehm que en 1987, tras estudiar los datos de 67 proyectos de software, concluyó que encontrar y solucionar un problema de software después de que haya sido entregado al cliente cuesta cien veces más que encontrar y arreglar el problema en las etapas iniciales de diseño. En 2001, Boehm postuló que, para sistemas pequeños, el factor de proporcionalidad se halla más próximo a cinco que a cien.

El segundo detonante fue el estudio The Chaos Report (1994) de Standish Group que arrojó unos datos preocupantes como se ha visto en la introducción.

Si bien los métodos ágiles son maneras de trabajar, en el capítulo 3, veremos herramientas software que ayudan a gestionar por ejemplo, las tareas, las tarjetas o fichas CRC Clase-Responsabilidad-Colaborador de XP, informes, gráficos de quemado para ver la velocidad real del progreso y poder estimar calendarios, etc. Estos paquetes son de fácil uso y permiten su uso a través de entorno web, incluso desde Internet, algo indispensable para el caso de *Open Source Software Development*.

0.3.2. Software

En primer lugar, repasaremos algunas de las herramientas más comunes para automatizar tareas, ya que como se verá, todos los métodos integran y hacen pasar tests muchas veces antes de entregar la versión final al cliente. En concreto, se verán herramientas para pruebas automáticas, integración continua, trabajo colaborativo, refactorización y estándares de codificación.

Respecto al software de más alto nivel, que permita enviar notificaciones por e-mail de forma automática, compilar automáticamente al detectar cambios en el repositorio donde contribuyen los desarrolladores distribuidos por todo el mundo, etc., destacan:

- XPlanner
- Evo Task Administrator
- Rally
- VersionOne Agile Team y Agile Enterprise
- TargetProcess
- ExtremePlanner
- Atlassian
- xProcess
- Microsoft Visual Studio

Se detallarán brevemente sus características principales y se han acompañado de diferentes capturas de pantalla. Todos estos paquetes ofrecen infinidad de opciones, especialmente los *framework* como *Microsoft Visual Studio* o *xUnit*, integraciones con otros programas para incorporar directamente los resultados de éstos (informes, errores...). La descripción detallada de alguno de estos programas supondría una extensión mayor a la de este proyecto y por tanto, no ha sido el objetivo examinar exhaustivamente estas herramientas sino los métodos que subyacen en ellas.

En el capítulo 4, se resumen en una tabla las características, fases y papeles de los métodos analizados. Seguidamente se hace una crítica de la filosofía ágil así como de sus principales prácticas. Ciertos aspectos como el proyecto C3, el lenguaje Smalltalk o artículos publicados en periódicos influyentes tuvieron su particular importancia para dar envergadura a los métodos ágiles, que a día de hoy, devuelven más de 2 millones de entradas en Google.

Se dedica un apartado a cubrir los informes de las estadísticas de Standish Group 1994-2004 y una de Scott Ambler (Agile Modeling) de 2006.

Por último, se comenta la complementariedad de ciertos métodos (XP y Scrum es una combinación muy aceptada) así como las similitudes o algunas diferencias de forma compacta. El propósito no es comparar de forma exhaustiva las responsabilidades de cada papel o las fases entre los diferentes métodos.

En los anexos, se encuentran las versiones originales en inglés de los documentos más destacados como el manifiesto ágil. También se dispone de un glosario, una breve introducción a CMM, CASE, UML, unas pinceladas de humor de Dilbert que dan una idea del mal enfoque o burla que se le puede dar a los métodos ágiles, etc.

CAPÍTULO

1

CONTENIDO

1.1. Marco histórico

1.2. La necesidad de nuevos métodos

1.3. El Manifiesto Ágil

1.4. Los 12 principios ágiles

1.5. Características comunes de los métodos ágiles

1.6. Manifiesto de proyecto ágil

INTRODUCCIÓN

Durante los últimos 30 años, han aparecido numerosos métodos para el desarrollo de software. La aparición de Extreme Programming por parte de Kent Beck en 1999, se considera el punto inicial de los métodos ágiles.

Diferenciar un método ágil, MA en adelante, de otro no es una ciencia exacta, pues los límites no están exactamente definidos. También se sabe que ciertos métodos no son adecuados para algunos programadores, incluso se podría considerar que cada programador tiene su propio método personal.

Los métodos ágiles, utilizan prácticas adaptativas, no basadas en predicciones, iterativas, centradas en la gente (cliente) y los equipos (programadores), orientadas a entregas incrementales, con mucha comunicación y necesitan que el cliente esté muy involucrado en el proyecto para recibir su *feedback*.

"La diferencia entre un atracador de bancos y un teórico del método CMM es que con un atracador se puede negociar"

Ken Orr, Cutter Consortium

1. INTRODUCCIÓN

1.1. Marco histórico

A finales de la década de los 90, empezó una batalla respecto a la ingeniería de software y los métodos de desarrollo: el diseño basado en patrones contra los métodos ágiles, con *Extreme Programming* a la cabeza. Los MA se enfrentaron a metodologías muy consagradas en una lucha que a veces se ha llamado *"la batalla de los gurúes"*, *"la venganza de los programadores"* o *"la muerte del diseño"*. Estructuralmente, los MA se asemejan a RAD (Desarrollo Rápido de Aplicaciones) y a otros métodos iterativos, pero dan prioridad a otros aspectos.

La historia, hizo salir a la luz grandes fracasos de la OTAN, el Departamento de Defensa, DoD, o la Food & Drug Administration, FDA, de los métodos tradicionales, ya que en muchos casos no compensaba su coste en tiempo, complejidad y dinero. Obviamente, también existen muchos casos de éxito de estas metodologías.

Los organismos y empresas han desarrollado métodos e ingeniería de software: CMM, Spice, Bootstrap, TickIt, derivaciones de la ISO9000, SDCE, Trillium, etc. Algunos son métodos, otros metodologías de verificación o estimación de conformidad. El libro "*The mythical Man-Month*" de Fred Brooks (1975) ya se avanzó en el tiempo para revelar los problemas que tendrían los métodos clásicos. Es popular su frase:

"Haga un plan para tirarlo. Lo hará de todos modos". Fred Brooks (1975)

Años más tarde, lo reiteró: "*No construya uno para tirar. El modelo en cascada es erróneo*".

Los modelos de los métodos clásicos difieren bastante en su naturaleza, pero casi siempre exaltan las virtudes de planear y poseen un espíritu normativo. Comienzan con el análisis completo de los requisitos del usuario. Después de un largo período de intensa interacción con usuarios y clientes, los ingenieros establecen un conjunto definitivo y exhaustivo de características, requisitos funcionales y no funcionales. Esta información se documenta en forma de especificaciones para la segunda etapa, el diseño, en el que los arquitectos, junto a otros expertos en temas puntuales (como bases de datos), generan la arquitectura del sistema. Posteriormente, los programadores implementan ese diseño bien documentado y finalmente, el sistema completo se prueba y se entrega.

A continuación se muestra una tabla con las grandes formas de proceso metodológico, y no técnicas específicas que pueden aplicarse a ellas, como las técnicas de ingeniería de software asistidas por computadoras (CASE, ver anexos), las herramientas de cuarta generación, las técnicas de modelado estático (Lai) o dinámico (Forrester), el modelo de software de sala limpia o los modelos evaluativos como el *Capability Maturity Model*[1] (CMM) del SEI, que popularmente se considera ligado a los modelos en cascada.

De hecho, CMM no está sujeto a un modelo particular, ni constituye un modelo de proceso de ingeniería de software en el sentido estricto de la palabra; es un canon de evaluación que establece criterios para calificar la madurez de un proyecto en cinco niveles que van de lo

[1] www.sei.cmu.edu/cmm - ver anexos para más información.

caótico a la optimización continua. Una buena referencia para los modelos clásicos anteriores a 1990 es *Wicked problems* de Peter DeGrace y Leslie Stahl.

Para CMM, el desarrollo de software es un proceso definido que puede ser especificado en detalle. En algunos métodos clásicos puede haber bucles e iteraciones, pero la mentalidad innata de su planificación es totalmente lineal. Los autores del manifiesto ágil reconocían tener un enemigo común: CMM, aunque no es estrictamente un método, ni un método en cascada en particular, se identificaba como el modelo no-iterativo más importante.

"La diferencia entre un asaltante de bancos y un teórico del CMM es que con un asaltante se puede negociar" – Ken Orr, Cutter Consortium[2]

MODELO	VERSIÓN ORIGINAL	CARACTERÍSTICAS
Cascada	Secuencial: Bennington 1956 Iterativo: Royce 1970 -Estándar DoD2167-A	Lista de requisitos, diseño del sistema, diseño de programa, codificación, pruebas, operación y mantenimiento.
Cascada con fases superpuestas	McConnell 1996: 143-144	Cascada con eventuales desarrollos en paralelo (Modelo Sashimi).
Iterado con prototipos	Fred Brooks 1975	Iterativo – Desarrollo incremental.
Desarrollo rápido (RAD)	J. Martin 1991 – Kerr/Hunter 1994 – McConnell 1996	Modelo lineal secuencial con ciclos de desarrollo breves.
V	Ministerio de Defensa Alemán 1992	Combinación de cascada con iteraciones.
Espiral	Barry Boehm 1988	Iterativo – Desarrollo incremental. Cada fase no es lineal, pero el conjunto sí.
Espiral win-win	Barry Boehm 1998	Iterativo – Desarrollo incremental Aplica teoría-W (espiral) a cada etapa.
Modelo de desarrollo concurrente	Davis y Sitaram 1994	Modelo cíclico con análisis de estado.
Entrega incremental (Staged delivery)	McConnell 1996	Fases tempranas en cascada. Fases posteriores descompuestas en etapas.

Tabla 4: Historia de los principales modelos precursores de los métodos ágiles.

El mero hecho de que los MA se expresen a través de un Manifiesto resultante de una tertulia y no mediante un estándar elaborado durante años en un organismo, da una idea de la diferencia de actitud. Además, mientras la terminología de CMM, EUP[3] y otros métodos ortodoxos es detallada y formal, las nomenclaturas de muchos de los MA están plagadas de palabras procedentes de campos variopintos: *Scrum*, gallinas, cerdos, *sprint*, pre-juego, juego, post-juego, montaje u organización (*staging*) o púas (*spikes*).

Junto a esas palabras informales, también hay expresiones y máximas como *"El Minimalismo es Esencial"*, *"Todo se puede cambiar"*, *"El 80% hoy en vez del 100% mañana"* en Lean, *"Mirando basura o desperdicios"* en LSD, el *"Cono del Silencio"* o el *"Esqueleto Ambulante"* en Crystal Clear, *"Ligero de equipaje"* en AM, *"Documentos para ti y para mí"* en ASD, *"Gran Jefe"* o *"YAGNI"* (acrónimo de "No vas a necesitarlo") en XP.

[2] www.cutter.com
[3] www.enterpriseunifiedprocess.info

MODELO EN CASCADA

El modelo más criticado por los MA es el modelo en cascada, por su rigidez, obstinación por exigir una estipulación previa y completa de los requisitos, su falta de adaptación a los cambios, distanciamiento de la visión del cliente y retrasos para entregar código ejecutable. El nombre de cascada resulta apropiado: el agua que cae de una cascada no puede (por ella misma) retroceder hacia arriba para volver a caer. Las primeras ideas del modelo iterativo aparecieron en los años 60 de la mano de Tom Gilb y en 1975 por Vic Basili y Joe Turner.

Al final de cada fase, se hace una revisión para aceptar pasar a la siguiente fase o no. El modelo en cascada es *document-driven*, es decir que la parte más importante de los *workproducts* que pasan de una fase a la siguiente son documentos.

En el modelo en cascada puro, las fases son discontinuas, sin solaparse. Es un método adecuado cuando los requisitos están muy bien fijados y no cambian, aunque no se produce código ejecutable hasta el último momento. Funciona bien cuando predominan los requisitos de calidad en vez de los costes y un calendario preestablecido. El hecho de no permitir cambios en las fases intermedias una vez pasadas, elimina una gran fuente de errores potenciales. De hecho, es posible retroceder en el modelo en cascada (el llamado modelo "salmón"), aunque es difícil y lento por la completa documentación que se debe rehacer. Otra debilidad es la tardanza en producir alguna entrega tangible para el cliente.

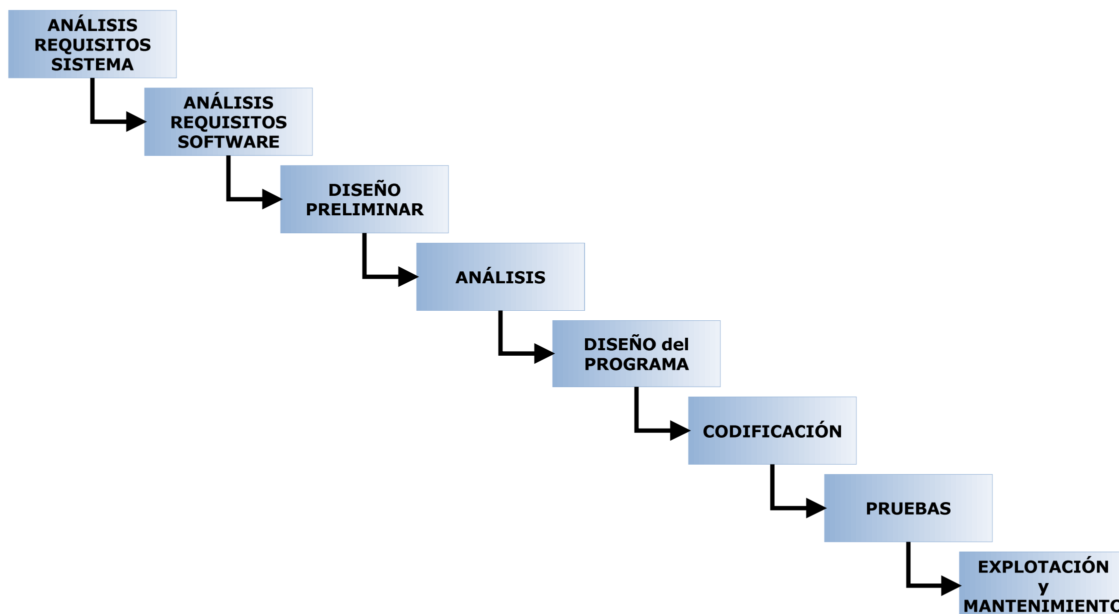


Ilustración 14. Modelo original en cascada de Royce (1970).

CODE-AND-FIX

El modelo "Programar-y-Arreglar" es un modelo raramente útil pero que también se utiliza, aún sin saberlo, ya que es lo más habitual cuando no se hace un plan exhaustivo. Cuando se junta con un calendario ajustado, entonces da lugar al enfoque *code-like-hell*, es decir, programar de forma radical sin control. En el modelo *code-and-fix*, se empieza con una idea general de lo que se necesita, teniendo o no alguna especificación formal, y se va programando, diseñando y testeando todo a la vez, *on the fly* (al vuelo) hasta tener una versión aceptable. Este modelo tiene algunas ventajas: no tiene sobrecarga (no se hacen planes exhaustivos, documentación, cumplimiento de normas, controles de calidad, etc.), enseguida se puede tener un resultado tangible del programa, y no se requiere ningún conocimiento

específico para usar este método. Puede ser útil para hacer pequeñas presentaciones de programas o prototipos para descartar, pero para proyectos de cierta envergadura, es peligroso.

MODELO EN ESPIRAL

El modelo en espiral (también llamado canela en rama, *cinnamon roll*, por su gráfico) es un modelo de ciclo de vida *risk-oriented* que divide un proyecto de software en varios mini-proyectos. Cada uno de éstos, afronta uno o varios riesgos principales, hasta que todos los problemas principales se han tratado. El concepto de riesgo a veces se entiende como requisitos mal especificados o arquitectura errónea, problemas potenciales de rendimiento, de la tecnología a usar, etc. Una vez se han tratado todos los problemas principales, el método en espiral termina como el modelo en cascada.

La idea básica del esquema es que se empieza a pequeña escala, en el centro de la espiral, se examinan los riesgos, se elabora un plan para tratarlos y se mira hacia la siguiente iteración. Cada iteración desplaza el proyecto a una escala mayor. Cada vez que nos alejamos del centro de la espiral, debemos estar seguros de que nos movemos en la dirección deseada. Cada iteración conlleva 6 pasos:

1) Determinar objetivos, alternativas y restricciones.
2) Identificar y resolver riesgos.
3) Evaluar alternativas.
4) Desarrollar las entregas, *deliverables*, para esa iteración y verificar que sean correctas.
5) Planear la siguiente iteración.
6) Planificar la siguiente iteración (si la hubiese).

En el modelo en espiral, las primeras iteraciones son las más "baratas". Se requiere menos esfuerzo para desarrollar el concepto que para desarrollar los requisitos, luego el diseño, la implementación y las pruebas. Es importante avanzar de iteración sólo cuando se han reducido los riesgos. De esta forma, cuando más aumentan los costes (esfuerzo) al pasar a otra iteración, más disminuyen los riesgos. Es decir, **cuanto más dinero y tiempo se emplea, menos riesgo se corre**, es exactamente lo necesario para desarrollar proyectos rápido y bien.

En resumen, las etapas podrían resumirse así:

1) Se realiza el análisis hasta que los desarrolladores piensan que han acabado.
2) Se realiza el diseño hasta que se piensa que se ha concluido con él. En el caso de que se encuentren fallos o problemas en el análisis, se abordan volviendo al análisis (es decir, se realiza otra iteración del análisis).
3) Se pasa el diseño a código. Si aparecen problemas en la traducción, se vuelve al diseño hasta que se solucionan (es decir, se realiza otra iteración del diseño).
4) Se hace que el código pase las pruebas oportunas. Si aparecen errores o fallos, se comienza otra iteración de la escritura de código.
5) Finalmente, se distribuye la aplicación. Los problemas que aparezcan se enviarán a las etapas anteriores.

Las metodologías basadas en el proceso en espiral progresan en el desarrollo de software mediante capas; cada capa o espiral representa una fase en el proceso. No existen fases prefijadas (captura de requisitos, análisis, diseño, etc.): en un proyecto habrá tantas capas como se necesiten. Un prototipo permite a los usuarios determinar si el proyecto va por buen camino, si debería volver a etapas anteriores o si debería concluirse. El proceso en espiral se introdujo para solucionar los problemas del proceso en cascada, y es la variante de éste más usada en la actualidad. La secuencia de pasos es aún lineal, pero admite *feedback*.

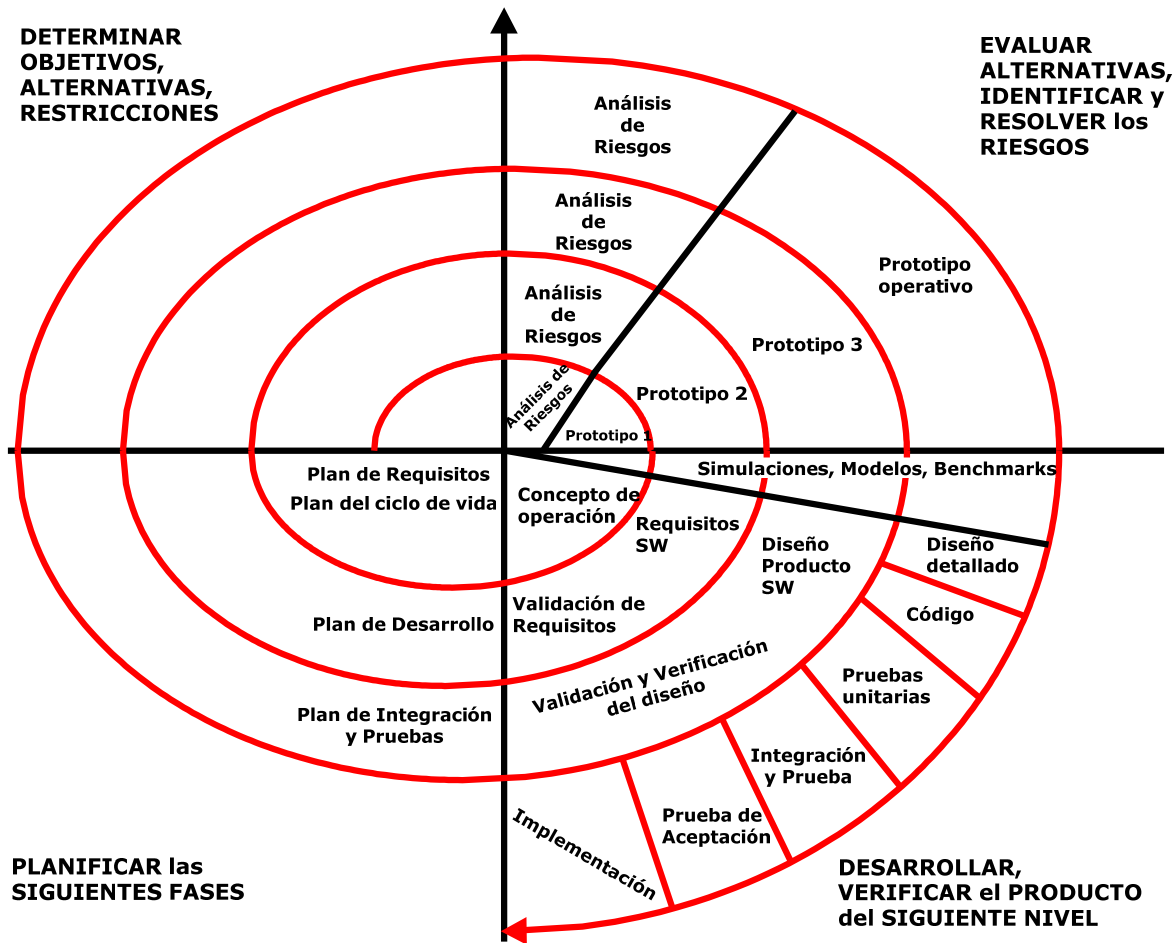


Ilustración 15. Modelo en espiral de Barry Boehm (1988).

El modelo en espiral considera que los pasos hacia atrás (las iteraciones posteriores a la primera) son errores. El modelo de desarrollo iterativo asume, en cambio, que siempre se van a cometer errores y que, por consiguiente, siempre habrá que efectuar varias iteraciones. Un proyecto basado en este último modelo se construye mediante iteraciones. Cada ciclo o iteración concluye con una versión del sistema en desarrollo que cumple un subconjunto de los requisitos exigibles al sistema final. En cada iteración aparecen todas las fases del modelo en cascada; pero cada iteración sólo está orientada a un subsistema, no al sistema completo.

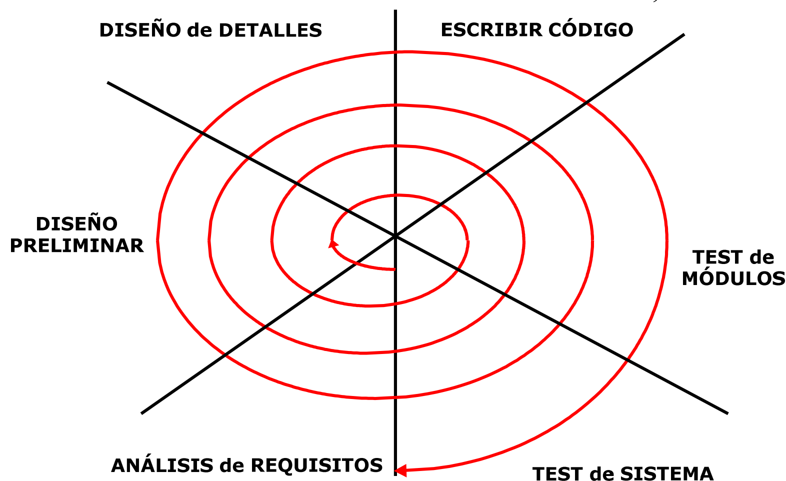


Ilustración 16. Modelo iterativo de desarrollo.

El modelo iterativo, al igual que los anteriores, se basa en una sucesión lineal de pasos o etapas. Frente a los otros, presenta algunas ventajas:

1) El desarrollo se divide en iteraciones; en cada una figura un subconjunto de las funciones exigidas al sistema.
2) Pueden abordarse en las primeras iteraciones aquellas funciones críticas o de alta prioridad para el cliente.
3) Los requisitos de las iteraciones aún no implementadas pueden cambiar durante el proyecto, sin modificar el código generado en las iteraciones terminadas.

MODELO EN V (EJÉRCITO ALEMÁN, 1992)

Puede notarse que su primera mitad es similar al modelo en cascada, y la otra mitad tiene como finalidad integrar y hacer pruebas asociadas a cada una de las etapas de la mitad anterior. Los planes de prueba son el nexo entre el desarrollo y la verificación. Se puede identificar una ventaja principal con respecto al modelo en cascada más simple, y se refiere a que este modelo involucra pruebas de cada una de las etapas del modelo de cascada. Sin embargo, tiene unas ciertas desventajas:

- El riesgo es mayor que el de otros modelos, pues en lugar de hacer pruebas de aceptación al final de cada etapa, las pruebas comienzan a efectuarse después de haber terminado la implementación, lo que puede traer como consecuencia un volver atrás de todo un proceso que costó tiempo y dinero.
- El modelo no contempla la posibilidad de retornar a etapas inmediatamente anteriores, cosa que en la realidad puede ocurrir.
- Se toma toda la complejidad del problema de una vez y no en iteraciones o ciclos de desarrollo, lo que disminuye el riesgo.
- Si los tests no se ejecutan hasta integrar, evidentemente, aparecerán algunos *bugs* que podían haberse detectado antes.
- Los usuarios del modelo en V, a menudo separan el diseño de tests de su implementación. El diseño de tests se hace cuando el documento de desarrollo apropiado está listo.
- Para un proyecto con muchas unidades, implementar los tests de unidad y los de integración por separado, puede ser lento y costoso. Podría ser mejor testear una unidad cuando se la agregara al sistema, usando el sistema para recibir los resultados o mensajes de los tests.

También tiene puntos favorables:

- Define fases tangibles en el proceso y da una secuencia lógica que las relaciona.
- La documentación de tests se escribe tan pronto como se puede, por ejemplo, los tests de integración se escriben cuando el diseño a alto nivel está acabado, y los tests de unidad se escriben cuando las especificaciones detalladas están acabadas.
- Da tanta importancia al desarrollo como a las pruebas.

La función de los tests está muy clara:

- Los tests de unidad comprueban que el código se corresponda con el diseño detallado.
- Los tests de integración comprueban si los componentes previamente testeados encajan.
- Los tests de sistema comprueban si el producto entero integrado cumple las especificaciones.
- Los tests de aceptación comprueban si el producto se corresponde con los requisitos finales del usuario.

A pesar de sus debilidades, se trata de un modelo más robusto y completo que el de cascada.

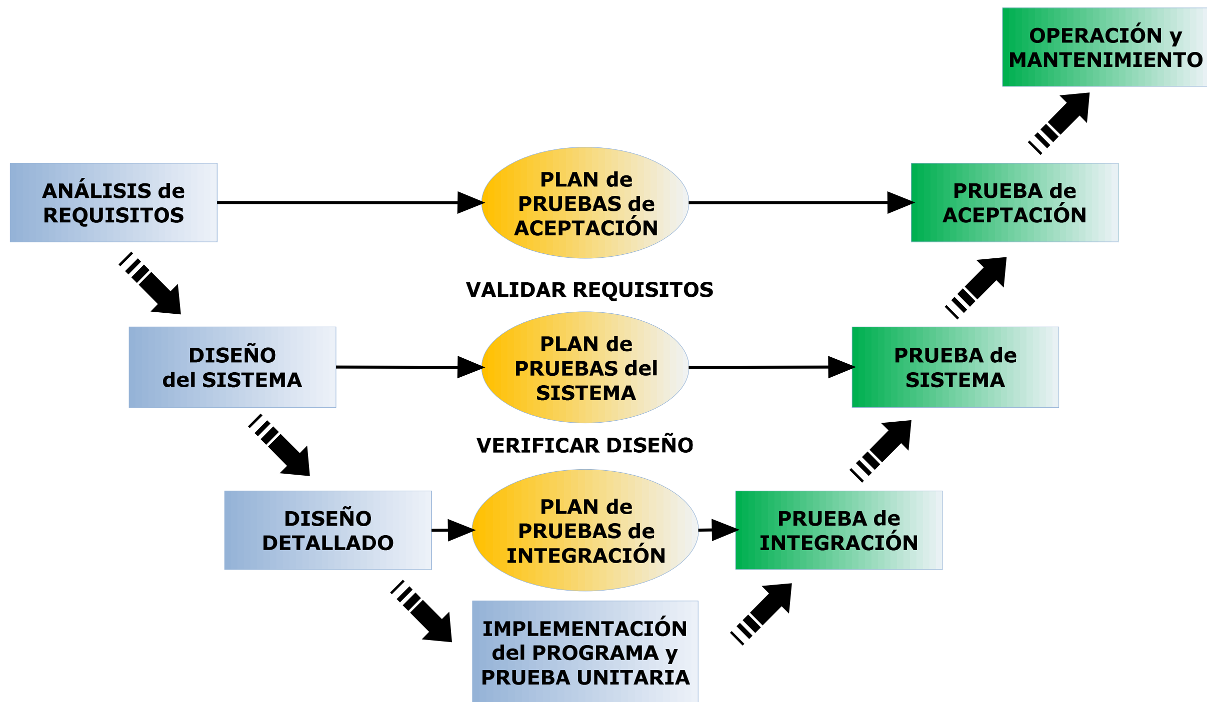


Ilustración 17. Modelo en V.

Muchos de los problemas ocasionados por el método en cascada puro son debidos a que las fases no están solapadas, a la excesiva documentación y a la dificultad para retroceder. Se han descrito diferentes modelos intentando mejorar estos puntos débiles:

MODELO SASHIMI (CASCADA CON FASES SUPERPUESTAS)

Peter DeGrace describe una de las modificaciones al método en cascada como el *modelo sashimi*. El nombre proviene de un modelo japonés (de Fuji-Xerox) para el desarrollo de hardware y se refiere al estilo japonés de presentar las rodajas superpuestas de pescado crudo. Sin embargo, uno de los problemas que ahora se plantean es definir los *milestones* o hitos, ya que se trabaja en paralelo. Esto también puede acarrear consecuencias si se hacen suposiciones erróneas.

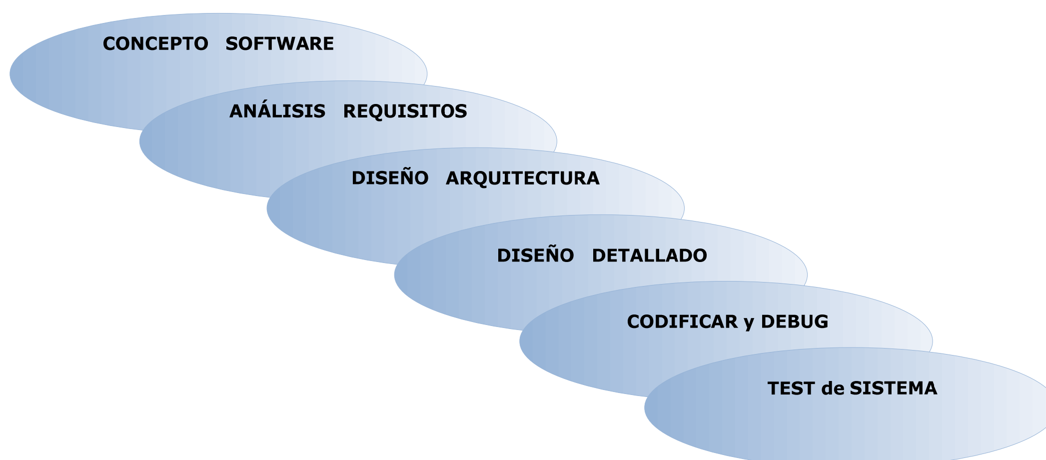


Ilustración 18. Una mejora del modelo en cascada, modelo Sashimi, cascada con fases superpuestas.

CASCADA CON REDUCCIÓN DE RIESGOS

Otra debilidad del modelo en cascada recae en la necesidad de definir todos los requisitos antes de empezar el diseño de la arquitectura. Una modificación, podría añadir una espiral de reducción de riesgos en el principio del modelo en cascada para tratar el riesgo de los requisitos. Se podría desarrollar un prototipo usuario-interfaz, entrevistas con usuarios, o cualquier otra forma de conocer los requisitos. El diagrama de este método es el de cascada original añadiendo una espiral para tratar riesgos que afectaría a las fases de análisis de requisitos y de diseño de arquitectura.

CASCADA CON SUBPROYECTOS

Otro problema del modelo en cascada puro es que supone tener completamente acabado el diseño de la arquitectura antes de empezar el diseño detallado. De igual forma, el diseño detallado debe estar completamente hecho antes de empezar a codificar y depurar, el *debug*.

Los sistemas tienen algunas partes conllevarán dificultades o sorpresas, pero otras partes que son habituales y se han implementado en otros proyectos, no comportarán problemas. Entonces, se plantea la cuestión ¿por qué retrasar la implementación de las partes fáciles sólo porque se esté esperando al diseño de una parte difícil? Si la arquitectura divide el problema en diferentes subsistemas o bloques independientes, cada uno podría evolucionar por separado. El único punto a tener en cuenta son las posibles interdependencias no previstas.

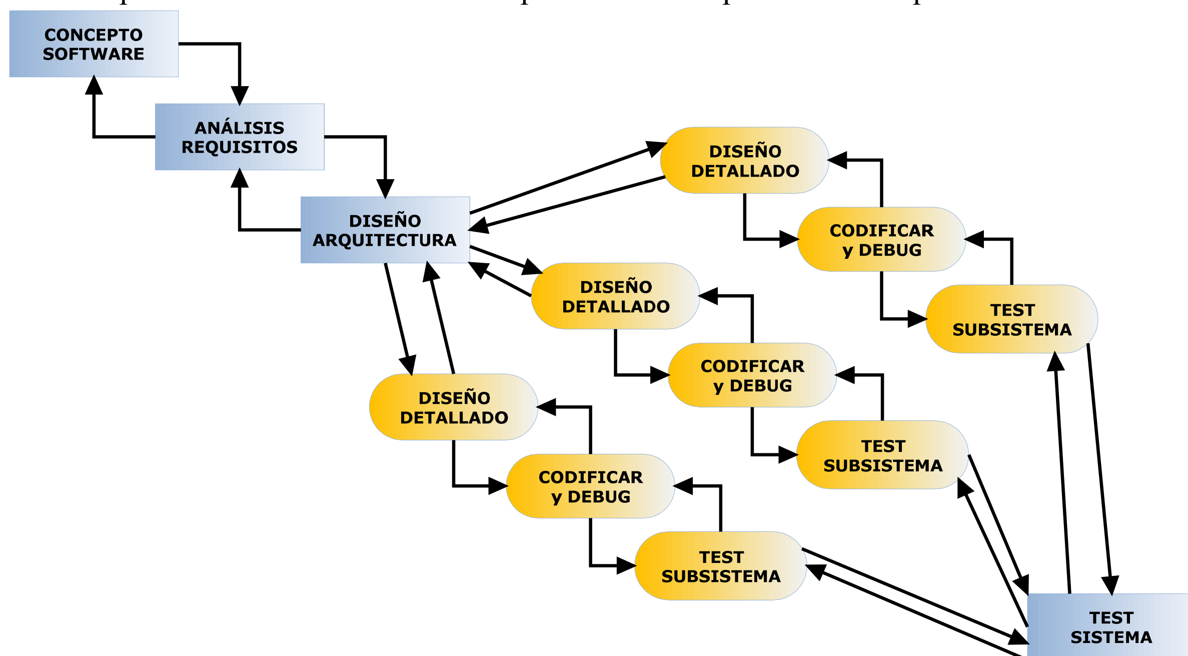


Ilustración 19. Otra mejora del modelo en cascada, cascada con subproyectos.

ENTREGA POR ETAPAS (*STAGED DELIVERY*)

Este modelo proporciona diferentes versiones del software al cliente, estableciendo de antemano qué contendrá cada versión.

La mayor desventaja es que precisa planear todas las dependencias para que el programa pueda funcionar de forma completa desde las primeras entrega, aunque evidentemente sin todas las características finales.

Este concepto de hacer una versión total pero incompleta es similar al *esqueleto ambulante* de los métodos Crystal, es un esqueleto que camina pero carece de la carne o funcionalidades).

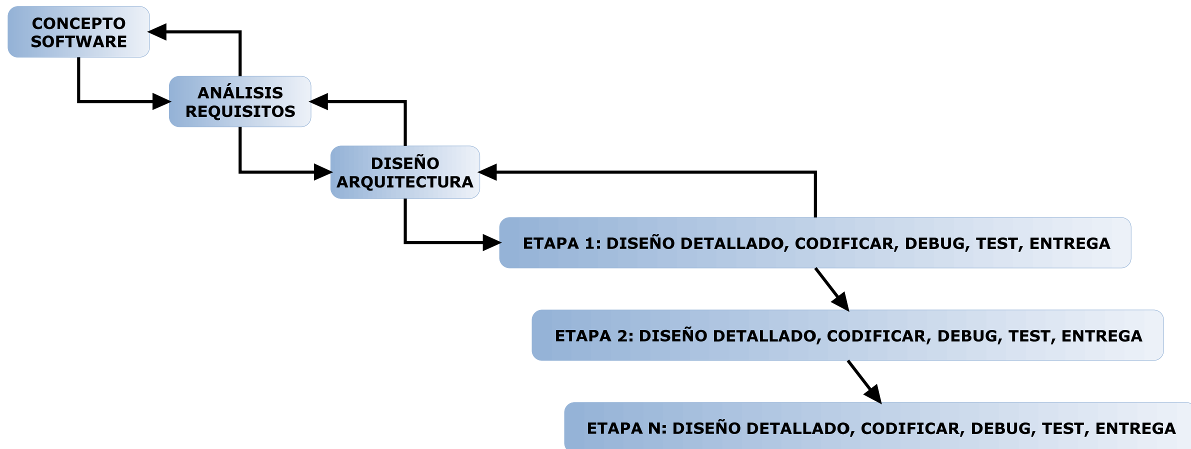


Ilustración 20. Evolución del método en cascada, entrega por etapas.

ENTREGA INCREMENTAL (EVOLUTIONARY DELIVERY)

Se desarrolla una versión del software, se muestra al cliente, y se refina según su *feedback*.

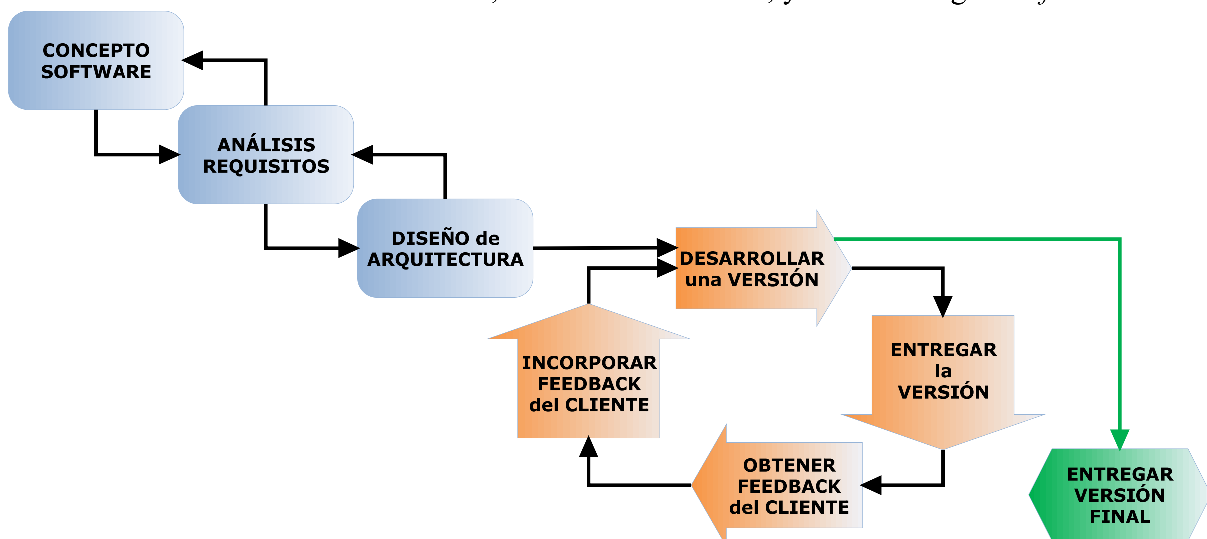


Ilustración 21. Entrega incremental.

1.2. La necesidad de nuevos métodos

En 1994, el grupo Standish[4], analizó más de 8.000 proyectos de software (el documento íntegro se encuentra en los anexos) y las conclusiones de *The CHAOS Report (1994)* fueron:

- Sólo el 16.2% de los proyectos se completó a tiempo, cumpliendo el presupuesto y con las funcionalidades inicialmente propuestas.
- El 31.1% de los proyectos se canceló antes de acabar.
- El 52.7% de los proyectos no cumplió presupuesto, tiempo o funcionalidades iniciales (o varios factores). Por ejemplo, costaron en media el 189% de su presupuesto inicial.
- Más del 25% se completó con un 25 a un 49% de las funciones y características especificadas originalmente.

[4] www.standishgroup.com/sample_research/chaos_1994_1.php

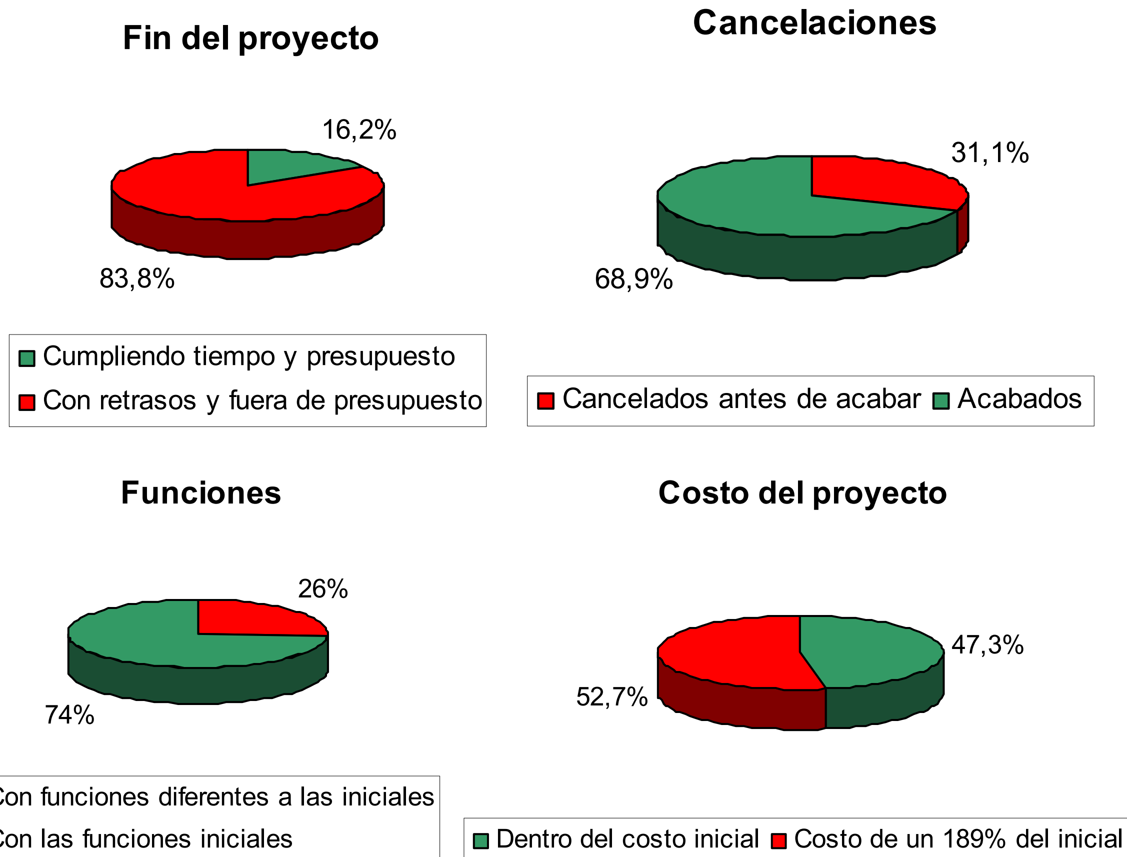


Ilustración 22. Consecuencias preocupantes de los métodos no ágiles.

En el apartado 4.6 se ve la evolución de estos factores hasta 2004 que ha realizado el grupo Standish. También se dispone de una amplia encuesta realizada en 2006 por Scott Ambler.

Otros ejemplos destacados de fracasos en el desarrollo de software:

London Ambulance Dispatching System (1992)

- Sistema para gestionar las llamadas de emergencias.
- Inversión: 1.2 millones de libras.
- Pérdidas: se estiman 20 vidas.
- Problemas: El sistema no distinguía llamadas distintas. Retenía llamadas durante horas. Usuarios sin formación. Implantación del sistema de manera apresurada.

Agencia Espacial Europea (1996)

- Sistema de navegación del Ariane 5. Evolución del Ariane 4.
- Inversión: 7 billones de dólares.
- Pérdidas: 2 satélites se desintegraron.
- Problema: Overflow al operar con la velocidad (5 veces mayor que en el Ariane 4). No se controlaban las excepciones.

Nike (2001)

- Sistema para automatizar la gestión, producción y venta.
- Inversión: 400 millones dólares.
- Pérdidas: 100 millones en ventas. Reducción de un tercio del valor de las acciones.

- Problema: El sistema intercambiaba órdenes de producción. Exceso de stock en algunos productos y falta en otros.

FBI (2005)

- Sistema para aumentar la seguridad de las redes y modernizar las aplicaciones de investigación.
- Inversión: 581 millones dólares.
- Pérdidas: 170 millones de dólares y 5 años de trabajo.
- Problema: Prisas tras el 11-S; Continuos cambios en los requisitos y en los responsables; poca preparación de los directores de proyecto.

Tras analizar estos casos, las causas de los problemas que se repiten son:

1) Planificación pobre.
2) Objetivos poco claros.
3) Objetivos cambiantes durante el proyecto.
4) Previsiones poco realistas.
5) Falta de soporte "superior".
6) Falta de implicación del usuario.
7) Falta de comunicación en el equipo.
8) Uso de técnicas inadecuadas.

La alta y diversa demanda de software para Internet, con constantes actualizaciones, así como el software para terminales móviles pone de manifiesto la necesidad de optimizar desde el punto de vista económico y temporal los métodos que se emplean para diseñar software.

MÉTODOS TRADICIONALES	MÉTODOS ÁGILES
Proceso controlado, gestionado	*Aleatorio, oportunista, guiado por sucesos (accident-driven)*
Proceso secuencial, lineal	*Procesos simultáneos, solapados*
Proceso replicable	*Ocurre de forma completamente única*
Proceso racional, determinado, orientado a proceso (goal-driven)	*Negociado, comprometido y caprichoso*
Procesos y herramientas	*Individuos e interacciones*
Documentación comprensible	*Software que funciona*
Negociación de contratos	*Colaboración con el cliente*
Seguir un plan	*Responder al cambio*
Se basan en normas	*Se basan en heurísticas para crear código*
Mayor o menor resistencia a los cambios	*Aceptan e incluso fomentan el cambio*
Se habla con el cliente sólo en reuniones	*El cliente forma parte del equipo*
Equipos grandes (>15-20 miembros)	*Equipos de trabajo pequeños o medianos*
Procesos con muchas normas	*Procesos con pocas reglas*
Mucha documentación	*Poca documentación*
Mucho análisis y diseño	*Poco análisis y diseño*
Conceden mucha importancia a la arquitectura	*Conceden poca importancia a la arquitectura*

Tabla 5. Diferencias entre métodos ágiles y tradicionales.

Uno de los problemas inherentes a los métodos orientados a proceso es que están completamente delimitados, congelados antes de empezar el diseño del software. Esta característica les hace inflexibles a la hora de realizar cambios en los requisitos.

Los métodos ágiles se oponen a los rígidos o pesados por varios motivos:

- La recopilación completa de los requisitos es cara y requiere tiempo.
- Los requisitos suelen cambiar a lo largo de la realización de los proyectos, con lo que el trabajo realizado puede ser inútil.
- No son métodos que puedan adaptarse fácilmente a los cambios, y menos aún a los cambios rápidos.
- Carecen de realimentación entre clientes y desarrolladores.
- Aunque el sistema final cumpla los requisitos iniciales, el cliente puede no estar satisfecho con los resultados finales (***"Esto es lo que pedí, pero no es realmente lo que necesitaba"***).

Se considera como nacimiento público del "*Agile Movement*" el momento en que el *Agile Software Development Manifesto*[5], escrito por 17 programadores y consultores en un taller en Snowbird (Utah, Estados Unidos), vio la luz en 2001. También surgió la Alianza Ágil, una organización sin ánimo de lucro que tiene como fines el mejor entendimiento de los métodos ágiles y la creación de condiciones favorables para discutir e intercambiar opiniones sobre ellos. Las versiones originales en inglés del Manifiesto así como de los 12 principios se encuentran en los anexos. La reunión que se sintetizó en el Manifiesto, tuvo 4 puntos:

1) Se necesitan métodos que respondan a cambios durante el proyecto. El término *"ágil"* es más apropiado que *"light"* ya que proyectos con muchos programadores o de alta seguridad, no son precisamente *light*.
2) El segundo punto son las 4 declaraciones del Manifiesto.
3) El tercero, los 12 principios ágiles.
4) Los métodos ágiles tienen la libertad de definirse ellos mismos.

1.3. El Manifiesto Ágil

Estamos descubriendo mejores maneras de desarrollar software,
tanto por nuestra propia experiencia como ayudando a terceros.
A través de esta experiencia hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas.
Software que funciona sobre documentación exhaustiva.
Colaboración con el cliente sobre negociación de contratos.
Responder ante el cambio sobre seguimiento de un plan.

Es decir, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

[5] www.agilemanifesto.org y http://agilealliance.org

1.4. Los 12 principios de los métodos ágiles

Nosotros seguimos estos principios:

13) Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.

14) Aceptamos requisitos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

15) Entregamos software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.

16) Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.

17) Construimos proyectos con profesionales motivados. Les damos el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.

18) El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.

19) Software que funciona es la principal medida de progreso.

20) Los procesos ágiles promueven el desarrollo sostenible. Patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

21) La atención continua a la excelencia técnica y los buenos diseños mejoran la agilidad.

22) La simplicidad, el arte de maximizar la cantidad de trabajo no realizado, es esencial.

23) Las mejores arquitecturas, requisitos y diseños surgen de equipos que se auto organizan.

24) A intervalos regulares, el equipo reflexiona sobre cómo ser más efectivo; entonces mejora y ajusta su comportamiento de acuerdo con sus conclusiones.

Como síntesis:

- Enfatizar el compartir, las relaciones de los desarrolladores y el papel humano en los contratos, como oposición a los procesos institucionalizados y herramientas de desarrollo. Se favorece el trabajo en pequeños grupos y sin barreras.

- El objetivo primordial es entregar software que funcione. Se crean nuevas versiones regularmente, a veces incluso a diario, pero normalmente una vez al mes. Los programadores deben escribir código simple, directo, optimizado, reduciendo el tiempo dedicado a la documentación a un mínimo.

- Se prioriza la relación y cooperación entre programador y cliente en vez de un contrato estricto. La intención es dar valor añadido tan pronto como el diseño empieza.

- El grupo de desarrollo, formado por los programadores y representantes del cliente, debe ser informado y estar autorizado a realizar cualquier cambio durante el proceso de creación. Debe quedar constancia de esto en el contrato.

Lo nuevo en los métodos ágiles no son las técnicas que utilizan, sino el hecho de reconocer que las personas son lo más importante para que el proyecto sea un éxito. [Boehm 2002] ilustra parte de la evolución de los diferentes métodos:

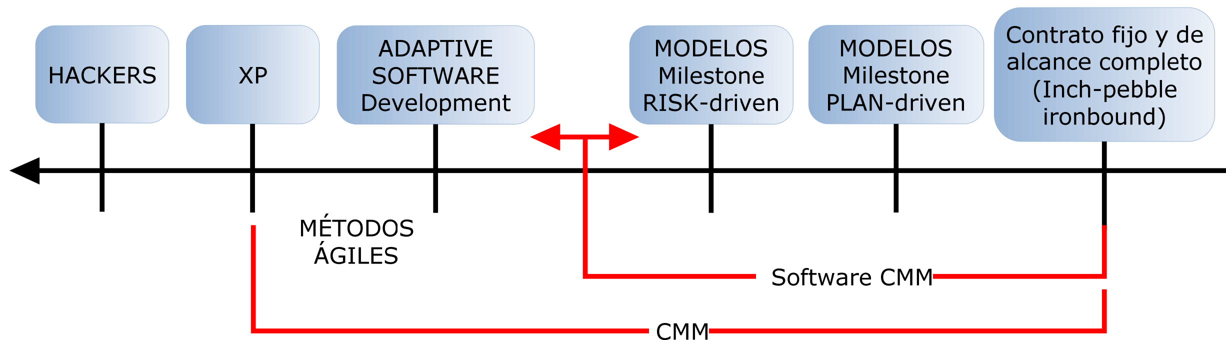


Ilustración 23. Eje temporal de la evolución de los métodos de desarrollo.

Un mismo método no puede ser el más eficiente en proyectos de muy diferente índole. Por tanto, la gestión de proyectos debe identificar cada proceso y aplicarle el método más eficiente. Esto lleva a la necesidad de tener tanto métodos ágiles como orientados a proceso.

Cockburn sintetiza que el núcleo de los métodos ágiles es tener reglas no demasiado estrictas y orientadas a la comunicación. Sus factores ideales para un proyecto son:

- Entre 2 y 8 personas en una habitación para facilitar la comunicación.
- Usuarios expertos *on-site* que proporcionan ciclos de *feedback* constantes y rápidos.
- Pequeños incrementos, mensuales o antes de 3 meses, para facilitar el testeo y corrección.
- Tests de regresión (realizados después de un cambio) totalmente automatizados: los tests unitarios *(unit test)* depuran el código de una clase, o un pequeño conjunto de clases, permitiendo mejoras continuas.
- Programadores experimentados, acabarán entre 2 y 10 veces antes que un equipo novel.

1.5. Características comunes de los métodos ágiles

Podemos destacar 9 pautas para acortar la duración o ciclo de vida de un proyecto:

1) Modular.
2) Iteratividad con cortos ciclos permitiendo verificaciones y correcciones rápidas.
3) Fijar tiempos (*time-bound*) con ciclos iterativos de 1 a 6 semanas.
4) Eliminar todas las actividades innecesarias.
5) Adaptarse con posibles nuevos riesgos que puedan aparecer.
6) Utilizar un método incremental para construir la aplicación en pequeños pasos.
7) Un enfoque convergente e incremental minimiza los riesgos.
8) Orientado a las personas, no a los procesos ni a la tecnología.
9) Forma de trabajar comunicativa y cooperativa.

[Favaro 2002] propone el desarrollo iterativo como el denominador común de los procesos ágiles. Los nuevos requisitos pueden ser introducidos, modificados o eliminados en las

sucesivas iteraciones. Este enfoque que puede retrasar la finalización del proyecto, necesita un contrato que permita estas dilataciones en los plazos de entrega.

[Highsmith y Cockburn 2001] afirman la importancia de tener satisfecho al cliente cuando se le entregue el proyecto y no sólo en el momento del inicio de éste con un contrato. Son necesarios métodos que acepten los cambios del cliente, y para ello, proponen unas reglas:

- Realizar la primera entrega en semanas, para lograr una victoria rápida y obtener el imprescindible *feedback* del cliente.
- Inventar soluciones simples, de forma que haya menos que cambiar y hacer las cosas más fáciles.
- Mejorar la calidad del diseño continuamente, haciendo la siguiente iteración más fácil de implementar.
- Testear constantemente para detectar fallos pronto y ahorrar dinero.

Un planteamiento para procesos ágiles sugerido por [Ambler 2002] es:

- La gente cuenta.
- Es posible realizar menos documentación.
- La comunicación es un asunto crítico.
- Las herramientas de modelado no son tan útiles como se pensaba normalmente.
- No se requiere un gran diseño *up-front* completo como en el método en cascada.

	MÉTODOS ÁGILES	OPEN SOURCE SOFTWARE	MÉTODOS ORIENTADOS A PROCESOS
PROGRAMADORES	Ágiles, entendidos, trabajando juntos (*collocated*), colaboradores	Equipos distribuidos geográficamente, colaboradores y entendidos	Orientados al plan; habilidades adecuadas; acceso a conocimientos externos
CLIENTES	Dedicados, entendidos, trabajando juntos, colaboradores, autorizados	Dedicados, entendidos, colaboradores, autorizados	Acceso a clientes entendidos, colaboradores, representativos y autorizados
REQUISITOS	Principalmente emergentes; cambios rápidos	Principalmente emergentes; cambios rápidos, siempre en desarrollo, nunca acaban	Conocidos a priori; mayoritariamente estables
ARQUITECTURA	Diseñada para los requisitos actuales	Abierta, diseñada para los requisitos actuales	Diseñada para los requisitos previstos
REFACTORING	Económico	Económico	Caro
TAMAÑO	Equipos y productos más pequeños	Equipos mayores, dispersados y productos más pequeños	Equipos y productos grandes
OBJETIVO VITAL	Dar valor rápido	Problema reto	Alta seguridad

Tabla 6: Boehm compara los métodos ágiles y los orientados a procesos (plan-driven) y añade el ejemplo del Open Source Software.

Puntos comunes a todos los métodos que se consideren ágiles:

AGILIDAD:

"Ágil" significa ser capaz de moverse de forma rápida pero con decisión, reaccionar ante los cambios con velocidad y destreza y cambiar la dirección manteniendo el equilibrio. Este equilibrio contrasta con métodos más tradicionales que se asemejan a una disciplina militar.

CAMBIO

Los métodos ágiles tratan los cambios como algo esperado, los aceptan y valoran mucho porque usan su información como *feedback* para adaptarse y ser más eficientes. Por contra, los métodos tradicionales, ven los cambios como enemigos y dedican mucho esfuerzo en controlarlos para poder retomar el plan inicial.

PLANNING

Cuando un proyecto ágil no avanza como estaba previsto, los cambios o desviaciones que ha habido le aportan información para mejorar el plan. Los métodos tradicionales aplican correctivos para no cambiar el plan. La cuestión es, *¿debería moldearse la realidad para encajar en el plan?* o *¿Debería reconstruirse el plan para amoldarse a la realidad?* La respuesta más sensata es una poco de cada; en el punto medio está la virtud según Aristóteles.

COMUNICACIÓN

Todos los MA promueven la comunicación entre los *stakeholders* o participantes (cliente, usuario final, contratista, etc.) y quitan importancia a la documentación que no ayude en la comunicación. Por tanto, son contrarios a la documentación que acabará archivada y olvidada en la mayoría de los casos. Los métodos tradicionales no están en contra de la comunicación, pero la formalizan en demasía según los MA. Evidentemente, llevado al extremo, la ausencia de documentación sería un problema, por eso se debe tener en cuenta que **todo aquello de lo que valga la pena hablar, vale la pena que quede constancia por escrito**.

APRENDIZAJE

Todos los MA se toman un proyecto como una experiencia de la que aprender. En el comienzo del proyecto, ni los clientes ni los programadores tienen un control absoluto de lo que se debe hacer. También aprenden del *feedback* del cliente. Los métodos tradicionales intentan que todo quede definido desde el principio, y si aparecen cambios, documentan estos problemas y aplican correctivos.

Como resumen podemos decir que un método de desarrollo de software ágil es:

- ✓ **Incremental**: versiones pequeñas de software, con ciclos rápidos.
- ✓ **Cooperativo**: desarrolladores y cliente siempre en contacto constante.
- ✓ **Directo**: el método en si es fácil de aprender y modificar, bien documentado.
- ✓ **Adaptativo**: capaz de tolerar cambios de última hora.

1.6. Manifiesto de proyecto ágil

El Manifiesto de proyecto ágil, Agile Project Manifesto, disponible en www.pmdoi.org, dice así (el original en inglés se encuentra en los anexos):

DECLARACIÓN DE INTERDEPENDENCIA

Enfoques ágiles y adaptables para unir personas, proyectos y valor.

Somos una comunidad de líderes de proyecto con éxito a la hora de entregar resultados. Para lograr estos resultados:

- **Aumentamos la ganancia en la inversión** (*increasing return*) centrándonos en el flujo continuo de valor.
- **Entregamos resultados fiables** comprometiendo a los clientes en interacciones frecuentes y propiedad compartida.
- **Esperamos la incertidumbre** y la manejamos gracias a las iteraciones, anticipación, y adaptación.
- **Liberamos creatividad e innovación** reconociendo que los individuos son la máxima fuente de valor, y creando un ambiente donde puedan diferenciarse.
- **Incrementamos el rendimiento** a través de la responsabilidad de grupo para una responsabilidad y resultados compartidos, buscando la efectividad del equipo.
- **Mejoramos efectividad y fiabilidad** a través de estrategias, procesos y prácticas específicas adaptadas a la situación.

©2005 David Anderson, Sanjiv Augustine, Christopher Avery, Alistair Cockburn, Mike Cohn, Doug DeCarlo, Donna Fitzgerald, Jim Highsmith, Ole Jepsen, Lowell Lindstrom, Todd Little, Kent McDonald, Pollyanna Pixton, Preston Smith y Robert Wysocki.

El título "Declaración de Interdependencia" tiene significados múltiples. Significa que los miembros de un equipo son parte de un todo interdependiente y no un grupo de individuos aislados. Significa que los equipos del proyecto, sus clientes, y sus participantes (*stakeholders*) también son interdependientes. Los equipos del proyecto que no reconocen esta interdependencia, raras veces tendrán el éxito.

Estos valores también forman un juego interdependiente. Mientras cada uno es importante independientemente de los otros, los seis forman un sistema de valores que proporciona una visión moderna de gestionar los proyectos, particularmente los complejos e indecisos. Las seis declaraciones (valor, incertidumbre, clientes, individuos, equipos, y contexto específico de la situación) definen un todo inseparable. Por ejemplo: es difícil entregar valor sin un cliente que valore algo. Es difícil tener equipos viables sin reconocer las contribuciones individuales. Es difícil manejar la incertidumbre sin aplicar las estrategias específicas circunstanciales.

Cada una de las declaraciones de valor tiene una forma distinta; el porqué un punto es importante precede a la descripción del valor. Por tanto, la "ganancia creciente en la inversión" es la razón de que centrarse en el flujo de valor continuo es importante. Las declaraciones de valor dan énfasis a la importancia de entregar resultados fiables (no es lo mismo que repetibles), gestionando la incertidumbre, liberando la creatividad e innovación, incrementando el rendimiento, y mejorando la efectividad.

Cada uno de los medios de las declaraciones expresa lo que ese grupo piensa y son los aspectos más importantes de la gestión de proyecto moderna, y también intentan diferenciar un estilo ágil y adaptable de gestionar proyectos. Por ejemplo, en la última declaración de valor, la frase “estrategias, procesos y prácticas específicas adaptadas a la situación” (*situationally specific strategies, processes, and practices*) indican que estos asuntos no deben ser demasiado estándares ni estáticos, sino dinámicos para adaptarse a las necesidades del proyecto y a los equipos. Otros estilos de gestión de proyectos son más propensos a la estandarización y a los procesos preceptivos.

Para más información sobre esta organización, se puede visitar www.apln.org o el foro de discusión en www.groups.yahoo.com/group/agileprojectmanagement.

Jim Highsmith, 17 de febrero de 2005.

CAPÍTULO

2

CONTENIDO

2.1. XP
2.2. Scrum
2.3. Crystal
2.4. FDD
2.5. RUP, EUP & AUP
2.6. DSDM
2.7. ASD
2.8. OSS
2.9. LSD
2.10. AM
2.11. Evo
2.12. ISD
2.13. MSF
2.14. PP

MÉTODOS ÁGILES

En este capítulo veremos los principales métodos ágiles:

Extreme Programming (Beck, 1999)
Scrum (Schwaber, 1995; Schwaber & Beedle, 2002)
Crystal (Cockburn, 2002)
Feature Driven Development (Palmer & Felsing, 2002)
Rational Unified Process (Kruchten, 1996 y 2000)
Enterprise Unified Process (Ambler, 2004)
Agile Unified Process (Ambler, 2005)
Dynamic Systems Development Method (Stapleton, 1997)
Adaptive Software Development (Highsmith, 2000)
Open Source Software development
Lean Software Development (Charette; Poppendieck, 2003)
Agile Modelling (Ambler, 2002)
Evolutionary Project Management (Gilb, 1976)
Microsoft Solutions Framework (Microsoft, finales de los 90)
Internet-Speed Development (finales de los 90)
Pragmatic Programming (Hunt & Thomas, 2000)

De cada método, observaremos el proceso (descripción de las fases para desarrollarlo), los papeles y responsabilidades de cada persona, la práctica (actividades y *workproducts* o artefactos definidos) y su adopción y experiencias.

"Haga un plan para tirarlo. Lo hará de todos modos"

Fred Brooks

2. MÉTODOS ÁGILES DE DESARROLLO DE SOFTWARE

2.1. Extreme Programming – XP

Extreme Programming (a veces escrito eXtreme Programming) o programación extrema, surgió como respuesta a la lentitud de los modelos tradicionales de desarrollo. En 1999, Kent Beck[6] puso por escrito la metodología de XP. Aunque las tácticas por separado que utiliza XP no son novedosas, la manera de unirlas sí. **El adjetivo "*eXtreme*" viene de llevar al extremo principios de sentido común:** ***"si diseñar es bueno, diseñemos todo el tiempo", "si las pruebas son buenas, probemos todo el tiempo"***. Se suele conocer como los "*three extremoes*" a Kent Beck, Ron Jeffries y Ward Cunningham. Las siglas XP no tienen relación con la versión Xp de Windows, donde significan "experience", aunque Microsoft utiliza prácticas ágiles.

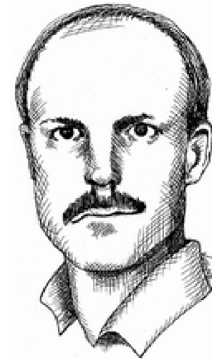


Ilustración 24. Kent Beck.

Cinco años después de ese libro, en 2004, Kent Beck publicó con Cynthia Andres la segunda edición de XP, que no es un refinamiento ni una ampliación de la primera sino que prácticamente vuelve a crear XP, aunque manteniendo los principios originales.

En la primera edición, XP se definió con 4 valores, 15 principios básicos, y 12 prácticas. Ahora, las doce prácticas han desaparecido. En el nuevo XP, hay 5 valores, 14 principios, 13 prácticas primarias y 11 prácticas secundarias. Las 24 nuevas prácticas reflejan las 12 originales sólo parcialmente. Dos de ellas, las metáforas y los estándares de codificación, ya no se proponen.

A mediados de la década de 1980, Kent Beck y Ward Cunningham trabajaban en un grupo de investigación de Tektronix. Allí idearon las tarjetas CRC "*Clase-Responsabilidad-Colaborador*" y sentaron las bases de lo que después serían los patrones de diseño y XP. Las tarjetas CRC son simples tarjetas de papel, de 4x6 o 3x5 pulgadas, y es una técnica que reemplaza a los diagramas en la representación de modelos.

En las tarjetas se escriben las responsabilidades, una descripción de alto nivel del propósito de una clase y las dependencias primarias. En su economía sintáctica y en su abstracción, anticipan lo que más tarde serán las tarjetas de historias de XP. Beck y Cunningham prohibían escribir en las tarjetas más de lo que cabía en ellas.

Ilustración 25. Ward Cunningham.

Aunque los principios y las prácticas que engloba Extreme Programming son formalmente más simples que otros métodos, Beck trabajó con Bill Wake para crear *XP 123* [7], es decir, el material estrictamente necesario para cuatro iteraciones, un par de meses.

[6] www.controlchaos.com

[7] www.xp123.com

Veremos que una de las prácticas más impactantes es la programación en parejas (a pares, conjunta o *pair programming*) que tuvo su origen cuando Kent y Ward empezaron a trabajar conjuntamente y mientras uno escribía código, el otro revisaba al instante.

La gestación de *Extreme Programming* empezó en 1996, cuando Kent Beck comenzó a trabajar en el proyecto Chrysler Comprehensive Compensation (C3) para la empresa Chrysler. Parte de las ideas de XP proceden de Ward Cunningham. El proyecto C3 era un proyecto escrito en *Smalltalk* que tenía como fin sustituir el sistema de nóminas de la compañía. Había comenzado en 1995, y Beck se incorporó a él a comienzos de 1996.

Beck recomendó a la empresa desechar el código ya escrito y comenzar de nuevo. Chrysler aceptó, y Beck tuvo las manos libres para inventar y aplicar la nueva metodología. En ocho meses, Beck y su equipo consiguieron producir un sistema piloto que contó con la aprobación del cliente. Sin embargo, en febrero de 2000, el proyecto fue cancelado porque había sobrepasado con creces el presupuesto original y sólo funcionaba para las nóminas de aproximadamente un tercio de los empleados de la empresa. El proyecto C3 llegó a tener unas 2.000 clases y unos 30.000 métodos. Para más detalles, ver el apartado 4.4.3.

Para Beck, **un plan no es una predicción**. Un plan se hace para que un grupo pueda colaborar, y para ayudar a tomar las decisiones sobre qué vía tomar, pero en ningún caso para imponer. Su misión más importante es cambiar los contratos sociales, cambiar la manera en que la gente trata a los demás y la forma que tienen muchas organizaciones de tratar a la gente.

XP se popularizó con el lanzamiento del libro *Extreme Programming Explained: Embrace Change* (1999), donde se reflejaban las experiencias obtenidas en el proyecto C3. Según el libro, el objetivo de XP es conseguir código ligero, conforme a las necesidades del cliente, y que pueda modificarse con rapidez, respondiendo a las necesidades cambiantes del cliente.

2.1.1. Los valores de XP (1999)

1) COMUNICACIÓN: Se hace énfasis en que la comunicación, para ser efectiva, debe involucrar a todos los participantes en el proyecto, y debe ser libre y sincera.

2) SIMPLICIDAD: Nunca debe perderse de vista que el objetivo de un proyecto es proporcionar valor al cliente; no es demostrar la pericia técnica del equipo ni construir una aplicación que resuelva más problemas que los del cliente. Cualquier arquitectura no debe ser más que una herramienta que cualquiera pueda entender y modificar sin necesidad de tener unos conocimientos especiales. Toda arquitectura debería comprenderse de un solo vistazo.

3) FEEDBACK: No se puede dirigir adecuadamente un proceso si no se dispone de *feedback* permanente sobre su progreso. El *feedback* puede provenir del cliente, de los programadores, de herramientas automáticas, etc. La arquitectura no debería ser más estática que el sistema al que modela. Una arquitectura que permita modificaciones sencillas debería capturar el estado o el tiempo de alguna manera.

4) CORAJE O VALENTÍA: A veces, hacer lo que es correcto requiere valor. Por ejemplo, hay que tener coraje para reescribir código que funciona pero ha alcanzado su límite de escalabilidad. Un programador XP debe ser lo bastante valiente como para desechar el código que no funciona y empezar de cero. También debe tener el coraje suficiente para involucrarse activamente en los proyectos.

Estos cuatro valores dan origen a cinco principios básicos:
- ✓ conseguir *feedback* rápido,
- ✓ no complicar las cosas con suposiciones (asumir que las cosas son simples),
- ✓ realizar cambios incrementales,
- ✓ abrazar el cambio, y
- ✓ generar productos de calidad.

Los cinco principios se manifiestan a través de las prácticas de XP.

2.1.2. Los nuevos valores de XP (2004)

En la nueva versión, XP se basa en cinco valores. Cuatro de ellos son prácticamente idénticos que el XP original, y se agrega el respeto como quinto valor. Los valores para XP segunda edición son:

1) COMUNICACIÓN: la mayoría de los problemas y errores provienen de la falta de comunicación. Debe haber comunicación entre los miembros del equipo y entre el equipo y los clientes. La comunicación más eficaz es la comunicación directa, interpersonal. También los artefactos deben ser fácilmente entendibles y estar actualizados.

2) SIMPLICIDAD: "*Haz lo más sencillo que podría funcionar*". Programar de forma sencilla –no simplista– requiere experiencia, ideas y trabajo duro. La simplicidad favorece la comunicación, reduce la cantidad de código y mejora la calidad. La idea subyacente es que las nuevas funciones se podrán agregar cuando se necesiten si el sistema es simple.

3) FEEDBACK: siempre debería poder compararse lo que está programado con lo que se quiere programar, respecto a las funciones que se necesiten. El *feedback* lo proporciona el contacto con el cliente y la disponibilidad de pruebas automatizadas que se desarrollan con el propio proyecto. Cuanto más simple es un sistema, más fácil es conseguir *feedback* sobre él.

4) VALOR (COURAGE): todos los métodos y procesos son herramientas para combatir y reducir nuestros miedos. Cuanto más miedo tengamos a un proyecto de software, mayores y más pesados serán los métodos que necesitaremos. La comunicación, la simplicidad y el *feedback* permiten adaptarse a los cambios grandes en los requisitos. También hay que tener valor para desechar código obsoleto.

5) RESPETO: los cuatro valores anteriores implican un quinto: el respeto entre los miembros y por su trabajo.

Los cinco valores no dan consejos específicos sobre cómo gestionar un proyecto, o cómo escribir código. Para este propósito, se utilizan las prácticas, y antes de las prácticas, se necesitan los principios que se detallarán más adelante.

2.1.3. Proceso (1999)

El ciclo de vida de XP consta de las siguientes fases [Beck 1999]: Exploración, Planteamiento, Iteraciones para entregar (*Release*), "*Productionizing*", Mantenimiento y Muerte.

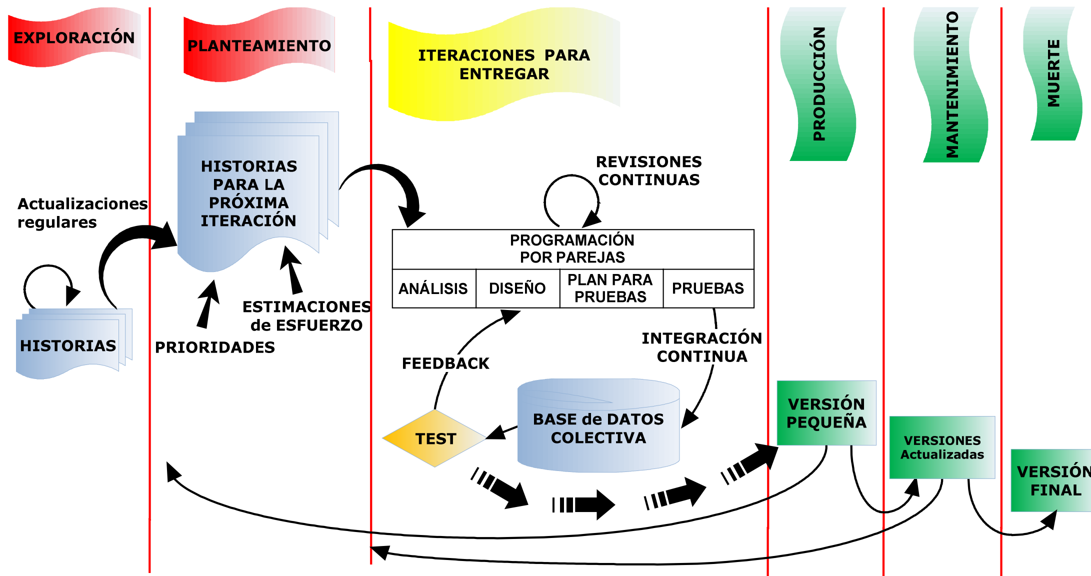


Ilustración 26. Pasos del proceso de Extreme Programming.

Al principio de cada iteración, los programadores traducen las historias de los usuarios en tarjetas CRC y desarrollan metáforas (apartado 2.1.6). Normalmente, la historia comprende un conjunto de clases y patrones, y da a los programadores un entendimiento común de cómo funcionan las cosas en cada etapa del proyecto y de cómo deberían funcionar.

En la **fase de Exploración**, los clientes escriben las historias en las tarjetas o fichas (*story cards*) lo que desean en la primera versión o *release*. Cada tarjeta describe una característica a implementarse, por ejemplo *"Cuando el cliente encuentra un producto que quiere comprar, lo añade a su carrito de la compra y continúa la compra"*. A partir de estas historias y de la prioridad que los clientes les asignan, se construye toda la planificación.

Al mismo tiempo, el equipo del proyecto se familiariza con las herramientas, tecnología y prácticas que usarán en el proyecto. La tecnología usada se testeará, y se explorarán las posibilidades de la arquitectura construyendo un prototipo del sistema. Esta fase necesita de unas semanas a unos meses, dependiendo en gran parte en lo familiarizado que estén los programadores con la tecnología a usar.

Consideremos un proyecto de implementación de un sistema de matriculación electrónica en una universidad. Asimismo, consideremos que el equipo XP que va a abordarlo se compone de ocho programadores. Tras reunirse con los clientes, el equipo XP ha obtenido las historias de los usuarios y ha estimado el número de puntos de cada una (1 punto = 1 semana teórica):

- Entrada en el sistema (2 puntos)
- Matriculación en el sistema (6 puntos)
- Consulta de las notas del curso actual (4 puntos)
- Consulta del expediente académico (3 puntos)
- Introducción de las notas de un alumno (3 puntos)
- Consulta del estado económico de la matrícula (1 punto)
- Pago electrónico de la matrícula (5 puntos)

Supongamos que el equipo ha determinado, basándose en sus proyectos anteriores, que su velocidad es de 2 puntos/semana real. En consecuencia, los ocho programadores podrán acabar 8 x (1/2) puntos por semana real (4 puntos).

En una iteración de tres semanas podrán acabar ocho puntos. O lo que es lo mismo, tres semanas de calendario equivaldrán a ocho semanas teóricas. Para la primera iteración, el cliente podrá elegir las historias que juzgue más necesarias o difíciles, pero de manera que la suma de sus puntos no supere los ocho puntos.

Faltarán, pues, otros 16 puntos (24-8) para cubrir todas las funciones del sistema. Si el cliente necesita todas las funciones, habrá que considerar un plazo total para el proyecto de nueve semanas o tres iteraciones (24 puntos divididos por 8 puntos/iteración).

El equipo puede comprometerse, en cada iteración, a implementar tantos puntos como fueron completados en la anterior iteración. Por ejemplo, si el equipo sólo acabase 6 de los 8 puntos de la primera iteración, para la segunda iteración únicamente propondría implementar historias cuya suma de puntos sea 6 puntos (o menos).

Tabla 7. Un ejemplo sencillo de organización de un proyecto XP.

Si bien podría parecer que las historias de usuario y los casos de uso (*use-case*, secuencia de acciones que realiza un actor en el sistema para conseguir un objetivo) son técnicas similares, no es así. **XP rechaza el uso de casos de uso**; los considera demasiado complicados y formales. Considera inútil el tiempo empleado en la redacción de casos de uso, en el uso de plantillas de casos de uso y en la notación UML de *includes*, *extends* y *uses*. Las historias de usuario son distintas de los típicos requisitos escritos, pues están enfocadas más hacia un nivel de descripción que permita estimar cuánto tiempo se necesitará para implementarlas. La planificación se descompone en dos partes: planificación de las versiones (*release planning*) y planificación de las iteraciones (*iteration planning*).

La **fase de Planificación de las versiones** establece el orden de prioridad de las fichas y las funciones de la primera versión. Los programadores estiman cuánto tiempo requiere cada función y se acuerda un calendario. El tiempo para la primera versión normalmente no excede de dos meses. La propia fase de planificación necesita un par de días.

La **fase de Planificación de las Iteraciones** para lanzar una versión incluye varias iteraciones de los sistemas antes de la primera versión. El calendario propuesto en la etapa de planificación se divide en un número de iteraciones de entre una y cuatro semanas. La primera iteración crea un sistema con la arquitectura del sistema completo. Esto es posible gracias a elegir las historias que serán la base del sistema, no detalles. El cliente decide las historias para cada iteración. Las pruebas funcionales, propuestas por el cliente, se realizan al final de cada iteración. Al final de la última iteración, el sistema está listo para la producción.

La **fase de Productionizing** requiere más pruebas y verificar el rendimiento del sistema antes de dárselo al cliente. Aquí, todavía pueden encontrarse nuevos cambios y se deberán hacer si se pactaron para esa versión determinada. Es posible que algunas iteraciones tengan que acelerarse. Las ideas pospuestas y sugerencias se documentan para realizarlas más adelante, por ejemplo, en la fase de mantenimiento.

Después de crear la primera versión para el uso del cliente, el proyecto debe seguir el desarrollo previsto y hacer nuevas iteraciones. Para lograrlo, la **fase de Mantenimiento**

requiere también un esfuerzo para las tareas de apoyo del cliente. Así pues, la velocidad de desarrollo puede disminuir después de que el sistema esté en producción. La fase de mantenimiento puede requerir incorporar más personal en el equipo y cambiar su estructura.

La **fase de Muerte** está cercana cuando el cliente ya no tiene ninguna funcionalidad o historia pendiente de ser implementada. Esto implica que el sistema satisface en todos los aspectos (rendimiento, fiabilidad, etc.) al cliente. Ahora es cuando se escribe la documentación ya que no se harán más cambios. Esta fase también puede ocurrir si el sistema no proporciona los resultados esperados o si se hace demasiado caro para continuar desarrollándolo.

2.1.4. Papeles y responsabilidades (1999)

Hay diferentes papeles en XP para cada tarea y propósito durante el proceso y su práctica:

PROGRAMADOR

Los programadores escriben las pruebas y mantienen el código del programa tan simple y optimizado como sea posible. Para que XP tenga éxito, debe haber comunicación y coordinación entre los programadores y los otros miembros del equipo. Se encargan de estimar el tiempo que llevará implementar las funciones del sistema asociadas a cada historia, definir las tareas que conlleva cada una e implementan las historias y las pruebas unitarias.

CLIENTE "EN CASA" (ON-SITE)

El cliente escribe las historias en fichas y las pruebas funcionales, decidiendo cuándo dar por válida cada funcionalidad. El cliente establece la prioridad de implementación de los requisitos o funcionalidades. En la segunda versión de XP, el papel del cliente ha cambiado ligeramente: originalmente era una sola persona, correspondía al perfil de un cliente real (preferentemente, un usuario final del sistema) y permanecía durante todo el proyecto en la misma habitación que el equipo de programadores. Después, el papel de Cliente se asignó a un solo analista de negocios, en lugar de a un cliente real o a un usuario. En la actualidad, el papel del cliente lo suele desempeñar un equipo de analistas.

TESTER

Los verificadores/comprobadores ayudan al cliente a escribir las pruebas funcionales que ejecutan regularmente. También informan de los resultados y mantienen las herramientas para verificar. De acuerdo con Beck, *"Un tester XP no es una persona separada, dedicada a romper el sistema y humillar a los programadores"*.

SUPERVISOR (TRACKER)

Se encarga de proporcionar *feedback*. Hace un seguimiento de las estimaciones hechas por el equipo dando *feedback* para saber cuán precisas son, para tener más precisión en estimaciones futuras. También sigue el progreso de cada iteración y evalúa si el calendario establecido es viable o si se ha de modificar parte del proceso.

ENTRENADOR O INSTRUCTOR

El *coach*, con un sólido conocimiento de XP, guiará a los miembros del equipo y es el responsable del proceso en conjunto. Vigila todo el proceso y se asegura de que el proyecto continúa siendo extremo, es decir, que se cumplen las doce prácticas. Beck comenta que la función más relevante del *coach* es la adquisición de juguetes y comida; otros dicen que está para servir café. Lo importante es que el *coach* se vea como un "facilitador" o ayudante, antes que como alguien que da órdenes.

CONSULTOR

El consultor o asesor es un miembro externo que posee el conocimiento técnico específico necesario. El consultor guía al equipo para resolver problemas específicos.

GESTOR O DIRECTOR (MANAGER, BIG BOSS)

El director toma las decisiones. Se comunica con el equipo del proyecto para determinar la situación actual y para distinguir cualquier dificultad o deficiencia en el proceso.

Algunos de estos papeles pueden combinarse en una sola persona; por ejemplo, los papeles de Gestor y Supervisor. **En un equipo XP no existen los papeles de Diseñador o de Arquitecto**, ni **tampoco existe el de Jefe Supremo que decide qué diseños son buenos** y cuáles malos: las decisiones de diseño las toma el equipo de programadores.

2.1.5. Las prácticas (1999)

XP es una colección de ideas y prácticas producto de metodologías ya existentes. El hecho de que el cliente tome las decisiones comerciales mientras los programadores deciden en los problemas técnicos, se deriva de las ideas de *"The Timeless Way of Building" de* C. Alexander (1979). La evolución en XP tiene sus raíces en las ideas de *Scrum* (Takeuchi y Nonaka, 1986) y en el lenguaje de patrones de Cunningham (1996). La idea de fijar los proyectos basada en las fichas del cliente proviene de Jacobsen (1994) y la entrega evolutiva (el cliente evalúa varias versiones antes de la definitiva) se adopta de Gilb (1988).

También el modelo en espiral, una mejora al modelo en cascada, ha tenido su influencia en XP. Las metáforas de XP se originaron en los trabajos de Lakoff y Johnson (1998), y Coyne (1995). Finalmente, el entorno donde trabajarán los programadores se adopta de Coplien (1998) y de deMarco y Lister (1999).

Entre los artefactos diferenciadores que se utilizan en XP, están las tarjetas de historias (*story cards*). Son tarjetas comunes de papel en que se escriben breves requisitos de una característica que es requisito (**jamás casos de uso**) y pueden adoptar el esquema CRC. Las tarjetas tienen una periodicidad de diez o veinte días.

Las tarjetas se usan para estimar prioridades, alcance y tiempo de realización; en caso de discrepancia, gana la estimación más optimista. XP también utiliza listas de tareas en papel o en una pizarra (jamás en el ordenador) y gráficos visibles pegados en la pared. Martin Fowler[8] admite que la preferencia por esta clase de herramientas hace que el mensaje entre líneas sea "*Los XPertos no hacen diagramas*". La gran preocupación de Fowler son los testeos y la colaboración, dos puntos resaltados por XP que está orientado a la gente, no a procesos.

XP tiene como objetivo permitir el desarrollo satisfactorio de software en equipos pequeños o medianos, a pesar los requisitos cambiantes. Las iteraciones breves, con las primeras versiones que proporcionan un *feedback* rápido, la participación del cliente, la comunicación y coordinación, la integración continua y las pruebas, la propiedad colectiva del código, la documentación limitada y la programación por parejas son las características principales de XP.

[8] www.MartinFowler.com

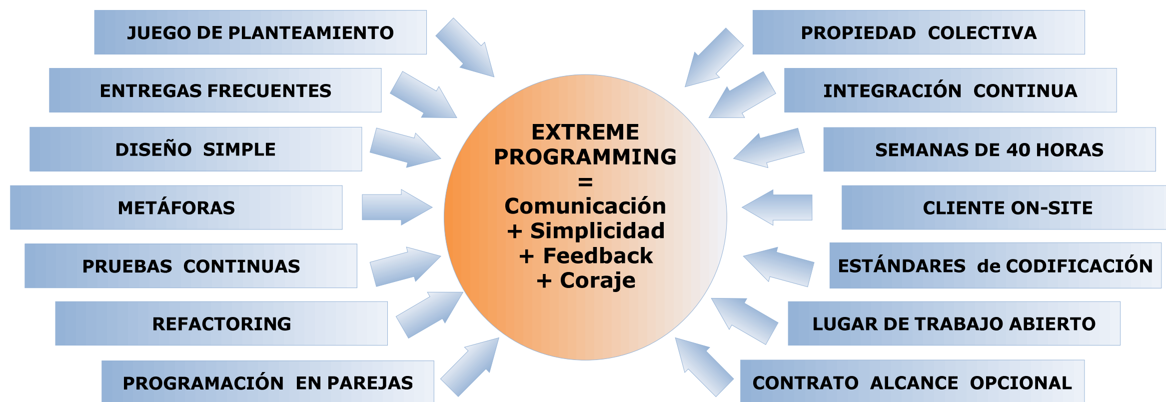


Ilustración 27. Valores y prácticas de Extreme Programming.

2.1.6. Principios de XP (1999)

Las prácticas, que Ward Cunningham sugiere rebautizarlas como *Xtudes* de XP, según [Beck 1999] son:

1. JUEGO DE PLANTEAMIENTO

El *planning game* necesita una alta interacción entre el cliente y los programadores. Los programadores estiman el esfuerzo necesario para implementar las historias del cliente y éste decide qué contendrán las versiones y cuándo estarán.

2. RELEASES PEQUEÑAS O ENTREGAS FRECUENTES

Un proyecto XP se divide en iteraciones o ciclos. La versión del sistema para cada ciclo proporciona el código para un pequeño conjunto de funciones, es decir, en cada ciclo o iteración únicamente se implementan unas cuantas historias de los usuarios. Por otro lado, un proyecto XP también es incremental: cada nuevo bloque de código se incorpora a la aplicación en cuanto está listo. Por tanto, se creará una versión pequeña pero operativa, por lo menos una vez cada 2 o 3 meses. Se realizan nuevas versiones incluso diariamente, pero al menos, una vez al mes.

3. METÁFORA

El sistema se define con unas metáforas entre el cliente y los programadores. Esta "historia compartida" guiará todo el desarrollo describiendo cómo trabaja el sistema. Una metáfora puede interpretarse como una arquitectura simplificada, una especie de imagen mental compacta del sistema.

4. DISEÑO SIMPLE

El énfasis se pone en diseñar la solución más simple posible. La complejidad innecesaria y código no necesario se eliminan inmediatamente. Los programas deben ser tan pequeños como sea posible. Para eliminar la duplicación de código, se utiliza la refactorización. **A menos código, menos errores**.

XP rechaza la idea de un análisis completo y la de comunicar el diseño de un sistema mediante diagramas UML y especificaciones textuales. En lugar de eso, propone que el diseño se comunique mediante conversaciones, metáforas, anotaciones en las tarjetas, refactorización y código. Sobre todo, mediante el propio código.

5. PRUEBAS CONTINUAS O INMEDIATEZ

Una aplicación XP sólo debe estar escrita para cumplir los requisitos del ahora: no debe pensarse en posibles ampliaciones futuras o en el qué pasará. El desarrollo del software estará condicionado por los resultados de las pruebas. Los tests de unidad se ejecutan constantemente. Los clientes escriben las pruebas funcionales. A esta filosofía se la llama desarrollo según pruebas, o ***test-driven***. Las pruebas y el código las escribe el mismo programador, pero la prueba debería realizarse sin intervención humana, y es a todo o nada. Hay dos clases de prueba:

- **la prueba unitaria o de unidad** (a nivel de método) que verifica una clase, o un pequeño conjunto de clases y son responsabilidad del programador.
- **la prueba funcional o de aceptación** que verifica todo el sistema, o una gran parte, y son propuestas por el cliente.

XP presenta una característica llamativa para los programadores tradicionales: **las pruebas para comprobar el código deben escribirse antes de escribir el código**. Esta forma de proceder es exactamente contraria a la de la programación convencional: por lo general no se programa con la intención de pasar pruebas. En XP, el propósito de una clase no es cumplir sus requisitos, sino pasar todas las pruebas que corresponden a esa clase. XP afirma que obrar así reduce las posibilidades de que cambios en algunas porciones del código, aparentemente inofensivos, puedan provocar graves fallos en el funcionamiento global.

Por regla general, una prueba unitaria es un programa que envía un mensaje a una clase y verifica que la respuesta es la predicha. Suelen implementarse como aserciones. **Cuando se va a incorporar nuevo código al sistema, se ejecutan todas las pruebas unitarias existentes, no sólo las asociadas al nuevo código**. Es la técnica **test-first**. Si en el sistema se introduce código nuevo que pasa las pruebas definidas para él, pero hace que fallen pruebas unitarias de otras clases, se rechaza el nuevo código. Con esta forma de trabajar, el programador siempre sabe que los fallos se deben al nuevo código incorporado, pues en la última versión funcionaban todas las pruebas. Estas pruebas se llaman **test de regresión**.

Las pruebas unitarias suelen usarse de forma automática (usando herramientas como JUnit, desarrollado por Erich Gamma y Kent Beck y que se verá en el capítulo 3), sin la interacción del usuario cada vez que se modifica el sistema. Se llaman así porque implican comprobar por separado cada unidad de código para asegurarse de que funciona por ella misma, independientemente de las otras unidades del sistema. Unidad suele corresponder a clase (en los lenguajes orientados a objetos) o a módulo (en los lenguajes estructurados). La intención que alienta estas pruebas es sencilla: si todas las clases del sistema pasan las pruebas unitarias y, aun así, el sistema presenta problemas, la causa reside en la manera en que se han juntado las clases. Aseguran, pues, que las clases de la aplicación funcionan tal y como fueron concebidas.

Según esta metodología, las pruebas unitarias deben cumplir estos requisitos:

- Estar organizadas en grupos o baterías,
- cubrir todas las clases del sistema,
- ejecutarse al cien por ciento, y
- ejecutarse cada vez que se hagan cambios en el código.

6. REESCRIBIR O REFACTORIZAR (REFACTORING)

Consiste en reestructurar el sistema quitando duplicidades, mejorar la comunicación, simplificar y agregar flexibilidad. Se lo ha parafraseado diciendo: *"Si funciona bien,*

arréglalo de todos modos". El término *refactoring*, introducido por W. F. Opdyke en su tesis doctoral, se refiere "al proceso de cambiar un sistema de software (orientado a objetos) de manera que no se altere el comportamiento exterior del código, pero se mejore su estructura interna". Normalmente se utilizan herramientas automáticas para hacerlo (DMS, GeNexus), y/o se implementan técnicas tales como aserciones (invariantes, pre- y poscondiciones) para expresar propiedades que deben conservarse antes y después de reescribir el código. Otras técnicas son transformaciones de grafos, métricas de software, refinamiento de programas y análisis formal de conceptos.

La preservación del comportamiento observable es imprescindible: en caso contrario, el código resultante de refactorizar no sería directamente comparable con el código previo. Para asegurar esto, suelen emplearse herramientas automáticas para la refactorización y herramientas para hacer pruebas automáticas, como *JUnit* (para Java), *CPPUnit* (para C++) o *Nunit* (para los lenguajes de la plataforma .Net).

El proceso general de una refactorización incluye los siguientes pasos:

- Se examina el código fuente.
- Se señalan los puntos donde conviene refactorizar.
- Se eligen los métodos de refactorización apropiados (extraer método, etc.).
- Se escriben pruebas unitarias para comprobar las funciones existentes. Si se sigue XP, este paso ya se habría hecho antes.
- Se aplican las refactorizaciones de una en una, por medio de alguna herramienta.
- En cada refactorización, se ejecutan las pruebas unitarias. Si fallan, hay que deshacer la última refactorización y ejecutarla en pasos más pequeños.

Por supuesto, para poder hacer estos pasos de una forma realista se precisa un entorno integrado de desarrollo como *Eclipse* o *NetBeans* y un sistema de control de versiones como *CVS*. Para más detalles sobre herramientas automáticas, ver el apartado 3.1.

Al igual que sucede con los patrones, existe un amplio catálogo de *refactorizaciones* comunes: reemplazo de iteración por recursividad; sustitución de un algoritmo por otro más claro; extracción de clase, interfaz o método; descomposición de condicionales; reemplazo de herencia por delegación, etc. El libro de referencia es *Refactoring: Improving the Design of Existing Code* de M. Fowler, K. Beck, J. Brant, W. Opdyke y D. Roberts.

7. PROGRAMACIÓN EN PAREJAS

La *pair programming* consiste en que dos personas escriben el código en una sola computadora, turnándose el uso del ratón y el teclado. Quien no escribe, piensa desde un punto de vista más estratégico y realiza lo que podría llamarse revisión de código en tiempo real. Los papeles pueden cambiarse varias veces al día. Si revisar el código es bueno, *¿por qué no revisarlo continuamente, incluso desde el mismo momento en el que se escribe por primera vez?*

8. PROPIEDAD COLECTIVA

Cualquiera puede cambiar cualquier parte del código cuando quiera, siempre que escriba antes la prueba correspondiente. Algunas veces los practicantes aplican el patrón organizacional *Code Ownership* de Coplien. Un programador puede modificar cualquier clase, no sólo las escritas por él.

9. INTEGRACIÓN CONTINUA

Cuando un trozo de código está listo, se integra en el sistema. Por tanto, el sistema se integra y se compila muchas veces cada día. Se realizan las pruebas para aceptar el nuevo código.

10. SEMANAS DE 40 HORAS

40 horas es el máximo; en RAD las horas extras eran una *best practice*. Aquí se considera que los programadores deben descansar y estar frescos para escribir código eficiente. No se permiten 2 semanas consecutivas haciendo horas extras. Si esto pasara, se trataría como un problema a ser resuelto.

11. CLIENTE EN CASA

Se necesita que haya al menos un representante de la empresa promotora con el que se pueda hablar fluidamente, de manera que pueda seguir en tiempo casi real el desarrollo del proyecto, así como proponer modificaciones. El cliente debe estar "en casa" (*on-site*); es decir, debe permanecer junto al equipo de XP durante todo el proyecto. En palabras de Kent Beck: *"Un verdadero cliente debe sentarse con el equipo, y estar disponible para contestar preguntas, resolver disputas y establecer las prioridades a pequeña escala"*. La participación del cliente es imprescindible en XP: se intenta que el cliente participe en las pruebas de software y que se atiendan lo antes posible todas las modificaciones que vaya formulando.

12. ESTÁNDARES DE CODIFICACIÓN

Los programadores siguen normas de programación. Se pone especial atención a la comunicación a través del código, mediante comentarios. Como en XP rige un cierto purismo de codificación, los comentarios no están muy bien vistos: **si el código es tan oscuro que necesita comentario, se debe reescribir o refactorizar**.

Estas son las 12 prácticas, aunque XP necesita de otras condiciones que podrían considerarse como prácticas:

13. LUGAR DE TRABAJO ABIERTO

Todos los miembros del equipo (programadores, comprobadores del código e instructores) trabajan juntos, en un mismo espacio. Los programadores y el cliente deben estar en un mismo sitio físico, o no muy alejados. Sería ideal una sala grande con pequeños departamentos o separadores y situar a los programadores en el centro.

14. REGLAS JUSTAS

El equipo tiene sus propias reglas, justas, a cumplir, pero también pueden cambiarse cuando se quiera. Los cambios deben ser acordados por todos y valorar su impacto.

15. CONTRATOS DE ALCANCE OPCIONAL

XP insiste en la importancia de que los proyectos se hagan con contratos de alcance opcional (*optional scope contracts*). "Alcance" hace referencia a las características o funciones del sistema que es objeto del contrato. Los contratos donde quedan fijados los plazos de entrega, el coste, los requisitos y la calidad (contratos de alcance global) implican riesgos para el cliente y los programadores. A veces, la implementación de una función resulta más complicada de lo esperada y se debe rebajar el tiempo dedicado a otras funciones o hacer horas extra no deseadas. Por otro lado, el cliente puede encontrarse con una aplicación que cumple los términos del contrato (requisitos, calidad y fecha de entrega), pero que no se corresponde a sus verdaderas necesidades. Si el cliente se da cuenta de este hecho antes de

que finalicen los planos de entrega, puede exigir modificaciones; lo cual conlleva el riesgo de que la calidad del sistema se resienta (el programador debe hacer más cosas en el mismo tiempo) o de gastar más dinero (el programador puede decir que se ha limitado a cumplir los requisitos). Si se da cuenta después, la única solución para que el sistema se adapte a sus necesidades consiste en firmar un nuevo contrato (y gastar, por tanto, más dinero).

Con los contratos de alcance opcional, XP intenta evitar esos problemas. Un contrato de este tipo establece el tiempo, el coste y la calidad del proyecto; a cambio, permite que el conjunto de funciones del sistema se ajuste cuando se necesite, de común acuerdo entre el cliente y el desarrollador. XP argumenta que el riesgo para el cliente es mínimo: como el contrato de alcance opcional trabaja con el modelo de XP, el cliente puede establecer un primer ciclo de desarrollo para reducir todo lo posible los riesgos que corre. Tras la primera versión, el cliente conocerá lo que va a obtener por su dinero, y si ha seleccionado o no a un desarrollador que puede satisfacer sus necesidades. Si está contento con los resultados, decidirá seguir adelante con el proyecto, lo cual conllevará más versiones (y más dinero) para el desarrollador.

Este tipo de contrato posibilita que, si el cliente detecta que sus requisitos no eran completos o no eran lo que en realidad quería, tenga la oportunidad de cambiarlos en el próximo ciclo. Como el cambio de alcance no afecta a la entrega del software o al pago del trabajo realizado en los otros ciclos, el programador no tiene ninguna objeción al cambio de requisitos. Al contrario: le supondrán nuevos ingresos.

Barry Boehm, tras estudiar los datos de 67 proyectos de software en 1987, concluyó que encontrar y solucionar un problema de software después de entregarse al cliente cuesta hasta 100 veces más que encontrar y arreglar el problema en las etapas de diseño iniciales. En 2001, Boehm postuló que, para sistemas pequeños, el factor de proporcionalidad se halla más próximo a 5 que a 100. Esta es la curva de Boehm, a la que Kent Beck añadió la de Extreme Programming, aunque sin una justificación como la de Boehm.

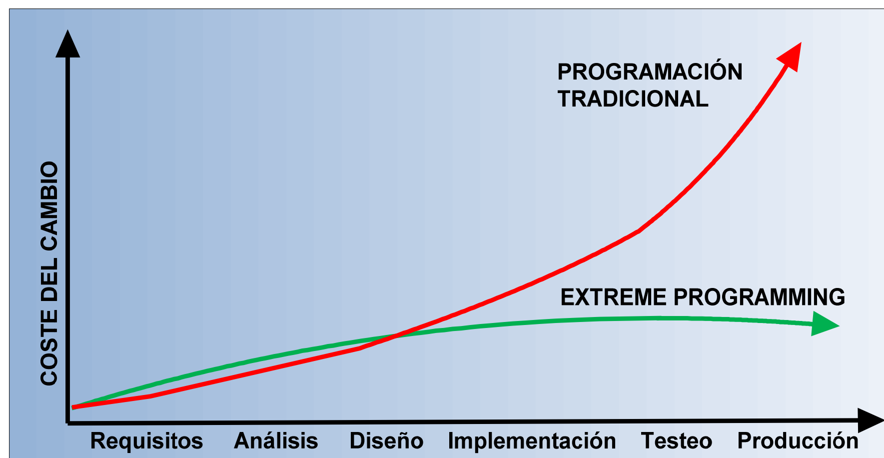


Ilustración 28. Coste de los cambios según Boehm y Kent Beck.

2.1.7. Las nuevas prácticas (2004)

El nuevo XP se basa en 13 prácticas primarias, y 11 prácticas secundarias (corolario). Primero deben aplicarse las prácticas primarias, y cada una de ellas puede brindar una mejora en el proceso de desarrollo de software. Las prácticas secundarias, requieren dominio de las primarias, y son difíciles de aplicar sin éstas. Las 24 prácticas deben aplicarse íntegramente para obtener todos los beneficios de XP.

Kent Beck no clasifica las prácticas en categorías, aunque podrían clasificarse en cuatro categorías (ciertas prácticas podrían pertenecer a dos grupos): Análisis de requisitos y Planning; Equipo y Factores humanos; Diseño, y Programar y lanzar versiones.

PRÁCTICAS PRIMARIAS

– Análisis de requisitos y Planning:

1) HISTORIAS: se describen todas las funciones del sistema usando historias, descripciones breves de funciones que el cliente podrá ver.

2) CICLO SEMANAL: el desarrollo del software se realiza semanalmente. Hay una reunión al principio de cada semana donde el cliente elige, según prioridades y teniendo en consideración el tiempo necesario por los programadores, las historias a programar durante la semana.

3) CICLO TRIMESTRAL: trimestralmente se evalúa el desarrollo y el progreso del proyecto.

4) SLACK (POCO SERIO): evitar hacer promesas que no puedan cumplirse. En cualquier plan, se incluyen algunas tareas que pueden eliminarse si se retrasa. De esta manera, habrá un margen de seguridad por si hay imprevistos.

– Equipo y Factores Humanos:

5) COMPARTIR ESPACIO: los equipos de desarrollo deben trabajar en un espacio sin divisiones para facilitar la comunicación.

6) EQUIPO COMPLETO: el equipo debe componerse de miembros con todas las habilidades necesarias para el proyecto, sentido de compañerismo y de ayuda mutua.

7) ÁREA DE TRABAJO INFORMATIVA: al área de trabajo debe tener carteles o pizarras para conocer el estado del proyecto y las tareas a realizar.

8) TRABAJO ENÉRGICO: los programadores deben descansar, para enfocar su trabajo y ser productivos. Por tanto, se limitarán las horas extras para compaginar la vida personal.

9) PROGRAMACIÓN POR PAREJAS: el código siempre está escrito por dos programadores en una única máquina.

– Diseño:

10) DISEÑO INCREMENTAL: XP se opone a un gran diseño completo inicial (*up-front*). El equipo escribe código lo más pronto posible para obtener *feedback* y mejorar el sistema

continuamente. El diseño es indispensable para obtener código bien hecho. La pregunta es cuándo diseñar. XP sugiere hacerlo incrementalmente durante la programación.

11) PROGRAMACIÓN PRIMERO COMPROBAR (TEST-FIRST): antes de actualizar y añadir código, es necesario escribir las pruebas para verificarlo. Esto resuelve cuatro problemas:

- *Cowboy coding* (programador en solitario, o como si lo estuviera): Es fácil dejarse llevar y programar todo lo que se tiene en mente. Las pruebas ayudan a detectar los problemas y demostrar que nuestro diseño es correcto.
- Encaje y unión (*coupling & cohesion*): si no es fácil escribir una prueba, significa que hay un problema de diseño, no de pruebas o codificación. Si el código se adapta bien y es muy cohesivo, se podrá probar fácilmente.
- Confianza: si escribe código que funciona y se documenta con pruebas automatizadas, los compañeros de equipo confiarán en ese programador.
- Ritmo: es fácil pasar horas programando, pero si se acostumbra al ritmo: probar-programar-refactorizar, probar-programar-refactorizar…, no le pasará.

– Programar y lanzar versiones:

12) BUILD DE 10 MINUTOS: El sistema debe construirse y todas las pruebas deben terminarse en diez minutos para ejecutarlo a menudo y obtener *feedback*.

13) INTEGRACIÓN CONTINUA: los programadores deben integrar los cambios cada dos horas para evitar problemas mayores al integrar grandes partes.

2.1.8. Las prácticas secundarias (2004)

– Análisis de requisitos y Planteamiento:

1) CLIENTE INVOLUCRADO: el cliente debe formar parte del equipo aportando *feedback* rápido y contribuyendo a la planificación trimestral y semanal.

2) DESPLIEGUE (O DISTRIBUCIÓN) INCREMENTAL: al reemplazar un sistema antiguo, empiece a reemplazar algunas funcionalidades y gradualmente reemplace todo el sistema. Evitar el enfoque "Todo o nada."

3) CONTRATO DE ALCANCE NEGOCIADO O LIMITADO: es preferible tener una serie de pequeños contratos a uno que englobe tiempo, coste y calidad global final.

4) PAY-PER-USE: normalmente el cliente paga por cada versión del software. Esto crea un conflicto entre el proveedor y el cliente que querría menos versiones, aunque con más funciones nuevas cada vez. Sin embargo, es la forma de obtener la información necesaria para mejorarlo al gusto del cliente y acabarlo antes.

– El equipo y los Factores Humanos:

5) CONTINUIDAD DE EQUIPO: los equipos de desarrollo deben ser los mismos en varios proyectos. La relación que comparten en un proyecto es provechosa y no tienen que dispersarse.

6) EQUIPOS QUE SE REDUCEN: cuando el equipo sea más capaz y productivo, mantenga su carga constante, pero gradualmente reduzca su tamaño y envíe esos miembros libres a formar más equipos.

– Diseño:

7) ANÁLISIS CAUSA-RAÍZ: cada vez que se encuentra un fallo, hay que eliminarlo y eliminar sus causas. Así, también evitaremos volver a tener el mismo error en otro punto.

Programar y lanzar versiones:

8) CÓDIGO Y PRUEBAS: el código y las pruebas son los únicos artefactos y se han de guardar. Los otros documentos pueden generarse a partir del código y las pruebas.

9) CÓDIGO COMPARTIDO: cualquiera del equipo debe poder cambiar cualquier parte de sistema cuando quiera. Esta práctica se llamó "propiedad del código colectiva" en el XP original.

10) BASE DE CÓDIGO ÚNICA: sólo hay una versión oficial de sistema. Se puede desarrollar una rama temporal, pero sólo usarse durante unas horas.

11) DESPLIEGUE DIARIO: cada noche se debe poner nuevo software en producción. Es arriesgado y costoso tener diferentes versiones en producción y en el equipo.

Hemos visto que no se citan algunas prácticas originales como estándares de codificación (considerado obvio) ni la metáfora que era lo más complejo de definir y entender y lo más difícil de llevar a cabo de las doce prácticas originales.

Después de este resumen, podemos analizar la relación entre las 12 prácticas originales y las actuales. De hecho, sólo algunas prácticas se corresponden con las originales. Algunas prácticas extienden las originales y algunas son nuevas. Por ejemplo, el "juego de planificación," se divide en cuatro nuevas prácticas (historias, ciclo semanal, ciclo trimestral y *slack*). El siguiente gráfico relaciona las prácticas antiguas y las nuevas.

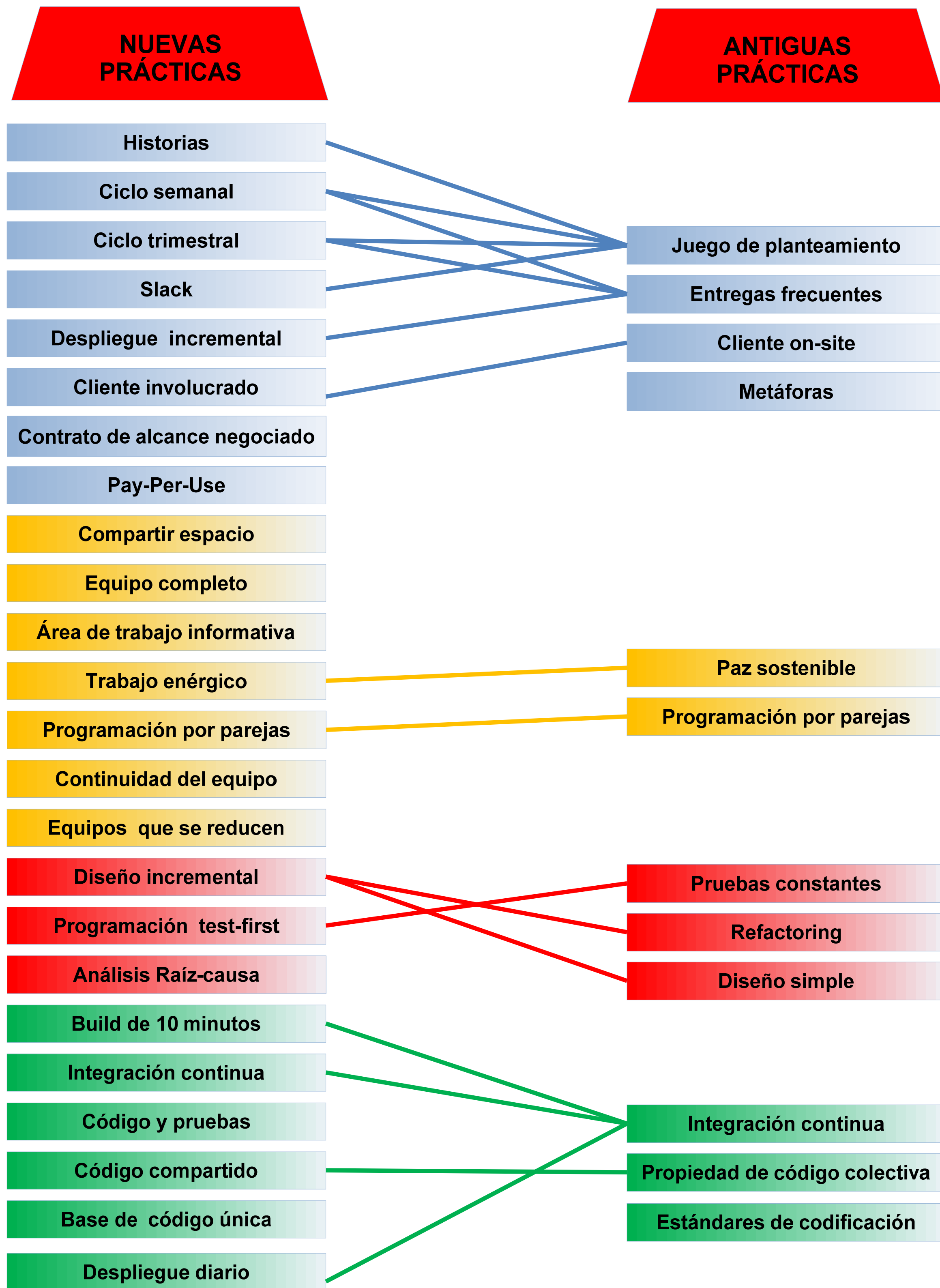


Ilustración 29. Relación entre las viejas y nuevas prácticas de XP.

2.1.9. Lemas de XP

XP utiliza unos lemas que son compartidos por toda la comunidad de desarrolladores:

PREGÚNTELE AL CÓDIGO

Cuando un programador se encuentra con que el código no funciona como quería o no acaba de comprender su comportamiento, pone en práctica este lema. Significa que, en lugar de tratar de comprender mentalmente a qué se debe el comportamiento indeseado del código, debe colocar puntos de interrupción, *breakpoints,* en el código para averiguar qué valores tienen las variables o cuál es el flujo de control que sigue realmente el programa (o escribir código adicional para averiguar qué esta sucediendo).

EL CÓDIGO HUELE MAL (CODE SMELLS)

Este lema es una metáfora olfativa. Decir que un código huele mal es una indicación de que algo va mal en el código. Hay muchos motivos para decir que un código huele mal: exceso de métodos estáticos en una clase, código duplicado, tamaño excesivo de los métodos, clases con demasiado código o con poco código, uso de métodos con el mismo nombre pero con distinto significado, sentencias *catch* vacías, demasiada documentación, antipatrones, mal uso de la herencia...

ESCUCHE AL CÓDIGO

Si le preguntamos al código y nos contesta, lo lógico es que escuchemos sus respuestas. Al menos eso piensan los programadores XP. Cuando un programador XP experimentado escucha al código con problemas, encuentra que el código huele mal (nótese la doble metáfora, porque son metáforas: eschuchar y oler). Los motivos para que el código sea “ruidoso” son pocos: exceso de comentarios monótonos, nombres demasiado largos y ruidos introducidos por el lenguaje de programación empleado (sintaxis adicional como “\n”, “\t”, etc.)

NO VA A NECESITARLO (YOU AREN’T GONNA NEED IT O YAGNI)

Este lema establece que las cosas deben implementarse cuando el programador verdaderamente las necesite, nunca cuando sólo prevea que puede llegar a necesitarlas. Un buen programador de XP nunca se preocupa por las futuras funciones del código, sino por las actuales. Si añade un método a una clase y se da cuenta de que va a necesitar otros métodos, no añadirá los otros métodos si no los necesita en ese momento. Es decir, prefiere trabajar menos ahora, aun a riesgo de trabajar más en el futuro. El motivo que alegan los programadores XP para este lema es que resulta mala idea hacer más cuando uno ya tiene demasiadas cosas que hacer. Los programadores extremos viven el día.

NO LO VA A NECESITAR MÁS (YOU DON’T NEED IT ANYMORE O YDNIA)

Esta regla se usa cuando el programador XP encuentra un trozo de código que nunca ha usado y que no parece necesario. La recomendación es borrarlo inmediatamente. Hay varios motivos para esto. A saber: evitar perder tiempo intentando comprender código que no se usa; evitar el trabajo de depuración del código inservible; evitar la refactorización del código inútil; no aumentar en vano el tiempo que tardarán en ejecutarse las pruebas unitarias y funcionales; e impedir que su refactorización impida una refactorización óptima del código útil.

UNA Y SOLAMENTE UNA VEZ

Este lema resume la idea de que un fragmento de código sólo debe existir en un lugar. La idea, por supuesto, no es nueva. En programación siempre se ha tendido a evitar el código redundante, por lo difícil que resulta mantenerlo. Para evitar esto XP propone la aplicación

del principio abierto/cerrado, refactorización, uso de patrones, rechazo total del "cortar y pegar", etc.

REFACTORICE SIN PIEDAD

Dado que el código quiere ser simple, hay que ayudarlo. Este lema nos recomienda evitar cualquier duplicación del método mediante el uso de la refactorización. Por ejemplo, si un programador extremo encuentra dos métodos que parecen iguales, refactorizará el código para combinarlos en uno. Si encuentra dos objetos con funciones comunes, los refactorizará hasta hacerlos uno.

HAGA LA COSA MÁS SIMPLE QUE POSIBLEMENTE PODRÍA FUNCIONAR

Esta regla significa que cuando se implementa una nueva función del sistema, debe hacerse de la manera más simple que uno piensa que puede funcionar (lo cual incluye sentencias *if*, *switch*, etc.). Las pruebas unitarias ya nos dirán si funciona o no. Si se ejecutan correctamente, el siguiente paso consistirá en aplicar el lema "*Refactorizar sin piedad"* hasta que se cumpla el lema "*Una vez y solamente una vez"*.

EL CÓDIGO FUENTE ES EL DISEÑO

Los desarrolladores extremos explican, por analogía con el ciclo del producto de un producto industrial, que la actividad de **la programación es análoga a la fase de diseño; y que la compilación es equivalente a la fase de manufacturación**. Algunos programadores XP se quejan de que esa metáfora les compara con los obreros de las fábricas de manufacturas. Generalmente, se acepta que la frase expresa la idea de que la actividad de programar es diseñar.

LA REGLA DE ORO

Los seguidores de XP utilizan esta expresión para referirse a la conocida frase *"No haga a los demás lo que no desea que le hagan a usted"*. Otras versiones de esta regla son: *"Quien tiene el oro hace las reglas"* (versión cínica) y *"Hágaselo a los demás antes de que se lo hagan a usted"* (versión anticipatoria). Usan *"Goal donor"* (donante de la meta) y *"Gold owner"* (dueño del oro) para referirse a las personas sobre las que habla la regla de oro, es decir, a los promotores (empresa o institución que solicita el proyecto, que puede coincidir o no con el cliente). *"Goal donor"* designaba en el proyecto C3 a quienes tenían la responsabilidad de fijar las metas y objetivos del proyecto. Asimismo, *"Gold owner"* designaba al promotor del proyecto. Nótese el juego de palabras que se pierde al traducirlo al castellano, ya que *gold* y *goal* se pronuncian casi igual en inglés.

¡MANTÉNGALO SIMPLE, ESTÚPIDO! (KEEP IT SIMPLE, STUPID! O KISS)

En teoría, el código debería ser tan simple que no necesitara comentarios.

2.1.10. Los nuevos principios de XP

En la segunda versión de XP, los principios son el puente entre los valores (sintéticos y abstractos) y las prácticas que dicen cómo desarrollar código. Los 14 principios de XP son:

1) HUMANIDAD: las personas desarrollan el software y por tanto, son la clave principal. Necesitamos encontrar un equilibrio entre las metas de las personas y las de la empresa. Si sobrestimamos las necesidades de personas, no trabajarán adecuadamente, con baja productividad y provocarán pérdidas en la empresa. Si sobrestimamos las necesidades de la empresa, habrá ansiedad, exceso de trabajo y conflictos, lo cual tampoco es bueno para la empresa.

En XP, las necesidades de las personas son:

- Seguridad básica – la necesidad de mantener el puesto de trabajo,
- Realización (logro) – sentirse útiles y valoradas en su propio trabajo,
- Pertenencia a un grupo – identificarse con un grupo,
- Crecimiento – la oportunidad de ampliar sus habilidades y perspectivas,
- Intimidad – la habilidad de entender y ser entiendo por otros.

2) ECONOMÍA: si produce software, también debe producir valor comercial. Dos aspectos de economía son importantes en XP: el valor actual (*present value*) y el valor de las opciones. El primero dice que un dólar hoy es mejor que un dólar mañana: cuanto antes se gane dinero con el software y cuanto más tarde se gaste, más ganancia creará. Esto se une con el valor de las opciones. Si puede diferir las inversiones del plan hasta que sean obvias, es muy beneficioso para el negocio. XP utiliza el diseño incremental, dedicándose a dar valor comercial, según el *feedback* del cliente.

3) BENEFICIO MUTUO: toda actividad debe beneficiar a todas las personas y organizaciones involucradas. Éste es quizás el principio más importante de XP, y el más difícil. Siempre hay soluciones fáciles a cualquier problema donde alguien gana y otros pierden, pero perjudican el ambiente. Se deben solucionar más problemas de los que se crean, beneficiándose usted mismo y al cliente, ahora y en el futuro.

4) AUTO-SIMILITUD (SELF-SIMILARITY): se debe poder reutilizar código para soluciones similares, aunque provengan de contextos diferentes. Por ejemplo, escribir pruebas que fallen, y entonces escribir código que las pase.

5) MEJORA: la perfección no existe, pero debe buscarse. En cada iteración el sistema debe mejorar en calidad y funcionalidades, usando el *feedback* del cliente, las pruebas automatizadas y la experiencia del propio equipo.

6) DIVERSIDAD: equipos donde todos sean iguales son cómodos, pero no es eficaz. Los equipos deben incluir diferentes conocimientos, habilidades y caracteres, para poder averiguar y resolver los problemas. Establecer diferencias, evidentemente crea conflictos que deben resolverse. Las opiniones diferentes son muy útiles.

7) REFLEXIÓN: un equipo eficaz no hace simplemente su trabajo sino que se pregunta cómo está trabajando, y por qué trabaja de esa manera; necesita analizar las razones del éxito –o fracaso– para aprender. Sin embargo, no se debe pensar demasiado ya que sino no se escribiría ningún código. **La reflexión viene detrás de la acción, y antes de la próxima acción**.

8) FLUJO: implica continuidad y firmeza para desarrollar software útil, a partir de pequeños incrementos. Esto permite comprobar si el sistema evoluciona hacia la dirección correcta.

9) OPORTUNIDAD: los problemas deben verse como una oportunidad para aprender y mejorar. Por ejemplo, si no se pueden hacer planes a largo plazo, use ciclos de la planificación más cortos. Si un programador comete demasiados errores, utilice programación por parejas.

10) REDUNDANCIA: los problemas críticos y difíciles deben resolverse de varias maneras diferentes. Así, si una solución falla, las otras evitarán un desastre. El costo de la redundancia se justifica fácilmente en estos casos. Los defectos del software deben buscarse, encontrarse y

arreglarse de muchas maneras (programación por parejas, pruebas automatizadas, participación activa del cliente...). Aunque se encontrarán muchos defectos de forma repetida, se mejorará la calidad.

11) FRACASO: si no puede hacerlo bien, equivóquese. Si no sabe cómo llevar a cabo una historia, intente llevarla a cabo de tres o cuatro maneras diferentes; aunque ninguna sea la ideal, habrá aprendido. **El fracaso es útil si le enseña algo**. Es bueno probar algo y equivocarse en lugar de tardar demasiado en empezar una acción, intentando hacerlo bien a la primera.

12) CALIDAD: siempre debe estar en el máximo. Aceptar una menor calidad no equivale a economizar ni a programar más rápido. La calidad no sólo es un factor económico: los miembros del equipo deben estar orgullosos de su trabajo porque mejora la autoestima y la efectividad. No se debe confundir la calidad con el perfeccionismo: provocar retrasos intentando buscar la perfección es peor que probar y equivocarse.

13) PEQUEÑOS PASOS (BABY STEPS): realizar cambios grandes, preparados durante un período largo de tiempo, es peligroso. Es más adecuado proceder iterativamente en pequeños pasos. Esto no significa ir despacio, sino que cada pequeño incremento se evalúa si va en la dirección correcta y se pierde menos tiempo si hay que retroceder.

14) RESPONSABILIDAD ACEPTADA: la responsabilidad sólo puede aceptarse. Es fácil pedir a programadores “Haga esto”, o “Haga que”, pero esa manera no funciona. Inevitablemente, usted preguntará menos de lo que podría lograrse o, probablemente, más de lo que puede conseguirse. De todas formas, la persona que recibe la orden, decidirá si ser responsable y aceptarla o no.

2.1.11. Adopción y experiencias

Beck sugiere adoptar XP gradualmente: *"Si usted quiere probar XP, por Dios, no intente tragárselo de una vez. Escoja el peor problema en su proceso actual y pruebe a resolverlo como lo haría XP.”* Al lado de los valores y las prácticas, los seguidores de XP usan acrónimos algo extravagantes pero eficaces:

- YAGNI – *You Aren’t Gonna Need It.* No vas a necesitarlo.
- TETCPB – *Test Everything That Can Possibly Broke.* Prueba todo lo que podría fallar.
- DTSTTCPW – *Do The Simplest Thing That Can Possibly Work.* Haz lo más sencillo que pueda funcionar.
- DRY – *Don’t Repeat Yourself.* No te repitas, no dupliques código.
- GoF – *Gang of Four.* Los cuatro autores de *Design Patterns. Elements of Reusable Object-Oriented Software*, E. Gamma, R. Helm, R. Johnson, y J. Vlissides.
- PLOPD – *Pattern Languages of Program Design.*
- POSA – *Pattern Oriented Software Architecture.*
- BUFD – *Big Up Front Design* es el mote peyorativo que se aplica a los grandes diseños preliminares de los métodos en cascada.
- “el Libro Verde” es *Planning Extreme Programming* de Kent Beck y Martin Fowler.
- “el Libro Blanco” es *Extreme Programming Explained* de Kent Beck.

Una de las ideas fundamentales de XP es que cada proyecto es único, y por tanto XP debe adaptarse. Respecto a cuánto pueden adaptarse las prácticas y principios XP para poder todavía hablar de XP, no hay una respuesta precisa. Es típico que no se utilicen todas las prácticas de XP sino sólo las necesarias.

El libro *Extreme Programming Installed* [Jeffries et al. 2001] describe una colección de las técnicas, cubriendo la mayoría de las prácticas de XP, que reutilizaron durante un extenso proyecto de software industrial utilizando XP. Se dedica especial atención a la estimación de la dificultad y duración de las tareas. Los autores sugieren usar *spikes* (púa o astilla) para lograr esto. Una púa es un trozo desechable de código, usado para comprender cómo podría resolverse un problema de programación. Es la versión ágil de la idea de prototipo. Se lo llama así porque "va de punta a punta, pero es muy fino".

[Maurer y Martel 2002] incluyen algunos números con respecto a la productividad de usar XP en un proyecto de desarrollo web: una media de 66.3% más de líneas de código, un 302.1% de aumento en el número de nuevos métodos y un 282.6% más de nuevas clases.

Los obstáculos más comunes surgidos en proyectos XP son la utopía de pretender que el cliente se quede en el sitio y la resistencia de muchos programadores a trabajar en parejas. Craig Larman señala como factores negativos la ausencia de énfasis en la arquitectura durante las primeras iteraciones (no hay arquitectos en XP) y la consiguiente falta de métodos de diseño arquitectónico.

Muchos de los ejemplos de proyectos que han utilizado XP, han adoptado Scrum como método de gestión. En el apartado de Adopción y experiencias de Scrum, 2.2.4, se detallan varios casos de éxito de la aplicación conjunta de XP + Scrum.

2.1.12. Limitaciones

Como afirma el propio Beck, la metodología de XP no es en absoluto adecuada en cualquier proyecto, ni tiene sus límites bien acotados. Sin embargo, se conocen algunos límites.

XP se dirige a equipos pequeños y medianos: entre tres y veinte miembros. El ambiente físico también es importante en XP y debe facilitar la comunicación y coordinación entre los miembros. Tener el equipo trabajando en dos plantas diferentes de un edificio es intolerable para XP. Sin embargo, la dispersión geográfica de los equipos no es necesariamente un problema si interaccionan entre ellos.

También la tecnología podría plantear obstáculos insuperables para que un proyecto triunfase con XP. Por ejemplo, una tecnología muy inflexible a la hora de permitir cambios o que necesitase mucho tiempo de espera para el *feedback*, no sería adecuada para XP.

2.1.13. Investigación actual

Algunas empresas han creado sus propias versiones de XP, como es el caso de *Standard & Poor's*, que ha elaborado plantillas para proyectos futuros. XP se ha combinado con Evo, a pesar que ambos difieren en su política de especificaciones; también es compatible con *Scrum*, al punto que existe una forma híbrida, inventada por Mike Beedle, que se llama *XBreed* [9] y otra bautizada *XP@Scrum* en la que *Scrum* se usa como método de gestión alrededor de prácticas de ingeniería de XP. Se lo ha combinado muchas veces con UP, aunque éste

[9] www.xbreed.net, Mezcla de Scrum y XP creada por Mike Beedle, firmante del manifiesto ágil.

recomienda pasar medio día de discusión de pizarra antes de programar y XP no admite más de 10 o 20 minutos. La diferencia mayor concierne a los casos de uso, que son norma en UP y que XP reemplaza por tarjetas de papel con historias simples que se refieren a características. Sin embargo, en las herramientas de RUP ahora hay plug-ins para XP.

Kent Beck ya no está interesado en las técnicas prácticas de XP (alta productividad, alta flexibilidad, etc.) sino en extender XP a otros ámbitos de dos formas:

1) **Las especialidades**: diseño del interfaz de usuario, diseño de gráficos, consultoría comercial tradicional. Sugiere que como no ha habido ningún cambio en la manera de funcionar de las consultoras, es una buena oportunidad para probar algo nuevo: estrechar la integración entre los asesores comerciales y la validación, con implementaciones. Trabaja en esa dirección con Lance, para crear XC, es decir, *Extreme Consulting*.

2) **Escalar XP**: Kent trabaja con Iona Technologies, proveedor de infraestructuras, para reducir el número de personas de los departamentos de ventas y marketing.

2.2. Scrum

Las primeras referencias al término *Scrum* apuntan a un artículo de Hirotaka Takeuchi e Ikujiro Nonaka, *The New Product Development Game* (1986) en el que se presentaron diversas mejores prácticas de empresas japonesas innovadoras que siempre resultaban ser adaptativas, rápidas y con capacidad de auto-organización. Otra palabra del mismo texto relacionada con modelos japoneses es *Sashimi* (cascada con fases solapadas) inspirado en una estrategia japonesa de desarrollo de hardware utilizada por Fuji-Xerox.

Ilustración 30. Caricatura de una reunión de Scrum.

El término *Scrum* originalmente proviene de una estrategia de rugby, volver a poner en juego un balón perdido, mediante la que todo el equipo coopera y decide rápido la siguiente acción. Como metodología ágil específicamente referida a ingeniería de software, *Scrum* fue aplicado por Jeff Sutherland[10] y elaborada más formalmente por Ken Schwaber [Schwaber 1995]. Poco después Sutherland y Schwaber se unieron para refinar y extender *Scrum*.

En *Scrum* se aplicaron principios de procesos de control industrial, junto con experiencias metodológicas de Microsoft, Borland y Hewlett-Packard. Un libro de referencia es [Schwaber y Beedle 2002]. Los primeros éxitos demostrados de *Scrum* fueron realizados por Schwaber y consistieron en la renovación de un software médico para IDX Corporation o un proyecto muy específico que involucró a 110 personas en Wang Laboratories.

Schwaber fundó una empresa y un método, MATE (*Methods and Toool Expert*), que demostró ser útil para consultorías. Por aquel entonces, las empresas gastaban millones en metodologías, y uno de sus clientes (entre ellos, IBM) les preguntó si podrían hacer unas ampliaciones a MATE para adecuarse mejor a sus necesidades. Poco después, cada cliente quería una versión de MATE personalizada.

Cuando Schwaber estuvo trabajando en DuPont, aprendió que había dos procesos químicos: los *procesos definidos* y los *procesos empíricos*, que requieren constante monitorización (temperatura, presión...) para hacer ajustes. Los empíricos fueron el origen de las reuniones diarias de *Scrum*, ya que el desarrollo de software requiere control y adaptación. Uno de los beneficios instantáneos de estas breves reuniones diarias es que eliminan la necesidad de otras reuniones más largas y generalmente poco productivas. *Scrum* debe verse más como un método de gestión no sólo de software, sino aplicable a otras disciplinas como la consultoría. *Scrum* reduce la gestión de proyectos a su esencia.

Ilustración 31. Ken Schwaber.

[10] http://jeffsutherland.com

Scrum se ha desarrollado para gestionar el proceso de desarrollo de sistemas. Aplica ideas de proceso industrial para controlar el desarrollo de los sistemas. Su enfoque reintroduce las ideas de flexibilidad, adaptabilidad y productividad. No define ninguna técnica de desarrollo de software específica para la fase de aplicación. *Scrum* se centra en cómo deben funcionar los miembros del equipo para que el sistema sea flexible y se adapte a unas condiciones constantemente cambiantes.

En *Scrum*, el desarrollo de los sistemas involucra el entorno y asuntos técnicos (como requisitos, horario, recursos o tecnología) que probablemente cambiarán durante el proceso. Así pues, el proceso es imprevisible y complejo, requiriendo flexibilidad. *Scrum* ayuda a mejorar las tácticas existentes en ingeniería identificando continuamente cualquier deficiencia o impedimento en el proceso, así como con las prácticas que se usan.

2.2.1. Proceso

El proceso de *Scrum* consta de tres fases: *pre-game*, desarrollo o *game* y *post-game.*

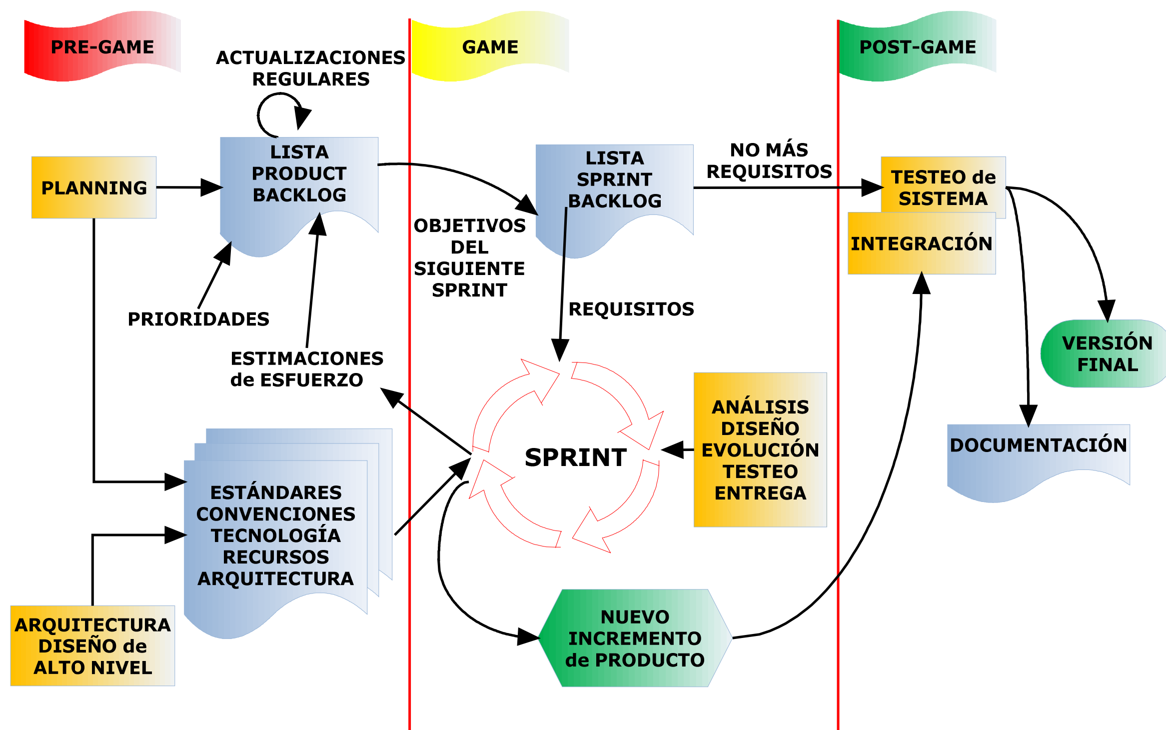


Ilustración 32. Proceso de *Scrum*.

A continuación se detallan las fases de *Scrum* según Schwaber y Beedle. La fase pre-game incluye dos subfases:

PRE-GAME, SUBFASE PLANNING

El planteamiento incluye la definición del sistema a desarrollar y asegurar la financiación. Se crea una lista de acumulación o pila de producto, *Product Backlog list* conteniendo todos los requisitos conocidos hasta el momento. Los requisitos los puede originar el cliente, los programadores o los departamentos de ventas, marketing o atención al cliente. Se priorizan los requisitos y se estima el esfuerzo necesario para su desarrollo. La *Product Backlog list* se actualiza constantemente con puntos nuevos y más detalles, así como con estimaciones más exactas y el nuevo orden de prioridades. El *planning* también incluye la definición del equipo del proyecto, herramientas y otros recursos, la valoración de riesgos, necesidades de

formación y la aprobación de la gestión de comprobaciones. En cada iteración, el equipo(s) de *Scrum* revisa la *Product Backlog list* actualizada. En la jerga de *Scrum*, se llaman *paquetes* a los objetos o componentes que necesitan cambiarse en la siguiente iteración.

PRE-GAME, SUBFASE DISEÑO DE ALTO NIVEL / ARQUITECTURA

En la fase de la arquitectura, se crea el diseño del sistema a alto nivel basándose en los requisitos actuales del *Backlog*. En el caso de una mejora a un sistema ya existente, se identifican los cambios necesarios en el *backlog* así como los problemas que puedan causar. En una reunión se revisa el diseño del plan para cumplir las propuestas. Además, se preparan los contenidos de las versiones que serán lanzadas.

LA FASE DESARROLLO (GAME)

Esta fase, la parte ágil de *Scrum*, se trata como una caja negra donde se espera lo imprevisible. Las diferentes variables técnicas y de entorno que pueden cambiar (calendario, calidad, requisitos, recursos, tecnologías, herramientas, e incluso métodos de desarrollo) se observan y controlan durante los *sprints*. En lugar de considerar estos puntos sólo al principio del proyecto, *Scrum* los controla constantemente para adaptarse a los cambios.

En la fase de desarrollo, el sistema funciona en *sprints* o carreras cortas. Los *sprints* son ciclos iterativos donde se desarrollan o mejoran las funcionalidades para producir los nuevos incrementos. Cada *sprint* incluye las fases habituales de desarrollo del software: requisitos, análisis, diseño, desarrollo y entrega. La arquitectura y el diseño del sistema evolucionan durante el desarrollo del *sprint*. **Un *sprint* dura de una semana a un mes**. Puede haber de tres a ocho *sprints*, por ejemplo, antes de que el sistema esté listo para lanzarse. También puede haber más de un equipo que construya cada incremento.

LA FASE POST-GAME

Contiene el cierre de la versión. Se entra en esta fase cuando se completan todos los requisitos. En este caso, no se incorpora ni mejora ninguna función. El sistema ya está listo para lanzarse (*release*) y ahora se integra, se pone a prueba y se documenta.

2.2.2. Papeles y responsabilidades

Los papeles en *Scrum* tienen tareas y propósitos diferentes durante el proceso y sus prácticas. Son: *Scrum Master*, *Product Owner*, equipo de *Scrum*, Cliente o usuario y Dirección o Gestión. Veamos sus funciones:

1) SCRUM MASTER

El *Scrum Master* es responsable de asegurar que el proyecto se realiza según las prácticas, valores y reglas de *Scrum* y que progresa como estaba previsto. Actúa recíprocamente tanto con el equipo del proyecto como con el cliente. También es responsable de resolver cualquier impedimento para seguir trabajando tan productivamente como sea posible.

2) PROPIETARIO DEL PRODUCTO

El propietario del producto es oficialmente responsable del proyecto, gestionando, controlando, y haciendo visible la *Product backlog list*. Es elegido por el *Scrum Master*, el cliente y la dirección. Toma las últimas decisiones de las tareas relacionadas con la *Product backlog list*, participa estimando el esfuerzo de desarrollo para los puntos del *Backlog* y los concreta en funcionalidades a desarrollar.

3) EQUIPO DE SCRUM

El equipo de *Scrum* tiene autoridad para decidir las acciones pertinentes para organizarse y lograr lo propuesto en cada *sprint*. El equipo de *Scrum* está involucrado, por ejemplo, en estimar el esfuerzo requerido para cada parte, crear y revisar la *Product Backlog list* e identificar problemas a tratar.

4) CLIENTE

El cliente participa en las tareas relacionadas con los puntos del *Backlog* para diseñar o mejorar el sistema.

5) DIRECTOR

La gestión o dirección toma la última decisión y se encarga de los documentos, normas y convenciones seguidas en el proyecto. La dirección también participa en identificar objetivos y requisitos. Por ejemplo, ayuda a seleccionar el *product owner*, valorar los progresos y reducir el *Backlog* con el *Scrum Master*.

2.2.3. La práctica

Scrum no requiere o proporciona ninguna práctica específica para el desarrollo del software. Sin embargo, requiere adoptar ciertas pautas y prácticas para evitar el caos causado por imprevistos y complejidades.

Las pautas de *Scrum* son:

- Equipos auto-dirigidos y auto-organizados. No hay *manager* que decida, sino "*miembros del equipo*" o "*cerdos*"; la excepción es el *Scrum Master* que debe ser 50% programador y que resuelve problemas, pero no manda. Los observadores externos se llaman "*gallinas*"; pueden observar, pero no interferir ni opinar.
- Una vez elegida una tarea, no se agrega trabajo extra. En caso que se agregue algo, se recomienda quitar alguna otra cosa.
- Encuentros diarios con esta tres preguntas indicadas:
    - ¿Qué has hecho desde la última reunión (día anterior)?
    - ¿Has encontrado algún obstáculo?
    - ¿Qué harás para la siguiente reunión (mañana)?

  Se realizan siempre en el mismo lugar, en círculo. La reunión diaria impide caer en el problema señalado por Fred Brooks: *"¿Cómo es que un proyecto puede retrasarse un año? Un día cada vez"*.
- Iteraciones de 30 días; se admite que sean más frecuentes.
- Demostración a participantes externos al final de cada iteración.
- Al principio de cada iteración, planeamiento adaptativo guiado por el cliente.

El nombre de los miembros del equipo y los externos se deriva de una típica fábula agilista:

¿Qué nombre ponemos al restaurante?

"Jamón y huevos"

No gracias. Yo estaré comprometido, pero tú solo involucrada.

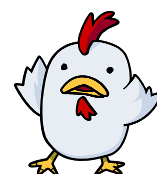


Ilustración 33. Fábula agilista sobre el cerdo y la gallina usados en Scrum.

Vemos que el papel del cerdo es mucho más decisivo porque él ha de poner parte de su cuerpo, mientras que la gallina sólo colabora con huevos. Los cerdos están comprometidos a cumplir los objetivos del *sprint*. Quienes sólo están involucrados y acuden a las reuniones como observadores son las gallinas. Una forma sencilla de saber a qué grupo se pertenece es: *"si te pueden echar por dejar que el proyecto fracase, eres un cerdo; si puedes mantener tu puesto de trabajo incluso si el proyecto fracasa, eres una gallina"*.

A continuación, se describen las prácticas de *Scrum*:

PRODUCT BACKLOG

El *Product Backlog* define todo lo necesario en el producto final, basándose en los conocimientos de ese momento. Por tanto, define el trabajo a hacer en el proyecto. Incluye una lista ordenada por prioridades y actualizada de requisitos técnicos para que el sistema se haga o mejore. Los puntos del *Product Backlog*, por ejemplo, pueden incluir características, funciones, parches para *bugs*, defectos, peticiones de mejoras o actualizaciones de tecnología. También se incluyen temas que requieren solución para poder hacer otros puntos de la lista. A la lista de *backlog* puede contribuir el cliente, el equipo del proyecto y los departamentos de ventas, marketing y atención al cliente.

Esta práctica incluye las tareas para crear la lista de *backlog*, actualizarla agregando, quitando o especificando puntos con sus respectivas prioridades. El *Product Owner* es responsable de mantener el *Product Backlog*.

Product Backlog:	Fecha: Estimado:
Tipo: Nuevo ☐ Mejora ☐ Arreglo ☐	Fuente:
Descripción:	
Notas:	

Ilustración 34. Ficha de Product Backlog utilizada en Scrum.

ESTIMACIÓN DE ESFUERZO

La estimación de esfuerzo es un proceso iterativo donde las estimaciones de cada punto se detallan más a fondo partiendo de la información disponible en cada momento. El *Product Owner* junto con el equipo se encarga de estas estimaciones.

SPRINT

El *sprint* consiste en adaptarse a las condiciones cambiantes del proyecto como requisitos, tiempo, recursos, conocimiento, tecnología, etc. El Equipo de *Scrum* se organiza para producir un nuevo incremento ejecutable en un *sprint* que dura aproximadamente un mes natural. Las herramientas activas del equipo son las reuniones para planear el *sprint*, el *Sprint Backlog* y las reuniones diarias de Scrum.

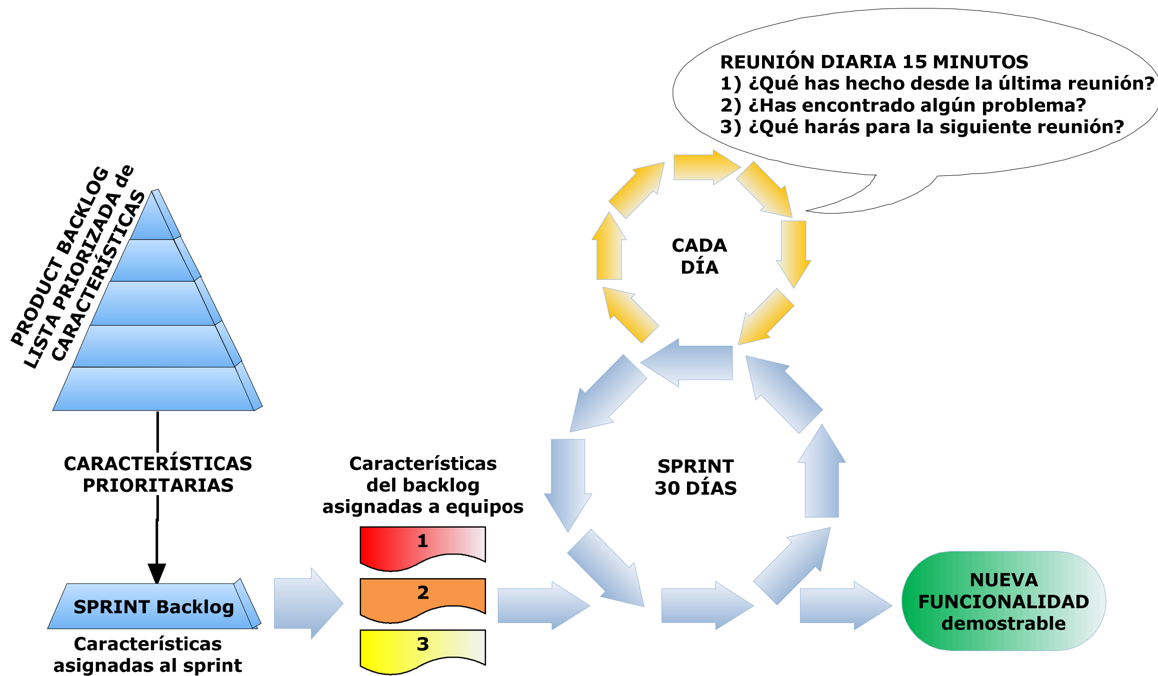


Ilustración 35. Prácticas de un sprint en *Scrum*.

REUNIÓN PARA PLANEAR EL SPRINT

El *Sprint Planning meeting* consta de dos fases y la organiza el *Scrum Master*. En la primera fase, los clientes, usuarios, la dirección, el *Product Owner* y el equipo, eligen los objetivos y las funcionalidades del próximo *sprint*. En la segunda fase, el *Scrum Master* y el equipo concretan la manera de conseguir esos objetivos (*product increment*) en el siguiente *sprint*.

SPRINT BACKLOG

Es el punto de partida para cada *sprint*, una lista de puntos de la lista del *Product Backlog* seleccionados para llevarse a cabo en el próximo *sprint*. El equipo de *Scrum* junto con el *Scrum Master* y el *Product Owner* seleccionan los puntos en la reunión para planear el *Sprint*, basándose en los puntos con prioridad y los objetivos de ese *sprint*. A diferencia del *Product Backlog*, el *Sprint Backlog* no se modifica hasta que el *sprint* (es decir, 30 días) termina. Cuando todos los puntos del *Sprint Backlog* están completos, se prepara una nueva iteración del sistema.

Sprint Backlog:	Fecha: Trabajo pendiente /Fecha:
Estado: Pendiente ☐ Activo ☐ Completo ☐	
Descripción:	
Notas:	

Ilustración 36. Ficha de Sprint Backlog utilizada en Scrum.

El registro incluye los valores que representan las horas de trabajo pendiente; en función de esos valores se acostumbra a elaborar un gráfico de quemado (se pueden ver ejemplos en el apartado de Crystal, 2.3), usados también en muchos otros métodos.

REUNIÓN DIARIA DE SCRUM

Se organizan reuniones de *Scrum* diarias para seguir el progreso del equipo y planear las reuniones: qué se ha hecho desde la última reunión y qué se hará para la siguiente. También se ponen sobre la mesa problemas y otros asuntos que puedan aparecer en esta reunión diaria de unos 15 minutos. Se busca y soluciona cualquier deficiencia o imprevisto del proceso. El *Scrum Master* dirige las reuniones. La dirección (*Management*), también puede colaborar en la reunión.

SPRINT REVIEW MEETING

En el último día del *sprint*, el equipo y el *Scrum Master* presentan los resultados del *sprint*, (el incremento producido) a la dirección, clientes, usuarios y *Product Owner* en una reunión informal. Los participantes evalúan la evolución y deciden sobre las siguientes actividades. La reunión de la revisión puede revelar nuevos puntos e incluso cambiar la dirección tomada del sistema que se está construyendo.

Al final de cada iteración de treinta días hay una demostración a cargo del *Scrum Master*. Las presentaciones en *PowerPoint* están prohibidas. En los encuentros diarios, las *gallinas* deben estar fuera del círculo. Todos tienen que ser puntuales; si alguien llega tarde, se le cobra una multa que se destinará a obras de caridad. Se permite usar *artefactos* de los métodos a los que *Scrum* acompañe, por ejemplo *Listas de Riesgos* si se utiliza UP, *Planguage* si el método es Evo, o los *Planes de Proyecto* sugeridos en la disciplina de Gestión de Proyectos de *Microsoft Solutions Framework*. No se permiten, en cambio, instrumentos como diagramas PERT, porque parten del supuesto de que las tareas de un proyecto se pueden identificar y ordenar.

2.2.4. Adopción y experiencias

Dado que *Scrum* no requiere ninguna práctica específica de la ingeniería, puede ser adoptado para gestionar cualquier práctica. Sin embargo, *Scrum* puede cambiar las descripciones de los puestos de trabajo y las costumbres del equipo considerablemente. Por ejemplo, el jefe de proyecto, *Scrum Master*, ya no organiza el equipo, sino que es el propio equipo quien se organiza y decide qué hacer. Esto se podría llamar un equipo auto-organizativo, como definieron Schwaber y Beedle en 2002. El *Scrum Master* trabaja para solventar los obstáculos del proceso y tomar decisiones en los *Scrums* diarios. Su papel ahora es más parecido al de un entrenador que al de jefe de proyecto.

Distinguimos dos tipos de situaciones donde *Scrum* puede adoptarse: un proyecto existente y un nuevo proyecto:

1) Un caso típico de adoptar *Scrum* en un proyecto ya existente donde el entorno de desarrollo y la tecnología a usar existen, pero el equipo del proyecto está luchando contra problemas debidos a que los requisitos cambian y la tecnología es compleja. En este caso, la introducción de *Scrum* se empieza con *Scrums* diarios con el *Scrum Master*. El objetivo del primer *sprint* debería ser "hacer una demostración de cualquier muestra de funcionalidad para el usuario con la tecnología escogida".

   Esto ayudará al equipo a confiar en si mismo, y al cliente en confiar en el equipo. Durante los *Scrum* diarios del primer *sprint*, se identifican los problemas del proyecto

y se solventan para permitir que progrese el equipo. Al final del primer *sprint*, el cliente y el equipo junto con el *Scrum Master* mantienen una *sprint Review* y deciden el siguiente paso. En caso de continuar con el proyecto, tendrá lugar una reunión para planear el *Sprint* donde se decidirán los objetivos y requisitos del siguiente *sprint*.

2) En caso de adoptar *Scrum* en un nuevo proyecto, es aconsejable trabajar primero con el equipo y el cliente durante varios días para construir el *Product backlog*. En este punto, el *Product Backlog*, puede consistir en *business functionality* y requisitos tecnológicos. El objetivo del primer *sprint* vuelve a ser demostrar una pieza clave de funcionalidad para el usuario en la tecnología elegida. Esto, requiere primero diseñar y construir un *system framework* inicial, es decir una estructura para el sistema a desarrollar, a la que puedan agregarse nuevas características. El *sprint backlog* debería incluir las tareas necesarias para alcanzar el objetivo del *sprint*.

Como el problema principal en este momento involucra la adopción de *Scrum*, el *Sprint Backlog* también incluye las tareas de establecer el equipo y los papeles de *Scrum*, así como las prácticas de gestión además de las tareas propias para llevar a cabo la demostración. Cuando el equipo esté trabajando con el *Sprint Backlog*, el *Product Owner* trabaja con el cliente para construir un *Product backlog* más amplio para poder planear el siguiente *sprint* después de la primera *Sprint Review*.

Existen informes, [Rising y Janof 2000] atestiguando experiencias exitosas con *Scrum*. Afirman que no está pensado para estructuras grandes y complejas, pero si se dividen en pequeños equipos aislados, se pueden utilizar elementos de *Scrum* satisfactoriamente. Las relaciones entre el subequipos deben definirse claramente. En estos casos particulares, siempre utilizaban la reunión diaria y decidieron que de dos a tres veces por semana era suficiente para mantener el *sprint* en el calendario acordado.

Por ejemplo, se han documentado experiencias de hasta 800 miembros, divididos en *Scrums* de *Scrums*, definiendo un equipo central que se encarga de la coordinación, las pruebas cruzadas y la rotación de los miembros. El texto que relata esa experiencia es *Agile Software Development Ecosystems* [Highsmith 2002].

Veamos otros ejemplos del éxito de Scrum:

MICROSOFT DEFIENDE SCRUM

El vicepresidente de la plataforma .NET de Microsoft, David Treadwell, elogió a Scrum como método ágil para la gestión de proyectos. Como se publicó en eWeek y SlashDot el 13-11-2005, Microsoft combinó los modelos de trabajo ágiles Scrum y Extreme Programming para finalizar el lanzamiento de las nuevas versiones: SQL Server 2005, Visual Studio 2005 Tool Suite y Biztalk Server 2006 Integration Server.

SCRUM + XP COMO MARCO PARA EL DESARROLLO DE VIDEOJUEGOS

En una entrevista a Clinton Keith, director técnico de la empresa desarrolladora de videojuegos High Moon Studios, y publicada en GameDAILY el 25-12-2005, Clinton afirma que en el desarrollo de un videojuego no es posible conocer cómo va ser hasta el final del proyecto, y por tanto los modelos ágiles resultan más apropiados que los formales que necesitan requisitos y planificaciones cerradas desde el principio. En un primer momento, introdujo Scrum para el desarrollo del último producto, Darklwatch porque *"era algo que podíamos adoptar rápidamente"*. Más tarde, incorporó también Extreme Programming para

superar problemas de documentación: *"En el pasado empleábamos largas documentaciones técnicas intentando crear el código definitivo a la primera. Desafortunadamente, los documentos no eran demasiado correctos, y los cambios en el código podían crear bastantes problemas al equipo si no empleábamos prácticas XP"*.

Clinton también comenta el cambio de cultura de gestión que implica la adopción de Scrum. Éste es uno de los principales problemas en su implantación: desde el modelo tradicional de gestión tipo PMI, no se comprende la cultura subyacente de mayor autonomía y poder del equipo, y a menudo se ve como una "pérdida de poder" que alimenta la resistencia al cambio de muchos gestores que sólo trabajan con modelos clásicos.

EN EL ICCS2006, SUTHERLAND AFIRMÓ QUE SCRUM ES LA METODOLOGÍA MÁS EFICIENTE

En la *International Conference on Complex Systems* de 2006, http://necsi.org/events/iccs6, Jeff Sutherland afirmó que Scrum es la metodología de desarrollo capaz de lograr récords de productividad y calidad, y que no sólo se puede aplicar en proyectos de pequeña envergadura.

Mostró el estudio *"Adaptive Engineering of Large Software Projects with Distributed / Outsourced Teams"* (versión íntegra en los anexos) que muestra cómo se ha trabajado y qué resultados se han obtenido en el desarrollo de un sistema Java de un millón de líneas de código, realizado por equipos distantes de más de 50 personas de las empresas SirsiDynix y StarSoft Development Laboratories, que han llevado a cabo el proyecto empleando Scrum + Extreme Programming.

Recordemos que el modelo de desarrollo ágil Scrum surgió del análisis de Takeuchi y Nonaka sobre las prácticas de producción de empresas como Honda, Canon, Fuji-Xerox, Brother o Toyota, que, por ejemplo, conseguía multiplicar por 4 la eficiencia y por 12 la calidad respecto a sus competidores.

En la ponencia *Adaptive Engineering of Large Software Projects with Distributed / Outsourced Teams*, Jeff Sutherland, Anton Viktorov y Jack Blout, afirman que la aplicación de Scrum o Scrum + Extreme Programming logran también en el desarrollo de software récords de eficiencia y calidad, y que la afirmación de que se trata de una metodología adecuada para proyectos y equipos pequeños no es cierta.

Los resultados de eficiencia han sido: un equipo de 57 personas en 14.5 meses (827 personas / mes) han desarrollado 671.688 líneas de código de gran calidad (con dos fases de refactorización incluidas). Considerando las medias de productividad que suelen obtener los equipos que trabajan con ciclos de vida en cascada o secuenciales, Scrum multiplica la eficiencia x10, ya que 54 personas/mes habrían obtenido los resultados que según afirma Mike Cohn en su libro *User Stories Applied: For Agile Software Development*, obtiene el desarrollo en cascada con 540 personas/mes.

LA EXPERIENCIA SCRUM EN GOOGLE (DICIEMBRE 2006)

Google ha trabajado siempre con principios de desarrollo ágil: pequeños equipos (tres personas), autogestionados y con elevado nivel de autonomía y capacidad de decisión. Sin embargo el crecimiento de la empresa le plantea retos importantes, porque cada vez son más y mayores los proyectos que tiene en marcha, cuenta con cinco centros de desarrollo repartidos en sedes diferentes, las funcionalidades a implantar se deben traducir a cada idioma, hay que mantener el material de marketing, etc.

Jeff Sutherland trabajó con el equipo del proyecto AdWords de Google para superar estos retos sin abandonar la organización ágil y con la experiencia preparó la charla "*Lessons learned at Google*" disponible en htttp://qcon.infoq.com/qcon/conference. En ella, tras una breve introducción en la que presenta los cuatro principios básicos del manifiesto ágil y un resumen de su charla sobre el origen de Scrum, da un repaso a los retos de crecimiento de Google.

Describe cómo para dar continuidad al propio espíritu ágil de Google, ha adaptado e implantado principios de Scrum en la organización. Cómo ha encajado la figura del propietario del proyecto o producto y del gestor o director de Scrum.

Los principios de Scrum que ha introducido en la forma de trabajar de Google han sido:

- La gestión de los requisitos del sistema (*backlog*), con priorización y estimación de cada funcionalidad.
- La reunión de seguimiento diario.

En la charla, comenta las dificultades iniciales. En el primer caso, los errores de los equipos al realizar estimaciones les llevó a retrasos en las planificaciones de desarrollo. Respecto a la resistencia contra las reuniones diarias, que las consideraban innecesarias, y los problemas habituales de tiempo y divagación.

Las prácticas estándar de Scrum que se han modificado para adaptarlas a las características de los proyectos y de la empresa han sido:

- Uso de un wiki para dar soporte a los requisitos del sistema (*backlog*).
- Considerar como criterio de avance (gráficos de quemado o *burn-down*) la ejecución de tareas en lugar del criterio estándar de Scrum de la ejecución de funcionalidades.

No se trata por tanto de la implantación del Scrum original en Google, sino del enriquecimiento de las prácticas de Google al incorporar y adaptar prácticas de Scrum.

Como Jeff expone en la charla:

Sabes que no estás realizando desarrollo iterativo cuando:

- Las iteraciones son de más de 6 semanas (Ken Schweber en *Scrum Metodology* habla de hasta 8 semanas).
- Las iteraciones no tienen el tiempo acotado.
- El equipo intenta cerrar todas las especificaciones antes de programarlas.
- Las iteraciones no producen código terminado.
- Las iteraciones no incluyen pruebas.

Sabes que no empleas Scrum cuando**:**

- Los requisitos del sistema (*product backlog*) no contienen estimaciones.
- No puedes generar un gráfico de avance del trabajo (*burn-down*) y por lo tanto no conoces la velocidad de avance.
- El equipo no sabe quién es el propietario del producto.
- Hay un gestor de proyecto que está implicado en el trabajo del equipo.

Termina la charla explicando las posibilidades de escalabilidad de Scrum para aplicarlo a equipos grandes que incluso pueden estar geográficamente dispersos, exponiendo brevemente el caso del desarrollo del proyecto ILS de SirsiDynix que presentó en ICCS2006. El video de la charla titulada "*Scrum Tuning: Lessons learned at Google*" también está disponible en http://video.google.es/videoplay?docid=8795214308797356840.

2.2.5. Limitaciones

Scrum es un método adecuado para equipos pequeños, de menos de 10 programadores. Schwaber y Beedle sugieren un equipo de cinco a nueve miembros del proyecto. Si deben trabajar más personas, deben formarse subequipos.

Pueden encontrarse esfuerzos por integrar XP y *Scrum*. *Scrum* proporciona el *Project management framework*, que se sostiene por las prácticas de XP. Uno de los beneficios de la unión es permitir escalar XP a proyectos mayores.

2.3. Familia de métodos Crystal

La familia de métodos *Crystal*[11], definida por Alistair Cockburn (pronunciado "Coburn", a la manera escocesa) en [Cockburn 2002], incluye diferentes métodos para seleccionar el más conveniente según cada proyecto en particular. Además de los métodos, el enfoque de *Crystal* incluye también los principios para adaptarlos a las circunstancias específicas variantes de cada proyecto.

Ilustración 37. Alistair Cockburn.

Cockburn se llama a sí mismo etno-metodólogo, debido a su estudio de las "culturas" de los métodos de desarrollo de software. También ha vivido durante 15 años en ciudades de todo el mundo, lo que le ha hecho observar que a pesar de funcionar de maneras muy diferentes, las empresas de todas las culturas acaban funcionando. Considera que lo más importante es la relación entre las personas: "*Pon de 4 a 7 personas que sean buenos ciudadanos (comportamiento hacia los otros) en una sala y obtendrás software.*" Una de las tácticas que utiliza para trabajar en grupo es interpretar el lenguaje del cuerpo, la comunicación no verbal[12].

Para Cockburn, las habilidades de una persona y las políticas (normas del proyecto dictadas por la organización) van en sentidos contrarios. Las políticas son específicas a un proyecto y no se transfieren. Sin embargo, las habilidades (cómo programar y crear tests, cómo diseñar la interfaz de usuario...) sí que son transferibles, aunque algunas, las que llama *soft skills*, no se pueden enseñar ni aprender.

El autor de los métodos *Crystal* considera una aberración la ingeniería de software, ya que ingeniería y desarrollo de software son muy diferentes. Por eso, se refiere al desarrollo de software como un juego cooperativo de invención y comunicación: todo lo que se hace es inventar y comunicar esa invención. Los programadores no son sólo su clientela, sino también su grupo de estudio para observarlos y aprender.

Los métodos se llaman *Crystal* evocando las facetas de una gema: cada faceta es otra versión del proceso, y todas se sitúan entorno a un núcleo idéntico. Hay cuatro variantes: *Crystal Clear* (Claro o transparente como el cristal) para equipos de 8 o menos integrantes; Amarillo, de 8 a 20; Naranja, de 20 a 50; Rojo, de 50 a 100. En un principio, Cockburn pretendía seguir con *Crystal Maroon* (100 a 200) o *Blue* (200 a 500), pero según dijo su creador, hasta que no esté en un proyecto que los necesite, no los crearía. *Crystal* no soporta equipos distribuidos. La metodología más exhaustivamente documentada es *Crystal Clear* (CC).

Cada miembro de la familia *Crystal* utiliza un color para indicar el peso de la metodología: cuanto más oscuro, más pesado. *Crystal* sugiere elegir el color apropiado para un proyecto basándose en su tamaño y criticidad. Los proyectos más grandes necesitarán más coordinación y metodologías más pesadas. Cuanto más crítico sea el sistema a desarrollar, mayor rigor se necesitará. Los caracteres C, D, E, y L indican las connotaciones que tendría un fallo del sistema (es decir, el nivel de criticidad): ***C****omfort* (C), ***D****iscretionary* money (D), ***E****ssential money* (E) y ***L****ife* (L).

[11] http://alistair.cockburn.us

[12] Por ejemplo, *El lenguaje del cuerpo*, de Allan Pease.

En otras palabras, criticidad C indica que una caída del sistema (*system crash*) debida a fallos, causa una pérdida de *comfort* (comodidad) para el usuario mientras que fallos en un sistema de vida crítico pueden causar literalmente la pérdida de una vida (sistemas de emergencia).

Las dimensiones de criticidad y tamaño se representan con un símbolo que indica la categoría. Por ejemplo, D6 identifica un proyecto con un el máximo de 6 personas que entregan un sistema de criticidad máxima de dinero discrecional.

CRITICIDAD del SISTEMA

L6	L20	L40	L80	L200	L500
E6	E20	E40	E80	E200	E500
D6	D20	D40	D80	D200	D500
C6	C20	C40	C80	C200	C500
TRANSPARENTE	AMARILLO	NARANJA	ROJO	GRANATE	AZUL

TAMAÑO del PROYECTO

Ilustración 38. Dimensiones de las metodologías *Crystal*.

Existen ciertas reglas, características y valores comunes a todos los métodos de la familia *Crystal*. En primer lugar, los proyectos siempre usan ciclos incrementales de desarrollo, con un incremento máximo de 4 meses, pero preferentemente entre uno y tres meses. El énfasis se pone en la comunicación y cooperación de las personas. Los métodos *Crystal* no limitan cualquier práctica de desarrollo o herramienta, y también permiten, por ejemplo, la adopción de prácticas de XP y *Scrum*.

Además, la visión de *Crystal* plasma los objetivos para reducir los pasos intermedios y los desarrolla según evoluciona el proyecto. Las metodologías más comunes son *Crystal Clear*, y *Crystal Orange*. *Crystal Orange Web* no trata con un único proyecto sino con una cadena de iniciativas que luego se juntan.

Los siete valores o propiedades de *Crystal Clear* son:

1) **Entrega frecuente**. Consiste en entregar software a los clientes con frecuencia, no sólo en compilar. La frecuencia dependerá del proyecto, pero puede ser diaria, semanal, mensual, etc.

2) **Comunicación osmótica**. Todos trabajan en la misma sala. El *Cono del Silencio* se describe como una reunión separada para que los asistentes se concentren mejor.

3) **Mejora reflexiva**. Tomarse un pequeño tiempo (desde unas horas hasta alguna semana o una vez al mes) para pensar con profundidad qué se está haciendo, cotejar notas, reflexionar y discutir.

4) **Seguridad personal**. Hablar cuando algo molesta: decirle amigablemente al manager que la agenda no es realista, o a un colega que su código necesita mejorarse. Esto es importante porque el equipo puede descubrir y reparar sus debilidades. No es provechoso encubrir los desacuerdos con gentileza y conciliación.

5) **Foco**. Saber lo que se está haciendo y tener la tranquilidad y el tiempo para hacerlo. Lo primero debe venir de la comunicación sobre dirección y prioridades, típicamente con el *sponsor* o Patrocinador Ejecutivo. Lo segundo, de un ambiente donde la gente no se vea obligada a hacer otras cosas incompatibles.

6) **Fácil acceso a usuarios expertos**. La importancia del contacto directo con expertos está contrastada. Un encuentro semanal o semi-semanal con llamadas telefónicas adicionales suele ser una buena pauta. Otra variante es que los programadores se entrenen para ser usuarios durante un tiempo. El equipo de desarrollo, de todas maneras, incluye un experto comercial.

7) **Ambiente técnico con prueba automatizada, gestión de configuración e integración frecuente**. Microsoft estableció la idea de las *builds* diarias, y no es una mala práctica. Muchos equipos ágiles compilan e integran varias veces al día.

2.3.1. Proceso

Todas las metodologías de *Crystal* proporcionan pautas de política de estándares, *workproducts* o artefactos, "*local matters*", herramientas, estándares y papeles que se deben hacer en el proceso de desarrollo.

Crystal Clear se dirige a proyectos muy pequeños, de categoría D6, máximo seis programadores. Sin embargo, ampliando la comunicación podría aplicarse a proyectos E8/D10. Un equipo que use *Crystal Clear* debería estar en un espacio compartido dadas las limitaciones de la estructura comunicativa.

Crystal Orange se dirige a proyectos medianos, de 10 a 40 miembros (categoría D40), y cuando el proyecto dure de uno a dos años. Los proyectos categoría E50 también pueden utilizar *Crystal Orange* con ampliaciones en los procesos de verificación y pruebas. En *Crystal Orange*, un proyecto se divide para varios equipos con grupos usando la estrategia de *Diversidad Holística* (ver 2.3.3). Sin embargo, este método no soporta un entorno de desarrollo distribuido. *Crystal Orange* enfatiza la importancia del *time-to-maket*, es decir, el tiempo que se tarda hasta que el producto está en el mercado. El equilibrio entre muchas entregas y cambios rápidos en los requisitos y resultados obliga a reducir el número de *deliverables*, pero manteniendo la comunicación eficientemente entre los equipos.

ESTÁNDARES DE POLÍTICA

Estas prácticas deben aplicarse durante el proceso de desarrollo. Tanto *Crystal Clear* como *Crystal Orange* proponen las siguientes normas:

- Entregas incrementales regulares. *Crystal Clear* propone entregas incrementales cada 2 o 3 meses y en *Crystal Orange* pueden ser hasta cada 4 meses. Es la única diferencia entre las normas de política.
- Seguimiento del progreso basado en las entregas de software y decisiones importantes más que en documentos escritos.
- Implicación directa del usuario.

- Test de regresión automatizado, es decir, aplicado después de algún cambio para ver si la resolución ha causado algún fallo en algo que antes funcionaba.
- Dos *user viewings* por versión.
- *Workshops* o talleres para poner a punto el producto y la metodología al principio y en la mitad de cada incremento.

Las normas de política de estas metodologías son obligatorias, pero pueden ser reemplazadas por las prácticas equivalentes usadas en XP o *Scrum*.

WORKPRODUCTS

Los requisitos para los *workproducts* difieren hasta cierto punto, según sea para *Crystal Clear* o *Crystal Orange*. Los puntos comunes para ambos son: secuencia de versiones, modelos de objetos comunes, manual del usuario, casos de test y migración de código.

Además, *Crystal Clear* incluye anotar descripciones de casos, mientras que *Crystal Orange* requiere el documento de requisitos. En *Crystal Clear*, el horario se documenta en las *user viewings* y entregas y el *workproduct* equivalente en *Orange* necesita una planificación más extensa del proyecto.

Los requisitos de documentación son menores en *Clear* (*workproducts* más ligeros):

CRYSTAL CLEAR	- borradores (*screen drafts*) - bocetos del diseño - notas si fueran necesarias
CRYSTAL ORANGE	- documentos con diseño de interfaz de usuario - especificaciones entre equipos - informes de estado

LOCAL MATTERS

Estos "asuntos internos" son procedimientos a aplicar, aunque su realización se deja al propio proyecto. No difieren demasiado para *Clear* y *Orange*: ambas metodologías sostienen que el propio equipo debe establecer y mantener las plantillas para los *workproducts*, así como tests de regresión, y las normas del interfaz de usuario. Por ejemplo, la documentación del proyecto es necesaria, pero su contenido y forma son un *local matter*. Además de esto, las técnicas para los papeles individuales en el proyecto no están definidas por *Crystal Clear* ni *Orange*.

HERRAMIENTAS

Las herramientas requeridas por el método *Crystal Clear* y *Orange* son:

CRYSTAL CLEAR	• compilador • herramienta para gestionar las versiones • herramienta de configuración-gestión • pizarra *(printing whiteboards*)	
CRYSTAL ORANGE	• control de versiones • programación • testeos • comunicación • *drawing*	• seguimiento de proyecto • medición de rendimiento • *screen drivers* para tests de interfaz gráfica de usuario (GUI)

Las pizarras se usan, por ejemplo, para reemplazar los documentos formales y resúmenes de las reuniones. Es decir, se usan para guardar y presentar material que de lo contrario debería escribirse como un documento formal después de cada reunión.

ESTÁNDARES

Crystal Orange propone seleccionar las normas de notación, las convenciones del plan, y las normas de formato y de calidad que se usarán en el proyecto.

ACTIVIDADES

En el gráfico se aprecia una parte del proceso incremental; en la sección 2.3.3 se detallan más.

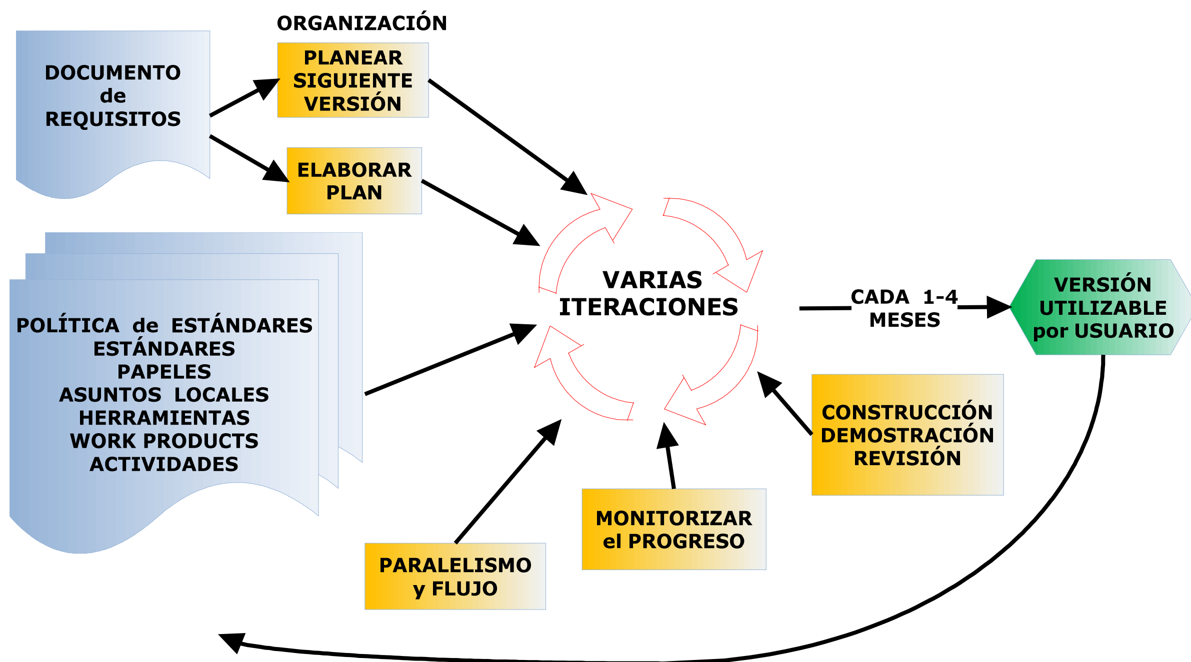


Ilustración 39. Un incremento de Crystal Orange.

Según Cockburn, la mayoría de los modelos de proceso propuestos entre 1970 y 2000 se describían como secuencias de pasos. Aunque se recomendaran iteraciones e incrementos, los modelos parecían dictar un proceso en cascada. El problema con estos procesos es que realmente están describiendo un workflow requerido, un grafo de dependencia: el equipo no puede entregar un sistema hasta que está integrado y funciona. No se puede integrar y verificar hasta que el código no está escrito y funcionando, y no puede diseñar y escribir el código hasta conocer los requisitos. Un grafo de dependencia se interpreta necesariamente en ese sentido, aunque no haya sido la intención original.

En lugar de esta interpretación lineal, CC enfatiza el proceso como un conjunto de ciclos anidados. En la mayoría de los proyectos se perciben siete ciclos:

1) el proyecto,
2) el ciclo de entrega de una unidad,
3) la iteración (múltiples entregas por proyecto, pero pocas iteraciones por entrega),
4) la semana laboral,
5) el período de integración, de 30 minutos a tres días,
6) el día de trabajo, y
7) la etapa de desarrollo de una sección de código, de pocos minutos a pocas horas.

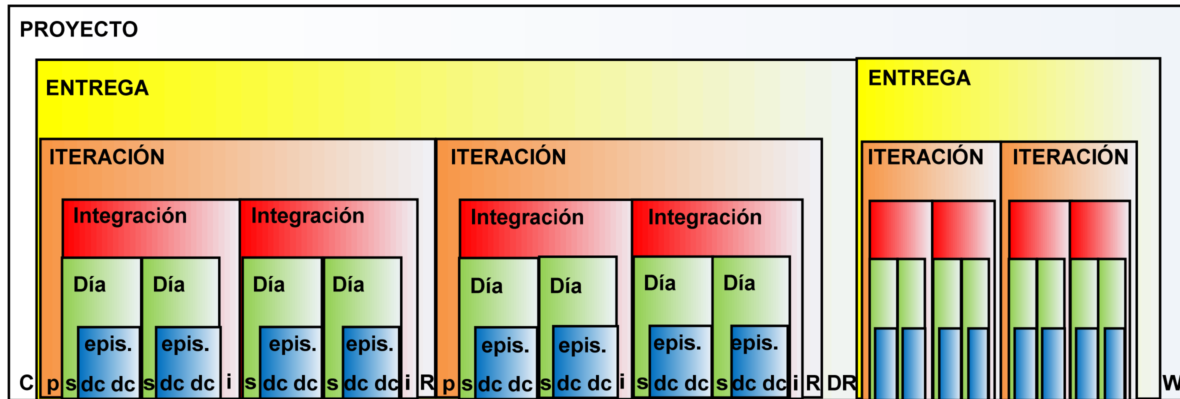


Ilustración 40. Ciclos anidados de Crystal Clear para mostrar las actividades diarias.

Cockburn subraya que interpretar no linealmente un modelo de ciclos es difícil. La figura muestra los ciclos y las actividades conectadas a ellos. Las letras denotan *Chartering*, planeamiento de iteración, reunión diaria de pie (*stand-up*), desarrollo, *check-in*, integración, taller de Reflexión, Entrega (*Delivery*), y empaquetado del proyecto (*Wrap-up*).

Proyecto	Iteración	Día	Integración	Etapa
Charter				
	Plan			
		Reunión diaria de pie		
				Diseñar, check-in Diseñar, check-in
			Construir y testear	
				Diseñar, check-in Diseñar, check-in
			Construir y testear	
		Reunión diaria de pie		
				Diseñar, check-in Diseñar, check-in
			Construir y testear	
				Diseñar, check-in Diseñar, check-in
			Construir y testear	
	Entregar			
	Reflexionar y celebrar			
	Plan			
Cierre				

Tabla 8. Lista de actividades en secuencia vertical cronológica.

2.3.2. Papeles y responsabilidades

La diferencia básica entre *Crystal Clear* y *Orange* es que en el primero sólo hay un equipo en un proyecto mientras que en *Orange*, hay múltiples equipos para un proyecto. En ambas

metodologías, un puesto de trabajo puede incluir varios papeles. En *Crystal Clear* los papeles principales que requieren personas separadas, así como los subpapeles son:

PAPELES PRINCIPALES	Sponsor Diseñador-programador Usuario
SUBPAPELES DE UN DISEÑADOR-PROGRAMADOR	*Business class designer* Programador Documentador de Software Probador de una unidad de clases
OTROS PAPELES	Coordinador Experto comercial Recopilador de requisitos

El experto comercial representa la visión comercial y debe conocer el contexto. Su misión es preocuparse del plan de negocio y estar alerta de qué es estable y qué cambia.

A continuación se describen los artefactos de los que son responsables cada papel en CC:

1. PATROCINADOR

Se encarga de la Declaración de Misión con Prioridades de Compromiso *(Trade-off)*. Consigue los recursos y define la totalidad del proyecto.

2. USUARIO EXPERTO

Junto con el experto comercial, crea la Lista de Actores-Objetivos y el Archivo de Casos de Uso y Requisitos. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación, etc.

3. DISEÑADOR PRINCIPAL

Produce la Descripción Arquitectónica. El Diseñador Principal tiene funciones de coordinador, arquitecto, mentor y programador más experto. Se supone que debe ser al menos un profesional de nivel 3[13].

4. DISEÑADOR-PROGRAMADOR

Junto con el Diseñador Principal se encarga de:

- *Borradores*
- *Modelo Común de Dominio*
- *Notas y Diagramas de Diseño*
- *Código Fuente*
- *Código de Migración*
- *Pruebas*
- *Empaquetado del sistema*

Cockburn no distingue entre diseñadores y programadores. Un programa en CC es “diseño y programa”; sus programadores son diseñadores-programadores.

5. EXPERTO COMERCIAL

Junto con el Usuario Experto crea la Lista de Actores-Objetivos y el Archivo de Casos de Uso y Requisitos. Debe conocer las reglas y políticas del negocio.

---

[13] En los métodos ágiles se definen tres niveles de experiencia. Un programador de Nivel 1 es capaz de seguir los procedimientos; uno de Nivel 2 es capaz de apartarse de los procedimientos específicos y encontrar otros distintos y uno de Nivel 3 es capaz de manejar con fluidez, mezclar e inventar procedimientos.

6. COORDINADOR

Con la ayuda del equipo, crea el Mapa de Proyecto, el Plan de Entrega, el Estado del Proyecto, la Lista de Riesgos, el Plan y Estado de Iteración y la Agenda de Visualización.

7. VERIFICADOR.

Confecciona el informe de *bugs*. Puede ser un programador a tiempo parcial, o un equipo.

8. ESCRITOR TÉCNICO

Elabora el manual del usuario.

Además de los papeles introducidos en *Crystal Clear*, *Crystal Orange* propone una amplia gama de papeles principales, agrupados en varios equipos:

- *Sistema de planificación*
- *Mentores del proyecto*
- *Arquitectura*
- *Tecnología*
- *Funciones*
- *Infraestructura*
- *Equipos externos de test*

Los equipos se dividen en grupos interfuncionales o *cross-functional* con diferentes papeles.

Crystal Orange introduce nuevos papeles:

- *Diseñador de interfaz usuario*
- *Diseñador de base de datos*
- *Ayudante técnico*
- *Analista/diseñador comercial*
- *Usuario experto*
- *Arquitecto*
- *Mentor de diseño*
- *Reutilización (Reuse point)*
- *Escritor*
- *Tester*

Crystal Orange también define las habilidades y técnicas necesarias para tener éxito en estos papeles. Un miembro del proyecto puede tener varios papeles. Por ejemplo, *reuse point e*s un papel de media jornada que puede llevarse a cabo por arquitecto o un diseñador-programador. Este papel se refiere a la identificación de componentes del software reutilizables y adquisición de componentes comerciales. El papel del escritor incluye la construcción de documentación externa, como las especificaciones de interfaz y manual del usuario. El analista-diseñador comercial se comunica y negocia con los usuarios para especificar los requisitos e interfaz, y repasar el diseño.

Cada grupo consiste, por lo menos, en:

- *1 diseñador-analista comercial*
- *1 diseñador de interfaz de usuario*
- *2-3 diseñadores-programadores*
- *1 diseñador de la base de datos*
- *(1 verificador)*
- *(ayudantes técnicos)*

La razón de tener grupos interfuncionales es reducir las entregas y reforzar la comunicación local.

2.3.3. La práctica

Las estrategias más documentadas son:

1) **Exploración de 360°**. Verificar o tomar una muestra del negocio del proyecto, los requisitos, el modelo de dominio, la tecnología, el plan del proyecto y el proceso. La exploración es preliminar al desarrollo y equivale a la fase inicial de RUP. Mientras

que en RUP esto puede tardar algunas semanas o meses, en *Crystal Clear* debe durar unos pocos días; máximo dos semanas si se requiere usar una tecnología nueva o inusual. El muestreo de valor de negocios se puede hacer verbalmente, con casos de uso u otros mecanismos de listas, pero debe resultar en una lista de los casos de uso esenciales del sistema. Respecto a la tecnología, conviene hacer algunos experimentos con *spikes* o púas como definieron Cunningham y Beck.

2) **Victoria temprana**. Es mejor buscar pequeños triunfos iniciales que aspirar a una gran victoria tardía. Usualmente, la primera victoria temprana consiste en la construcción de un Esqueleto Ambulante. Conviene no utilizar la técnica de *"lo peor primero"* de XP, porque puede bajar la moral. La preferencia de Cockburn es ***"lo más fácil primero, lo más difícil segundo"***.

3) **Esqueleto ambulante**. Es un paso que debe ser simple, pero completo. Podría ser una rutina de consulta y actualización en un sistema cliente-servidor, o la ejecución de una transacción en un sistema transaccional de negocios. Un Esqueleto Ambulante no suele ser robusto; sólo camina, y carece de la "carne" o funcionalidad de la aplicación real, que se agregará incrementalmente. Es diferente de una *spike*, porque ésta es "*la más pequeña implementación que demuestra un éxito técnico plausible*" y después se tira, porque puede ser peligrosa; una *spike* sólo se usa para saber si se está en la dirección correcta. El Esqueleto debe producirse con buenos hábitos de producción y pruebas de regresión, y está destinado a crecer con el sistema.

4) **Rearquitectura incremental**. Se ha demostrado que no es conveniente interrumpir el desarrollo para corregir la arquitectura. Más bien, la arquitectura debe evolucionar en etapas, manteniendo el sistema en ejecución mientras se modifica.

5) **Radiadores de información**. Es una lámina pegada en algún lugar que el equipo pueda observar mientras trabaja o *camina*. Tiene que ser comprensible para el observador informal, entendida de un vistazo y renovada periódicamente para que valga la pena visitarla. Puede estar en una página web, pero es mejor si está en una pared. Podría mostrar el conjunto de la iteración actual, el número de pruebas pasadas o pendientes, el número de casos de uso o historias entregadas, el estado de los servidores, los resultados del último Taller de Reflexión... Una variante creativa es un sistema de semáforos implementado por Freeman, Benson y Borning.

En cuanto a las técnicas, se favorecen:

1) **Entrevistas**. Se suele entrevistar a más de un responsable para tener visiones más ricas. Cockburn ha elaborado una plantilla de dos páginas para tal fin. La idea es averiguar cuáles son las prioridades, la lista de características deseadas, los requisitos más críticos y los más negociables. Si se trata de una actualización o corrección, saber cuáles son las cosas a mantener y los errores a evitar.

2) **Talleres de reflexión**. El equipo debe detenerse 30-60 minutos para reflexionar sobre sus convenciones de trabajo, discutir inconvenientes o mejoras y planear el período siguiente. De aquí puede salir material para colgar como Radiador de Información.

3) **Planeamiento Blitz (relámpago)**. Una técnica puede ser el Juego de Planteamiento de XP. En este juego, se ponen tarjetas indexadas en una mesa, con una historia o función

visible en cada una. El grupo finge que no hay dependencias entre tarjetas, y las alinea en secuencias de desarrollo preferidas. Los programadores escriben en cada tarjeta el tiempo estimado para desarrollar cada función. El patrocinador o embajador del usuario escribe la secuencia de prioridades, teniendo en cuenta los tiempos estimados y el valor que aporta cada función al cliente. Las tarjetas se agrupan en períodos de tres semanas llamados iteraciones que se agrupan en entregas o *releases*, usualmente no más largas de tres meses. Pueden usarse tarjetas CRC. Cockburn propone otras variantes del juego, como la ***Jam Session*** o sesión de improvisación, como en jazz. Las diferencias respecto el juego de XP son varias:

- en XP las tarjetas tienen historias; en CC, listas de tareas;
- el juego de XP asume que no hay dependencias; el de CC, que sí las hay;
- en XP hay iteraciones de duración fija; en CC, no se presupone duración.

4) **Estimación Delphi con estimaciones de habilidad**. La técnica se llama así por analogía con el oráculo de Delfos, descrita por primera vez en el clásico *Surviving Object-Oriented Projects* de Cockburn, reputado como uno de los mejores libros sobre el paradigma de objetos. En el proceso Delphi se reúnen los expertos responsables y proceden para proponer el tamaño del sistema, su tiempo de ejecución, la fecha de las entregas según dependencias técnicas y de negocios y para equilibrar las entregas en paquetes de igual tamaño.

5) **Encuentros diarios de pie**. La palabra clave es brevedad, 5-10 minutos como máximo. No se trata de discutir problemas, sino de identificarlos. Los problemas sólo se discuten en otros encuentros posteriores, con quienes tienen que tratarlos. La técnica proviene de *Scrum*. Se deben hacer de pie para que la gente no escriba en sus ordenadores portátiles, garabatee papeles o se quede dormida.

6) **Miniatura de procesos**. La "*Hora Extrema*" fue inventada por Peter Merel para introducir a la gente en XP en 60 minutos y proporciona pautas canónicas que pueden usarse para ponerlo en práctica. Presentar CC puede llevar entre 90 minutos y un día. La idea es que la gente pueda conocer y probar la nueva metodología.

7) **Gráficos de quemado o de avance (burn-down)**. Su nombre viene de los gráficos de quemado de calorías de los regímenes dietéticos; también se usan en otros métodos. Se trata de una técnica para descubrir con antelación demoras y problemas en el proceso. Para ello, se hace una estimación del tiempo restante para programar lo que resta al ritmo actual, lo cual sirve para tener dominio de proyectos cuyas prioridades cambian brusca y frecuentemente. Estos gráficos ilustran la velocidad del proceso, analizando la diferencia entre las líneas proyectadas y efectivas de cada entrega. Existen multitud de tipos de gráficos de este tipo: considerando todo lo que se ha de implementar (pueden aumentar los puntos durante el desarrollo), lo que ya está diseñado, implementado, testeado, etc. Se pueden consultar muchos hechos por Ron Jeffries[14].

[14] www.xprogramming.com/xpmag/BigVisibleCharts.htm

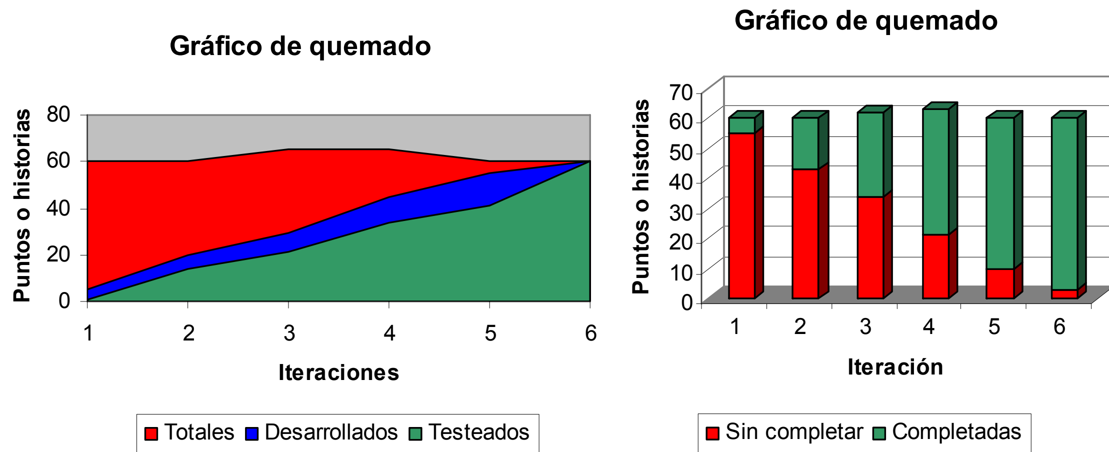


Ilustración 41. Dos ejemplos de gráficos de quemado para mostrar el progreso del proyecto.

8) **Programación en parejas**. Mucha gente siente que la programación en parejas de XP involucra una presión excesiva; la versión de *Crystal Clear* establece proximidad, pero cada cual hace su trabajo asignado, prestando un ojo a lo que hace su compañero, que tiene su propia máquina. Esta es una ampliación de la Comunicación Osmótica al contexto de la programación.

A pesar de que *Crystal Clear* no contempla el desarrollo de software propiamente dicho, involucra unos veinte productos de trabajo o artefactos. Los más destacables son:

1) **Declaración de la misión**. Texto de menos de una página, describiendo el propósito.

2) **Estructura del equipo**. Lista de equipos y miembros.

3) **Metodología**. Comprende papeles, estructura, proceso, productos de trabajo que usan y métodos de revisión.

4) **Secuencia de entrega**. Declaración o diagrama de dependencia; muestra el orden de las entregas y el contenido de cada una.

5) **Cronograma de visualización y entrega**. Lista, plantilla de hoja de cálculo o herramienta de gestión de proyectos.

6) **Lista de riesgos**. Descripción de riesgos por orden descendente de prioridad.

7) **Estado del proyecto**. Lista hitos, fecha prevista, fecha efectiva y comentarios.

8) **Lista de actores-objetivos**. Lista de dos columnas, plantilla de hoja de cálculo, diagrama de caso de uso o similar.

9) **Casos de uso anotados**. Requisitos funcionales.

10) **Archivo de requisitos**. Contiene qué se debe construir, quiénes han de utilizarlo, de qué manera proporciona valor y qué restricciones afectan al diseño.

Todas las metodologías *Crystal* usan varias prácticas, como el desarrollo incremental. En la descripción de *Crystal Orange*, el incremento incluye otras actividades que se describen a continuación:

ORGANIZACIÓN (STAGING)

La organización incluye la planificación del próximo incremento del sistema. Debería programarse para lanzar una versión operativa (*working release*) cada tres o cuatro meses como máximo, pero preferiblemente cada uno a tres meses. El equipo selecciona las funcionalidades a desarrollar y exponen las fechas que consideran factibles.

CORRECCIÓN / MODIFICACIÓN Y REPASO (REVISION & REVIEW)

Cada incremento incluye varias iteraciones que a su vez constan de: construcción, demostración y revisión de los objetivos de la nueva versión.

SUPERVISIÓN (MONITORING)

El progreso se supervisa según las entregas, su progreso y estabilidad. El progreso se controla con hitos o *milestones* (*start, review 1, review 2, test, deliver*) y niveles de estabilidad: muy inestable, estable y suficientemente estable para revisar (*wildly flunctuating, flunctuating y Stable enough to review*). Tanto *Crystal Clear* como *Orange* requieren supervisión.

PARALELISMO Y FLUJO (FLUX)

Cuando el equipo de supervisión concede la estabilidad "*stable enough to review*" a las entregas, ya puede empezar la próxima tarea. En *Crystal Orange*, esto significa que los múltiples equipos pueden continuar con el máximo paralelismo con éxito. Para asegurar esto, los equipos de supervisión y arquitectura revisan sus planes de trabajo, estabilidad y sincronización.

ESTRATEGIA DE DIVERSIDAD HOLÍSTICA

Crystal Orange utiliza este método para dividir grandes equipos funcionales en grupos *cross-functional*. La idea esencial es incluir varias especialidades en un mismo equipo. A su vez, también permite formar equipos pequeños con la habilidad (*know-how*) especial necesaria, y también tiene en consideración asuntos como la ubicación, comunicación, documentación y coordinación de múltiples equipos.

TÉCNICA AJUSTE DEL MÉTODO (METHODOLOGY-TUNING)

Es una de las técnicas básicas de *Crystal Clear* y *Orange*. Propone *Project interviews* y talleres en equipo para elaborar una metodología *Crystal* a medida para cada proyecto en particular. Una de las ideas centrales del desarrollo incremental es permitir arreglar o mejorar el proceso de desarrollo. En cada incremento, el proyecto puede aprender y utilizar estos nuevos conocimientos adquiridos en el próximo incremento.

USER VIEWINGS

En *Crystal Clear* se proponen dos *user viewings* para cada versión. En *Crystal Orange*, las revisiones del usuario (*user review*) deben realizarse tres veces en cada incremento.

TALLERES DE REFLEXIÓN

Tanto *Crystal Clear* como *Orange* incluyen una regla que un equipo debería considerar antes y después de los *increment reflection workshops*. De hecho, la recomendación se extiende también a los talleres intermedios.

Ni *Crystal Clear* ni *Orange* definen prácticas específicas o técnicas a usar por los miembros del proyecto respecto a sus tareas para el desarrollo de software. *Crystal* permite la adopción de prácticas de otras metodologías como XP y *Scrum* para reemplazar algunas de sus propias prácticas, como los talleres de reflexión.

2.3.4. Adopción y experiencias

Uno de los primeros informes referidos a la experiencia de haber utilizado *Crystal* fue para el proyecto *Winifred*, que duró 2 años y estaba formado por 20 - 40 miembros, según el momento. El objetivo era reescribir el obsoleto sistema de un *mainframe*.

El proyecto encontró problemas en el primer incremento. Problemas de comunicación en el propio equipo y entre equipos, una larga fase de requisitos retrasó el diseño de la arquitectura varios meses, muchos movimientos de técnicos y las asignaciones del trabajo poco definidas dieron lugar a *Crystal Orange*. Una solución fue la adopción de un proceso iterativo para cada incremento. Esto proporcionó una oportunidad a los equipos para identificar sus propias debilidades y reorganizarse. También, las iteraciones incluyeron los *functional viewings* con los usuarios para definir los requisitos definitivos. Esto mantuvo a los usuarios constantemente involucrados en el proyecto.

Además, las lecciones aprendidas en el primer incremento animaron a aquellos encargados de definir las responsabilidades de cada uno, métodos de comunicación, la propiedad definida de las entregas y el apoyo de la dirección.

En el proyecto Winifred, equipos diferentes podían tener números diferentes de iteraciones según sus tareas. Para adaptarse a las iteraciones y viceversa, se planeó una comunicación entre equipos, por ejemplo usando entregas. En conclusión, los factores de éxito fueron usar incrementos, ciertos individuos clave, y organizar el equipo a acostumbrarse a las entregas.

2.3.5. Limitaciones

Las metodologías *Crystal* no cubren proyectos *life-critical*, es decir, donde pueda peligrar la vida de alguien. Otra restricción identificada por Cockburn es que sólo los equipos que trabajan en una misma sala (*co-located*) pueden dirigirse con estas metodologías. Cockburn también ha identificado otras limitaciones. Por ejemplo, *Crystal Clear* tiene una estructura de comunicación relativamente restringida y sólo es conveniente para un único equipo localizado en un mismo espacio de la oficina. Además, *Crystal Clear*, no tiene sistemas de validación, por tanto no es conveniente para sistemas *life-critical*.

Crystal Orange también exige localizar sus equipos en un mismo edificio de oficinas y no es aconsejable para sistemas de máxima criticidad, ya que tampoco pone demasiado énfasis en las verificaciones. Tampoco tiene estructuras de subequipos, limitándose a proyectos que involucren hasta 40 personas.

2.4. Feature Driven Development – FDD

FDD, creado por [Palmer y Felsing 2002], es un enfoque ágil y adaptativo para los sistemas en vías de desarrollo. FDD no cubre el proceso de desarrollo de software por completo, sino que se centra en las fases de diseño y construcción. Sin embargo, se ha planteado para trabajar con las otras actividades de un proyecto de desarrollo de software y no requiere ningún modelo de proceso en particular. El enfoque de FDD incluye el desarrollo iterativo con las mejores prácticas eficaces en la industria. Enfatiza los aspectos de calidad a lo largo del proceso e incluye entregas frecuentes y tangibles, junto con una precisa supervisión del progreso del proyecto.

FDD es, además, marca registrada de la empresa Nebulon Pty[15]. Aunque hay coincidencias entre la programación guiada por características y el desarrollo guiado por características, FDD no implementa necesariamente FOP (*Feature Oriented Programming*). FOP es una técnica de programación orientada o guiada por funciones o *features* y centrada en el usuario, no en el programador. Su objetivo es sintetizar un programa conforme a las características requeridas. En términos de FOP, los objetos se organizan en módulos o capas conforme a características.

FDD consiste en cinco procesos secuenciales y proporciona los métodos, técnicas y pautas a seguir por los participantes del proyecto para entregar el sistema. Además, FDD incluye los papeles, artefactos, objetivos, y *timelines* requeridos en un proyecto. A diferencia de otros métodos ágiles, FDD afirma ser adecuado para el desarrollo de sistemas críticos.

FDD fue publicado por primera vez en [Coad et al. 2000]. Posteriormente fue desarrollado por Jeff DeLuca, Peter Coad y Stephen Palmer, basándose en un extenso proyecto real: DeLuca fue contratado para salvar un sistema muy complejo para el cual el contratista anterior había creado, tras dos años, 3.500 páginas de documentación y ninguna línea de código.

2.4.1. El proceso

FDD consiste en cinco procesos secuenciales durante los que se lleva a cabo el diseño y construcción del sistema. La parte iterativa de los procesos FDD (Diseño y Construcción) soporta desarrollo ágil con adaptaciones rápidas a los cambios de última hora en los requisitos. Típicamente, una iteración de una característica implica de una a tres semanas para el equipo.

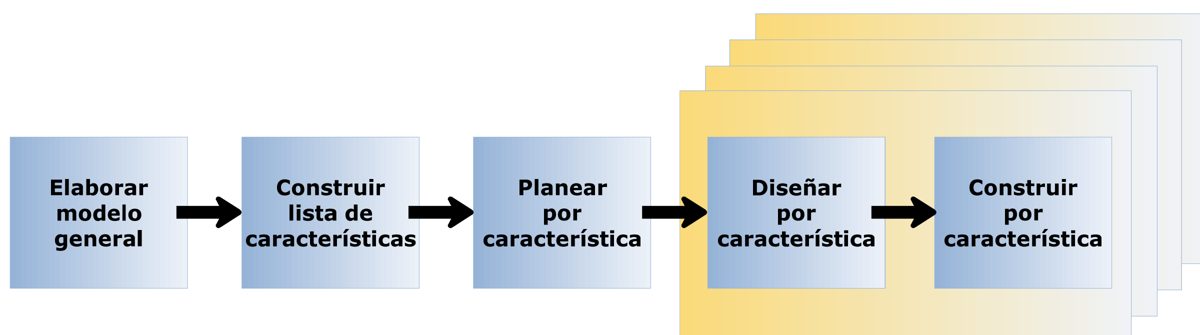


Ilustración 42. Procesos de FDD. Las dos últimas fases son iterativas.

[15] www.nebulon.com/fdd y también www.featuredrivendevelopment.com.

A continuación, se describen los cinco procesos según Palmer y Felsing:

1) ELABORAR UN MODELO GENERAL

Cuando empieza el desarrollo de un modelo global, los *domain experts* (ver 2.4.2) ya son conscientes de las limitaciones, contexto y requisitos del sistema a construir. Es probable que en esta fase, ya exista la documentación de requisitos como los casos o las especificaciones funcionales. Sin embargo, FDD no se dirige explícitamente a recoger y gestionar los requisitos. Los expertos del dominio presentan el llamado *walkthrough* o ensayo donde se informa a los miembros del equipo y al arquitecto jefe (*chief architect*) de la descripción de alto nivel del sistema.

El dominio general se vuelve a dividir en diferentes áreas de dominio y se realiza un ensayo más detallado para cada uno de ellos por los miembros del dominio. Después de cada ensayo, un equipo de desarrollo trabaja en pequeños grupos para producir modelos de objeto para el área del dominio. El equipo de desarrollo entonces discute y elige los modelos apropiados de objeto para cada área del dominio. Simultáneamente, se construye un diagrama del modelo global para el sistema.

2) CONSTRUIR UNA LISTA DE CARACTERÍSTICAS (FEATURES)

Los ensayos, modelos de objeto y la documentación existente de los requisitos son una buena base para construir una lista extensa de características del sistema a desarrollar. En la lista, el equipo de desarrollo presenta cada una de las funciones valoradas por el cliente e incluidas en el sistema. Las funciones se presentan a cada área del dominio y estos grupos de funciones se basan en conjuntos de características principales (*major feature sets*).

Además, los conjuntos de características principales se dividen en conjuntos de características que representan las diferentes actividades dentro de las áreas del dominio. Los usuarios y patrocinadores repasan la lista de características del sistema para validarla y completarla.

3) PLANEAR POR CARACTERÍSTICA

Este proceso incluye la creación de un plan a alto nivel donde los conjuntos de características van en secuencia según su prioridad y dependencias, y se asignan a los programadores jefe (*chief programmers*, ver 2.4.2). Además, se asignan a los diseñadores individuales (propietarios de las clases, ver 2.4.2) las clases identificadas en el proceso de elaborar un modelo general. También pueden establecerse horarios y *milestones* o puntos de referencia para conjuntos de características.

4) DISEÑAR POR CARACTERÍSTICA (ITERATIVA)

Se selecciona un grupo pequeño de características del conjunto de características, *feature set(s),* y los propietarios de las clases forman los *feature teams* necesarios para desarrollar las características seleccionadas. Diseñar y construir por característica son procedimientos iterativos durante los cuales se confeccionan las características seleccionadas.

5) CONSTRUIR POR CARACTERÍSTICA (ITERATIVA)

Una iteración debería durar entre unos días y 2 semanas. Puede haber varios *feature teams* (ver 2.4.2) diseñando concurrentemente y construyendo su propio conjunto de características. Este proceso iterativo incluye inspección del diseño, codificación, comprobar las unidades e integración. Después de una iteración exitosa, las características completadas se incorporan al

programa principal (*main build*) mientras la iteración de diseñar y construir vuelve a empezar con un nuevo grupo de características del conjunto (*set*) de características.

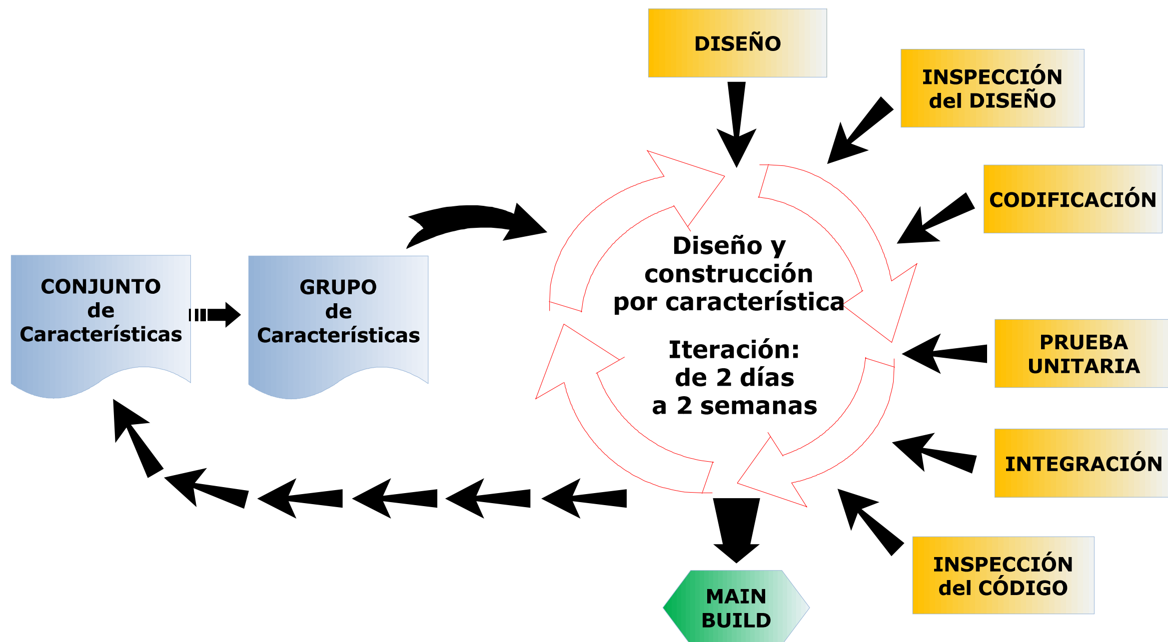


Ilustración 43. Procesos Diseño y Construcción por característica de FDD.

2.4.2. Papeles y responsabilidades

FDD clasifica sus papeles en tres categorías:

PAPELES CLAVE	- *Manager del proyecto* - *Arquitecto jefe* - *Jefe de desarrollo* - *Programador jefe* - *Propietarios de las clases* - *Experto del dominio*
PAPELES DE APOYO	- *Gestor de versiones* - *Experto del lenguaje de programación* - *Ingeniero de las build* - *Toolsmith para herramientas varias* - *Administrador de Sistemas*
PAPELES ADICIONALES	- *Probadores* - *Distribuidores o Desplegadores (deployers)* - *Escritores técnicos*

Un miembro del equipo puede tener varios papeles, y un mismo papel puede ser compartido por varias personas. Los miembros y sus respectivas responsabilidades son:

Manager del Proyecto

El director del proyecto es el líder administrativo y financiero del proyecto. Una de sus tareas es proteger al equipo de distracciones externas y permitirle avanzar proporcionado condiciones favorables. En FDD, el director tiene la última palabra respecto alcance (*scope*), horarios (*schedule*) y personal del proyecto (*staffing*).

ARQUITECTO JEFE

El diseñador principal es responsable del diseño global del sistema y de hacer los talleres con el equipo. También es quien toma las últimas decisiones en todos los temas de diseño. Si fuera necesario, este papel puede ser dividido en dos: arquitecto del dominio y arquitecto técnico.

MANAGER DE DESARROLLO

El director de desarrollo dirige las actividades de desarrollo diarias y resuelve cualquier conflicto que puede ocurrir dentro del equipo, así como problemas de provisión de recursos o *resourcing*. Las funciones de este papel pueden combinarse con las del arquitecto jefe o *Project Manager*.

PROGRAMADOR JEFE

El programador principal es experimentado y participa en el análisis de requisitos y diseño de los proyectos. También es responsable de encabezar los equipos pequeños en el análisis, diseño y desarrollo de nuevas características. Entre sus funciones, deberá:

- Seleccionar las características del conjunto de características a ser desarrolladas en la próxima iteración del diseño y construcción por característica.
- Identificar las clases (y propietarios) que se necesitarán en cada iteración.
- Cooperar con otros programadores resolviendo asuntos técnicos y de provisión de recursos.
- Informar del progreso del equipo semanalmente.

PROPIETARIOS DE LAS CLASES

Los propietarios de las clases trabajan bajo la orientación del programador principal en las tareas de diseño, codificación, testeo y documentación. El propietario de una clase es responsable del desarrollo de la clase que se le ha designado. Los propietarios de las clases forman los ***feature teams***. En cada iteración, participan cuando sus clases están incluidas en la característica que se está programando.

EXPERTO DEL DOMINIO

El *Domain Expert* puede ser un usuario, un cliente, un patrocinador, analista comercial o una mezcla de éstos. Su función es saber cómo deberían funcionar los diferentes requisitos del sistema para explicarlo a los diseñadores y asegurar que entregan un sistema adecuado.

GESTOR DEL DOMINIO

El *Domain Manager* dirige a los expertos del dominio y resuelve sus diferencias de opiniones acerca de los requisitos para el sistema.

GESTOR DE VERSIONES

El *Release Manager* controla el progreso repasando los informes de los programadores principales y manteniendo breves reuniones con ellos. Informa del progreso al jefe del proyecto.

EXPERTO DEL LENGUAJE

El llamado *language lawyer/guru* es un miembro del equipo que debe poseer un conocimiento exhaustivo de un lenguaje de programación específico o tecnología. Este papel es particularmente importante cuando el equipo del proyecto trata con alguna nueva tecnología.

BUILD ENGINEER

Es la persona responsable de preparar, mantener y construir el *build process*, incluyendo la memoria de las versiones y la documentación.

TOOLSMITH

Se encarga de construir pequeñas herramientas para los equipos de desarrollo, testeo y conversión de datos. También, puede preparar y mantener bases de datos y sitios web para propósitos específicos del proyecto.

ADMINISTRADOR DE SISTEMAS

Las tareas de un administrador de sistema comprenden:

- Configurar, gestionar y solucionar problemas de los servidores y la red.
- Desarrollar y probar entornos utilizados por el equipo del proyecto.
- También puede ser involucrado en el *productionizing* del sistema.

TESTER

Los probadores verifican y validan (V&V) que el sistema cumpla las especificaciones y requisitos del cliente. Puede ser un equipo independiente o una parte del equipo del proyecto.

DISTRIBUIDOR O DESPLEGADOR (DEPLOYER)

Su trabajo consiste en convertir los datos existentes al formato requerido por el nuevo sistema y participar en el despliegue de nuevas versiones. Puede ser un equipo independiente o una parte del equipo del proyecto.

ESCRITOR TÉCNICO

Prepara la documentación para el usuario final. Puede ser parte del equipo o independiente.

2.4.3. La práctica

FDD consiste en un conjunto de mejores prácticas, que no son nuevas, pero que la mezcla específica de ellas, hacen que los cinco procesos de FDD sean únicos para cada caso. Palmer y Felsing también defienden que deberían usarse todas las prácticas disponibles para conseguir el mayor beneficio, ya que ninguna práctica domina por completo todo el proceso. FDD involucra las prácticas siguientes:

- ***Modelado de los objetos del dominio***: Exploración y explicación del dominio del problema. Los resultados se engloban en el *framework* donde se agregan las funcionalidades.

- ***Desarrollar por característica***: Desarrollar y seguir el progreso a través de una pequeña lista de funciones importantes para el cliente descompuesta por funcionalidades.

- ***Propiedad individual de la clase (código)***: Cada clase tiene una única persona responsable de la fiabilidad, rendimiento, e integridad conceptual de la clase.

- ***Equipos de características***: Referido a los equipos pequeños y formados dinámicamente.

- ***Inspección***: Referido al uso de los mejores mecanismos para detectar fallos.

- ***Builds regulares***: Asegurar que siempre haya disponible un sistema que funcione para mostrar. Son las bases donde se añadirán nuevas funcionalidades.

- ***Gestión de configuración***: Permite la identificación y control de las últimas versiones de cada fichero de código fuente completado.

- ***Informes de progreso***: Se informa a todos los niveles organizativos necesarios del progreso basándose en partes completadas.

El equipo del proyecto debe poner todas las prácticas anteriores en uso para cumplir con las reglas de FDD. Sin embargo, pueden adaptarlas según su nivel de experiencia.

FDD suministra varios artefactos para la planificación y control de los proyectos. En www.nebulon.com/articles/fdd/fddimplementations.html se encuentran diversos formularios y tablas con información real de implementaciones de FDD: Vistas de desarrollo, Vistas de planificación, Informes de progreso, Informes de tendencia, Vista de plan (<acción><resultado><objeto>), etc. Se han desarrollado también algunas herramientas que generan vistas combinadas o específicas.

Plan - Implementación característica "Autorización" (9)

Identificador	Descripción	Program. jefe	Propiet. clase	Walkthrough		Diseño		Inspección de diseño		Código		Inspección de código		Pasar a construir	
				Plan	Real	Plan	Real	Plan	Real	Plan	Real	Plan	Real	Plan	Real
MD127	Validar los límites transaccionales de un CAO contra una instrucción de implementación	CP	ABC	23 Dic 98	23 Dic 98	31 Ene 99	31 Ene 99	01 Feb 99	01 Feb 99	10 Feb 99		18 Feb 99		20 Feb 99	
MD128	Rechazar una instrucción de implementación para un conjunto de líneas	CP	ABC	STATUS: Inactivo NOTA: [añadido por CK: 3/2/99] Ya no aplica											
MD129	Confirmar una instrucción de implementación para un conjunto de líneas	CP	ABC	23 Dic 98	23 Dic 98	31 Ene 99	31 Ene 99	01 Feb 99	01 Feb 99	10 Feb 99		18 Feb 99		20 Feb 99	
MD130	Determinar si los documentos se han completado para un prestador	CP	ABC	23 Dic 98	23 Dic 98	31 Ene 99	31 Ene 99	01 Feb 99	01 Feb 99	5 Feb 99		8 Feb 99		10 Feb 99	
				NOTA: [agregado por SL: 3/2/99] Bloqueado en AS											
MD131	Validar los límites transaccionales de un CAO contra una instrucción de desembolso	CP	ABC	23 Dic 98	23 Dic 98	31 Ene 99	31 Ene 99	01 Feb 99	01 Feb 99	5 Feb 99		8 Feb 99		10 Feb 99	
				NOTA: [agregado por SL: 3/2/99] Atrasado según estimaciones iniciales.											
Suma de los progresos para estas características en "Autorización": 55% Fecha estimada de finalización: Marzo 1999															

Tabla 9. Ejemplo de un plan de características, con la típica codificación de colores.

2.4.4. Adopción y experiencias

Algunos *agilistas* sienten que FDD es demasiado jerárquico para ser un método ágil, porque demanda un programador jefe, que dirige a los propietarios de clases, quienes dirigen equipos de características. Otros críticos se sienten decepcionados por la ausencia de procedimientos detallados para las pruebas del software. Los promotores del método alegan que las empresas ya tienen implementadas sus herramientas de prueba. **Un rasgo llamativo de FDD es que no exige la presencia del cliente**.

FDD se usó por primera vez para el diseñar una aplicación bancaria extensa y compleja a finales de los 90. Según Palmer y Felsing, FDD es adecuado para nuevos proyectos, mejora de proyectos, actualización del código existente, y proyectos para crear una segunda versión de una aplicación existente. Sus creadores también sugieren adoptar el método gradualmente.

2.4.5. Limitaciones

Los padres de FDD afirman que "*es digno de ser considerado seriamente por cualquier organización de desarrollo de software que necesite entregar sistemas de software de calidad y business-critical a tiempo*", a pesar del poco énfasis en las pruebas constantes.

2.5. Rational / Enterprise / Agile Unified Process – RUP, EUP y AUP

RUP fue desarrollado por Philippe Kruchten, Ivar Jacobsen y otros en la *Rational Corporation* para complementar UML, *Unified Modelling Language* (ver anexos), un estándar en lo referido a métodos para modelar software (*modelling* o *modeling*). Agile UP es la versión ágil o simplificada de RUP y también se analizará. Cronológicamente, 1988: Objectory 1.0, 1998: RUP 5.0, febrero 2004: EUP y septiembre 2005: AUP. RUP es un enfoque iterativo para la programación orientada a objetos (OO), y abarca casos de uso para modelar los requisitos y construir la base de un sistema. Implícitamente no descarta otros métodos, aunque su método propuesto, UML, se adapta muy bien a la programación OO.

2.5.1. Proceso de RUP

La duración de un proyecto de RUP se divide en cuatro fases [Kruchten 2000]: Concepción, Elaboración, Construcción y Transición. Estas fases se dividen en iteraciones, cada una con el propósito de crear un fragmento de software operativo. La duración de una iteración puede variar de dos semanas o menos a seis meses.

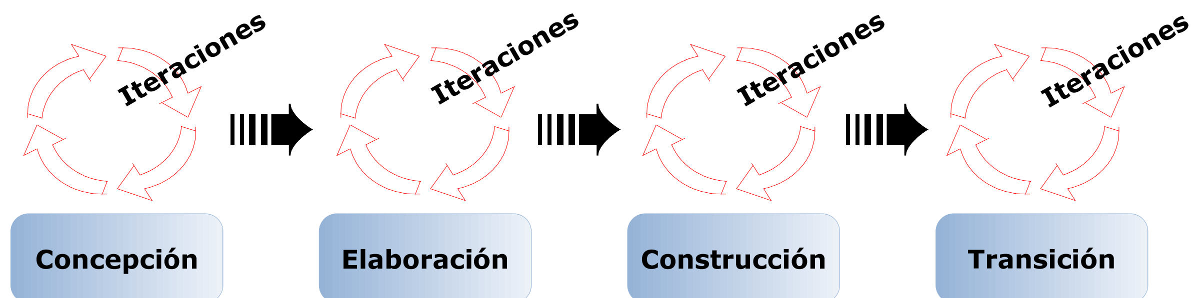


Ilustración 44. Fases de RUP.

A continuación, se detallan las fases de RUP según Kruchten:

1) FASE CONCEPCIÓN (INCEPTION)

En la fase de concepción, se definen los objetivos del ciclo de vida o duración del proyecto para tener en cuenta las necesidades de cada participante, como el usuario final, comprador, o contratista. Esto conlleva, por ejemplo, establecer las posibilidades, límites, y criterio de aceptación del proyecto. Se identifican los casos de uso críticos que determinarán la funcionalidad del sistema. Se diseñan posibles arquitecturas, y se estima el calendario y costo para el proyecto entero. Además, se hacen las estimaciones para la siguiente fase.

2) FASE DE ELABORACIÓN

La fase de la elaboración establece la base de la arquitectura del software. Se analiza el problema o reto, teniendo en cuenta que se construirá el sistema por completo. Se define el plan del proyecto. RUP asume que la fase de la elaboración proporcionará una arquitectura suficientemente sólida y que los requisitos y planes son suficientemente estables. Se describe el proceso y el entorno de desarrollo en detalle. RUP pone énfasis en la automatización de herramientas y también se considera en esta fase. Después de esta fase, se han identificado y descrito la mayoría de los casos y todos los actores, la arquitectura de software está descrita, y se ha creado un prototipo ejecutable de esa arquitectura. Al final de la fase de la elaboración, se contrasta lo que inicialmente fue planeado mediante la realización de riesgos, confirmar la estabilidad de lo que el producto tiene que ser, la estabilidad de la arquitectura, y el gasto de recursos.

3) Fase de Construcción

Aquí se desarrollan los componentes y características restantes y se integran en el producto para pasar las pruebas. RUP considera la fase de construcción un proceso industrial donde se enfatiza gestionar los recursos y controlar los costos, horarios, y calidad. Los resultados de la fase de construcción (versiones alfa, beta, etc.) se crean tan rápidamente como sea posible, pero sin olvidar la calidad. Durante esta fase, se hacen una o más versiones, antes de la siguiente fase.

4) Fase de transición

Se entra en esta fase cuando el software está suficientemente maduro (determinado, por ejemplo, a partir del número e importancia de los cambios pedidos) para ser mostrado a los usuarios. Según su *feedback*, se crearán nuevas versiones para corregir problemas o terminar características pospuestas. La fase de la transición consiste en probar las versiones beta, preparar a los usuarios y gestores de los sistemas, y pasar el producto a los departamentos de marketing, distribución y ventas. Ahora se escribe la documentación.

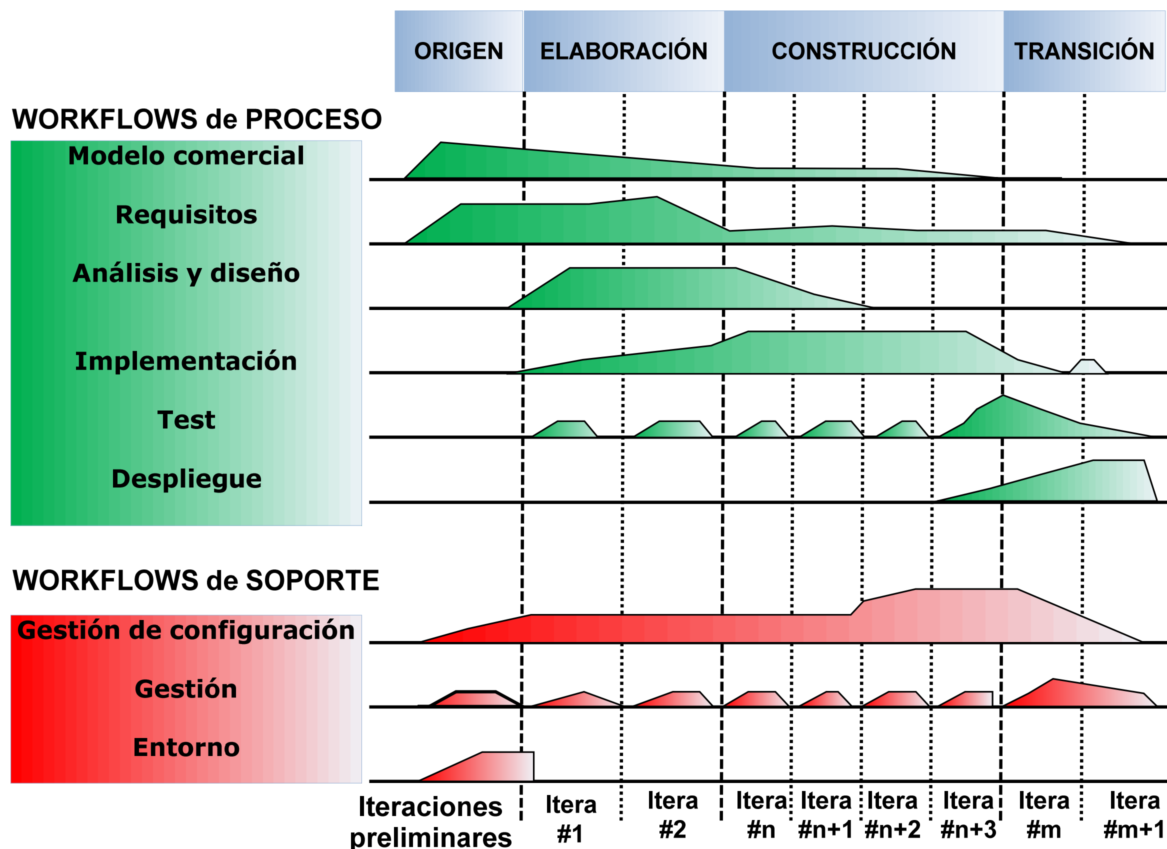


Ilustración 45. Iteraciones de RUP durante el ciclo de vida.

A lo largo de las fases, nueve *workflows* transcurren en paralelo. Cada iteración incluye de forma más o menos profunda, los nueve *workflows*:

- *Business Modelling*
- *Requisitos*
- *Análisis & Diseño*
- *Implementación*
- *Test*
- *Entorno*
- *Gestión de configuraciones*
- *Gestión de cambios*
- *Gestión de proyecto*

Aunque la mayoría tiene nombres autoexplicativos, describimos los dos menos comunes:

El **Business Modelling** se usa para asegurar cumplir con las necesidades comerciales del cliente. Es de utilidad analizar el sector y la forma de trabajar del cliente. Se construye simulando casos y actores para poner a prueba el software. El *business modelling* se hace a menudo en las fases concepción y elaboración, aunque puede ser omitida si no se estima necesaria.

El **Environment** *workflow* (*workflow* es volumen de trabajo) se diseña sólo para apoyar el trabajo de desarrollo. Sus funciones incluyen llevar a cabo y configurar el propio RUP, seleccionar y adquirir las herramientas pertinentes, desarrollar herramientas internas, preocuparse del mantenimiento hardware y software, y de la formación (*training*). Prácticamente todo el trabajo del entorno se hace durante la fase de concepción.

2.5.2. Papeles y responsabilidades de RUP

La manera de RUP de asignar papeles es *activity-driven*, es decir, hay un papel para cada actividad. RUP define 30 papeles, llamados trabajadores. Existen 9 categorías que dan lugar a los 30 papeles de RUP:

– Requisitos (5)	– Test (2)	– Configuración y Gestión de cambios (2)
– Análisis y diseño (6)	– Entorno (3)	– Gestión de proyecto (2)
– Implementación (3)	– Despliegue (4)	– *Business Modeling* (3)

Algunos de los nombres de los puestos son: varios jefes (de configuración, de control de cambios, de despliegue o distribución y de proyecto), participante (*stakeholder*), analista de sistemas, especificador de requisitos, diseñador de interfaz de usuario, arquitecto de software, ingeniero de proceso, especialista de herramientas (*tool specialist*), varios revisores (de diseño, código, requisitos, arquitectura y proyecto), programador, diseñador, implementador, integrador, diseñador de tests y testeador.

En una cooperación continua con los miembros de la organización comercial, un **Business-Process Analyst** lidera y coordina la definición de los casos de uso, que describen los procesos importantes y actores (por ejemplo, clientes) en el negocio de la organización. Su objetivo es crear una abstracción de alto nivel del negocio que se está modelando y también define el *business object model* (describe cómo interactúan procesos y actores) y crea un glosario con la jerga empleada en el negocio.

El modelo *use-case business* se divide en partes, cada una de las cuales se elabora en un caso de uso, *business use case*, por un **Business Designer**. De esta forma, se identifican y documentan los papeles y entidades de la organización.

Un crítico del modelo comercial, *Business-Model Reviewer* repasa todos los artefactos producidos por el analista, *Business-Process Analyst,* y el diseñador comercial, *Business Designer*.

El *workflow* de *Environment* tiene dos papeles:

- **Constructor del curso**: crea el material del curso (diapositivas, guías didácticas, ejemplos...) para los usuarios finales.
- **Toolsmith**: desarrolla las herramientas para facilitar la automatización de tareas tediosas o repetitivas y para mejorar la integración entre las herramientas.

El número de las personas requerido en un proyecto de RUP varía considerablemente, dependiendo de los límites con los que se llevan a cabo los *workflows*. Por ejemplo, el

workflow de *Business model* (junto con sus papeles correspondientes) puede omitirse completamente, si el *business model* no tiene importancia en el producto final.

2.5.3. La práctica de RUP

Las piedras angulares de RUP son seis que quedan bajo las fases de los *workflow* y los papeles de RUP. Las prácticas son:

PRÁCTICA	DESCRIPCIÓN
Desarrollo iterativo	El software se desarrolla en pequeños incrementos e iteraciones breves (permiten identificar riesgos y problemas rápidamente) para reaccionar adecuadamente a ellos.
Gestionar requisitos	- Identificar los requisitos cambiantes y adaptar el software a ellos. - Los requisitos pueden priorizarse, filtrarse y seguirse. - La comunicación funciona mejor con requisitos bien definidos.
Arquitectura basadas en componentes	- La arquitectura es más flexible usando componentes: se aíslan partes más probables de cambiar y se manejan más fácilmente. - Construir componentes reutilizables ahorra tiempo en el futuro.
Software de visualización de modelos	- Se construyen modelos ya que los sistemas complejos son imposibles de entender en su integridad. - Usando un método de visualización común como UML, pueden capturarse la arquitectura del sistema y diseño objetivamente para compartirlo con todos los grupos involucrados.
Verificar la calidad	Verificar la calidad en cada iteración para encontrar fallos pronto, reduciendo el costo si se debieran de arreglar al final.
Consideración de cambios	Cualquier cambio en los requisitos debe tratarse y reflejarse en el software. La madurez del software también puede medirse eficazmente según la frecuencia e importancia de los cambios hechos.

Tabla 10. Las prácticas de RUP.

2.5.4. Adopción y experiencias de RUP

Kruchten defiende que RUP puede adoptarse muchas veces, total o parcialmente, "*out of the box*", es decir, sin modificaciones. En muchos casos, sin embargo, se recomienda una configuración completa (haciendo modificaciones a RUP) antes de implementarlo.

El propio proceso de implementación es un programa iterativo de seis pasos que se repiten hasta que el nuevo proceso ha sido completamente llevado a cabo –que es esencialmente un ciclo *planear-hacer-comprobar-actuar*. Cada incremento aporta nuevas prácticas a desarrollar, y se amolda a las prácticas precedentes. A partir del *feedback* del ciclo anterior, se actualiza el *development case* si fuera necesario. Se recomienda probar primero en un proceso adecuado.

A pesar de la necesidad de una adaptación profunda, RUP se ha implementado en muchas organizaciones. El éxito de RUP también puede ser, en cierta medida, debido a que *Rational Software* vende herramientas populares que apoyan las fases de RUP, por ejemplo, *Rational Rose*, una herramienta de modelado UML o *ClearCase*, para gestionar la configuración del software.

2.5.5. Enterprise Unified Process (EUP)

El *Unified Process* se ha convertido en un estándar del desarrollo de software basado en componentes y orientado a objetos. Sin embargo, las empresas necesitan algo más, un proceso

que refleje el ciclo de vida completo del software. Respecto a RUP, EUP es una extensión que añade dos fases adicionales, *Producción* y *Retirement* o retiro, así como nuevas disciplinas: Soporte y Operaciones y varias disciplinas de empresa. Es habitual primero adoptar RUP y después añadir las extensiones de EUP.

FASE DE PRODUCCIÓN

EUP añade esta quinta fase al *Unified Process*, para representar la parte del ciclo de vida después de que el sistema haya sido desplegado. Su propósito es mantener el software en producción hasta que sea reemplazado por una nueva versión, desde una versión menor que corrija algunos *bugs*, hasta una versión renovada, o sea retirado y eliminado de la producción.

FASE DE RETIRO

Esta sexta fase de EUP, representa la parte del ciclo de vida donde el sistema se elimina totalmente de la producción. Al retiro a veces se le llama *dispensation* (exención) o *disposal* (deshacerse). Las actividades para retirar un software se suelen centrar en convertir y archivar datos, gestionar la configuración de los componentes software, y en asegurarse de que su retirada no tendrá efectos colaterales en otros sistemas. Habitualmente los sistemas se retiran cuando su sucesor ya está funcionando.

DISCIPLINA SOPORTE Y OPERACIONES

Esta disciplina está destinada a operar y dar soporte al software y requiere procesos definidos. Al igual que muchas de las otras fases, también ocupa varias. En la fase de Construcción, y quizás tan pronto como en la fase de Elaboración, hará falta diseñar un plan de Soporte y Operaciones, documentos y manuales de formación. Durante la fase de Transición, se seguirán mejorando estos artefactos con la experiencia adquirida.

DISCIPLINAS DE EMPRESA

Las *core enterprise disciplines* son siete: *Enterprise Administration, Enterprise Architecture, Enterprise Business Modeling, People Management, Portfolio Management, Software Process Improvement (SPI),* y *Strategic Reuse*. Se centran en asuntos que implican todo el proyecto y que la mayoría de las organizaciones afronta a diario. Estas disciplinas engloban las actividades requeridas para extender el *Unified Process* y convertirlo en un ciclo de vida de un sistema de IT completo en vez de sólo un proceso de desarrollo. El trabajo más significativo ocurre antes de la fase De concepción (*inception*).

En los anexos se detallan más en profundidad las disciplinas.

2.5.6. Agile Unified Process (AgileUP, AUP o dX)

Robert Martin, presentó dX, una versión mínima de RUP (ahora llamada Agile RUP) donde se tienen en cuenta los puntos de vista del desarrollo de software ágiles. dX imita deliberadamente los principios de XP, pero adaptando las actividades de RUP, aparte de ser un pequeño cambio, pues “dX” es “XP” boca abajo. Agile Modelling (AM) propuesto por S. Ambler en 2002, es un nuevo enfoque, donde RUP puede utilizarse para las fases *Business Modelling*, definir requisitos, análisis y planteamiento.

La simplificación basada en el Rational Unified Process (RUP) de IBM, AUP, actualmente está en su versión 1.1. El ciclo de vida de AgileUp tiene parte de comportamiento en cascada a grosso modo, pero es iterativo en sus pasos y proporciona versiones incrementales.

PRINCIPIOS DE AGILE UP

1) Los empleados saben lo que hacen, conocen a alto nivel el proyecto.
2) Simplicidad y documentación concisa.
3) Agilidad, seguimiento de los principios ágiles.
4) Centrarse en actividades que proporcionen valor al cliente.
5) Independencia de herramientas software usadas.
6) Adaptar el producto a las necesidades.

FASES

Agile UP se caracteriza por ser "*serial in the large*", es decir, comportamiento en cascada o serie en su conjunto (aunque partes son iterativas), como se desprende de sus cuatro fases.

Fase	Objetivos	Milestone
1. Concepción	Identificar el alcance inicial del proyecto, una arquitectura principal para el sistema, y obtener los fundamentos del proyecto inicial y la aceptación de los participantes.	Objetivos del Ciclo de Vida (LCO)
2. Elaboración	Comprobar la arquitectura del sistema.	Arquitectura de Ciclo de Vida (LCA)
3. Construcción	Construir software que funciona a base de incrementos regulares, primero ocupándose de las necesidades de mayor importancia para los participantes.	Capacidad Operacional Inicial (IOC)
4. Transición	Validar y desplegar el sistema en el entorno de producción.	Versión de Producto (PR)

Tabla 11. Fases y milestones de AUP.

DISCIPLINAS DE AGILE UP

Las disciplinas o prácticas se ejecutan de forma iterativa, definiendo las actividades que los miembros del equipo realizan para construir, validar y entregar software que funcione y cumpla los requisitos de los participantes.

Disciplina	Objetivos
Modelo	Entender el negocio de la organización, el dominio del problema que ataca el proyecto, e identificar una solución viable.
Implementación	Transformar el modelo en código ejecutable y realizar un testeo básico, en particular, tests unitarios.
Test	Realizar una evaluación objetiva para asegurar la calidad. Incluye encontrar defectos, validar que funciona como se preveía, y que cumple los requisitos.
Despliegue	Planear la entrega del sistema y ejecutar el plan para que los usuarios finales dispongan del sistema.
Gestión de configuración	Gestionar el acceso a los *workproducts* del proyecto. Esto incluye controlar las versiones de los *workproducts* y sus cambios.
Gestión de proyecto	Dirigir las actividades del proyecto: gestión de riesgos, asignar tareas, hacer seguimientos, coordinar y asegurar que se cumple el calendario y el presupuesto.
Entorno	Respaldar el resto del esfuerzo asegurando que el equipo dispone de las guías, estándares y herramientas soft y hard cuando las necesite.

Tabla 12. Disciplinas de AgileUP.

MILESTONES

Existen cuatro puntos o *milestones* en AUP que señalan el fin de una fase. En cada uno, se debería considerar hacer una revisión para verificar que el equipo ha cumplido con éxito el criterio del *milestone*. Los *milestones* son:

1) Milestone de la fase Concepción: Objetivos del ciclo de vida (LCO)

Los participantes evalúan el estado del proyecto y deben estar de acuerdo en:

- Acuerdo del alcance.
- Definición inicial de requisitos.
- Acuerdo del plan: coste inicial y estimación de calendario.
- Identificación y aceptación de riegos.
- Aceptación del proceso por parte de todos los participantes.
- Viabilidad comercial, técnica y operacional.
- Plan de proyecto adecuado para la siguiente fase, elaboración.
- Acatamiento de los planes (*portfolio*) de la organización.

2) Milestone de la fase Elaboración: Arquitectura del ciclo de vida (LCA)

Se evalúa el estado del proyecto y los participantes deben estar de acuerdo en:

- Visión estable y realista del proyecto.
- Arquitectura estable y suficiente para satisfacer los requisitos: se han hecho prototipos de la arquitectura en puntos críticos para afrontar los mayores riesgos.
- Aceptación de riesgos: se han documentado junto con las estrategias a seguir.
- Viabilidad: el proyecto todavía se ve viable en todos los sentidos.
- Planes detallados para las siguientes iteraciones inmediatas de construcción.
- La arquitectura refleja la realidad de empresa.

3) Milestone de la fase Construcción: Capacidad de operación inicial (IOC)

En este punto, los participantes deben estar de acuerdo en:

- Estabilidad del sistema: el software y la documentación de soporte son aceptables (estables y maduras) para entregar el sistema a los usuarios.
- Participantes preparados para el sistema, quizá todavía formándose.
- Aceptación de riesgos: se han evaluado y documentado los riesgos y las estrategias para vencerlos son aceptables.
- Aceptación de coste y estimaciones de tiempo.
- Planes detallados para las siguientes pocas iteraciones de transición.
- Cumplimiento con la empresa: los *workproducts* deben ser los esperados.

4) Milestone de la fase Transición: Versión del producto (PR)

En este *milestone*, los participantes deben estar de acuerdo en:

- Aceptación de los *Business stakeholders*: están satisfechos y aceptan el sistema.
- Aceptación del equipo de operaciones y soporte: están conformes con los procedimientos y la documentación.
- Aceptación de coste y estimación de tiempo para futuras ampliaciones.

PAPELES

Consideraciones sobre los papeles o *roles*:

1) Un papel puede llevarlo más de una persona.
2) Una misma persona puede tomar varios papeles.
3) Un papel no es una posición.

4) Se debería luchar para llegar a ser un "especialista generalizador", *generalizing specialist*, que tiene más de una especialidad, un conocimiento general de todo el proceso y una buena comprensión del dominio en el que se trabaja.

Papel	Descripción	Disciplina(s)
DBA ágil	Un administrador de base de datos (DBA) colabora con el equipo para diseñar, testear, desarrollar y mantener las bases de datos.	Implementación
Modelador ágil	Crea y desarrolla modelos, ya sean esbozos, tarjetas como en XP o complejos archivos de herramientas CASE, de forma colaborativa. Los modelos ágiles son sólo apenas lo suficientemente buenos.	Modelado Implementación
Cualquiera	Una persona con cualquier otro papel.	Gestión de configuración Gestión de proyecto
Manager de configuración	Es responsable de proporcionar estructura de CM global y el entorno al equipo de desarrollo.	Gestión de configuración
Desplegador	Despliega el sistema a los entornos de pre-producción y producción.	Despliegue
Desarrollador	Desarrolla, escribe, testea y construye software.	Modelado Implementación Despliegue
Ingeniero de proceso	Desarrolla, adapta y mantiene los materiales necesarios para el software (descripciones de proceso, plantillas, consejos, ejemplos)	Entorno
Manager de proyecto	Dirige el equipo, lo protege, crea relaciones con los participantes y coordina interacciones con ellos, planea, dirige y asigna recursos, establece prioridades, y mantiene al equipo centrado.	Modelado Test Despliegue Gestión de proyecto
Revisor	Evalúa los *workproducts* del proyecto, a menudo "*works in progress*", proporcionando *feedback*.	Test
Participante (stakeholder)	Alguien que es un usuario directo o indirecto, jefe de usuarios, manager senior, miembro del personal de operaciones, soporte (*help desk*), desarrolladores trabajando en otros sistemas que integran o interactúan con el que se está desarrollando, o profesionales del mantenimiento potencialmente afectados por el desarrollo y/o despliegue del proyecto.	Modelado Implementación Test Despliegue Gestión de proyecto
Escritor técnico	Responsables de confeccionar la documentación para los participantes, como material para cursos, documentación para soporte y operaciones y usuarios finales.	Despliegue
Manager de tests	Responsable de planear, gestionar y defender las actividades de testeo y calidad.	Test
Tester	Escribe, dirige y registra los resultados de los tests.	Test
Especialista de herramientas	Responsable de seleccionar, adquirir, configurar y mantener herramientas.	Entorno

Tabla 13. Descripción de los papeles y disciplinas donde actúan.

ENTREGAS O DELIVERABLES

Agile UP distingue entre:

- *Deliverable*, el cual es imprescindible de hacer de forma permanente como un *workproduct* del sistema.
- Otros *workproducts* del proyecto que después se descartarán.
- *Workproducts* de la empresa que los mantiene el departamento de IT y los comparte entre varios proyectos.

ENTREGAS IMPRESCINDIBLES

Los mínimos *deliverables* para Agile UP, ordenados por prioridad son:

Deliverable	Descripción
Sistema	El software y hardware funcionando y la documentación para ser desplegado en producción.
Código fuente	El código del programa, que debe seguir unos estándares.
Conjunto de tests de regresión	Una colección de *test cases*, y el código para ejecutarlos en el orden correspondiente. Se deben incluir tests de aceptación, unitarios, de sistema, etc. Es recomendable automatizarlos y ejecutarlos ante cualquier cambio.
Scripts de instalación	Código para instalar el sistema en el entorno de (pre)producción. También serán útiles scripts de desinstalación por si falla la instalación.
Documentación del sistema	Ayudará a los participantes a trabajar con el sistema y a los desarrolladores a mejorarlo. Comprende documentación general, de operaciones, soporte y usuario. Debe ser tan concisa (*lean*) como se pueda.
Notas de la versión	Deberían resumir los puntos más importantes de la versión actual.
Modelo de requisitos	Debe describir los requisitos que el modelo debe cumplir. Comprende varios *workproducts*, incluyendo tests de aceptación, oportunidades de automatización, modelo de proceso de negocio, normas del negocio, modelo de la organización, glosario del proyecto, requisitos técnicos, modelo de casos de uso y modelo de interfaz de usuario.
Modelo de diseño	Describe el diseño del proyecto, y debe ser tan simple como se pueda. Incluyen varios *workproducts*, un documento resumen del sistema y varios modelos: de despliegue, de objeto, de datos físicos (PDM), de amenazas de seguridad, y de interfaz de usuario. El mejor sitio para documentar el diseño son los tests unitarios y el código fuente.

Tabla 14. Deliverables mínimos para Agile UP.

OTROS *WORKPRODUCTS* DEL PROYECTO

Otros artefactos, incluyen:

Workproduct	Descripción
Tests de aceptación	Describen los requisitos según los participantes.
Oportunidades de automatización	Una lista de actividades que se hacen de forma manual y repetitiva.
Presupuesto	Cantidad de fondos, cuándo se recibirán, condiciones...

Modelo de proceso del negocio	Descripción de las actividades del negocio, cómo se mueve la información. Los DataFlow Diagrams (DFD) y UML permiten visualizarlos fácilmente.
Especificaciones de las reglas del negocio	Aparecen las reglas que implementa el sistema. Una regla de negocios (*business rule*) define o limita un aspecto del negocio que se pretendía imponer. A menudo se centran en problemas de control de acceso, cálculos comerciales o política de la organización.
Esquema de datos	Es el esquema del origen de los datos. En el caso de bases de datos relacionales, se describe mediante *Data Definition Language* (DDL); para orígenes de datos tipo XML, se utiliza la definición de esquemas para XML o DTD para XML. Los esquemas de datos evolucionarán junto con el código.
Informe de defectos	Un tipo de petición de cambio que define un problema del sistema. Puede ser un e-mail o un fichero de hoja de cálculo. Es recomendable usar software para seguir los *bugs*, especialmente en la fase de transición y una vez el sistema esté desplegado en producción.
Modelo de despliegue	Describe cómo organizar los aspectos de hardware, middleware y software del sistema. Deben ser simples y suele usarse UML, un diagrama de red o esbozos.
Plan de desarrollo	Describe el planteamiento para desplegar el sistema en producción. Es útil conocer las dependencias con otros grupos, identificar el alcance de la versión, trabajar con la gente de soporte y operaciones, prever la formación, etc.
Modelo del dominio	Representa las principales entidades del negocio, sus relaciones y responsabilidades. Se empieza creando un modelo “fino” del dominio que debería conducir el desarrollo del modelo de objeto y esquema de datos.
Estimación	Coste previsto para completar el proyecto (se corrige durante el proceso).
Estimación de individuos	Una estimación del tiempo y/o coste, hecha por un individuo.
Modelo de objeto	Describe el esquema de objeto, el código que constituye el software. Para la estructura estática, son ideales los diagramas de clases, componentes y paquetes de UML. Para los aspectos dinámicos, se pueden usar diagramas de secuencia, comunicación y de máquina de estados de UML.
Documentación de operaciones	Captura los procedimientos e información de soporte para hacer funcionar el sistema una vez esté en producción.
Evaluación de organización	Describe la organización en la que se desplegará el sistema, incluyendo una indicación de la capacidad de la organización para adoptar el nuevo sistema.
Modelo de organización	Estructura de la gente involucrada en el proyecto: miembros del equipo, participantes y sus papeles.
Modelo de datos físicos (PDM)	Esquema físico de soporte de datos, por ejemplo una base de datos relacional o un archivo XML. Se recomienda notación UML.
Glosario	Vocabulario específico del negocio del proyecto.
Visión general del proyecto	Resume las metas y planes. Contiene una declaración de objetivos, visión y organización.
Plan del proyecto	Consta de planes de iteraciones, calendario general, lista de riesgos, estimaciones y presupuesto.
Recursos de proyecto	Consta de la financiación, hardware/software e instalaciones.

Calendario del proyecto	Indica las actividades, sus relaciones y los *milestones*. Es típico utilizar un diagrama de Gantt o Pert.
Prototipo Proof-of-concept	Código que demuestra que el enfoque técnico tomado funciona. Se crea en la fase de Elaboración. Demuestra que la arquitectura funciona de principio a fin.
Registro de revisiones	Los resultados, incluyendo las acciones a llevar a cabo en puntos difíciles, de una revisión.
Lista de riesgos	Lista de los riesgos identificados y estrategias para atenuarlos (si las hay).
Modelo de amenaza de seguridad	Examina las amenazas del sistema, a menudo representadas con diagramas de despliegue UML.
Documentación de soporte	Incluye guías *trouble-shooting*, información para contactar con el equipo, etc.
Documento general del sistema	Destinado al personal responsable de mantener y desarrollar el sistema.
Proceso adaptado	Guías y plantillas adaptadas al proyecto en curso.
Requisitos técnicos	Cuestiones como usabilidad, seguridad y rendimiento. Suelen llamarse requisitos "no funcionales".
Plantillas	Impresos electrónicos para rellenar un tipo de *workproduct* concreto.
Modelo de test	Contiene el conjunto de tests de regresión e informes de defectos.
Estrategia de test	Descripción del enfoque de los tests. Se puede reutilizar el de otros proyectos.
Herramientas	Todo el software necesario para desarrollar el sistema.
Material de formación	Notas del curso, documentación general, asignaciones... para ayudar a los usuarios finales y al personal de soporte y operaciones.
Casos de uso	Los *use case* describen algo de valor para un participante y son requisitos primarios para AUP.
Modelo de casos de uso	Están formados por cero o más diagramas de casos de uso con sus descripciones, y descripciones de los actores.
Documentación de usuario	Los manuales, documentación de ayuda, etc. que ayudan a los usuarios finales a comprender el sistema. Debe aparecer cómo contactar con el personal de soporte.
Modelo de interfaz de usuario	Describe el interfaz con el sistema. Prototipos en papel o dibujos en la pizarra suelen ser suficientes para saber qué hay que hacer. Deberían hacerse en las etapas de concepción, quizá elaboración (en construcción el código debe funcionar).

Tabla 15. Otros artefactos secundarios de AUP.

WORKPRODUCTS DE LA EMPRESA

Los equipos de la propia empresa (arquitectos, administradores de bases de datos...) a menudo proporcionarán los siguientes *workproducts* para ayudar al proyecto:

Entrega	Descripción
Modelo de arquitectura de empresa	Representa los *frameworks*, redes, configuraciones de despliegue, infraestructura técnica de soporte y la infraestructura del dominio para una organización.
Guía de desarrollo de empresa	Estándares y pautas aplicables a todos los sistemas de la organización, incluyendo estándares de codificación, reglas de la red, estándares de datos...
Dirección de empresa	Estándares y pautas (*guidance*) a seguir dentro de la organización: guías de desarrollo, recursos humanos, modelado, usabilidad...
Declaración de misión de la empresa	Declaración de las estrategias a seguir para lograr la visión de la empresa.
Visión de la empresa	Declaración de los objetivos primarios de una organización.
Guía de recursos humanos	Contratación, ascensos, transferencias, formación, educación...
Guía de modelado	Técnicas como convenciones de nombres, estilo de presentaciones, notación...
Arquitectura de referencia	Arquitectura funcional con el conjunto de tests y *workproducts* para poder usarla.
Guía de usabilidad	Guía para desarrollar la interfaz del programa y usarla de forma efectiva.

Tabla 16. Workproducts facilitados por la empresa.

2.5.7. Limitaciones

RUP generalmente no se ha considerado particularmente “ágil”. El método fue ideado originariamente para desarrollar el proceso de producción de software por completo, y por consiguiente contiene extensas (poca ágiles) pautas para cada fase. El método RUP completo lista más de cien “artefactos” en las diferentes fases del proceso, siendo necesario quedarse con los imprescindibles. La cuestión, es saber cuánto se puede omitir para seguir siendo considerado RUP.

Una de las mayores desventajas de RUP es que no proporciona ninguna guía de implementación como *Crystal,* que lista la documentación requerida y papeles con respecto a la criticidad del software y tamaño del proyecto.

2.6. Dynamic Systems Development Method – DSDM

Desde su origen en 1994, DSDM[16], se popularizó en el reino Unido para RAD, *Rapid Application Development*. DSDM es un *framework* para el desarrollo de RAD sin ánimo de lucro y no propietario, mantenido por el *DSDM Consortium*. Sus creadores, 16 expertos en RAD, afirman que además de servir como un método también proporciona un *framework* de controles para RAD, complementado con consejos sobre cómo usarlos eficazmente.

La idea fundamental detrás de DSDM es que **en lugar de fijar la cantidad de funcionalidad en un producto, y después ajustar tiempo y recursos para alcanzar esa funcionalidad, se prefiere ajustar la cantidad de funcionalidad conforme a un tiempo y recursos fijados**.

2.6.1. Proceso

DSDM consiste en cinco fases [Stapleton 1997]: estudio de viabilidad, estudio comercial, iteración del modelo funcional, iteración de diseño y construcción, e iteración de implementación. Las primeras dos fases son secuenciales, y sólo se hacen una vez; las tres últimas fases durante las que se desarrolla el trabajo, son iterativas e incrementales.

Para DSDM, las iteraciones son *time-boxes* o cajas de tiempo. Una *time-box* dura un período de tiempo predefinido, y la iteración debe completarse en ese tiempo. El tiempo permitido para cada iteración se decide de antemano y garantiza los resultados que se pretenden conseguir. Una duración típica de la *time-box* es de unos días a unas semanas.

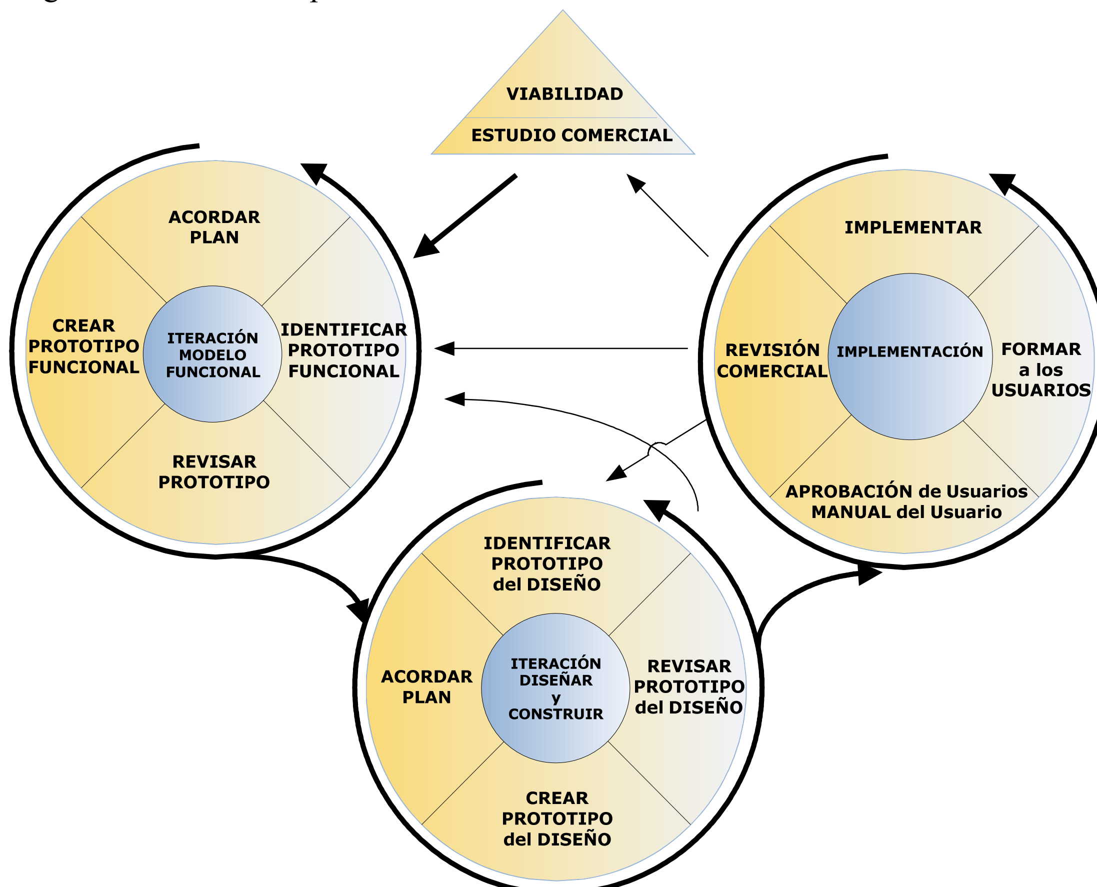


Ilustración 46. Diagrama de procesos de DSDM.

[16] Página web del DSDM Consortium: www.dsdm.org

Las fases de DSDM, con sus documentos resultantes son:

1) ESTUDIO DE VIABILIDAD

El estudio de viabilidad no debería durar más de unas semanas y evalúa la conveniencia de DSDM para un proyecto dado. Se tiene en cuenta el tipo de proyecto y la gente que lo usará. Además, se preocupa por las posibilidades técnicas de llevar a cabo el proyecto y los riesgos que conllevaría. Se preparan dos *workproducts*: un informe de viabilidad, y un esbozo del plan para el desarrollo. Opcionalmente, puede hacerse un prototipo rápido si el negocio o tecnología son nuevos, para decidir si continuar con la siguiente fase o no.

2) ESTUDIO COMERCIAL

En el estudio comercial se analizan las características esenciales del negocio y la tecnología. Se recomienda organizar talleres con un número suficiente de expertos en clientes para poder considerar todos los posibles casos del sistema y establecer las prioridades del desarrollo. Se describen los procesos comerciales afectados y los tipos de usuario en una **Definición de Área Comercial**, *Business Area Definition*. Se presentan descripciones de alto nivel de los procesos en formato conveniente, como diagramas ER.

En la fase de estudio comercial también se elabora:

- Una **Definición de Arquitectura de Sistema**, el primer boceto de la arquitectura de sistema, pero se permitirá cambiarla en el curso del proyecto.
- Un **Plan de esbozos de prototipos** (*Outline Prototyping Plan*) que recoja la estrategia de prototipos para las siguientes fases, y un plan para la gestión de configuración.

3) ITERACIÓN DEL MODELO FUNCIONAL

La fase de iteración del modelo funcional es la primera fase iterativa e incremental. En cada iteración, se decide el contenido y el enfoque adecuado a partir de los resultados de iteraciones anteriores. Se realiza el análisis, la codificación y los prototipos. Los prototipos no se desechan completamente, sino que se mejoran gradualmente hasta poder incluirlos en el sistema final.

En los diferentes pasos de las fases, se generan en total cinco *outputs* o resultados:

1) Un **Modelo Funcional** con el código prototipo y los modelos de análisis. El testeo continuo aquí es fundamental.
2) Las **Funciones Priorizadas** son una lista priorizada de las funciones que se entregan al final de la iteración.
3) Los **Documentos de la Revisión del Prototipo Funcional** (*Functional prototyping review documents*) recogen los comentarios de los usuarios del incremento actual para tenerse en cuenta en las siguientes iteraciones.
4) Se enumeran los **Requisitos no funcionales**, principalmente para ser tratados en la próxima fase.
5) El **Análisis de riesgo de más desarrollos** es un documento importante, porque de la próxima fase (iteración diseñar y construir) en adelante, los problemas que surjan serán más difícilmente atacables.

4) ITERACIONES DISEÑAR Y CONSTRUIR

El resultado es un sistema testeado que cumple por lo menos el conjunto mínimo de requisitos. Diseño y Construcción es una fase iterativa, y tanto el diseño como los prototipos funcionales

son revisados por los usuarios, ya que el desarrollo posterior vendrá determinado por sus impresiones.

5) IMPLEMENTACIÓN

En la última fase, el sistema se transfiere del entorno de desarrollo al propio entorno de producción. Se forma a los usuarios y se les entrega el software. Si existen muchos usuarios o se requiere mucho tiempo, esta fase podría iterarse. En esta fase se entrega:

- el software,
- el manual, y
- un Informe de Revisión de Proyecto.

DSDM define cuatro posibles cursos de desarrollo:

- Si el sistema cumple todos los requisitos, no se necesita hacer nada más.
- Si hay una cantidad importante de requisitos que tiene que ser dejada de lado (por ejemplo, si no fueron descubiertos hasta que el sistema estaba en desarrollo), el proceso debe realizarse de nuevo, de principio a fin.
- Si alguna funcionalidad menos crítica tiene que ser omitida, podría ejecutarse el proceso desde la fase del modelo funcional.
- Si algunos problemas técnicos no pueden tratarse debido a la falta de tiempo, pueden hacerse ahora iterando las fases de diseño y construcción.

2.6.2. Papeles y responsabilidades

En la versión actual de DSDM, 4.2, se definen 10 papeles:

- PATROCINADOR EJECUTIVO. Es la persona que posee autoridad y responsabilidad financiera, y quien tiene la última palabra en las decisiones importantes.

- VISIONARIO. Es el usuario que tiene la percepción más exacta de los objetivos del sistema y del proyecto. Asegura que se cumplan los requisitos esenciales y que el proyecto vaya en la dirección adecuada desde el punto de vista de los requisitos.

- USUARIO EMBAJADOR. Proporciona al proyecto conocimiento de la comunidad de usuarios a la que se dirige el proyecto y difunde información sobre el progreso del sistema a otros usuarios. Es clave para asegurar un diseño adecuado. Revisa la documentación, crea la documentación para el usuario y supervisa tanto los tests de los usuarios como su formación.

- USUARIO ASESOR (*Advisor*). Representa otros puntos de vista importantes; puede ser alguien del personal de IT o un auditor financiero. Aprueba los diseños y los prototipos.

- JEFE DE PROYECTO. El *project manager* se asegura que se entreguen todos los aspectos del proyecto. Tiene la responsabilidad de coordinar e informar a los gestores, elaborar el calendario, supervisar el progreso, controlar la gestión de riesgos, tratar problemas inesperados, etc.

- COORDINADOR TÉCNICO. Define la arquitectura del sistema y es responsable de la calidad técnica del proyecto, del control técnico y la configuración del sistema. Es el encargado del control de versiones.

- LÍDER DEL EQUIPO. Se encarga que todo el equipo vaya en la misma dirección. Gestiona el control de cambios y la documentación. Informa al jefe de proyecto del progreso.

- **DESARROLLADORES**. Los *developers* modelan e interpretan los requisitos, convirtiéndolos en prototipos y código entregable. No hay distinción entre analista, diseñador y programador.

- **TESTEADOR**. Realiza los tests (no de usuario ni tests de unidad) según la estrategia de tests del plan de desarrollo. El *tester* es parte del equipo de desarrollo. El usuario embajador es el responsable de todos los tests de usuario.

- **ESCRIBA**. El *scribe* registra los requisitos, acuerdos y decisiones alcanzadas en las reuniones, talleres y sesiones acerca de los prototipos. Distribuye la documentación.

2.6.3. La práctica

Nueve prácticas, llamadas principios en DSDM, definen la ideología y la base para todas las actividades en DSDM.

PRÁCTICA DSDM	DESCRIPCIÓN
Compromiso activo del usuario	Tienen que estar presentes unos pocos usuarios entendidos a lo largo del desarrollo del sistema para asegurar *feedback* rápido y preciso.
Equipos de DSDM autorizados para tomar decisiones	No se pueden tolerar procesos que se tarde mucho en decidir; los ciclos de desarrollo deben ser rápidos. Los usuarios involucrados deben tener el conocimiento suficiente para decidir.
Entrega frecuente de productos	Pueden corregirse las decisiones erróneas, con el *feedback* adecuado y si los ciclos de entrega son cortos.
Adecuar las entregas al propósito comercial	Construir el producto adecuado antes de construirlo bien: **"*Build the right product before you build it right*"**.
Desarrollo iterativo e incremental	Los requisitos del sistema cambian. Si el sistema evoluciona con el tiempo gracias al desarrollo iterativo, pueden encontrarse y corregirse los errores pronto.
Los cambios durante el desarrollo son reversibles	Durante el desarrollo, es fácil tomar un camino equivocado. Con iteraciones cortas y asegurando poder volver al paso anterior, puede corregirse el camino.
Establecer una baseline a alto nivel para los requisitos	Congelar los requisitos centrales sólo debe hacerse a alto nivel; se debe permitir que los requisitos de los detalles cambien. Conforme se avanza, se deben congelar más requisitos.
Pruebas a lo largo de todo el proceso	Con el tiempo justo, se suele retardar las pruebas hasta el final. Los programadores y usuarios deben irlos testeando al desarrollarlos. Probar también es incremental. Se enfatizan los tests de regresión.
Cooperación y colaboración entre los participantes	La organización debe comprometerse con DSDM, y sus departamentos de IT y comercial deben cooperar. Elegir qué partes se entregan y cuáles no es un compromiso que requiere acuerdo común.

Tabla 17. Principios de DSDM.

2.6.4. Adopción y experiencias

En el DSDM Consortium hay documentos específicos sobre la convivencia de su metodología con Microsoft Solutions Framework. También hay casos de éxito (particularmente el de Fujitsu European Repair Centre) en que se empleó Visual Basic como lenguaje, SQL Server como base de datos y Windows como plataforma de desarrollo e implementación.

Para facilitar la adopción del método, el *DSDM Consortium* ha publicado un cuestionario para ayudar a decidir si DSDM es conveniente para un proyecto. El cuestionario cubre tres áreas: comercial, de sistema y técnica. Las preguntas principales a considerarse son:

PREGUNTA	DESCRIPCIÓN
¿La funcionalidad va a ser bastante visible en la interfaz del usuario?	Si se intenta la implicación del usuario a través de prototipos, los usuarios deben verificar (sin entender los detalles técnicos) que el software hace lo esperado.
¿Puede identificar todos los tipos de usuarios finales?	Es importante ser capaz de identificar e involucrar a todos los grupos de usuarios del software.
¿La aplicación es computacionalmente compleja?	Existe peligro para desarrollar con una aplicación compleja desde cero con DSDM. Se obtienen mejores resultados si ya hay partes disponibles.
¿La aplicación es grande? ¿Puede dividirse en componentes pequeñas?	DSDM se ha usado para el desarrollo de sistemas muy grandes, algunos han durado 3 años, aunque la funcionalidad se desarrolla en pequeños pasos.
¿El proyecto realmente es "urgente" (time constrained)?	La implicación del usuario activo es muy importante. Si los usuarios no participan (porque realmente no tiene una fecha límite establecida estrictamente), los programadores pueden frustrarse y empezar a hacer suposiciones sobre qué se necesita y cuándo.
¿Los requisitos son flexibles y sólo especificados a alto nivel?	Si los requisitos detallados han sido acordados antes de que los programadores empiecen su trabajo, no se lograrán los beneficios de DSDM.

Tabla 18. Preguntas antes de adoptar DSDM.

2.6.5. Limitaciones y actualidad

El equipo mínimo de DSDM es de dos personas y puede llegar a seis, pero puede haber varios equipos en un proyecto. El mínimo de dos personas involucra un programador y un usuario. El máximo de seis es un valor encontrado en la práctica. DSDM se ha aplicado a proyectos grandes y pequeños. La condición previa para su uso en sistemas grandes es que se puedan dividir en equipos pequeños.

Stapleton sugiere que DSDM se aplica más fácilmente a sistemas comerciales que a diseñar aplicaciones científicas o de ingeniería.

DSDM fue desarrollado originalmente y continúa siendo mantenido por un consorcio que consiste en varias compañías miembro. Los manuales de DSDM y la documentación de ayuda están disponibles para los miembros del consorcio a cambio de una cuota anual. Como ejemplo de continuación después de su creación, en 2001 se lanzó e-DSDM, una versión del método adaptada para los e-Business y proyectos de e-Commerce [Highsmith 2002].

Mike Griffiths, de Quadrus Developments, ha elaborado en particular la combinación de DSDM con XP, llamada EnterpriseXP (www.enterprisexp.org). También hay documentos conjuntos de DSDM y Rational, con la participación de Jennifer Stapleton, que demuestran la compatibilidad de DSDM con RUP, a pesar de sus fuertes diferencias terminológicas.

2.7. Adaptive Software Development – ASD

El Desarrollo Adaptativo de Software [17], creado por el miembro del *Cutter Consortium* James A. Highsmith III, y se publicó el año 2000. Muchos de los principios de ASD provienen de las primeras investigaciones de Highsmith en los métodos de desarrollo iterativos. El predecesor más notable de ASD, "*RADical Software Development*", fue desarrollado por Highsmith y S. Bayer en 1994.

Ilustración 47. Jim Highsmith.

ASD está dirigido principalmente a problemas en sistemas complejos y grandes. El método propone un desarrollo incremental, iterativo, con continuos prototipos. En esencia, ASD trata de "hacer equilibrios al límite del caos".

Entre las influencias de Highsmith, haber colaborado para la US Air Force, le hizo destacar la importancia de los tests:

"Si un componente tiene una fiabilidad del 99.9%, cuando juntamos 200 componentes, la fiabilidad se reduce a un 82%, y si ponemos 1.000 bajaría hasta un 37%"

Este factor multiplicativo hace que realizar software o equipos muy fiables sea algo realmente concienzudo y complejo. Dio formación en Microsoft y observó la empresa, dándose cuenta de que aunque se trabajaba de forma diferente (FDD, mucha comunicación, *milestones* o hitos incrementales...), se conseguían buenos resultados. También tiene una gran experiencia en RAD. Cuando trabajó para Nike, se dio cuenta de que necesitaban una nueva manera de gestionar el proceso creativo, regido por los resultados, *result-driven*.

Su objetivo es proporcionar un *framework* o esqueleto con la suficiente guía para evitar caer en el caos, pero sin ser excesiva para no suprimir nuevas ideas ni la creatividad. **Un plan se ha de ver como una hipótesis, no como una predicción**.

2.7.1. Proceso

Un proyecto de ASD se lleva a cabo en ciclos de tres fases [Highsmith 2000]: Especular, Colaborar, y Aprender.

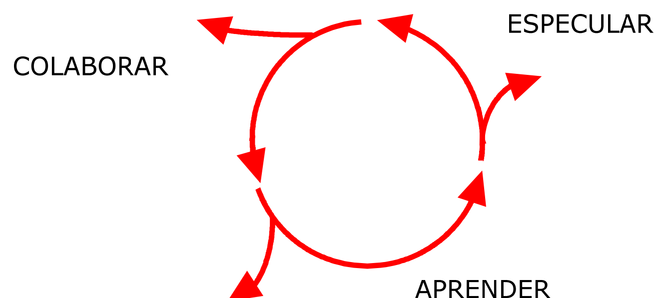


Ilustración 48. El ciclo de ASD.

Los nombres de las fases ponen énfasis al papel del cambio en el proceso. Por ejemplo, se usa "*Especulación*" en vez de "*Planning*", ya que en un plan, una incertidumbre generalmente se

[17] www.adaptivesd.com

ve como una debilidad, y si hay desviaciones indican fracaso. De igual forma, "*Colaborar*" resalta la importancia del trabajo en equipo para tratar con condiciones muy cambiantes. "*Aprender*" resalta la necesidad de reconocer y reaccionar ante los errores, y el hecho que los requisitos pueden cambiar durante el desarrollo.

La fase de Inicio de Proyecto define las piedras angulares del proyecto, y se empieza definiendo la misión del proyecto. La misión, básicamente pone un marco general para el producto final, y todo el desarrollo tiene como fin lograr la misión.

Una de las cosas más importantes al definir la misión del proyecto es deducir qué información necesaria para llevar a cabo el proyecto. Se definen las facetas importantes de la misión en tres artículos:

- Una carta constitucional de visión de proyecto,
- datos del proyecto, y
- un contorno de especificación de producto.

La fase de Inicio de Proyecto prepara el programa global a seguir, así como los planes y objetivos de los ciclos de desarrollo. Los ciclos típicamente duran de 4 a 8 semanas.

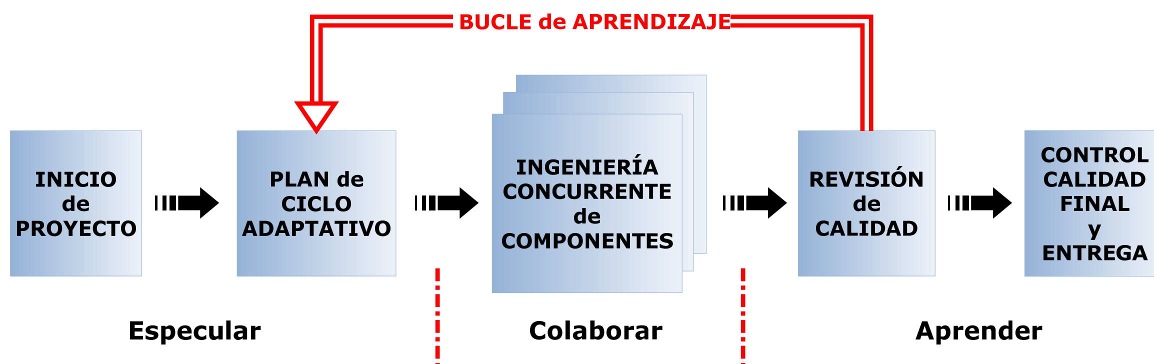


Ilustración 49. Fases del ciclo de vida de ASD.

ASD está orientado a componentes, no a tareas. En la práctica, esto significa que se prefieren los resultados y su calidad en lugar de las tareas o el proceso usado para llegar al resultado. La manera con que ASD plasma este punto de vista es a través de ciclos de desarrollo adaptativos que contienen la fase Colaboración, donde varias componentes pueden estar en desarrollo a la vez. Planear los ciclos es una parte del proceso iterativo y las definiciones de los componentes se refinan continuamente para reflejar cualquier nueva información o cambio en los requisitos de las componentes.

La base para los ciclos siguientes (el bucle de aprendizaje) se adquiere de sucesivas revisiones de calidad, centradas en demostrar la funcionalidad del software desarrollado durante el ciclo. Al realizar las revisiones, es vital la presencia del cliente y de un grupo de expertos (*customer-focus group*). Sin embargo, dado que las revisiones de calidad son bastante escasas (sólo al final de cada ciclo), la presencia del cliente en ASD se apoya con sesiones JAD, desarrollo de aplicaciones compartidas (*Joint Application Development*).

Una sesión de JAD es esencialmente un taller donde diseñadores y representantes del cliente discuten las características del producto. ASD no propone periodicidad para las sesiones de JAD, pero son particularmente importantes al principio.

La fase final en un proyecto ASD es el QA final (control de calidad) y la entrega. ASD no considera cómo debe llevarse a cabo la fase, pero remarca la importancia de poner en práctica las lecciones aprendidas. Como resumen, la naturaleza adaptable de desarrollo en ASD se caracteriza por las propiedades de la tabla siguiente.

CARACTERÍSTICA	DESCRIPCIÓN
Orientado a proceso (misión)	Las actividades de cada ciclo deben contrastarse con la misión del proyecto global. La misión puede ajustarse.
Basado en componentes	Las actividades de desarrollo no deberían estar orientadas a tarea, sino enfocadas en hacer software que funcione, construyendo pequeños trozos del sistema.
Iterativo	Un método en cascada serie sólo se adapta a entornos bien definidos. Si hay muchos cambios, el esfuerzo debe ponerse en “rehacer” y no en “hacerlo bien a la primera”.
Time-Boxed	La ambigüedad en proyectos complejos puede aliviarse fijando fechas tope factibles regularmente. Una gestión *time-boxed* obliga a tomar las decisiones pronto.
Tolerante a cambios	Es más importante adaptarse a los cambios que controlarlos. Los diseñadores deben evaluar constantemente si es probable que los componentes que hacen cambien.
Risk-Driven	El desarrollo de los puntos de alto riesgo (menos conocidos, o más críticos) debe empezar lo más pronto posible.

Tabla 19. Características de ciclos de desarrollo adaptativos.

2.7.2. Papeles y responsabilidades

El proceso ASD se origina de la cultura organizativa y de gestión, pero sobre todo, de la importancia de la colaboración entre equipos y del trabajo en equipo. A pesar de esto, Highsmith, no describe la estructura del equipo en detalle. Igualmente, se citan muy pocos papeles o responsabilidades:

- Un patrocinador ejecutivo que tiene la responsabilidad global del producto.
- Los participantes en una sesión JAD:
    - un ayudante para planear y moderar la sesión,
    - un secretario para tomar actas,
    - el jefe del proyecto,
    - representantes del cliente, y
    - representantes de los programadores.

2.7.3. La práctica

ASD propone muy pocas prácticas para el trabajo diario. Básicamente, expresamente cita tres:

- desarrollo iterativo,
- planning basado en características (basadas en componentes), y
- revisiones en grupo enfocadas al cliente.

De hecho, quizás el problema más significante con ASD es que sus prácticas son difíciles identificar, y deja muchos puntos sin especificar.

2.7.4. Adopción y experiencias

Los principios e ideas subyacentes de ASD son pautas razonables, pero se dan pocas directrices para aplicar el método. La filosofía del libro de Highsmith es más construir una cultura organizativa adaptativa, que no dar pautas específicas. Como consecuencia, el método ofrece poca base para adoptar ASD en una organización. Por tanto, **ASD requiere complementarse con otros métodos**. Esto no implica que ASD sea inútil, simplemente profundiza en la filosofía del movimiento ágil.

2.7.5. Limitaciones y actualidad

ASD no tiene limitaciones innatas para su aplicación. Un rasgo interesante de ASD es que no fuerza a que los equipos estén localizados en la misma oficina como otros métodos.

A menudo, los equipos tienen que trabajar con las limitaciones de tiempo y espacio, por no citar las de la propia organización. Highsmith toma nota de esto, y puntualiza que la mayoría de las dificultades en el desarrollo distribuido se deben a problemas en habilidades sociales, culturales y del propio equipo. ASD ofrece las técnicas para reforzar la colaboración entre equipos mediante estrategias para compartir información, el uso de herramientas de comunicación, y maneras de introducir el rigor gradualmente en el trabajo del proyecto para apoyar el desarrollo distribuido.

No ha habido ninguna investigación significante en ASD, quizás debido al hecho que el método deja muchos cabos sueltos. El creador de ASD ha llevado los temas más allá, centrándose en las bases del desarrollo ágil de software ágiles (personas, relaciones e incertidumbre) y lo ha plasmado en el libro *Agile Software Development Ecosystems* [Highsmith 2002].

2.8. Open Source Software Development – OSS

La visión del método ágil basado en software libre o de código abierto, tiene varias similitudes con otros métodos ágiles, aunque también tiene sus propias peculiaridades. OSS ofrece un nuevo paradigma de desarrollo de software que ofrece una manera innovadora de crear las aplicaciones. El interés por OSS ha aumentado notablemente después de varias historias de éxito como el servidor *Apache*, el lenguaje *Perl*, el gestor de correo *SendMail* o el sistema operativo *Linux*. Microsoft incluso ha admitido que Linux es su competidor más duro en sistemas operativos para servidores.

La filosofía de OSS implica que el código fuente está disponible libremente para modificarse y redistribuirse sin cargo alguno. [Feller y Fitzgerald 2000] resumen las motivaciones y pautas para el desarrollo de OSS:

1) **Tecnológica**: necesidad de código robusto, ciclos de desarrollo más rápidos, las más estrictas normas de calidad, fiabilidad y estabilidad, y más plataformas y estándares abiertos.
2) **Barata**: la necesidad colectiva de costo y riesgo compartido.
3) **Socio-política**: satisfacer el reto de programadores particulares, la reputación de los colegas, el deseo de trabajo útil, con sentido, y la comunidad orientada al idealismo.

La mayoría de los proyectos OSS pertenecen a herramientas de desarrollo u otras plataformas usadas por profesionales que han participado a menudo en el desarrollo de ellos, por tanto, teniendo los papeles de cliente y programador al mismo tiempo. OSS no es una recopilación de prácticas estrictamente definidas y publicadas para desarrollar software. En cambio, está muy detallado en lo que se refiere a diferentes licencias para la distribución del software y como una manera colaboradora de individuos ampliamente dispersados para producir software a partir de incrementos pequeños y frecuentes. La *Open Source Initiative*[18] hace un seguimiento y concede licencias para software que cumple las definiciones de OSS. Varios investigadores, (por ejemplo, Gallivan 2001 o Crowston y Scozzi 2002), sugieren que un proyecto OSS representa una organización virtual.

2.8.1. Proceso

Aunque Cockburn menciona que OSS difiere de otros métodos ágiles en aspectos filosóficos, económicos y estructura de grupo, OSS sigue muchas veces las mismas líneas de pensamiento y prácticas que otros métodos ágiles. Por ejemplo, el proceso desarrollo de OSS empieza con versiones frecuentes sin esperar demasiado, y le faltan muchos de los mecanismos tradicionales usados para coordinar planes, fechas y procesos definidos. Típicamente, un proyecto de OSS consiste en las siguientes fases visibles según [Sharma et al. 2002]:

Ilustración 50. Fases visibles de un proyecto OSS.

[18] www.opensource.org

Aunque es posible separar las fases de OSS, el interés está en cómo se gestiona este proceso. [Mockus et al. 2000] lo resumió:

- ✓ Los sistemas se construyen con un número potencialmente grande de voluntarios.
- ✓ El trabajo no se asigna; cada uno escoge la tarea que le interese.
- ✓ No existe ningún diseño explícito a nivel de sistema.
- ✓ No hay plan del proyecto, calendarios o lista de entregas.
- ✓ El sistema crece con incrementos pequeños.
- ✓ Los programas se testean frecuentemente.

Según [Feller y Fitzgerald 2000], el proceso OSS se organiza como un desarrollo enorme en paralelo y esfuerzo de depuración. Esto implica la cooperación y contribuciones desinteresadas de los programadores individuales. Hay también, sin embargo, señales que esta idea de gratis o libre está cambiando. El proceso de desarrollo de OSS no incluye normas predefinidas sobre la documentación o costumbres. Sin embargo, el proceso involucra costumbres y tabúes que sólo se aprenden con la experiencia.

2.8.2. Papeles y responsabilidades

El proceso de desarrollo OSS parece bastante libre y descabellado. Sin embargo, debe tener alguna estructura, o de lo contrario nunca habría podido lograr éxitos tan notables como ha hecho en los últimos años. También empresas consolidadas (IBM, Apple, Oracle, Corel, Netscape, Intel, Ericsson...) muestran interés en OSS. En estos esfuerzos, sobre todo en los mayores proyectos de OSS, las compañías citadas han tomado el papel de coordinador y de mediador, actuando así como la dirección del proyecto.

Otro nuevo fenómeno es la aparición de sitios web almacén como *Freshmeat* y *SourceForge*[19] que disponen de código y aplicaciones libres. Estos sitios normalmente proporcionan servicios gratis para programadores como alojamiento web, backups, archivos antiguos, etc. Dado que no hay reuniones cara a cara en el contexto de OSS, la importancia de Internet es obvia. El seguimiento de las versiones aquí es aún más importante al haber una multitud de programadores y muy variopintos.

Una estructura típica de desarrollo de OSS está compuesta por varios niveles de voluntarios y podría ser esta:

1) **Líderes del proyecto**: tienen la responsabilidad global por el proyecto y normalmente han escrito el código inicial.

2) **Programadores voluntarios**: crean y presentan código. Pueden dividirse en:
   a) Miembros Senior o programadores con más autoridad global.
   b) Programadores Periféricos que crean y presentan cambios del código.
   c) Contribuyentes Ocasionales.
   d) Ayudantes y diseñadores acreditados.

3) **Otros individuos** que comprueban, identifican *bugs* e informan de fallos.

4) **Posters** que frecuentemente participan en *newsgroups* y discusiones, pero no programan.

[19] http://sourceforge.net , http://freshmeat.net

[Sharma et al. 2002] expone que los proyectos de OSS normalmente los dividen los arquitectos o programadores principales en tareas más pequeñas y manejables de las que se ocuparán individuos o grupos. Los programadores voluntarios se dividen en individuos o grupos pequeños que seleccionan las tareas que harán libremente. Por tanto, la división modular racional del proyecto global es esencial.

2.8.3. La práctica

Empezar o adquirir la propiedad de un proyecto de OSS pueden hacerse de varias maneras:

- fundar uno nuevo,
- cogerlo si el dueño anterior lo cede, o
- tomar voluntariamente un proyecto a punto de morir.

A menudo, los futuros usuarios definen el proyecto y también programan. El proceso es continuo, ya que el desarrollo está evolucionando eternamente. Aunque pueden estar participando cientos de voluntarios, normalmente sólo un grupo pequeño realiza la parte principal del trabajo.

Sharma describe algunos de los aspectos organizativos centrales en OSS, por ejemplo la división de tareas, el mecanismo de coordinación, la distribución de toma de decisiones, los límites organizativos, y la estructura informal del proyecto.

Para trabajar adecuadamente, los individuos geográficamente dispersados así como los grupos pequeños de programadores, deben comunicarse eficazmente, sobre todo cuando normalmente no se encuentran cara a cara.

2.8.4. Adopción y experiencias

[Lerner y Tirole 2001] resumen parte de las experiencias acerca de los caminos de desarrollo de Linux, Perl, y Sendmail, describiendo el desarrollo desde el esfuerzo de un solo programador a un equipo de decenas de miles geográficamente distribuidos.

Uno de los problemas y al mismo tiempo ventaja, es que continuamente se están mejorando, y que el producto nunca está "acabado". OSS depende totalmente de Internet que posibilita la colaboración.

Otra cuestión respecto al software de OSS es la estabilización y puesta a la venta en el mercado de código al nivel de aplicación. Aunque el código OSS no es un producto comercial, hay empresas que han empaquetado las partes útiles para ellos y las comercializan. Redhat y SOT para Linux son dos ejemplos de ello.

2.8.5. Limitaciones y actualidad

El propio método de OSS no conlleva ningún dominio especial para la aplicación ni pone ningún límite para el tamaño del software. Las tendencias futuras también confían en aprovechar la colaboración global entre programadores, mientras que los esfuerzos de OSS están produciendo software que después se integra y se vende. Desde una perspectiva legal, OSS debe verse más como una estructura autorizadora que se aprovecha de las condiciones de la Licencia Pública General, GNU www.gnu.org.

El propio proceso de desarrollo OSS puede llevarse a cabo con métodos de desarrollo de software específicos diferentes, aunque estos métodos deben permitir las características de OSS. Obviamente, el proceso puede tener los aspectos ágiles, pero por otro lado, el propio proceso de desarrollo puede retardarse por varias razones, por ejemplo, falta de programadores interesados y hábiles.

Están haciéndose estudios para explicar cómo un método casi sin normas funciona con éxito. Un tema peliagudo son los aspectos legales referentes al uso de componentes de OSS. Las compañías de software están mostrando interés en averiguar las posibilidades de usar la metodología de OSS como ayuda.

Internet es crucial para funcionar y poner los prototipos al alcance de todos. Sin embargo, la comunidad es áspera, como el mundo de los negocios convencionales, es decir, si encuentra clientes, usted sobrevive, pero sin clientes usted se muere. En otras palabras, sólo si el prototipo introducido capta interés y atención, empezará a atraer a programadores.

[Bergquist y Ljungberg 2001] concluyen: ***"Uno podría entender la cultura OSS como un tipo de fusión de colectivismo e individualismo: dar a la comunidad es lo que hace un héroe al individuo en los ojos de los otros. Los héroes son las influencias importantes, pero también los líderes poderosos."***

[Crowston y Scozzi 2002] han investigado varios proyectos OSS usando los datos disponibles en Sourceforge. Sus conclusiones principales son:

1) La disponibilidad de competencias es un factor clave en el éxito de proyectos.

2) Es importante para los programadores poder reconocer las necesidades de los clientes. Los proyectos relacionados con el desarrollo de sistemas y administración han sido más exitosos.

3) Los proyectos gestionados por administradores muy conocidos tienen más éxito, ya que atraen a nuevos voluntarios más fácilmente.

2.9. Lean Software Development - LSD

Lean Development es el método menos divulgado entre los reconocidamente importantes. La palabra "*lean*" significa magro, sin grasa; en su sentido técnico apareció por primera vez en 1990 en el libro de James Womack *La máquina que cambió el mundo*. LD, iniciado por Bob Charette, se inspira en el éxito del proceso industrial *Lean Manufacturing*, bien conocido en la producción manufacturera y automovilística desde la década de los 80. Parte de LD surgió de buscar maneras de construir sistemas para las compañías de telecomunicaciones.

Ilustración 51. Bob Charette.

Lean Production a veces se ha traducido como "producción exacta". Este proceso tiene como precepto la eliminación de desperdicios –*waste*– a través de la mejora constante, haciendo que el producto fluya a instancias del cliente para hacerlo lo más perfecto posible.

Un aspecto importante para Charette, fue el uso de plantillas (*templates*) para identificar situaciones anteriores parecidas y de esa forma acelerar entre un 20 y un 40% la velocidad del desarrollo. Las plantillas son como componentes pero a más alto nivel, son funciones orientadas al dominio del problema, como control de proceso, e-Commerce, facturas, etc.

El paso de Charette por la escuela de teoría económica de Austria, le hace ver también el lado económico: tener a un representante profesional del cliente siempre a disposición implicaría un gasto muy elevado. Trabajó en una empresa de telecomunicaciones con un problema bastante habitual: querer aplicar el mismo método, CMM, para todas sus prácticas, aún cuando para el 60% de ellas, LD era muchísimo mejor. Dell también vio que ERP no era lo suficientemente flexible y lo cambió.

LD ha evolucionado a *Lean Software Development* (LSD), siendo su referencia Tom y Mary Poppendieck [20] [Poppendieck, 2003]. Se ha hablado de LD-2, adoptando las prácticas adaptativas de XP, pero a día de hoy todavía no se ha publicado ningún libro sobre LD-2.

2.9.1. Proceso

Los procesos a la manera americana corrían con sus máquinas al 100% de capacidad y mantenían inventarios gigantescos de productos y suministros; Toyota, en contra de la intuición, resultaba más eficiente manteniendo suministros en planta para un solo día, y produciendo sólo lo necesario para cubrir las órdenes pendientes. Esto es lo que se llama *Just in Time Production*. Con JIT se evita además que el inventario quede obsoleto, o empiece a actuar como un freno para el cambio. Toyota implementaba además las técnicas innovadoras del *Total Quality Management* de Edward Deming, que sólo algunos matemáticos y empresarios conocían en Estados Unidos. Hasta el día de hoy, la foto de Deming en Toyota es más grande que la del fundador, Toyoda Sakichi (después se cambió la 'd' por 't').

Otros aspectos esenciales de *Lean Manufacturing* son la relación participativa con el empleado y el trato que le brinda la compañía, una especificación de principios, disciplinas y métodos iterativos, adaptativos, auto-organizativos e interdependientes en un patrón de ciclos

[20] www.poppendieck.com

de corta duración que tiene más un aire de familia con el patrón de procesos de los MA[21]. Existe unanimidad de intereses, consistencia de discurso y complementariedad entre las comunidades Lean de producción y desarrollo de software.

Mientras que otros MA se centran en el proceso de desarrollo, Charette sostenía que para ser verdaderamente ágil también se debería conocer bien el negocio. LD se inspira en doce valores centrados en estrategias de gestión:

1) Satisfacer al cliente es la máxima prioridad.
2) Proporcionar siempre el mejor valor por la inversión.
3) El éxito depende de la activa participación del cliente.
4) Cada proyecto LD es un esfuerzo de equipo.
5) Todo se puede cambiar.
6) Soluciones de dominio, no puntos.
7) Completar, no construir.
8) Una solución al 80% hoy, en vez de una al 100% mañana.
9) El minimalismo es esencial.
10) La necesidad determina la tecnología.
11) El crecimiento del producto es el incremento de sus prestaciones, no de su tamaño.
12) Nunca llevar LD más allá de sus límites.

Otra preceptiva algo más amplia es la de Mary Poppendieck, cuidadosamente decantadas del *Lean Manufacturing* y de *Total Quality Management* (TQM), que sólo coincide con la de Norton que veremos posteriormente, en algunos puntos:

1) Eliminar desperdicios: Entre los desperdicios se encuentran diagramas y modelos que no agregan valor al producto.
2) Minimizar el inventario: Suprimir artefactos como documentos de requisitos y diseño.
3) Maximizar el flujo: Utilizar desarrollo iterativo.
4) Solicitar demanda: Soportar requisitos flexibles.
5) Otorgar poder a los trabajadores.
6) Satisfacer los requisitos del cliente: Trabajar junto a él, permitiéndole cambiar de idea.
7) Hacerlo bien la primera vez: Verificar pronto y *refactorizar* cuando sea preciso.
8) Abolir la optimización local: Alcance de gestión flexible.
9) Asociarse con quienes suministran: Evitar relaciones de adversidad.
10) Crear una cultura de mejora continua.

Uno de los sitios primordiales del modelo son las páginas consagradas a LSD que mantiene Darrell Norton[22], donde se promueve el desarrollo del método aplicando el *framework* .NET de Microsoft. Norton ha reformulado los valores de Charette reduciéndolos a siete y suministrando 22 herramientas similares a patrones organizacionales para su implementación en ingeniería de software.

[21] www.strategosinc.com/principles.htm
[22] http://codebetter.com/blogs/darrell.norton

2.9.2. Los 7 principios y las 22 herramientas por Darrell Norton

PRINCIPIO 1: ELIMINAR DESPERDICIOS (VER LOS DESPERDICIOS, *VALUE STREAM MAPPING*)

Desperdicio es todo lo que no agregue valor a un producto, desde el punto de vista del cliente. Este principio equivale a la reducción del inventario en manufactura, y aquí es el conjunto de artefactos intermedios. No implica eliminar toda la documentación, sino dedicarse a lo que agregue valor para el cliente. Este principio es la base de los otros principios. El primer paso para adoptar LSD es aprender a ver los desperdicios; el segundo, encontrar sus orígenes y eliminarlos.

Ilustración 52. La base de todos los principios de LSD.

- HERRAMIENTA 1: VER LOS DESPERDICIOS

Existen 7 tipos de desperdicio:

a) **El inventario** (trabajo parcialmente hecho). El inventario tiene tendencia a quedarse obsoleto. No se puede saber si funcionará o no hasta que el software realmente esté en producción; no se sabrá hasta entonces si se resolverá el problema comercial del cliente. ¿Qué pasa si el sistema nunca llega a la producción?

b) **Sobre-procesar** (procesos extras). El papeleo consume recursos, retarda la contestación, esconde los problemas de calidad, se pierde, se degrada y se queda obsoleto. Porque se requiera la entrega del papeleo no significa que agregue el valor. Si hay que hacer papeleo, debe ser breve y a alto nivel.

c) **Sobre-producción** (características extras): Todo el código en el sistema tiene que ser inspeccionado, compilado, integrado, y probado cada vez que se cambia, y después debe mantenerse. Todo aumento de complejidad del código es un punto de fracaso potencial. Hay gran posibilidad que el código extra quede obsoleto antes de que se use.

d) **Transporte** (cambiar de tareas o proyecto): Asignar personas a varios proyectos es una fuente de desperdicio. Cada vez que un programador cambia a otro proyecto, se pierde un tiempo hasta que vuelve a centrarse en la nueva tarea. Normalmente, pertenecer a varios equipos causa más interrupciones y más cambios de tarea.

e) **Esperas**: Una de las pérdidas de tiempo mayores es esperar a que las cosas pasen o que algo acabe. Los retrasos empezando un proyecto, al proveerse de personal, debidos a la documentación de requisitos excesiva, retrasos en las revisiones y aprobaciones, retrasos probando, y los retrasos en la distribución son un desperdicio.

f) **Movimiento** (*motion*): Cuando un diseñador tiene una pregunta, ¿cuánto "movimiento" necesita para averiguar una respuesta? Las personas no son las únicas cosas que mueven; los artefactos también. Los requisitos pueden cambiar de analistas a diseñadores, los documentos de plan de diseñadores a programadores, los del código a las pruebas, y así sucesivamente. Cada salto que da un artefacto tiene probabilidad de perderse. Las grandes cantidades de conocimiento tácito permanecen siempre con el creador del documento.

g) **Defectos**: La cantidad de desperdicio causada por un defecto es el producto del impacto del defecto por el tiempo que lleva sin detectarse. Deben encontrarse cuanto antes, probar inmediatamente, integrar a menudo, y entregar pronto a producción.

- HERRAMIENTA 2: OBSERVAR EL MAPA DE FLUJO DE VALOR

Hacer un mapa de flujo de valor (*value stream mapping*) ayuda a enfocar la atención en los productos y en el cliente en lugar de en las organizaciones, recursos, tecnologías, procesos, etc. La manera de construirlo es pasar por todos los pasos del proceso desde el punto de vista del cliente, señalando los tiempos cuando se aporta valor al cliente y cuándo no (esperas).

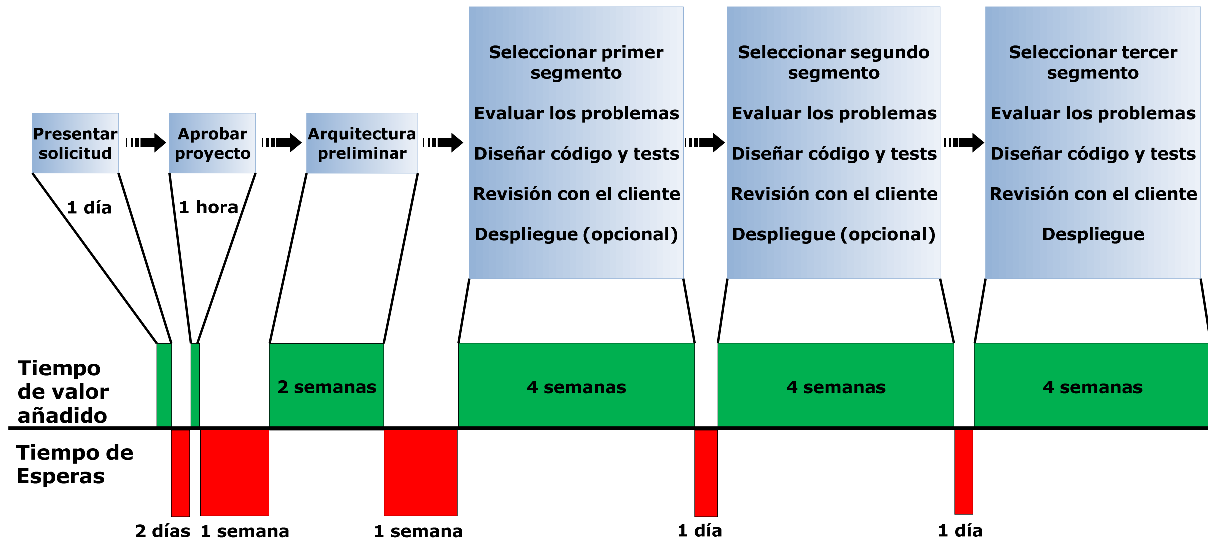


Ilustración 53. Mapa de flujo de valor (Value stream mapping) de un método ágil.

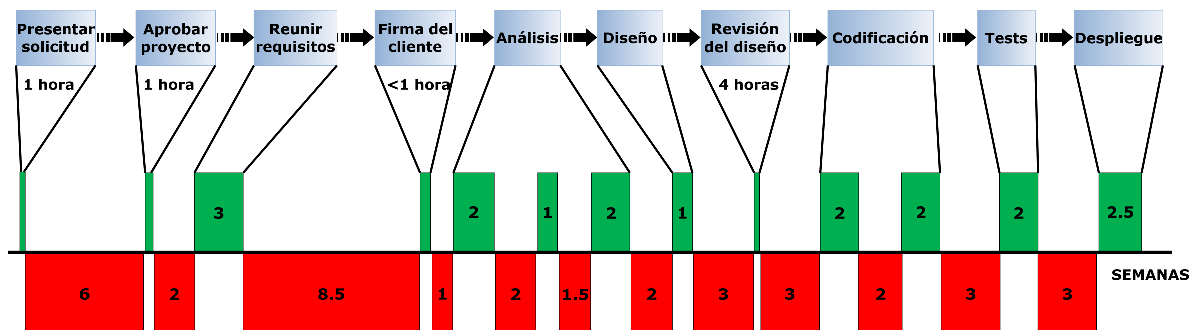


Ilustración 54. Mapa de flujo de valor de un método tradicional.

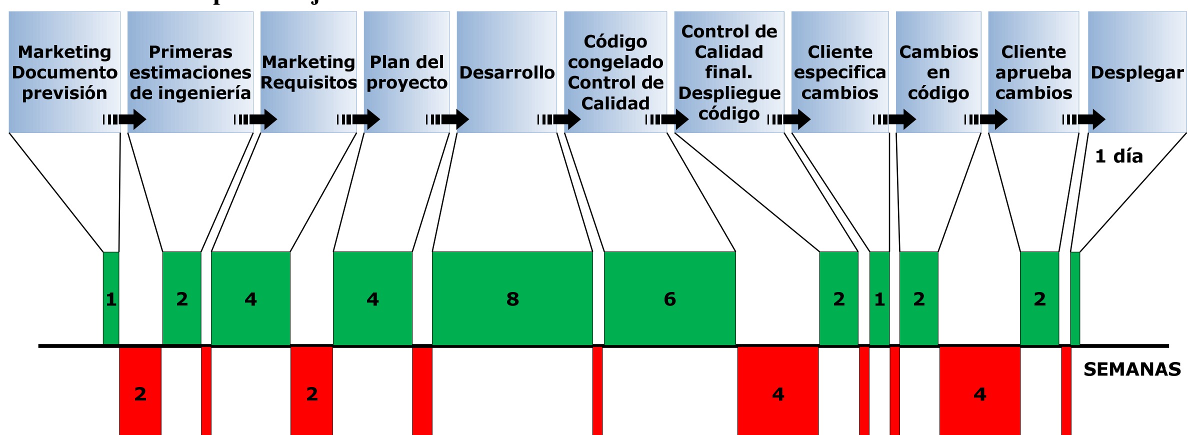


Ilustración 55. Mapa de flujo de valor de Kent Beck y su XP.

PRINCIPIO 2: AMPLIFICAR EL CONOCIMIENTO (*FEEDBACK*, ITERACIONES, SINCRONIZACIÓN, *DESARROLLO SET-BASED*)

El desarrollo se considera un ejercicio de descubrimiento gracias al *feedback*, aumentándolo cuando se encuentran problemas. Cuando una organización desarrolla software como un problema, hay una tendencia a imponer más disciplina, un proceso secuencial más riguroso donde:

- se documenten los requisitos muy concisamente,
- todos los acuerdos con el cliente queden por escrito,
- se controlen los cambios más cuidadosamente,
- cada requisito se debe implementar en el código,
- se use el control determinista en un ambiente dinámico, retrasando la respuesta del *feedback*.

Y todo esto, en general, hace que las cosas empeoren.

- HERRAMIENTA 3: FEEDBACK

- Cuando un problema crece, hay que asegurar que existen todos los lazos de *feedback*, y aumentar la frecuencia del *feedback* en las áreas que abarque el problema.

- En lugar de reunir más requisitos de los usuarios, muéstreles varias capturas de pantalla del programa y obtenga sus impresiones.

- En lugar de permitir que los defectos aumenten, ejecute los tests justo después de escribir el código.

- En lugar de añadir más documentación o detallar más la planificación, compruebe ideas escribiendo código.

- En vez de estudiar con detalle qué herramienta usar, pruebe las tres mejores candidatas.

- HERRAMIENTA 4: ITERACIONES

Un punto de partida universal para un enfoque ágil son las iteraciones:

- Los lazos de *feedback* frecuentes refuerzan el control.
- Las iteraciones son puntos de sincronización (equipo y cliente ven qué se ha hecho).
- Las iteraciones obligan a tomar decisiones.

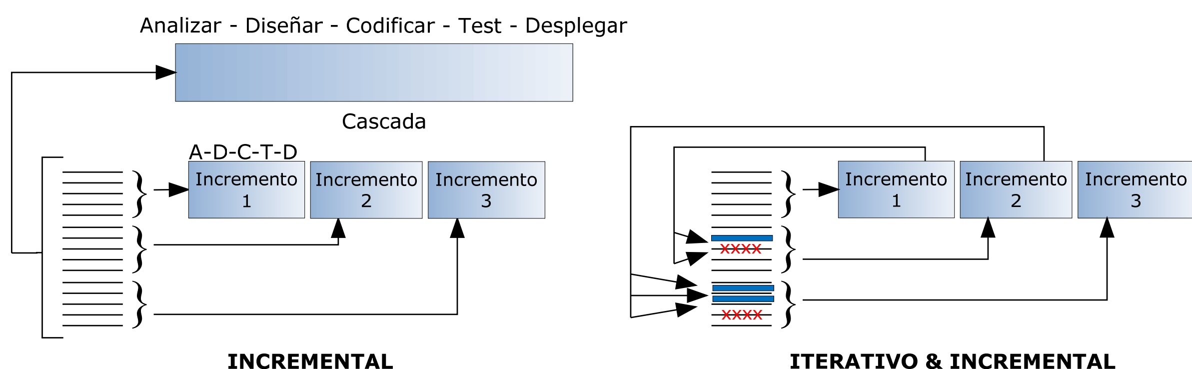


Ilustración 56. Método en cascada y método iterativo.

- HERRAMIENTA 5: SINCRONIZACIÓN

Siempre que varios individuos estén trabajando en una misma actividad, deben sincronizarse, especialmente si el proceso es complejo.

Integre diariamente dentro de cada equipo (el registro al menos diariamente en el almacén local).

Integre al menos una vez por semana los diferentes equipos (el registro al menos semanalmente en el "almacén central").

Las *build* más frecuentes proporcionan *feedback* mucho más rápido. Las *builds* y sus tests deben automatizarse.

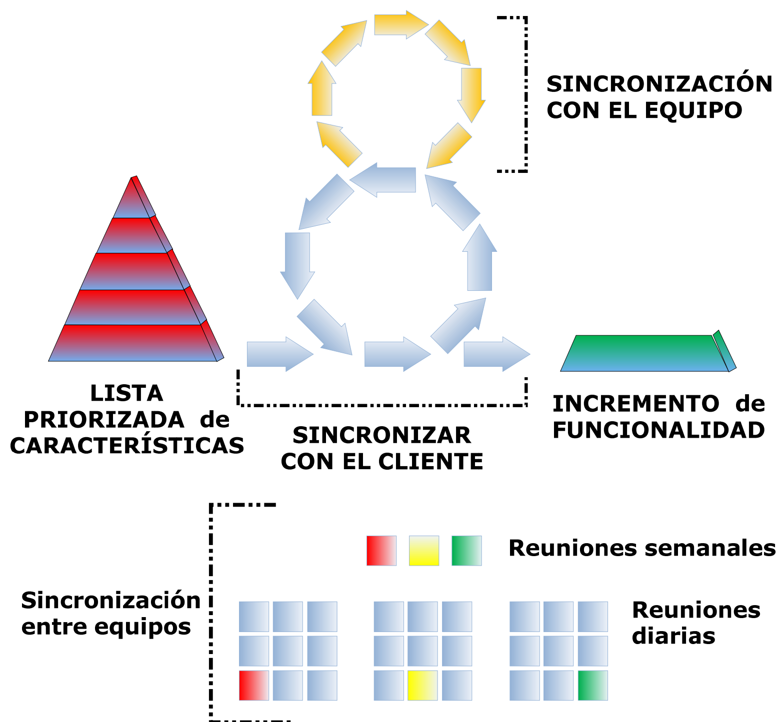


Ilustración 57. Sincronización total en LSD.

- HERRAMIENTA 6: DESARROLLO SET-BASED

En este tipo de desarrollo, la comunicación trata de las limitaciones, no de las opciones. Así se retrasa la toma de decisiones hasta que tengan que ser tomadas. Elabore varias opciones, comunique las limitaciones y deje que surjan las ideas.

Cuando tenga un problema difícil:

- desarrolle un conjunto de soluciones alternativas al problema,
- mire cuán bien funcionan, y
- fusione los puntos fuertes de todas ellas o escoja una de las alternativas.

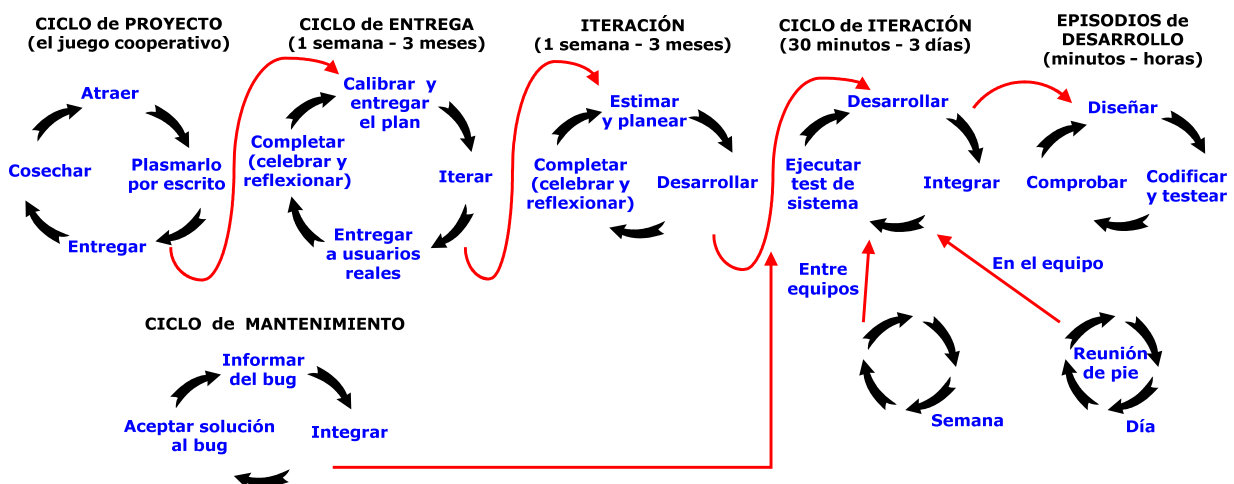


Ilustración 58. Ciclo completo de desarrollo de un proyecto aplicando el método Lean.

PRINCIPIO 3: DECIDIR TAN TARDE COMO SEA POSIBLE (OPTIONS THINKING, THE LAST RESPONSIBLE MOMENT, MAKING DECISIONS)

Esperar a tomar decisiones es conveniente en todos los dominios que haya incertidumbre porque brindan una estrategia basada en opciones fundadas en la realidad, no en especulaciones. En un entorno que cambia, la decisión tardía, que mantiene las opciones abiertas, es más eficiente que un compromiso prematuro. En otras palabras, **no hay que planificarlo todo antes de comenzar**. En un entorno cambiante, los requisitos detallados corren el riesgo de estar equivocados.

- Establecer los requisitos antes de que el desarrollo empiece normalmente es una manera de protegerse contra los errores graves. El problema del desarrollo secuencial es que obliga a los programadores a centrarse mucho en algo concreto en lugar de buscar una visión global.

- La manera más fácil de cometer graves errores es ir a los detalles demasiado rápido.

- Cuando hay riesgo de errores graves, es mejor inspeccionar el conjunto y retrasar las decisiones específicas.

- Con el desarrollo concurrente (coexistente, en paralelo), se empiezan a programar las características de mayor valor según el plan conceptual de alto nivel, mientras todavía se están definiendo los requisitos de los detalles.

- El desarrollo coexistente requiere programadores especializados para anticiparse a los problemas y colaborar con los clientes y analistas.

- HERRAMIENTA 7: “OPTIONS THINKING”

- Normalmente no damos a nuestros clientes la opción de cambiar sus ideas. Las opciones permiten aprender y tomar decisiones basadas en hechos, no en especulaciones.

- Fijar un plan precozmente restringe el aprendizaje, enfatiza el impacto de los defectos, limita la utilidad del producto, y aumenta el costo de cambiar.

- Los planes y predicciones no son malos, pero debe evitarse tomar decisiones irrevocables basadas en la especulación. Es mejor desarrollar opciones, y comunicárselas al cliente para que decida.

- Se necesita experiencia para saber qué opciones mantener abiertas.

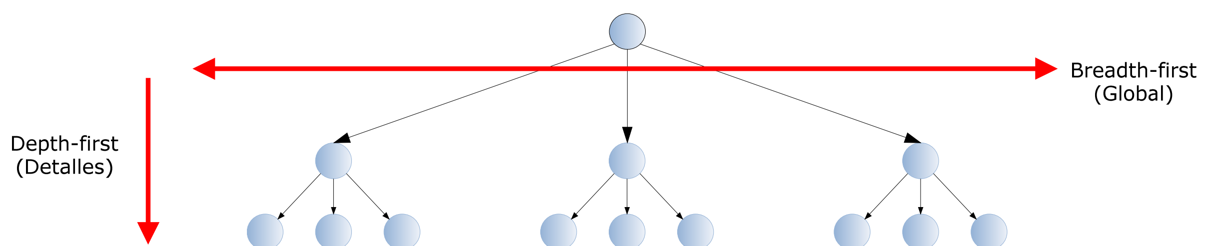


Ilustración 59. Enfoques Breadth-first (global) y Depth-first (detalles).

- HERRAMIENTA 8: EL ÚLTIMO MOMENTO RESPONSABLE

- El desarrollo concurrente permite retrasar los compromisos hasta el último momento responsable, es decir, hasta el momento en el que no tomar una decisión elimina una alternativa importante.

- Comparta el trabajo parcialmente completado: utilice el lazo de *feedback*. Un buen trabajo es un proceso de descubrimiento, hecho a través de ciclos exploratorios breves y repetitivos.

- Desarrolle un sentido sobre cómo aceptar los cambios (usar componentes con interfaces o la abstracción, evitar la repetición, encapsular las variaciones, evitar funciones extras).

- Desarrolle un sentido de lo que realmente sea importante en el dominio.

- Desarrolle un sentido sobre cuándo deben tomarse las decisiones.

- HERRAMIENTA 9: TOMAR DECISIONES

- Tomar decisiones de forma **intuitiva** es un enfoque más maduro que la forma **racional**, y normalmente también lleva a mejores decisiones.

- Tomar decisiones de forma **racional** implica descomponer un problema, quitando el contexto, aplicando técnicas analíticas, y exponiendo el proceso y resultados a la discusión. Tiene el problema de la visión túnel al ignorar los instintos de personas con experiencia.

- Es importante tener personas experimentadas que tomen sabias decisiones.

PRINCIPIO 4: ENTREGAR TAN RÁPIDO COMO SEA POSIBLE (*PULL SYSTEMS,* TEORÍA DE COLAS, COSTO DEL RETRASO).

- Se deben favorecer ciclos cortos de diseño, implementación y *feedback*. **El cliente recibe lo que necesita hoy, no lo que necesitaba ayer**. No significa apresurarse y hacer un trabajo mal hecho, sino dar valor a los clientes en cuanto lo pidan.

- A los clientes les gusta una entrega rápida que a menudo se traduce en una mayor flexibilidad comercial. Las compañías pueden entregar antes de que el cliente cambie de idea y tienen menos recursos atados durante el desarrollo.

- El principio *Entregar tan Rápido como sea Posible* complementa a *Decidir tan Tarde como sea Posible*: Cuanto más rápido pueda entregar, más podrá retrasar las decisiones. Por ejemplo, si usted puede hacer un cambio en el software en una semana, entonces no tendrá que decidir exactamente qué va a hacer hasta una semana antes de que se necesite un cambio.

- Las entregas rápidas le permiten guardar sus opciones abiertas hasta que haya reducido la incertidumbre y pueda tomar decisiones basándose en hechos.

- HERRAMIENTA 10: SISTEMAS DE ARRASTRE (PULL SYSTEMS)

- Debe estar claro para todo el mundo y en todo momento, qué debe hacer cada uno para contribuir de la forma más eficaz.

- Puede decirles qué hacer, Mandar & Controlar *(Command & Control)*, o dejar que ellos lo deduzcan, *auto-organización*. En un ambiente cambiante, sólo la segunda forma funciona.

- Para una auto-organización eficaz, deben definirse maneras para comunicarse, ya que ningún plan se adaptará perfectamente cuando las condiciones varíen mucho.

- Los radiadores de información o mecanismos de *feedback*: Una de las características de un sistema de este tipo es el control visual, o *gestión a la vista*. Todos deben poder ver qué está pasando, qué debe hacerse, qué problemas hay y cuánto se progresa.

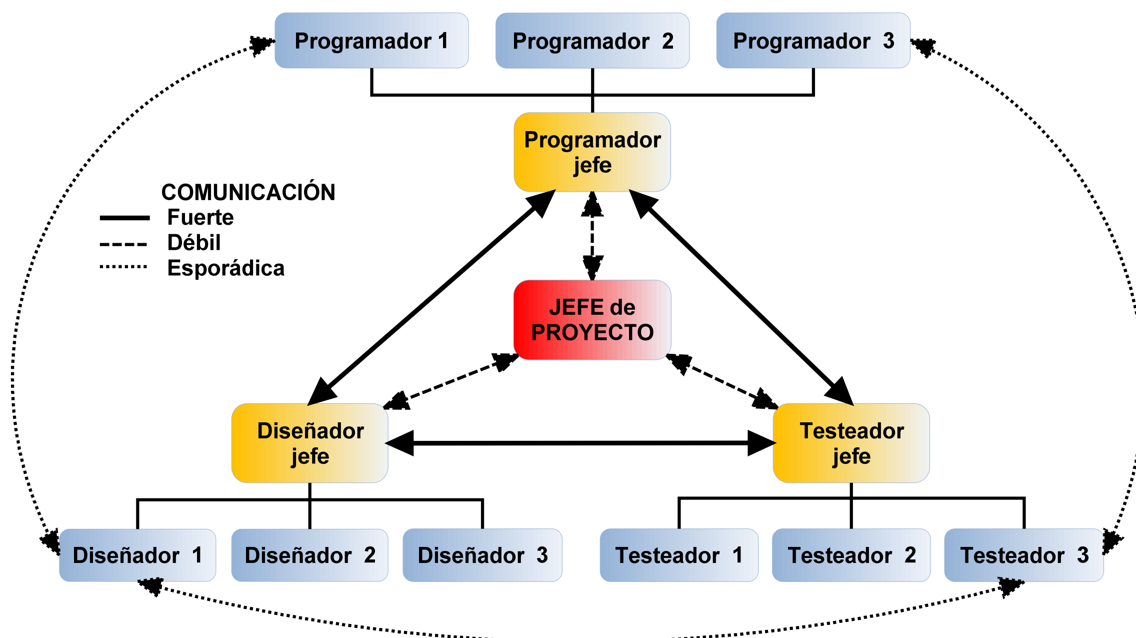


Ilustración 60. Método rígido Mandar & Controlar (Command &Control), contrario a Scrum.

- HERRAMIENTA 11: TEORÍA DE COLAS

- La medida fundamental de una cola es el **tiempo del ciclo**, y cuanto menor sea, mejor.

- Tasa de llegadas constante: cuando las llegadas llegan uniformemente, no en ráfagas, el tiempo de espera se acorta. Una manera de controlar la tasa de llegada de trabajo es entregar el trabajo en pequeños incrementos.

- Tiempo de servicio constante: La manera más fácil de quitar la variabilidad del tiempo que se tarda en servir una petición es aumentar el número de servidores. Trabajar con paquetes pequeños permitirá que los equipos procesen en paralelo.

- Inactividad (*slack*): Los tiempos de ciclo cortos no son posibles si se cargan excesivamente los recursos. Un uso al máximo no proporciona valor; normalmente será perjudicial.

- HERRAMIENTA 12: COSTO DEL RETRASO

- Cree un modelo económico simple (cuenta de pérdidas y ganancias) de un nuevo producto para los próximos años. Consiga estimaciones sobre los retrasos en las ventas y la cuota de mercado. El gráfico mostrará la relación entre las ventas, la cuota de mercado y las ganancias. Si el retraso significa "pérdidas iniciales por precio alto" o "pérdidas a largo plazo por cuota de mercado", el costo del retraso puede ser alto.

- Mantenga el modelo simple, asegúrese que todos entienden y compran en el modelo económico. Pida ayuda de sus contables si fuera necesario.

- Los modelos económicos pueden ayudarle a justificar el costo de reducir el tiempo del ciclo, eliminar cuellos de botella, y adquirir herramientas que le permitan entregar tan rápido como sea posible.

- Use el modelo económico de su negocio para resolver las decisiones de desarrollo.

- Dé un modelo económico al equipo, para deducir lo que es importante para la empresa. Todos tienen la misma información.

Modelo de la aplicación

- Si no está en el desarrollo de un nuevo producto, desarrolle un modelo para la aplicación desde el punto de vista del cliente. Una mirada simple al modelo económico del cliente ayuda al equipo a tomar las decisiones de intercambio de aplicación.

- Primero, identifique los factores económicos de su cliente relacionados con la aplicación, como por ejemplo, niveles de contratación de personal, ayuda del sistema, satisfacción del cliente o software del *call-center*.

- Segundo, traduzca los factores a expresiones económicas.

- Entonces, comprenda los objetivos que su cliente quiere lograr con la aplicación.

- Finalmente, modele cada meta separadamente en su propia columna y compare cada uno con la base. Es mejor no hacer cálculos demasiado precisos; las estimaciones burdas son útiles.

PRINCIPIO 5: OTORGAR PODER AL EQUIPO (AUTODETERMINACIÓN, MOTIVACIÓN, LIDERAZGO, APTITUD)

Los programadores experimentados son quienes pueden tomar las decisiones más adecuadas. No significa abandonar el liderazgo, sino permitir a las personas que agregan valor usar todo su potencial.

La mayoría de los programas de mejora de la calidad (*MBO, TQM, ZeroDefects, Optimized Operations, Service Excellence, ISO9000, Total Improvement Program, Customers First...*) no cambiaron demasiado la manera de hacer el trabajo. De hecho, aumentaron la intensidad de factores que llevan al descontento en el trabajo (política, supervisión, documentación,

administración) en lugar de aumentar factores que contribuyen a la satisfacción del trabajo (logro, reconocimiento, responsabilidad).

El "*Pensamiento Lean*" o *Lean Thinking* da importancia a la inteligencia de los trabajadores de primera línea, *frontline*, quienes deben determinar y mejorar continuamente su manera de trabajar.

Aunque el desarrollo del software no puede tener éxito sin gente disciplinada y motivada, la experimentación y *feedback* son más eficaces que intentar conseguir las cosas bien a la primera.

El factor crítico en la motivación no es ninguna medida, sino otorgar poder: mover las decisiones hasta el nivel más bajo de la organización y fomentar la capacidad de esas personas para tomar las decisiones de forma acertada.

En una *organización Lean*, las personas que agregan valor son el centro de energía orgánica. Los trabajadores de primera línea, tienen autoridad en el diseño del proceso y la responsabilidad de tomar decisiones; son el enfoque de recursos, información y preparación.

- HERRAMIENTA 13: AUTODETERMINACIÓN

Es difícil implementar con éxito métodos de mejora, y aún más difícil hacer que duren. General Electrics tiene un programa de mejora exitoso, llamado "GE Work-Out":

- En un Work-Out, unos 50 trabajadores se reúnen dos o tres días y proponen ideas que los ayudarán a encontrarse mejor en sus puestos de trabajo, y propuestas específicas para suprimir procesos intermedios o llevar a cabo prácticas que aportarán valor más rápido.
- Los jefes deciden si aplicar o no cada propuesta, allí mismo o en un par de días.
- Quienes hacen las propuestas, serán responsables de llevarlas a cabo.
- Una combinación de herramientas simples, acción inmediata, y posibilitar que participe toda la compañía, hicieron de este método un éxito.
- Los trabajadores dicen a sus jefes cómo permitirlos hacer sus trabajos.
- Un Work-Out enseña a los jefes a escuchar a sus trabajadores, los ayuda a llevar a cabo sus ideas, y se asegura de que se implementen rápidamente.

- HERRAMIENTA 14: MOTIVACIÓN

Pensamientos de William McKnight (líder de 3M durante 1930-1950):

- *"Contrate personas buenas, y déjelos solos."*
- *"Si pone cercos alrededor de las personas, conseguirá ovejas. Dé a las personas la libertad que necesitan."*
- *"Anime, no critique. Deje que las personas sigan una idea."*
- *"Dé una oportunidad - ¡y rápido!"*

La motivación intrínseca requiere un sentimiento de pertenecer, de seguridad, de competencia, y de progreso (los pilares de la motivación):

✓ **Pertenecer**: Todo el mundo debe tener clara la meta y comprometerse a conseguirla; el equipo debe ganar o perder unidos, como un equipo.

✓ **Seguridad**: No mate la motivación con "una mentalidad de cero defectos".

- ✓ **Competencia**: Es muy motivador ser parte de un equipo ganador, pero muy desmotivador si el fracaso es inevitable. Un ambiente de trabajo indisciplinado no genera un sentimiento de libertad sino de perdición. Un sentido de competencia viene del conocimiento y habilidad, *feedback* positivo, valores morales elevados, y de encontrarse con un desafío difícil. Un líder que delega y confía en los trabajadores debe asegurarse que el equipo va por el buen camino y guiarlo.

- ✓ **Progreso**: Necesidad de sentir haber logrado algo para reafirmar el propósito. En cada iteración, el equipo pone sus esfuerzos ante los clientes y averigua cómo se ha hecho. Cuando el equipo alcanza un objetivo particularmente importante, es hora de una celebración.

- HERRAMIENTA 15: LIDERAZGO

Líderes respetados:

- Detrás de un equipo apasionado, hay un líder apasionado.
- Escriben el concepto inicial, establecen el ritmo del desarrollo y determinan cómo tomar las decisiones.
- No tienen autoridad directa sobre las personas que trabajan en el proyecto.

Master developers (llamados ingenieros de sistemas, programadores jefes o arquitectos):

- En la mayoría de los grandes sistemas, un individuo o pequeño equipo de programadores virtuosos asume la responsabilidad principal del diseño. Ejercen la dirección gracias a su experiencia, no porque les hayan cedido autoridad.
- Están sumamente familiarizados con el dominio de la aplicación y la tecnología, y tienen experiencia en comunicar su perspectiva técnica.
- Entienden a los clientes y al resto de programadores, las limitaciones del sistema, las interacciones, los requisitos no expresados, condiciones de excepción e intuyen los posibles cambios.
- Miran el sistema con un nivel bastante alto de abstracción, aunque conocen y entienden los detalles que el cliente quiere.
- Son el punto central de comunicación, y parte del equipo, involucrándose en los detalles del trabajo. Mantienen el liderazgo necesario para tomar las decisiones acertadas, progresar rápido, y desarrollar software de calidad.

JEFES: HACEN FRENTE A LA COMPLEJIDAD	LÍDERES: HACEN FRENTE A LOS CAMBIOS
- El plan y Presupuesto - Organización y personal - Seguimiento y control	- Determinar el camino - Alinear las personas - Motivar

- HERRAMIENTA 16: APTITUD, COMPETENCIA (*EXPERTISE*)

- Un poco de conocimiento puede escribirse (código) y compartirse con documentación, pero mucho conocimiento es conocimiento tácito que sólo se compartirá hablando.
- Las normas normalmente las elabora una comunidad competente o, cuando sea necesario, el equipo del programa.
- Normalmente es mejor trabajar con normas existentes que desarrollar unas propias.
- Donde las normas parezcan fallar y el trabajo sea de poca calidad, sería conveniente promover comunidades competentes y pedirles que desarrollen normas.

- Los programadores aprecian las normas razonables, sobre todo si tienen influencia en ellas y se mantienen actualizadas.

PRINCIPIO 6: INTEGRIDAD INCORPORADA (INTEGRIDAD PERCIBIDA, INTEGRIDAD CONCEPTUAL, REFACTORING, TESTEO).

La integridad conceptual significa que los conceptos del sistema trabajan en una arquitectura coherente y en armonía. La integridad proviene del liderazgo, la experiencia relevante, la comunicación efectiva y la disciplina saludable. Los procesos, los procedimientos y las mediciones no son substitutos adecuados para obtener integridad. La integridad no implica hacer un gran diseño en cascada, sino en no esperar al final para integrar.

La **integridad externa (percibida)** significa que la totalidad del producto logra un equilibrio de función, utilidad, fiabilidad, y economía que encantan a clientes. La medida de integridad percibida es aproximadamente equivalente a la cuota de mercado, o quizás un término mejor podría ser *mindshare* en vez de *marketshare*.

La **integridad interior (conceptual)** significa que los conceptos centrales del sistema trabajan con cohesión, sin problemas. La integridad conceptual es un requisito para la integridad percibida. Surge conforme el sistema evoluciona y madura.

La manera de construir un sistema con alta integridad conceptual y percibida es que haya comunicación entre cliente y desarrollo, y también entre los procesos del desarrollo.

- HERRAMIENTA 17: INTEGRIDAD PERCIBIDA

- Para lograr la integridad percibida los ingenieros jefes deben entender a los clientes.

- Los clientes valorarán un buen diseño cuando lo vean, pero no pueden preverlo de antemano. Si cambian las circunstancias, también lo harán sus percepciones sobre la integridad.

- Un diseño orientado a modelos, *model-driven*: Los modelos del dominio deben entenderse y ser utilizables directamente por el cliente y los programadores.

- La mejor manera de mantener el "conocimiento institucional" de un sistema para un buen mantenimiento es adjuntar al código una serie de pruebas automatizadas y un modelo a alto nivel actualizado. Los programadores no suelen utilizar la documentación inicial de diseño, pues pocas veces coincide con lo que después se construye.

- HERRAMIENTA 18: INTEGRIDAD CONCEPTUAL

- Se logra integridad conceptual cuando los componentes encajan y funcionan bien juntos; la arquitectura logra un equilibrio eficaz entre la flexibilidad, mantenimiento y eficacia.

- La clave para lograr la integridad conceptual es una comunicación efectiva entre los equipos cuando se toman decisiones.

- Usar partes existentes quita muchos grados de libertad, reduciendo la complejidad y la necesidad de comunicación.

- Usar una solución integrada para resolver problemas; para asegurar el flujo de información técnica:
  - Entender el problema y resolverlo al mismo tiempo, no secuencialmente.
  - No hay que esperar a tener la información completa para empezar a entregarla.
  - La información se transmite con frecuencia y en pequeñas series.
  - La información fluye en dos sentidos, no uno.
  - Se prefiere transmitir la información cara a cara, no con documentos.

- Predecir el futuro suele ser una pérdida de tiempo y recursos. Es mejor hacer un primer acercamiento global, *breadth-first*, y entender los fundamentos bien. Permita que surjan detalles y planee refactorizaciones regulares.

¿Cómo asegurar que surja una buena arquitectura?

1) Usar partes existentes y software ya disponible en el mercado (COTS, *Commercial off-the-shelf*) cuando sea posible.
- Arreglando tantos puntos del sistema como sea factible con software y normas existentes, se requiere menos comunicación, y despeja el camino para la comunicación del resto del sistema.

2) Resuelver los problemas de forma integrada:
- Empezar a programar antes de tener todos los detalles diseñados.
- Mostrar el software parcialmente completado a clientes y usuarios para conseguir *feedback*.
- Los programadores deben tener acceso a los clientes para resolver sus preguntas.
- Ejecute las pruebas de *"usabilidad"* cada vez que acabe alguna parte.
- Cree y ejecute las pruebas del cliente en cada iteración, no sólo al final.

3) Asegurarse de que haya programadores experimentados en todas las áreas críticas.

4) Los sistemas complejos requieren el liderazgo de un *Master Developer*.

- HERRAMIENTA 19: REFACTORING

- Se empieza con algo que funciona, se buscan debilidades, y se mejora el diseño. "Se necesitan de cinco a seis intentos para conseguir el producto adecuado."

- Para mantener la integridad conceptual, se debe refactorizar cuando el sistema empiece a perder las siguientes características:
  - simplicidad,
  - claridad (convenciones de nomenclatura, lenguaje común, comentarios),
  - conveniencia o idoneidad del uso, y
  - ninguna repetición: cuando un cambio tiene que hacerse en más de un lugar, la probabilidad de error crece exponencialmente; la duplicación es un enemigo de la flexibilidad, es el principio DRY "No te repitas a ti mismo".

- Un buen diseño evoluciona durante la vida de un sistema, pero esto no pasa por accidente; el código pobre no mejora ignorándose:
  - no añadir funcionalidades cuando se descubra un fallo,

- encontrar la raíz del problema y arreglarlo antes de seguir, y
- no pasar demasiado tiempo perfeccionando detalles insignificantes.

- **Como la publicidad, el *refactoring* no cuesta, paga**.

- HERRAMIENTA 20: TESTEOS

- Distinguimos dos tipos:
  Developer tests: el programador se asegura que hace lo que pretende.
  Customer tests: comprueban lo que busca el cliente, durante todo el desarrollo.

- Si no hay suficiente tiempo, se reasigna el esfuerzo dedicado a documentar los requisitos, a escribir las pruebas del cliente.

- Las pruebas deben automatizarse tanto como se pueda y ejecutarse con la *build* diaria.

- Asegúrese de que las pruebas son correctas y completas, gestione sus versiones, considérelas parte de la entrega y continúe usándolas y mejorándolas.

- Mantenga un amplio conjunto de pruebas durante la duración del sistema. Así el sistema puede repararse y refactorizarse a lo largo de su vida útil.

- Si las pruebas están claras y bien organizadas, son un recurso inestimable para entender cómo trabaja el sistema desde el punto de vista de un diseñador y de un cliente.

PRINCIPIO 7: VER LA TOTALIDAD (MEDIDAS, CONTRATOS).

- No implica ignorar los detalles, sino tener cuidado con la tentación de perfeccionar detalles antes que la estructura general.

- Cuando un problema empeora, los jefes aplican aún más agresivamente las mismas políticas que están causando el problema.

- Un problema subyacente produce síntomas que no pueden ignorarse. Sin embargo, el problema subyacente es difícil confrontar, ya que las personas se dirigen a los síntomas y no a la raíz del problema. Los arreglos hacen que el problema subyacente crezca y pase inadvertido porque se han solucionado sus síntomas, pero no la causa.

- Los cinco “por qué”: dirigirse a la causa raíz, siga preguntándose por qué y no se pare cuando encuentre la primera respuesta razonable, ya que puede que sólo sea el síntoma. Todavía queda un “por qué” más a preguntarse antes de llegar a la raíz del problema.

- *"Yo tengo seis sirvientes fieles que me cuidan bien y acertadamente. Sus nombres son Qué, Dónde, Cuándo, Cómo, Por qué y Quién.”* – Kipling

- Sub-optimización:
  - Cuanto más complejo es un sistema, mayor tentación de dividirlo en partes y tratarlas localmente.
  - La gestión local tiende a crear medidas de rendimiento locales que a menudo disminuyen el rendimiento global.

- HERRAMIENTA 21: MEDIDAS

- Perfeccionar cada tarea suele ser una estrategia global muy mala.

- Es muy difícil medir todo lo que es importante en el trabajo intelectual, sobre todo cuando cada esfuerzo es único y reina la incertidumbre.

- Si no puede medir las cosas importantes, con mediciones parciales es muy probable llegar a medidas sub-óptimas. Si no puede medir todo lo necesario para perfeccionar la meta global, gastará menos dinero sin medidas parciales sub-óptimas.

- La manera de estar seguro que todo se mide es la **agregación**, no la desagregación. Es decir, mueva las medidas un nivel arriba, no uno abajo.

- Deberían usarse las **medidas de información** (obtenidas agregando los datos para ocultar el rendimiento individual), no medidas de rendimiento.

- No rastree los defectos por programador: menos del 20% de los defectos de calidad están bajo el control del trabajador; el resto depende de la gestión.
  - No se buscan las raíces de los problemas al buscar defectos en los individuos.
  - Hay que animar a toda la organización para encontrar la causa de los defectos.

- HERRAMIENTA 22: CONTRATOS

- Los jefes de proyecto tienen cuatro variables que pueden ajustar: **tiempo**, **coste**, **calidad**, y **posibilidades** o **alcance** *(scope)*. De estas cuatro variables, fije el tiempo, el costo y la calidad, pero no el alcance. Priorice las funciones, pero no especifique en el contrato el conjunto fijo de funcionalidades a entregar. Pase de un alcance fijo a uno negociable[23]: entregando primero las funciones prioritarias, aportará la mayoría del valor comercial antes de completar la lista de deseos del cliente.

- Según un estudio del Grupo Standish, el 45% de las funciones no se usan nunca (consultar el apartado 4.6.1 para un resumen de las estadísticas o los anexos para el estudio completo).

- Barry Boehm y Philip Papaccio concluyeron en 1988 que la mejor manera de desarrollar software de bajo coste y de calidad superior es escribir menos código.

- Un control inflexible del alcance tiende a extender, no a reducir el alcance. Ahorre dinero usando algún formulario de contrato de alcance opcional.

2.9.3. Papeles y responsabilidades

Dado que LSD es más una filosofía de gestión que un proceso de desarrollo, no se detalla con precisión el tamaño del equipo, la duración de las iteraciones, los papeles o las etapas que componen el proceso.

2.9.4. La práctica

Veamos las pautas para llevar a cabo los 7 principios de LSD mediante las 22 herramientas que propone Norton, mediante ejemplos concretos:

[23] Contratos de alcance opcional en www.xprogramming.com/ftp/Optional+scope+contracts.pdf

PRINCIPIO 1: ELIMINAR DESPERDICIOS

1) Haga una lista con las 10 o 15 actividades más importantes en su organización. Póngase en el lugar del cliente y valore de 1 a 5 cada actividad, otorgando un 1 a las actividades que no le preocupan y 5 a las que el cliente valora mucho. Las actividades con menos puntuación son desperdicios. Elija dos actividades de las que tengan menor puntuación y desarrolle un plan para reducir a la mitad el tiempo que se les dedica.

2) A sus próximas siete reuniones del equipo, tómese algún tiempo para discutir cada uno de los 7 desperdicios: inventario, procesos extra, funcionalidades extra, cambios de tarea, esperas, movimientos y defectos. Pregúntese:
- ¿Está de acuerdo que ese desperdicio realmente lo es? ¿Por qué?
- ¿Cuánto tiempo se desperdicia de media en una semana?
- ¿Qué puede hacerse para reducir ese tiempo?

3) Desarrolle un mapa de flujo de valor para su organización. Empiece con algún nuevo proyecto y trace un calendario de su progreso. Averigüe cuánto de ese tiempo se dedica a actividades que el cliente valora y cuánto se pierde en esperas. Coja la mayor causa de retraso y elabore un plan para reducir ese tiempo a la mitad.

PRINCIPIO 2: AMPLIFICAR EL CONOCIMIENTO

1) Coja su problema más difícil y busque una manera de aumentar el *feedback*.

1.1) Aumente el *feedback* que venga de los equipos de desarrollo, haciendo al final de cada iteración las siguientes preguntas a cada equipo:
- ¿El equipo estaba provisto adecuadamente de personal para esta iteración?
- ¿Alguien ha necesitado recursos que no estaban disponibles?
- ¿Cómo pueden cambiarse las cosas para que vayan mejor o más rápido?
- ¿Qué está funcionando bien?

1.2) Aumente el *feedback* de los clientes hacia los equipos de desarrollo al final de cada iteración preguntándose:
- ¿Cuán bien esta sección resuelve el problema?
- ¿Cómo podría mejorarse?
- ¿Cómo afecta esta iteración al producto final?
- ¿Qué necesita para poner esta parte del sistema en producción?

1.3) Aumente el *feedback* del producto al equipo de desarrollo:
- Tenga programadores para escribir el código y ejecutar las pruebas.
- Tenga analistas, clientes, o verificadores que escriban y ejecuten las pruebas del cliente mientras los programadores escriben el código.
- Ponga diseñadores para ayudar a los clientes mientras hacen las pruebas.
- Obtenga la opinión del cliente antes de acabar las funciones.

1.4) Aumente el *feedback* dentro del equipo:
- Integre a los probadores o *testers* en el equipo.
- Involucre al personal de operaciones desde el principio del proyecto.
- Establezca la política que el equipo de desarrollo mantiene el producto.

2) Empiece las iteraciones con una sesión entre clientes y programadores. Los clientes deben indicar qué características son prioritarias. Los programadores deben comprometerse a hacer esas funciones en el tiempo que dura el tiempo de una iteración, *time-box*.

3) Ponga un gráfico de progreso o avance del proyecto (gráficos de quemado) en una zona visible por todos para que el equipo vea lo que se ha de hacer y lo que ya está hecho.

4) Si divide un sistema en varios equipos, intente tener una arquitectura divisible que les permita trabajar tan independientemente como sea posible. Sincronice los equipos tan a menudo como sea posible para integrar sus códigos y ejecutar las pruebas automatizadas.

5) Si el trabajo de los equipos funciona a nivel de interfaz máquina, hágalo también con la interfaz de usuario.

6) Busque el mayor problema que haya y pida al equipo que piense tres maneras de arreglarlo. En lugar de escoger una, haga que exploren las tres al mismo tiempo.

PRINCIPIO 3: DECIDIR TAN TARDE COMO SEA POSIBLE

1) Piense en ejemplos de su vida donde ha usado las opciones para retrasar decisiones. ¿Por ejemplo, decidió pagar a interés fijo su hipoteca? ¿le ha beneficiado? Encontrará que la mayoría de los ejemplos han sido favorables o neutrales, pocos perjudiciales.

2) Pida a un equipo o departamento que enumere decisiones a tomar. Agrúpelas en dos categorías: fáciles y difíciles de tomar. Después discuta qué información necesitaría para convertir cada decisión difícil en fácil. Escoja tres decisiones difíciles y aplique la Herramienta 8 (el último momento responsable) para retrasar esas decisiones.

3) Evalúe su personalidad: ¿se decanta por mirar primero la globalidad, *breadth-first* o los detalles, *depth-first*? Busque a alguien que sea lo contrario, y trabaje con él sobre cómo afrontar el siguiente proyecto.

4) Seleccione algunos procesos críticos y piense reglas simples para que las personas entiendan su postura y puedan tomar decisiones independientes.

PRINCIPIO 4: ENTREGAR TAN RÁPIDO COMO SEA POSIBLE

1) Ponga en un lugar visible:
- El objetivo de la iteración actual,
  - lo que ya se ha hecho,
  - lo que se está haciendo, y
  - lo que todavía no se ha hecho.
- La misión del proyecto global, y
  - lo que ya se ha hecho para cumplir la misión del proyecto, y
  - lo que todavía tiene que hacerse para acabar la misión.

2) Al final de la próxima iteración, repase el proceso para saber si todos saben qué hacer. Pida al equipo que dedique su tiempo como quiera. ¿Qué les ayudaría a tomar decisiones de forma más rápida o mejor? Escoja las dos mejores ideas y llévelas a cabo en la próxima iteración.

3) Realice un gráfico con el tiempo que se tarda en cada cola. Busque patrones, ¿la variabilidad es alta o baja? ¿Hay una tendencia ascendente o descendente?

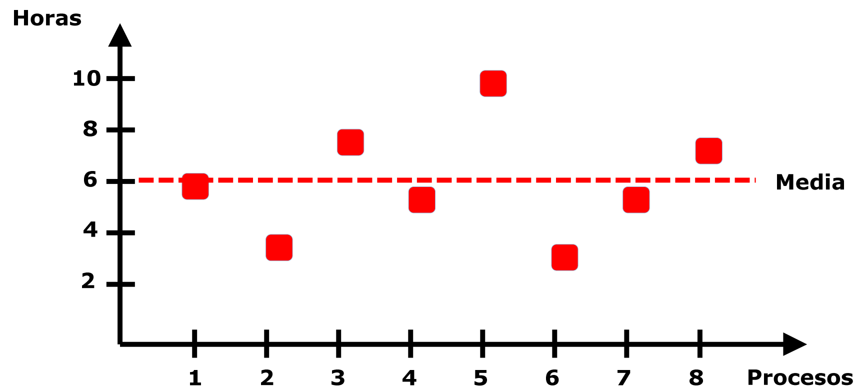


Ilustración 61. Tiempo de ciclo de los procesos.

4) Escoja la cola que representa su mayor cuello de botella e intente reducirla.

5) Pida al departamento de finanzas un modelo económico simple de cada equipo que refleje el costo del retraso, mantenimiento, funciones reducidas...

PRINCIPIO 5: OTORGAR PODER AL EQUIPO

1) Al final de cada iteración, revise el proceso con el equipo preguntándose:
- ¿Qué está retrasando el trabajo y qué está en la buena dirección?
- ¿Qué ayudaría a que las cosas fueran más rápido, mejor o fueran más baratas?

2) Haga una lista de prácticas malas y buenas. Decida qué puntos de la primera lista puede eliminar y cuáles de la segunda puede llevar a cabo. Llévelo a cabo. Repita esto después de cada iteración.

3) Asegúrese de que el equipo de desarrollo empieza cada iteración apuntando la meta de la iteración. La meta debe ser una o dos frases relacionadas con la iteración y el valor comercial que aportará. Coloque la meta en un lugar visible y úselo como fin cuando el equipo esté peleándose con una decisión difícil.

4) Use la programación por parejas, *pair programming* (comparte información) o diseñe revisiones dentro del *framework*. Si las revisiones del plan son buenas, asegúrese de resaltar el aprender y compartir en lugar de buscar errores.

5) Haga que cada miembro del equipo escriba el área donde el equipo es poco competente. Compare y elija con ellos el mejor candidato para proponer un plan y mejorar en esa área. Estrategias:
- Compre a todos los que lo pidan un libro sobre el tema y reúnase una vez por semana para discutir un capítulo.
- Consiga un gurú del tema que trabaje en parejas con ellos por un tiempo.
- Designe un comité de 3 personas para establecer las convenciones del equipo en esa área. Asegúrese de que evalúan cualquier norma, corporativa o de la industria, antes de crear otras.

PRINCIPIO 6: INTEGRIDAD INCORPORADA

1) Escoja uno de sus sistemas actuales y averigüe si tiene un lenguaje común (de calle). Hable con los clientes y cree un glosario de los términos que ellos consideran importantes al hablar sobre el sistema. Compare y combine este glosario con el vocabulario (técnico) del equipo de desarrollo. Pida a los programadores que identifiquen en el código los nombres que ellos usan para cada palabra en el glosario combinado. Finalmente, busque si existe alguna clase importante en los sistemas que no esté en el glosario. Si descubre que hay dos o tres vocabularios diferentes en uso, explique al equipo de desarrollo por qué es importante usar el lenguaje del dominio o área, incluso entre ellos.

2) Convoque una reunión invitando a cualquiera de las siguientes personas que normalmente no estarían allí, como por ejemplo las personas que se encargarán de:

- la prueba del sistema,
- despliegue o distribución del sistema,
- la formación de los usuarios,
- ser responsable para hacer funcionar el sistema en producción,
- el trabajo de *help-desk* para usuarios del sistema,
- mantener el sistema, y
- desarrollar o mantener cualquier sistema que acceda a los mismos datos.

Haga un *brainstorming* entre todos para conocer qué es lo que más les preocupa. Priorice los problemas y elija los 3 más importantes. Forme un comité mixto para tratarlos. Vuelva a reunirse en dos semanas para asegurarse que se han resuelto los problemas, y repita el proceso.

3) Ponga cinco hojas de papel en una pared de la sala del equipo. Etiquételas con:

- Simplicidad
- Claridad
- Conveniencia o idoneidad de uso
- Ninguna Repetición
- Ninguna función extra

Cada programador escribirá en un papel que corresponda algo del sistema actual que no cumpla esa norma. Por ejemplo, si se detecta repetición, anotarían a los culpables en la hoja "Ninguna repetición". Cuando al *refactorizar* se ha eliminado algún punto de la lista, se tacha. Al final de cada iteración, deje un día o dos para eliminar los peores puntos de estas listas.

4) Estime el tiempo medio del ciclo desde que se escribe una función hasta...

... ejecutar la prueba del programador.
... integrarla en el sistema y ejecutar las pruebas automatizada de los programadores.
... ejecutar la prueba del cliente.
... ejecutar la prueba de *"usabilidad"*.
... el despliegue o distribución.

Después, apunte un objetivo para el tiempo de ciclo de cada punto de la lista. Aborde esta lista de arriba a abajo: trabaje con el equipo para reducir el tiempo de cada punto hasta conseguir el tiempo propuesto.

PRINCIPIO 7: VER LA TOTALIDAD

1) Asegúrese que su sistema de medición de fallos es un sistema de medida informativo y no un sistema de medida de rendimiento.
    a. ¿Los defectos pueden identificar al programador que los causó? ¿Por qué? Si no hay ninguna buena razón, elimine la identidad de la persona del defecto; no guarde los nombres.
    b. Si hay una razón para conocer la identidad del programador (por ejemplo, si debe arreglar el código), entonces enseñe los informes que le atañen a él solo. Junte todos los informes de fallos; no los muestre públicamente ni los agrupe por programador.

2) Si contrata personal externo o es un contratista, el primer paso para usar los métodos ágiles bajo el contrato es deducir una manera de hacer el alcance opcional. Pida a su equipo legal protección adecuada a su compañía sin usar especificaciones de alcance fijado.

2.9.5. Adopción y experiencias

La familiaridad de muchas empresas con los principios de *Lean Production* & *Lean Manufacturing* ha facilitado la penetración en el mercado de su análogo en ingeniería de software. Existen abundantes casos de éxito documentados empleando LD y LSD, la mayoría en Canadá. Algunos de ellos son los de Canadian Pacific Railway, Autodesk y PowerEx Corporation. Se ha aplicado prácticamente a todo el espectro de la industria.

2.9.6. Limitaciones

Igual que *Agile Modelling*, que cubría sobre todo aspectos de modelado y documentación, LD y LSD han sido pensados como complemento de otros métodos, y no como una metodología excluyente a implementar en la empresa. LD prefiere concentrarse en las premisas y modelos derivados de *Lean Production*, que hoy constituyen lo que se conoce como el canon de la Escuela de Negocios de Harvard. Para las técnicas concretas de programación, LD promueve el uso de otros MA que sean consistentes con su visión, como XP o sobre todo *Scrum*.

2.10. Agile Modelling – AM, AMDD

Agile Modelling (AM) fue propuesto por Scott Ambler no tanto como un método ágil cerrado en sí mismo, sino como complemento de otras metodologías, ágiles o convencionales. Ambler recomienda su uso con XP, Microsoft Solutions Framework, RUP o EUP. En el caso de XP y MSF, los usuarios de AM podrían definir mejor los procesos de modelado que faltan en ellos, y en el caso de RUP y EUP, el modelado ágil permite hacer más ligeros los procesos que ya usan. AM es una estrategia de modelado (de clases, de datos, de procesos) pensada para contrarrestar la idea de que los métodos ágiles no modelan y no documentan. Podría definirse como un proceso de software basado en prácticas cuyo objetivo es orientar el modelado de una manera efectiva y ágil. AM actualmente se encuentra en su versión 2.

Agile Model Driven Development (AMDD) es la versión ágil de Model Driven Development (MDD). De forma resumida, sus ideas más destacables son:

1) Al principio del proyecto, se dedica cierto tiempo a hacer el modelo inicial, particularmente para conocer los requisitos fundamentales e identificar una posible arquitectura.

2) Durante los ciclos de desarrollo, típicamente se modelará durante varios minutos para luego implementar durante algunas horas o incluso días hasta necesitar hacer más modelos.

3) Las prácticas de AM de "Modelar con otros" y "Propiedad colectiva" proporcionan las revisiones necesarias.

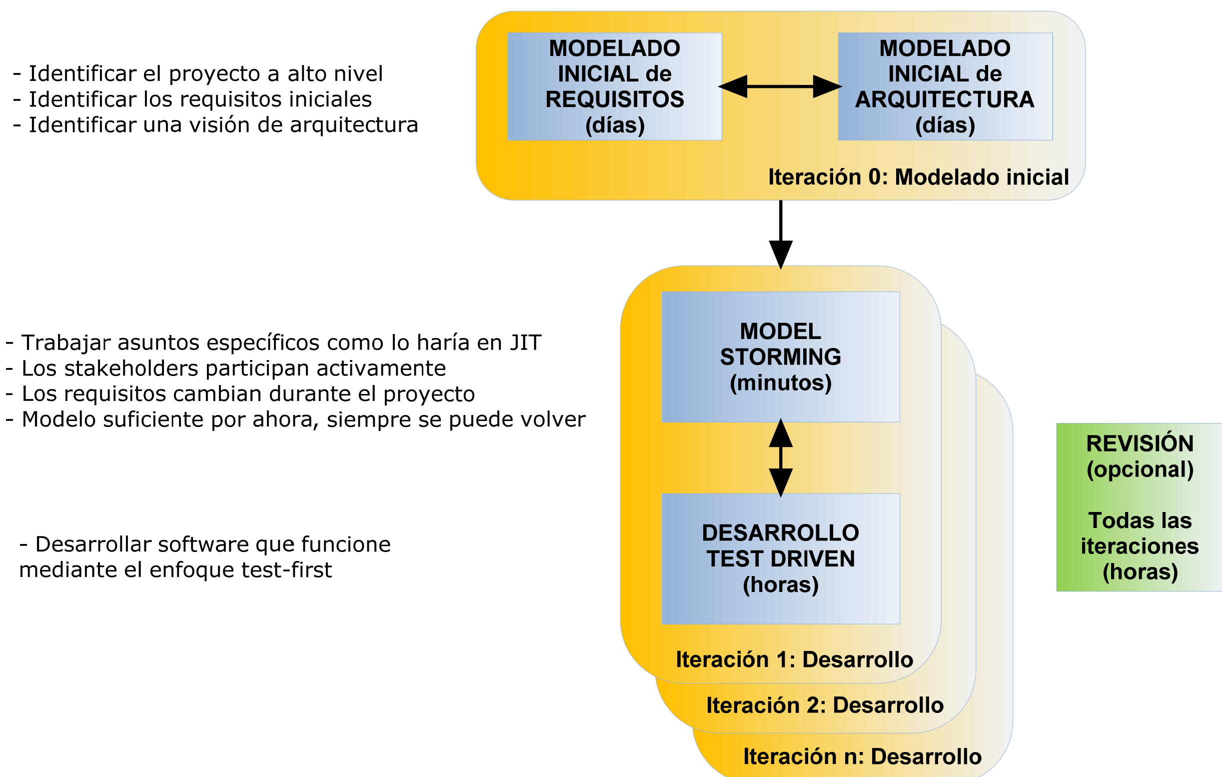


Ilustración 62. Funcionamiento general de Agile Model Driven Development.

2.10.1. Proceso

Los principales objetivos de AM son:

1) Definir y mostrar de qué manera se debe poner en práctica una colección de valores, principios y prácticas que conducen al modelado ágil o "de peso ligero".
2) Aplicar las técnicas de modelado en procesos de desarrollo ágiles.
3) Aplicar las técnicas de modelado independientemente del proceso de software usado.

Los valores de AM incluyen los de XP (1999): comunicación, simplicidad, *feedback* y coraje, añadiendo humildad. Los principios de AM dan lugar a la definición de sus alcances:

1) AM es una actitud, no un proceso preceptivo. Comprende una colección de valores a los que los modeladores ágiles adhieren principios en los que creen y prácticas que aplican. Describe un estilo de modelado; no es un recetario de cocina.
2) AM es un suplemento para otros métodos. El primer punto es el modelado y el segundo la documentación.
3) AM es una tarea de conjunto de los participantes. No hay "yo" en AM.
4) La prioridad es la efectividad. AM ayuda a crear un modelo o proceso cuando se tiene un propósito claro y se comprenden las necesidades de la audiencia; contribuye a aplicar los artefactos correctos para afrontar la situación inmediata y a crear los modelos más simples posibles.
5) AM es algo que funciona en la práctica, no una teoría académica. Las prácticas han sido discutidas desde 2001 en comunidad (www.agilemodeling.com/feedback.htm).
6) AM no es una píldora mágica que arregle todos los problemas si hay limitaciones y si no se cambia el enfoque.
7) AM es para el programador promedio, pero no reemplaza a la gente competente.
8) AM no es un ataque a la documentación, pero debe ser mínima y relevante.
9) AM no es un ataque a las herramientas CASE.
10) AM no es para cualquiera.

Los principios de AM especificados por Ambler incluyen:

1) **Adoptar la simplicidad**. La solución más simple es la mejor.
2) **Abrazar el cambio**. Aceptar que los requisitos cambian.
3) **Software que funcione es el primer objetivo**. Debe ser de alta calidad y coincidir con lo que el usuario espera.
4) **Habilitar el esfuerzo siguiente es el segundo objetivo**. Garantizar que el sistema es suficientemente robusto para admitir mejoras posteriores.
5) **Cambio incremental**. No esperar hacerlo bien la primera vez.
6) **Maximizar el papel del cliente**.
7) **Modelar con un propósito**. Si no se puede justificar para qué se está haciendo algo, ¿para qué molestarse?
8) **Modelos múltiples**. Múltiples paradigmas en convivencia, según se requiera.
9) **Trabajo de calidad**.
10) ***Feedback* rápido**. No esperar a que sea demasiado tarde.
11) **Viajar ligero de equipaje**. No crear más modelos de los necesarios.

Y dos principios complementarios:

12) **El contenido es más importante que la representación**. Pueden ser notas, pizarras o documentos formales.
13) **Comunicación abierta y honesta**.

2.10.2. Papeles y responsabilidades

Como AM se debe usar como complemento de otras metodologías, nada se especifica sobre métodos de desarrollo, tamaño del equipo, papeles, duración de iteraciones, trabajo distribuido y criticidad; todo eso dependerá del método que se utilice.

Los principios y prácticas pueden dar impresión de puro sentido común, sólo que en variante *caórdica* (combinación de caos y orden). En realidad, los aspectos pertenecientes al uso de herramientas de modelado y documentación, así como la articulación con metodologías específicas, las técnicas de bases de datos y el uso de artefactos están especificados cuidadosamente, como se puede constatar en el sitio web de AM y en los textos de Ambler.

Los diagramas de UML y los artefactos del Proceso Unificado, por ejemplo, han sido explorados en extremo detalle describiendo cómo debería ser su tratamiento en un proceso ágil EUP, *Enterprise Unified Process*, regido por principios *caórdicos*. EUP combina UP y AM. Se han documentado algunos estudios de casos de AM en proyectos de tamaño medio. Los métodos RUP, EUP y AgileUP o AUP están desarrollados en el apartado 2.5.

2.10.3. La práctica

AM propone unas prácticas principales:

1) Colaboración activa de los participantes.
2) Aplicación de los artefactos correctos.
3) Propiedad colectiva de todos los elementos.
4) Crear diversos modelos en paralelo.
5) Crear contenido simple.
6) Iterar a otro artefacto cuando nos quedemos bloqueados.
7) Modelo en incrementos pequeños.
8) Modelar con otros miembros del equipo.
9) Demostrarlo con código.
10) Utilizar las herramientas más simples (CASE, o mejor pizarras, tarjetas, post-it).
11) Exhibir públicamente los modelos para que los vea todo el equipo.
12) Diseñar modelos de manera simple.
13) Guardar la información en un único sitio y sin repetirla.

Y también unas prácticas complementarias:

14) Aplicación de estándares de modelado.
15) Aplicación adecuada de patrones de modelado.
16) Descartar los modelos temporales.
17) Formalizar modelos de contrato.
18) Actualizar sólo cuando lo que se estaba haciendo era perjudicial.

Artefacto	Soporte	Descripción
Diagrama de actividad	Pizarra blanca	Durante el análisis de modelado, se usan diagramas de actividad de UML para explorar la lógica del escenario, casos de uso, etc.
Diagrama de clases	Pizarra blanca	Los diagramas de clases muestran las clases con sus interrelaciones (incluyendo herencia, agregación, asociación), y las operaciones y atributos de las clases.
Definición de limitaciones	Tarjetas (Index card)	Una limitación *(constrain)* es una restricción de los grados de libertad para dar una solución. Las limitaciones son requisitos globales del proyecto.
Modelo CRC	Tarjetas	Un modelo Class-Responsibility-Collaborator es una colección de tarjetas índice estándar, cada una de las cuales ha sido dividida en 3 partes, indicando el nombre, las responsabilidades y los colaboradores de la clase. Se usan durante el modelado de análisis para el modelado conceptual.
Data flow diagram (DFD)	Pizarra blanca	Un DFD muestra el movimiento de los datos en un sistema, entre procesos, entidades y discos.
Entity/Relationship (E/R) diagram (data diagram)	Pizarra blanca	Los diagramas ER muestran las entidades principales, sus atributos, y las relaciones entre entidades. Igual que los diagramas de clases, pueden usarse para modelado conceptual.
Flow chart	Pizarra blanca	Los organigramas o cuadros sinópticos se usan de forma similar a los diagramas de actividades.
Diagramas de robustez	Pizarra blanca	Los diagramas de robustez se usan para analizar escenarios de uso e identificar clases candidatas y los principales elementos de interfaz de usuario (pantallas, informes,...).
Diagrama de secuencia	Pizarra blanca	Los diagramas de secuencia se usan para modelar la lógica de los escenarios de uso.
Diagrama de gráfico de estado	Pizarra blanca	Representan los diferentes estados y las transiciones entre ellos que muestra una entidad. Modelan el ciclo de vida de entidades con comportamientos complejos.
Caso de uso del sistema	Papel	Es una secuencia de acciones de un actor. Incluye decisiones de alto nivel, como componentes de páginas HTML o informes. También reflejan decisiones fundamentales sobre la arquitectura, como usar ADSL o teléfono móvil para acceder a la cuenta bancaria
Prototipo de Interfaz de Usuario	Pizarra blanca	Modelan la interfaz de usuario del sistema. Para la mayoría de la gente, la interfaz es el sistema.
Escenario de uso	Tarjetas	Es la descripción de una posible manera de hacer usar el sistema.
Diagramas de casos de uso	Pizarra blanca	Muestra una colección de casos de uso, actores, sus asociaciones y opcionalmente un límite del sistema o el alcance de las diferentes versiones.

Tabla 20. Posibles artefactos para el modelado de análisis.

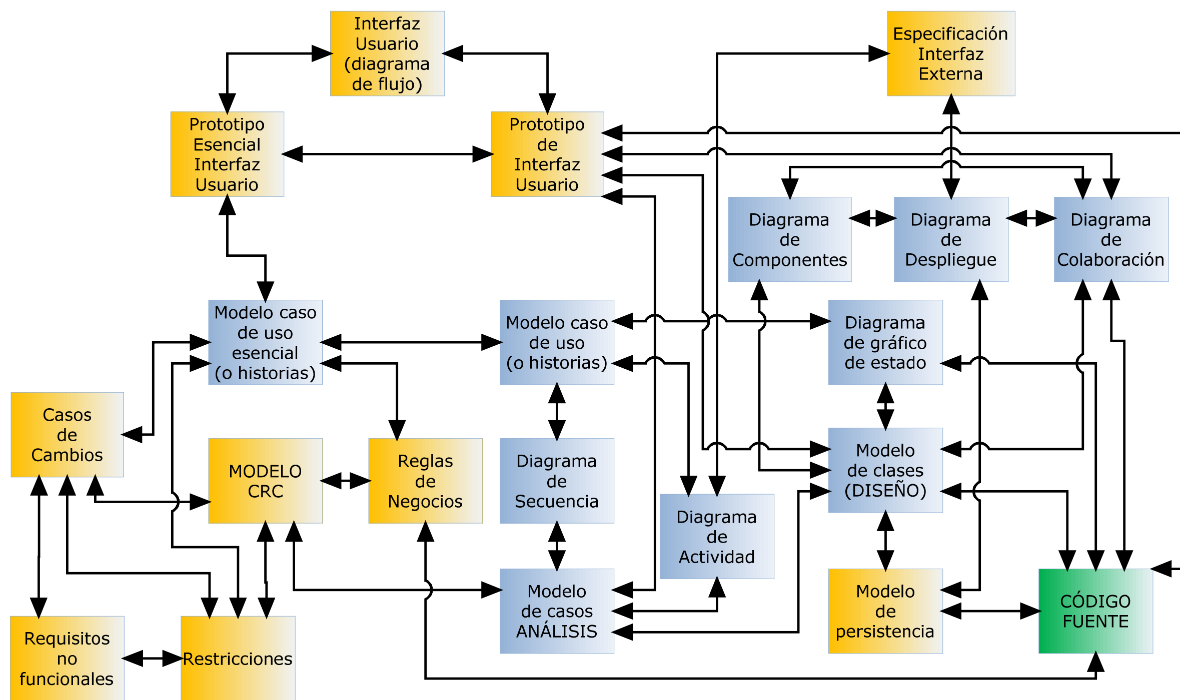


Ilustración 63. Diagrama de artefactos de Agile Modeling / EUP.

2.11. Evolutionary Project Management - Evo

Evo, creado por Tom Gilb[24], es el método iterativo ágil más antiguo, elaborado inicialmente en Europa. También es conocido como *Evolutionary Delivery*, *Evolutionary Management*, *Requirements Driven Project Management* y *Competitive Engineering*. En 1976, Gilb trató temas de desarrollo iterativo y gestión evolutiva en su clásico *Software metrics*, el texto que inventó el concepto e inauguró el campo de las métricas de software. Luego desarrolló en profundidad esos temas en una serie de columnas en *Computer Weekly UK*. En 1981 publicó *Evolutionary Development* en *ACM Software Engineering Notes* y *Evolutionary Delivery versus the 'Waterfall Model'* en *ACM Sigsoft Software Requirements Engineering Notes*.

En la década de 1980, Gilb cayó bajo la influencia de los valores de W. Edward Deming y del método **Planear-Hacer-Estudiar-Actuar** (PDSA) de Walter Shewhart, que se constituyeron en modelos conceptuales subyacentes a Evo. Deming y Shewhart son considerados los padres del control estadístico de calidad; sus ideas, desarrolladas en las décadas de 1940 y 1950, se han aprovechado, por ejemplo, en la elaboración de estrategias como Six Sigma[25], en *Lean Development* o en la industria japonesa.

En los años 90, Gilb continuó el desarrollo de Evo, que tal vez fue más influyente por las ideas que éste proporcionara a XP, *Scrum* e incluso UP que por su éxito como método particular. El texto *Rapid Development* [McConnell 1996], que examina prácticas iterativas e incrementales para desarrollo de software, menciona ideas de Gilb en 14 secciones; todas esas referencias están ligadas a *best practices*.

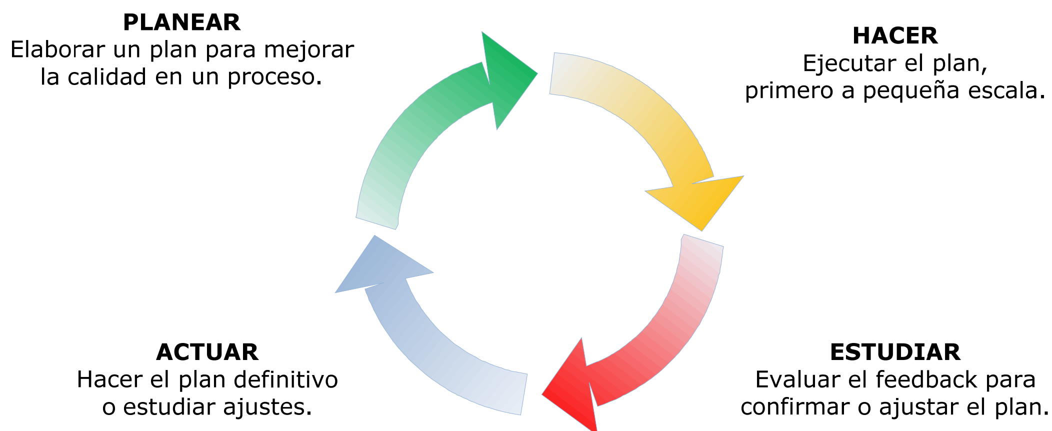


Ilustración 64. Método PSDA: Planear-Hacer-Estudiar-Actuar.

En las breves iteraciones de Evo, se efectúa un progreso hacia las máximas prioridades definidas por el cliente, entregando algunas partes útiles para algunos participantes y solicitando su *feedback*. Esta es la práctica que se ha llamado ***Planteamiento Adaptativo Orientado al Cliente*** y ***Entrega Evolutiva***. Otra idea distintiva de Evo es la clara definición, cuantificación, estimación y medida de los requisitos de rendimiento que necesitan mejoras.

[24] www.gilb.com

[25] Six Sigma es una metodología de gestión de calidad, centrada en el control de procesos y cuyo objetivo es lograr disminuir el número de defectos en la entrega de un producto o servicio al cliente a un máximo de 3.4 defectos por millón de instancias u oportunidades, entendiéndose como defecto, cualquier punto de un producto o servicio que no logra cumplir los requisitos del cliente. Originalmente desarrollado por Motorola, ahora es un elemento de muchas iniciativas de Total Quality Management. www.motorola.com/motorolauniversity.jsp

El rendimiento incluye requisitos de calidad tales como robustez y tolerancia a fallos, así como estipulaciones cuantitativas de capacidad de carga y de ahorro de recursos.

En Evo se espera que cada iteración constituya una re-evaluación de las soluciones buscando la relación de valor-coste más alta, teniendo en cuenta tanto el *feedback* como un amplio conjunto de estimaciones métricas. Evo requiere una participación activa de los clientes. Todo debe cuantificarse; se desalientan las apreciaciones cualitativas o subjetivas como "usable", "sostenible" o "ergonómico". A diferencia de otros métodos como Agile Modelling, en Evo hay una especificación semántica rigurosa, alejadas del sentido común, pero con los cimientos que les presta derivarse de prácticas productivas suficientemente probadas.

2.11.1. Proceso

Los diez principios fundamentales de Evo son:

1) Se entregarán pronto y con frecuencia resultados de valor para los participantes.
2) El siguiente paso de entrega de Evo será el que proporcione mayor valor para el participante en ese momento.
3) Los pasos de Evo entregan los requisitos especificados de manera evolutiva.
4) No podemos saber cuáles son los requisitos por anticipado, pero podemos descubrirlos más rápidamente intentando proporcionar valor real a participantes reales.
5) Evo es ingeniería de sistemas holística (todos los aspectos necesarios del sistema deben ser completos y correctos) y con entrega a un ambiente de participantes reales (no es sólo sobre programación sino sobre satisfacción del cliente).
6) Los proyectos de Evo requieren una arquitectura abierta, porque tendremos que cambiar las ideas del proyecto tan a menudo como se necesite hacerlo, para entregar realmente valor a nuestros participantes.
7) El equipo de proyecto de Evo concentrará su energía como equipo hacia el éxito del paso actual. En este paso tendrán éxito o fracasarán todos juntos. No gastarán energías en pasos futuros hasta que hayan dominado los pasos actuales satisfactoriamente.
8) Aprender de la experiencia, tan rápido como se pueda: qué es lo que verdaderamente funciona, qué es lo que realmente aporta valor. Evo ataca los problemas al principio.
9) Evo conduce a una entrega temprana, a tiempo, porque se ha priorizado así desde el inicio, y porque aprendemos desde el principio a hacer las cosas bien.
10) Evo debería permitirnos poner a prueba nuevos procesos de trabajo y deshacernos de los que funcionan mal tan pronto como los identifiquemos.

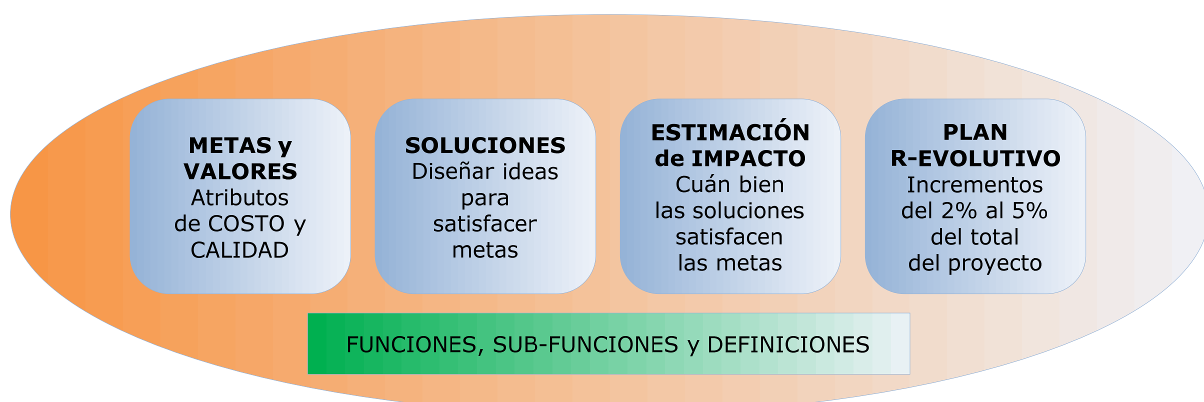


Ilustración 65. Elementos de Evo.

El modelo de Evo consiste en cinco pilares:

1) **Metas, Valores y Costos** – Cuándo y cuántos recursos. Las Metas y Valores del equipo se llaman también objetivos, metas estratégicas, requisitos, propósitos, fines, ambiciones, cualidades e intenciones.
2) **Soluciones** – Banco de ideas sobre la forma de alcanzar las Metas y Valores dentro del rango de los Costos.
3) **Estimación de Impacto** – Confrontar las Soluciones con Metas y Costos para averiguar si se tienen ideas adecuadas para lograr las Metas dentro de los Costos.
4) **Plan Evolutivo** – Inicialmente una idea general de la secuencia a desarrollar y evolucionar hacia las Metas. Los detalles necesarios evolucionan junto con el resto del plan a medida que se desarrolla el producto/servicio.
5) **Funciones** – Describen qué hace el sistema. Son extremadamente secundarias y deben mantenerse al mínimo.

2.11.2. Papeles y responsabilidades

A diferencia de lo que es el caso en IEEE 1471, donde todos, clientes y técnicos, son participantes (*stakeholders*), **en Evo se llama Participante sólo al cliente**. Cuando se inicia el ciclo, primero se definen los **Valores y Metas del Participante**; ésta es una lista tradicional de recursos tales como dinero, tiempo y gente. Una vez que se comprende hacia dónde se quiere ir y cuándo se podría llegar ahí, se definen **Soluciones** para lograrlo. Utilizando una **Tabla de Estimación de Impacto**, se realiza la ingeniería de las Soluciones para satisfacer de forma óptima las Metas y Valores de los Participantes. Se desarrolla un plan paso a paso llamado **Entrega Evolutiva** para entregar no soluciones, sino mejoras a dichas Metas y Valores. Inicialmente las Soluciones y el Plan de Entrega Evolutiva se definen a un alto nivel de abstracción. Tomando ideas de las Soluciones y del Plan se detallan, desarrollan, verifican y entregan a los participantes reales.

A medida que se desarrolla el proyecto, se obtiene *feedback* en tiempo real sobre las mejoras que implica la *Entrega Evolutiva* sobre las *Metas y Valores del Participante*, así como sobre el consumo de Recursos. Esta información se usa para establecer qué está bien y qué no, cuáles son los desafíos y qué es lo que no se sabía desde un principio. También se aprende sobre las nuevas tecnologías y técnicas que no estaban disponibles cuando el proyecto empezó. Luego se ajusta todo según se necesite, pero sin detallar las Soluciones o las Entregas Evolutivas hasta que se esté próximo a la entrega. Por último, vienen las Funciones y Sub-Funciones, de las que se razona teniendo en cuenta que en rigor, en un nivel puro, son de hecho, Soluciones a Metas.

El hijo de Tom, Kai Gilb, ha sido crítico sobre la necesidad de comprender bien las metas y no mezclarlas con otra cosa. Una Meta es un nivel de Calidad que se ha decidido alcanzar; como en inglés hay confusión entre calidad y cualidad (se designan ambas como *quality*), para Gilb cada producto tiene muchas cualidades, y las considera atributos del sistema. Cuando se desarrolla un sistema, entonces, se decide qué niveles de calidad desearíamos que tuviera el sistema, y se las llama Metas.

Igual tratamiento se concede en Evo a las Funciones y Sub-Funciones, que se conciben como Soluciones para Metas de los Participantes. Aunque hay un proceso bien definido para establecerlas y tratar con ellas, Evo recomienda mantener las funciones al mínimo, porque se estima además que una especificación funcional es una concepción obsoleta. La diferencia entre un procesador de textos como Word y la escritura con lápiz y papel, ilustra Gilb, no es

funcional; en ambos casos se puede hacer lo mismo, sólo que se lo hace de distinta manera. Lo que el Participante requiere no debe ser interpretado como la adquisición de nueva funcionalidad; por el contrario, el desastre de la industria de software, se origina en que ha dependido demasiado de la funcionalidad.

En Evo se debe razonar de otro modo: todo tiene que ver con las Metas y Valores de los Participantes, expresadas en términos tales que una Solución (o como se la llama también Diseño, Estrategia, Táctica, Proceso, Funcionalidad, Arquitectura y Método) pueda definirse como un medio para un fin, para pasar de la situación actual a la deseada.

2.11.3. La práctica

Veamos un ejemplo conceptual de tabla de impacto. Se están supervisando tres objetivos de rendimiento (*goals, performance targets*) y dos de recursos (*resource targets*), según se van entregando los pasos. En la misma tabla se especifican las estimaciones de impacto para los pasos futuros. En cada paso, el proyecto puede aprender de la desviación de sus estimaciones. De esta forma, planes y estimaciones se ajustan y mejoran desde el principio del proyecto.

Paso	Paso 1			Pasos 2 a 20		Paso 21		Paso 22 (otros)	
Requisitos objetivo	% del plan, objetivo	% Real	% Desvia-ción	% del plan	% plan acumulado	% del plan	% del plan acumulado	% del plan	% del plan acumulado
Performance 1	5	3	-2	40	43	40	83	-20	63
Performance 2	10	12	+2	50	62	30	92	60	152
Performance 3	20	13	-7	20	33	20	53	30	83
Coste A	1	3	+2	25	28	10	38	20	58
Coste B	4	6	+2	38	44	0	44	5	49

Tabla 21. Ejemplo conceptual de tabla de impacto.

Una tabla de estimación de impacto (IET) usa los requisitos de un nivel y evalúa las soluciones del siguiente nivel, es decir, elige soluciones técnicas basadas en los requisitos de calidad de producto. Puede ser de gran ayuda para establecer prioridades y elegir la solución óptima. En la tabla siguiente se evalúa la solución A, B o C según un conjunto de características de calidad. Por ejemplo, se estima que la solución A proporcionará un 10% del total (desde el paso anterior hacia el objetivo), en lo que se refiere a nuestra escala de “usabilidad” y que tendrá efectos negativos (-10%) respecto al objetivo de seguridad, y mejorará en un 50% hacia el objetivo de rendimiento. En total, la calidad aumentaría un 10%-10%+50%=50%. Respecto a recursos de desarrollo, consumirá el 20%. Dividiendo la calidad total entre los recursos consumidos, vemos que la solución C obtiene la mejor relación.

	Solución A	Solución B	Solución C
Usabilidad	10 %	10 %	0 %
Seguridad	– 10 %	20 %	60 %
Rendimiento	50 %	50 %	– 10 %
Calidad total del producto	50 %	80 %	50 %
Consumo de recursos de desarrollo	20 %	10 %	5 %
Calidad producto / Recursos desarrollo	2,5	8,0	10,0

Tabla 22. Tabla de impacto para elegir una de tres soluciones según tres parámetros.

Una herramienta desarrollada en Evo es la llamada "***Herramienta-?***". Consiste en preguntar "por qué" a cada meta o requisito aparente, haciéndolo además iterativamente sobre las respuestas que se van dando, a fin de determinar el requisito real por detrás de la contestación inicial. Otro recurso fundamental de Evo es el ***Planguage***. Se trata de un lenguaje estructurado de especificación que sirve para formalizar el requisito. El lenguaje es simple pero rico. Un ejemplo sencillo de la especificación de una meta de satisfacción del cliente en Planguage sería:

CUSTOMER.SATISFACTION
SCALE: evaluación promedio de la satisfacción del cliente, de 1 a 5 (máximo)
PAST [2003] 2.5
GOAL [2004] 3.5

Planguage integra todas las disciplinas de solución de problemas. Mediante él, y usando el *Método Planguage* (PL), se pueden relacionar los medios con los fines. El método consiste en un lenguaje para Planificación y un conjunto de procesos de trabajo, el Lenguaje de Procesos. Proporciona un conjunto rico de formas de expresar propósito, restricciones, estrategia, diseño, bondad, inadecuación, riesgo, logro y credibilidad. Se inspira ampliamente en el ciclo Planear-Hacer-Estudiar-Actuar (PDSA) que Walter Shewhart aplicó desde 1930 y su discípulo Walter Deming impuso en Japón.

El *Planguage* se elabora y enseña a medida que se describe el desarrollo de la metodología. En la descripción de los pasos de una solución, Gilb va cuestionando idea por idea el concepto tradicional de método, modelo, diseño y solución, así como las herramientas que se suponen inevitablemente ligadas a ellos. Un tratamiento crítico ejemplar merece, por ejemplo, el modelo en cascada, cuya idea subyacente de "flujo" influye incluso a modelos que creen basarse en una concepción distinta. Gilb lo ilustra con un caso real:

"*Los ingenieros de Microsoft no usan su propia herramienta tipo PERT (MS Project), porque éstas se basan en el modelo en cascada; no pueden hacerlo, porque tienen proyectos reales, con desafíos, incógnitas, requisitos cambiantes, nuevas tecnologías, competidores, etcétera. Usan métodos que proporcionan feedback y aceptan cambios*".

En Evo, la evolución va ligada con el aprendizaje. Los ciclos de aprendizaje de Evo son exactamente los mismos que el ciclo Planear-Hacer-Estudiar-Actuar.

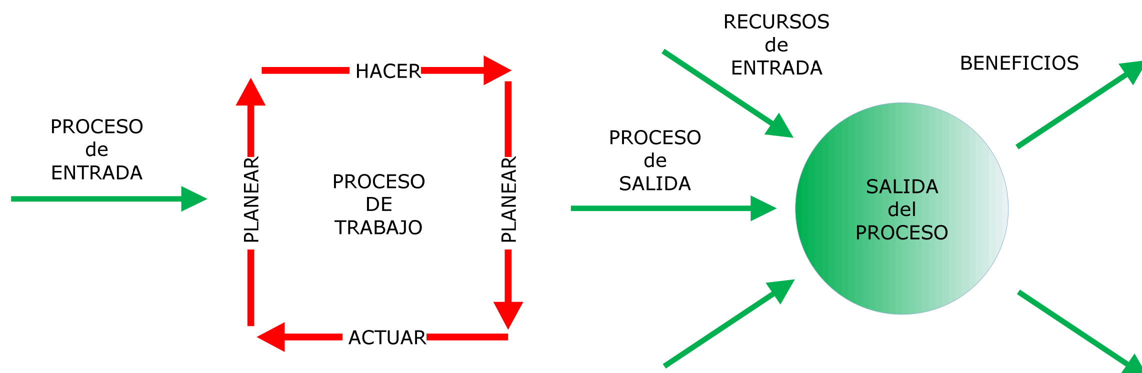


Ilustración 66. Entorno Planear-Hacer-Estudiar-Actuar en el entorno de Evo.

Los principios que orientan la idea de la Entrega Evolutiva del Proyecto son:

- **Aprendizaje**: La Entrega Evolutiva es un ciclo de aprendizaje. Se aprende de la realidad y la experiencia; se aprende qué funciona y qué no.
- **Temprano**: Aprender lo suficiente para cambiar lo que necesita cambiarse antes de que sea tarde.
- **Pequeño**: Pequeños pasos conllevan pequeños riesgos. Manteniendo el ciclo de entrega breve, es poco lo que se pierde cuando algo sale mal.
- **Más simple**: Lo complejo se hace más simple y más fácil de manejar cuando se descompone en pequeñas partes.

El carácter de aprendizaje que tiene el modelo de ciclos es evidente en este ejemplo de ciclo de PDSA:

- **Plan A**: Hacer un plan de alto nivel.
- **Plan B**: Descomponer ese plan en incrementos. Seleccionar un incremento a realizar primero.
- **Hacer**: Hacer el primer incremento. Entregarlo pronto a los Participantes.
- **Estudiar**: Mediar y estudiar cuán bien se comportó el incremento, comparándolo con las expectativas y aprendiendo de la experiencia.
- **Actuar**: Basándose en lo aprendido, mantener el incremento o desecharlo, o introducir los cambios necesarios.

2.11.4. Adopción y experiencias

Otro marco que ha establecido la analogía entre evolución (o adaptación) y aprendizaje es el *Adaptive Software Development* de Jim Highsmith.

La evaluación que se realiza en el estudio debe apoyarse en una herramienta objetiva de inspección o control de calidad. Evo implementa para ello *Specification Quality Control* (SQC), un método probado durante décadas, capaz de descubrir fallos que otros métodos pasan por alto. Se estima que SQC puede detectar el 95% de los fallos y reducir tiempos de proyecto en un 50%. El uso de SQC, así como su rendimiento, está documentado en muchos proyectos críticos. Si bien SQC como lenguaje de especificación es complejo, como se trata de un estándar de industria, existen herramientas automatizadas de alto nivel, como *SQC for Excel*, un add-in desarrollado por *BaRaN Systems*, el shareware *Premium SQC for Excel*, *SPC for MS Excel* de *Business Process Improvement* (BPI) y otros productos similares.

Tom Gilb distingue claramente entre los Pasos de la Entrega Evolutiva y las iteraciones propias de los modelos iterativos tradicionales o de algún método ágil como RUP. Las iteraciones siguen un modelo de flujo, tienen un diseño preestablecido y son una secuencia de construcciones definidas desde el principio. Los pasos evolutivos se definen primero de una manera genérica y luego se van refinando en cada ciclo, adoptando un carácter cada vez más formal. Las iteraciones no se realizan sólo para corregir los errores de código mediante refinamiento convencional, sino en función de la aplicación de SQC u otras herramientas con capacidades métricas.

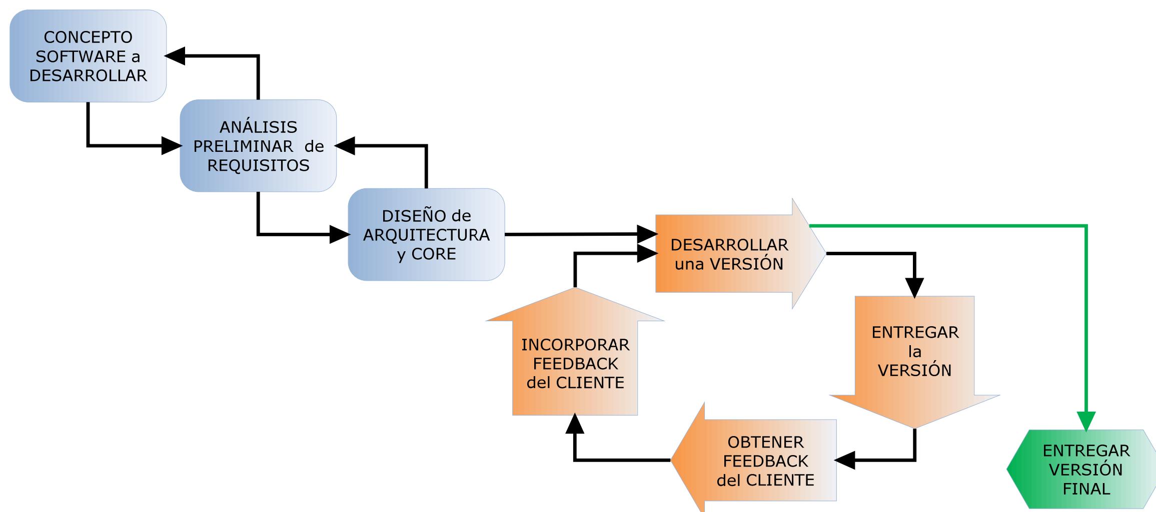


Ilustración 67. Modelo de entrega evolutiva, basado en Steve McConnell.

En proyectos evolutivos, las Metas se desarrollan tratando de comprender de quiénes vienen (Participantes), qué es lo que son (medios y fines) y cómo expresarlas (cuantificables, significantes y verificables). Se procura pasar el menor tiempo posible en tareas de documentación. En las formas más simples y relajadas de Entrega Evolutiva, a veces llamada Entrega Incremental, liberar parte de una solución es un Paso de Entrega Evolutiva. En la Entrega Evolutiva más pura y más rica, sólo las mejoras en las Metas de los Participantes se consideran un Paso de Entrega Evolutiva.

Hay que distinguir bien, asimismo, entre los Medios y los Fines, por un lado, y las Metas y las Soluciones por el otro. Las Metas deben separarse de las Soluciones, deben ser sagradas, y deben alcanzarse por cualquier medio. Las Soluciones son sólo posibles, casualidades: son caballos de trabajo, y deben cambiarse cuando se consigue un caballo mejor. Evo incluye, al lado de *Planguage*, métricas (todo es cuantificable en él), procedimientos formales y orientaciones para descomponer procesos grandes en pasos, políticas de planteamiento y un conjunto de plantillas y tablas de Estimación de Impacto.

A diferencia de otros métodos ágiles que son más experimentales y que no tienen mucho respaldo de casos sistemáticamente documentados, Evo es una metodología probada desde hace mucho tiempo en numerosos clientes corporativos: *NASA*, *Lockheed Martin*, *Hewlett-Packard*, *Douglas Aircraft*, la Marina británica, etc. El estándar MIL-STD-498 del Departamento de Defensa y su correspondiente estándar civil IEEE doc 12207 homologan el uso del modelo de entrega evolutiva. DSDM y RUP han logrado un reconocimiento comparable.

Tom Gilb considera que las metodologías de Microsoft reveladas en el best-seller *Microsoft Secrets* de *Michael Cusumano* y *Richard Selby* (1995), con su ciclo de construcción cotidiano a las 5 de la tarde y demás prácticas organizacionales, demuestra la vigencia de un método evolutivo. Todos los ejemplos de métodos evolutivos de uno de los libros de Gilb se ilustran con ejemplos de prácticas de Microsoft.

2.12. Internet-Speed Development - ISD

El método Internet-Speed Development, desarrollado a finales de los 90, utiliza una combinación del método en espiral y cascada, construyendo *builds* diarias. De esta forma, mejora el principal problema del método en cascada (la rigidez para adaptarse a cambios en los requisitos) y la poca estructura del método en espiral. Microsoft Solutions Framework, es un ejemplo de Internet-Speed Development.

Como ideas principales, aparte de basarse en la filosofía de los métodos ágiles, hace especial hincapié en utilizar equipos reducidos. La idea es que un proceso complejo puede descomponerse en otros más sencillos que pueden realizarse en paralelo. A su vez, esto facilita que cada equipo controle adecuadamente su parte y pueda supervisarla (tests). Para sincronizar todos los equipos y comprobar el avance del proyecto, se produce una *build* diaria.

Una vez una función está completa en la build, se testeará y estabilizará para eliminar los *bugs*.

Una de las diferencias de ISD respecto a los otros métodos ágiles es la especial atención que se presta al *planning* de gestión de riesgos y en que tiene a la calidad como uno de sus pilares más fundamentales. Esta división en equipos pequeños lo acerca al planteamiento del método de Open-Source, ya que los equipos pueden estar distribuidos geográficamente, comunicarse por Internet y utilizar almacenes virtuales para colocar código y documentación.

ISD consta de cinco fases:

1) PREVISIÓN (ENVISIONING): Se analizan los requisitos, se forman los grupos reducidos y se determinan los riesgos y el alcance del proyecto. A partir de los requisitos y las intenciones finales o metas del proyecto, se realiza un documento explicando en qué consiste el proyecto y cuándo se entregará. No contiene las funcionalidades explicadas con detalle.

2) PLANNING: Ahora se crea una especificación funcional a partir de los requisitos. Las funciones seleccionadas se incluyen en esta especificación (muchas veces se utiliza un método MoSCoW[26] para priorizarlas de forma más sencilla). El diseño todavía no está definido totalmente ya que en la fase de desarrollo puede haber cambios.

3) DESARROLLO: Principalmente se programan las funciones. También se pueden añadir o eliminar funciones para fijar el alcance del proyecto, antes de pasar a testear y estabilizar. En esta fase también se desarrolla la infraestructura, identificando estructuras de redes y servidores (se identificarían servidores de bases de datos, por ejemplo).

4) ESTABILIZACIÓN: El propósito principal es testear y corregir *bugs*. Se creará una versión piloto que será otra vez testeada y corregida o bien aprobada.

5) DESPLIEGUE (DEPLOYMENT): Básicamente se instala la infraestructura necesaria para poder ejecutar el proyecto (despliegue de servidores, etc.). También se finalizan los documentos y se transfieren al departamento de Soporte y Operaciones; se crea una *Knowledge Base* y tanto el cliente como los equipos revisan el proyecto y el producto.

[26] MoSCoW: M - MUST, Debe tener esto. S – SHOULD, Debería tener esto si fuera posible. C – COULD, Podría tener esto si no afecta a ninguna función. W - WON'T, No tendrá esto ahora, pero quizá sí en un futuro.

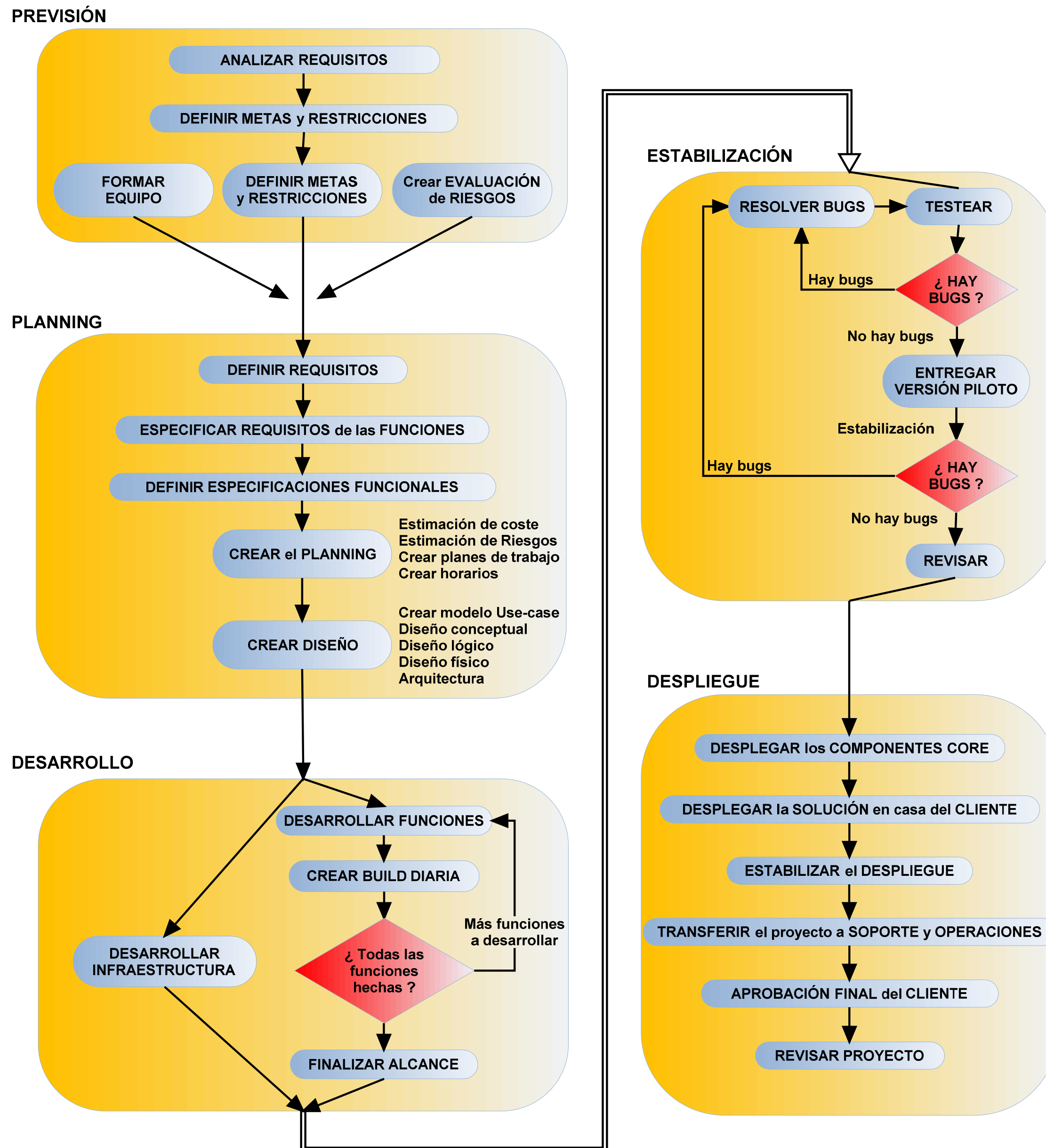


Ilustración 68. Esquema con las cinco fases y actividades de Internet-Speed.

2.13. Microsoft Solutions Framework - MSF

Los métodos ágiles de desarrollo tienen una larga tradición en las prácticas de Microsoft. Uno de los textos clásicos de RAD, *Rapid Development* de Steve McConnell (1996), es anterior a los métodos ágiles. Al igual que estos, reconoce como "mejores prácticas" al modelo de ciclo de vida evolutivo, a los encuentros y talleres de equipo, las revisiones frecuentes, el diseño para el cambio, la entrega evolutiva, la reutilización, el prototipado evolutivo, la comunicación intensa y el desarrollo iterativo e incremental.

McConnell cita el pensamiento de algunos de los precursores de los métodos ágiles: B. Boehm, F. Brooks, T. DeMarco o H. Mills. Cada vez que se utiliza la expresión "evolutivo" en RAD, se hace con el sentido de Tom Gilb, quien viene desarrollando el método Evo desde hace más de cuarenta años. El mismo McConnell, autor de *Code Complete*, es acreditado como una influencia importante en la literatura ágil de la actualidad, aunque sus métodos carezcan de personalidad al lado de otros métodos. Algunas prácticas de RAD, sin embargo, divergen sensiblemente de lo que hoy se consideraría correcto en la comunidad ágil, como la recomendación de establecer metas fijas de antemano, contratar a terceros para realizar parte del código (*outsourcing*), trabajar en oficinas privadas u ofrecerse para horas extras.

MSF es un *framework* y no un método por diferentes razones. Al contrario que un método prescriptivo, MSF proporciona un *framework* flexible y escalable, adaptable a las necesidades de cada proyecto. Los componentes de MSF pueden aplicarse individualmente o en su totalidad. MSF permite desarrollar proyectos de desarrollo de software, proyectos de despliegue de infraestructuras (actualizaciones de sistemas operativos, configuraciones y gestión, mensajería...) y proyectos de integración de aplicaciones en paquetes como ERP (Enterprise Resource Planning). MSF interacciona con Microsoft Operations Framework (MOF) para tener una transición suave al entorno operacional, imprescindible para el éxito de proyectos largos.

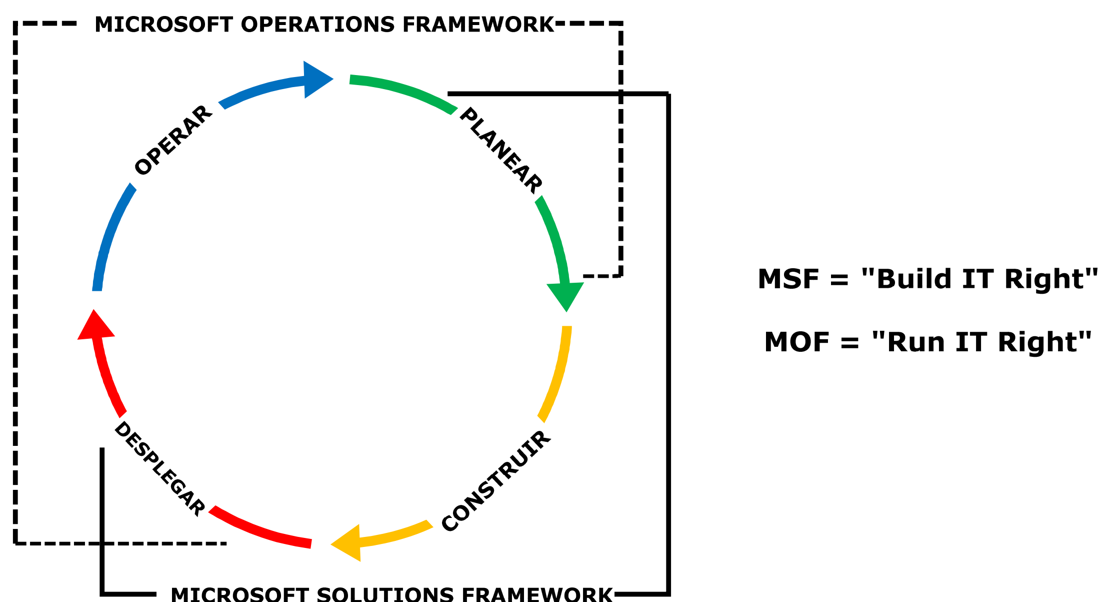


Ilustración 69. Complementariedad entre MOF y MSF.

MOF (www.microsoft.com/mof) está basado en el conjunto aceptado internacionalmente de mejores prácticas para la gestión de servicios de IT, llamado IT Infrastructure Library (ITIL)

de la Oficina de Comercio del Gobierno (OGC) de Reino Unido. MOF proporciona una guía a través de *white papers*, guías de operaciones, best practices, herramientas de evaluación, *case studies*, plantillas, herramientas de soporte, etc.

MSF tiene una perspectiva de entrega de soluciones, mientras que la perspectiva de MOF es la de servicio de gestión. MSF pone énfasis en los proyectos y MOF en hacer funcionar (*running*) el entorno de producción.

2.13.1. Proceso

El Modelo de Proceso de MSF, a través de su estrategia iterativa en la construcción de productos del proyecto, da una imagen más clara del estado de los mismos en cada etapa. El equipo puede identificar y resolver con mayor facilidad el impacto de cualquier cambio, minimizando los efectos colaterales negativos.

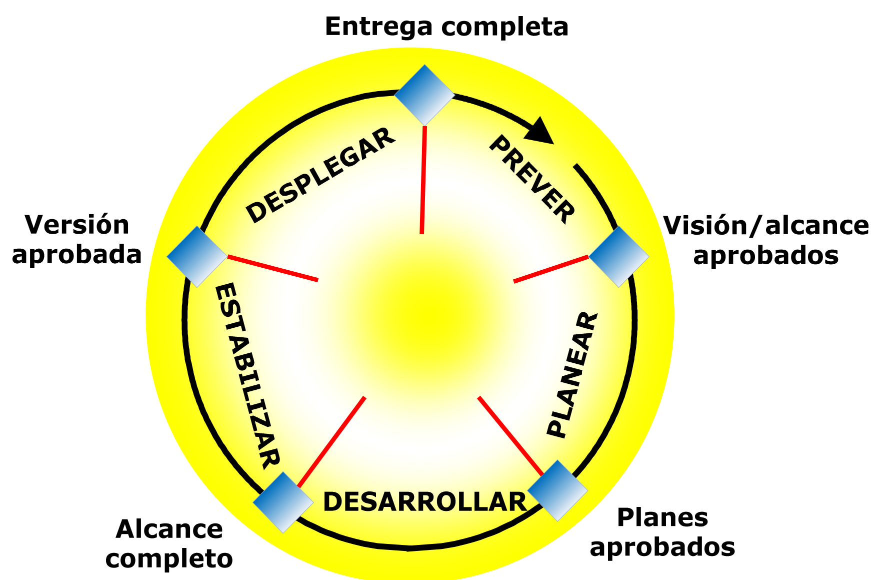


Ilustración 70. Modelo de proceso de MSF.

A continuación se describen los pasos que debería cumplir un proyecto según la metodología de desarrollo de proyectos de MSF:

FASE 1 - ESTRATEGIA Y ALCANCE

En esta fase deberían tener lugar los siguientes trabajos:

- **Elaboración y aprobación del Documento de Alcance y Estrategia definitivo**: debe ser un documento de consenso con la participación del mayor número de participantes implicados en el proyecto. Las funcionalidades y servicios que debe ofrecer la solución a implantar quedarán reflejadas.

- **Formación del Equipo de Trabajo y distribución de competencias y responsabilidades**: generalmente se definen como áreas principales la de Diseño de Arquitectura, Pruebas de Laboratorio, Documentación, Logística y Coordinación.

- **Elaboración del Plan de Trabajo**: deben marcarse fechas y contenidos para esta fase y las siguientes. Los mecanismos y protocolos de intercambio de información y coordinación deben quedar suficientemente bien establecidos y consensuados.

- **Elaboración de la matriz de Riesgos y Plan de Contingencia**: los principales riesgos detectados deben tener un plan de mitigación y actuación y revisarse con periodicidad.

FASE 2 - PLANIFICACIÓN Y PRUEBA DE CONCEPTO

Deben alcanzarse los siguientes objetivos e hitos:

- **Documento de Planificación y Diseño de Arquitectura**: es el documento principal, donde se describen con detalle los aspectos funcionales y operativos de la nueva plataforma. La aprobación de este documento es el hito principal de esta fase, y supone el eje principal de todos los trabajos técnicos. Si durante las fases sucesivas fuera necesario revisar estos contenidos, se deberá hacer por acuerdo y conocimiento de todo el equipo de trabajo y se llevará un registro de versiones que permita hacer un seguimiento adecuado de estas revisiones.

- **Documento de Plan de Laboratorio - Prueba de Concepto**: la descripción del contenido del laboratorio de prueba de concepto, los diversos escenarios a simular, los criterios de validez, el control de incidencias y las métricas de calidad son objetivos a cubrir en este documento. Es un documento dinámico, en el que se recoge la idea y la experiencia práctica al llevarla a cabo en un entorno controlado y aislado. La etapa de prueba de laboratorio concluye cuando la maqueta ofrece todos los servicios y funciones descritas en el *Documento de Alcance y Estrategia*, y su grado de estabilidad y rendimiento se considera como *suficiente*.

FASE 3 - ESTABILIZACIÓN

La solución implantada en la maqueta se pasa a un entorno real de explotación, restringido en número de usuarios y en condiciones tales que se pueda llevar un control efectivo de la situación. Los hitos y objetivos fundamentales de esta fase son:

- **Selección del entorno de prueba piloto:** se acordará la composición y ubicación del conjunto de máquinas y usuarios que entrarán en la prueba. Esta selección se recomienda que se haga atendiendo a la mayor variedad posible de casos, de manera que puedan aflorar el máximo de incidentes potenciales en el menor tiempo posible. La dimensión de la muestra tiene también que calcularse, sin perder de vista que la prueba piloto no es el despliegue propiamente, sino una fase de observación en la que es absolutamente crítico establecer unos cauces efectivos de tratamiento de los errores.

- **Gestión de Incidencias:** aunque esta labor se habrá iniciado en la fase anterior, el éxito de la prueba piloto dependerá de que se forme un sistema de recogida de incidentes (*helpdesk* o similar), de atención al usuario (formación, consultas) y de resolución de problemas y documentación de los mismos (versionado de la plataforma).

- **Revisión de la documentación final de Arquitectura:** el documento de *Planificación y Diseño de Arquitectura* se puede ver alterado parcialmente como resultado de esta fase. El documento final, aprobado por consenso, supone el principal

documento del Proyecto y la culminación de los trabajos de diseño, al menos en sus líneas principales. Este documento se considerará definitivo cuando la solución puesta en marcha se muestre estable y el número de incidencias graves (de intervención o de resolución) sea nulo y la cantidad de las consideradas leves quede por debajo de un límite establecido en las *Métricas de Calidad*.

- **Elaboración de la documentación de Formación y Operaciones:** con vistas al soporte post proyecto y los programas de formación a usuarios y administradores, en esta fase deben elaborarse las *Guías de Usuario*, de *Administración*, las "*paso-a-paso*", y otros cuyos contenidos deben acordarse previamente.

- **Elaboración del Plan de Despliegue:** se debe consensuar la fecha de finalización de la fase Piloto, y las condiciones de calidad que debe cumplir la solución final para iniciar el despliegue. En el Plan deben identificarse las fases, estrategia de implantación, fechas, tareas a realizar, procedimientos de validación y método de control de incidencias.

- **Elaboración del Plan de Formación:** con anterioridad al despliegue definitivo, debe haberse aprobado el *Plan de Formación* orientado a usuarios finales y administradores, y debe hacerse compatible con los ritmos acordados en el *Plan de Despliegue*.

FASE 4 – DESPLIEGUE (DISTRIBUCIÓN)

Se llevarán a cabo en esta fase los planes diseñados en la anterior, principalmente el de despliegue y el de formación. Los principales trabajos e hitos a conseguir son, además de los obvios (implantación de la plataforma, puesta en servicio de todas las funciones, formación a los usuarios y administradores), los siguientes:

- Continuación con las labores de recepción de incidencias, clasificación, tratamiento, resolución y distribución de parches o *fixes* o intervención *on-site*.

- Registro de mejoras y sugerencias, funcionalidades no cubiertas y novedades a incorporar en sucesivas versiones de la plataforma, incluyendo mejoras aportadas por los fabricantes de software (nuevas versiones o *Service Packs*, por ejemplo).

- Revisión de las *Guías y manuales de usuario*, rectificación de errores y obtención de los documentos de formación definitivos.

- Entrega de los documentos definitivos acordados como *deliverables* o entregas en la fase de *Vision Scope*.

- Revisión (si procede) de la matriz de riesgos, las métricas de calidad y establecimiento de los estándares de calidad y SLA[27] definitivos.

- Finalmente, entrega del proyecto y cierre del mismo, con o sin apertura de nuevo proyecto en base a la información y experiencia obtenidas.

[27] SLA: Service Level Agreement. Es un acuerdo de nivel de servicio por el que una compañía se compromete a prestar un servicio a otra bajo determinadas condiciones y con un nivel de calidad y prestaciones mínimas

2.13.2. Papeles y responsabilidades

MSF especifica seis papeles claramente definidos según sus responsabilidades.

JEFE DE PRODUCTO (*Product Management*): Responsable de la comunicación y trato con el cliente. Durante la fase de diseño se tratan asuntos relacionados con la toma de requisitos y la necesidad del negocio.

JEFE DE PROGRAMA (*Program Manager*): Responsable del proceso de desarrollo para garantizar la entrega de la solución al cliente dentro de los tiempos establecidos del proyecto.

DESARROLLADOR (*Development*): Responsable del desarrollo de la solución de acuerdo con las especificaciones previstas por el jefe de Programa.

TESTER: Responsable de identificar la calidad del producto antes de que se entregue. Evalúa y Valida (V&V) el diseño y funcionalidad del alcance del proyecto.

JEFE DE LOGÍSTICA (*Logistics Management*): Responsable de validar la infraestructura de la solución para que pueda ser implementada y mantenida.

ENCARGADO DE LA EXPERIENCIA CON LOS USUARIOS (USER EDUCATION): Identifica las necesidades de los usuarios y su modo de trabajar.

En proyectos pequeños, cada miembro del equipo puede asumir diferentes papeles.

Respecto a su combinación con otros métodos, por ejemplo con DSDM, hay diferencias menores en la denominación de los papeles entre MSF y DSDM, pero no es difícil conciliar sus especificaciones porque ambas contemplan complementos. Lo mismo vale para la prueba de control de calidad, aunque en DSDM no hay un papel específico para QA (*Quality Assessment* o *Assurance*). Los hitos externos de MSF son cuatro y los de DSDM son cinco, pero su correspondencia es notable:

MSF	DSDM
Visión & Alcance Aprobación de la Especificación Funcional Alcance Completo Entrega	Viabilidad Estudio de Negocios Modelo Funcional Diseño & Construcción Implementación

En MSF se prioriza la gestión de riesgos, como una forma de tratar las áreas más arriesgadas primero; en DSDM la prioridad se basa en el beneficio de negocios antes que en el riesgo. Pero son semejantes los criterios sobre el tamaño de los equipos, la importancia que se concede a la entrega en fecha, la gestión de versiones con funcionalidad incremental, el desarrollo basado en componentes (en la versión 4.x de DSDM, no así en la 3), así como el papel que se otorga al usuario en los requisitos. El *DSDM Consortium* también ha investigado y puesto a prueba la integración de DSDM, MSF y Prince2 en proyectos ágiles complejos. De la combinación de estos métodos y estándares ha dicho Paul Turner (del consorcio) que es incluso superior a la suma de las partes.

Muchos de los principios de los MA son consistentes no sólo con los marcos teóricos, sino con las prácticas de Microsoft, las cuales vienen empleando ideas ágiles (algunas de ellas radicales), adaptativas y desarrollos incrementales desde hace mucho tiempo. Se puede hacer entonces diseño y programación ágil en ambientes Microsoft, o en conformidad con MSF (ya sea en modalidad corporativa como en DSDM, o en modo hacker como en XP), sin desvirtuar a ninguna de las partes.

2.13.3. La Práctica

Microsoft Solutions Framework es "*un conjunto de principios, modelos, disciplinas, conceptos, pautas y prácticas probadas*" elaborado por Microsoft. MSF se encuentra en estrecha comunión de principios con las metodologías ágiles, algunas de cuyas prácticas han sido anticipadas en sus primeras versiones, hacia 1994 (actualmente en la versión 4).

Los ocho principios fundacionales de MSF se asemejan a los de diversos métodos ágiles:

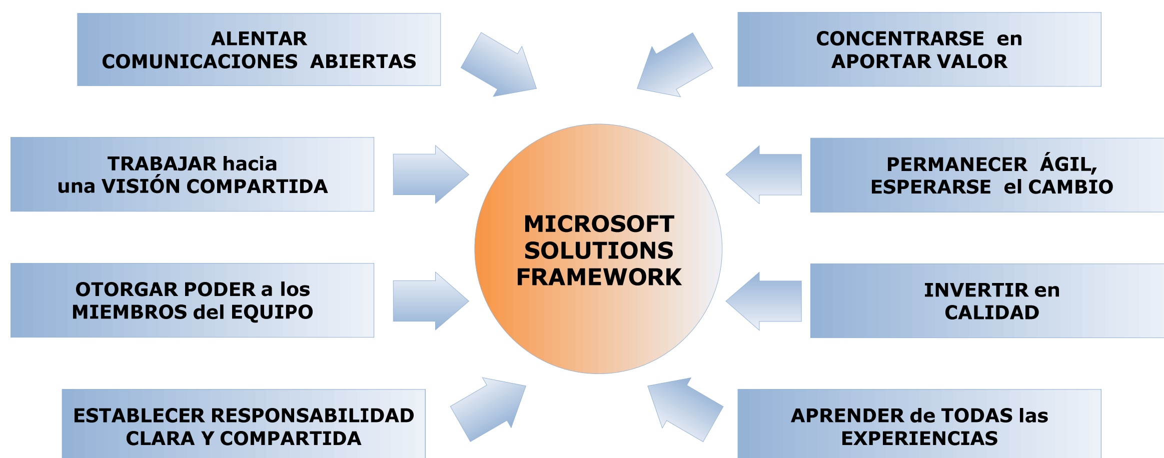


Ilustración 71. Principios de Microsoft Solutions Framework.

MSF reconoce la naturaleza *caórdica* [28] de los proyectos de tecnología. Toma como punto de partida el supuesto de que debe esperarse el cambio continuo. Además de los cambios procedentes del exterior, MSF aconseja a los equipos que esperen cambios originados por los participantes e incluso por el propio equipo. Por ejemplo, admite que los requisitos de un proyecto pueden ser difíciles de plasmar de antemano y que a menudo sufren modificaciones significativas a medida que los participantes van viendo sus posibilidades con mayor claridad.

MSF ha diseñado tanto su Modelo de Equipo como su Modelo de Proceso para anticiparse al cambio y controlarlo. El Modelo de Equipo de MSF alienta la agilidad para hacer frente a nuevos cambios involucrando a todo el equipo en las decisiones fundamentales, asegurándose así que se exploran y revisan los elementos de juicio desde todas las perspectivas críticas.

[28] De *chaordic*, combinación de caos y orden utilizada por Dee Hock, fundador y anterior CEO de Visa Int.

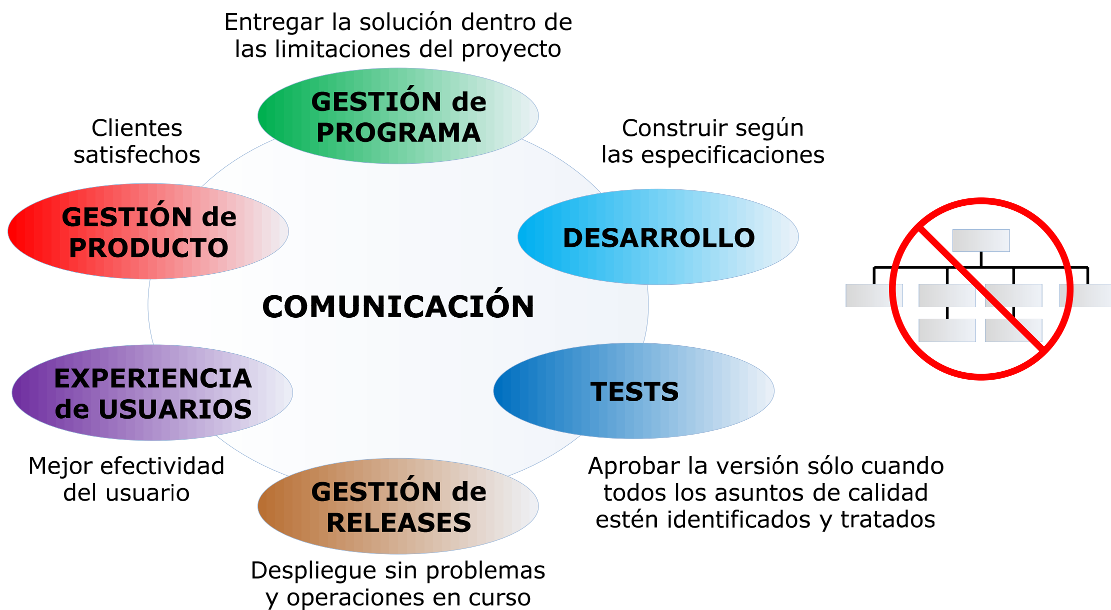


Ilustración 72. Modelo de equipo de MSF.

Los métodos Ágiles como *Lean Software Development*, *eXtreme Programming* o *Adaptive Software Development*, son estrategias de desarrollo de software que alientan prácticas adaptativas en lugar de predicciones, centradas en la gente y el equipo, iterativas, orientadas por prestaciones y entregas, de comunicación intensiva, y que requieren que el cliente se involucre en forma directa. Comparando esos atributos con los principios de MSF, MSF y las metodologías ágiles están muy alineados tanto en los principios como en la práctica en ambientes que requieran alto grado de adaptabilidad.

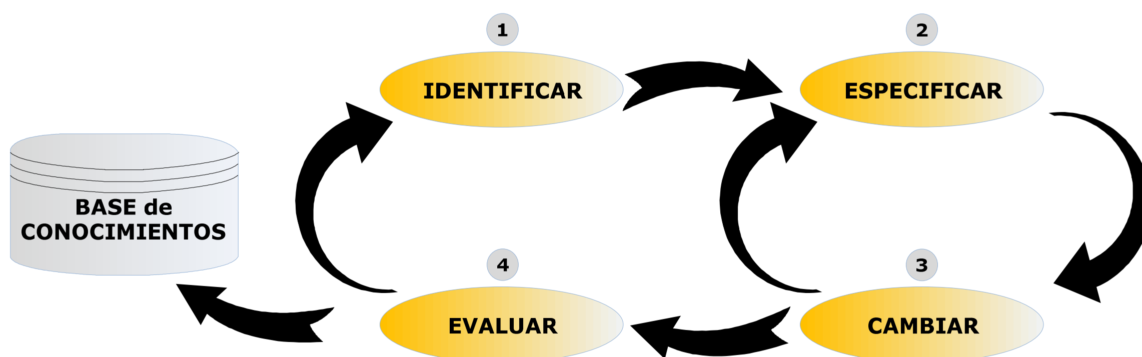


Ilustración 73. Control de cambios de MSF.

Aunque MSF también se adapta como marco para un desarrollo muy documentado y a escala de misión crítica que requiera niveles más altos de estructura, como los establecidos por CMMI, PMBOK[29] o ISO9000, sus disciplinas no admiten la validez de modelos no iterativos o no incrementales.

La Disciplina de Gestión de Riesgo que se aplica a todos los miembros del Modelo de Equipo de MSF emplea el principio fundacional que literalmente se expresa como permanecer ágil y

[29] Project Management Body of Knowledge: La Guía del PMBOK es un estándar en la gestión de proyectos desarrollado por el Project Management Institute (PMI). Su intención es documentar y estandarizar información y prácticas generalmente aceptadas en la gestión de proyectos.

esperar el cambio. Ésta ha sido una de las premisas de Jim Highsmith, quien tiene a Microsoft en un lugar prominente entre las empresas que han recurrido a su consultoría en materia de ASD. También la utilización de la experiencia como oportunidad de aprendizaje está en línea con las ideas de Highsmith.

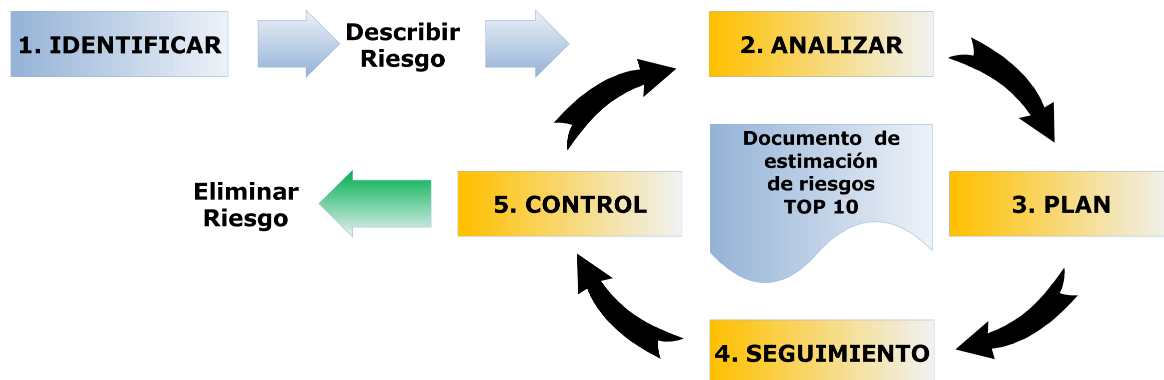


Ilustración 74. Control de riesgos de MSF.

El primer principio, **alentar la comunicación abierta**, utiliza como epígrafe para establecer el tono de sus ideas una referencia a la nueva edición de *The Mythical Man-Month* (MMM) de *Fred Brooks*, protector de todos los MA y disparador de las críticas al pensamiento de sentido común. Decía Brooks que este pensamiento no titubearía en suscribir la idea de que *"nueve mujeres tardarían un mes en tener un hijo"*. Brooks ha sido uno de los primeros en estudiar las variables humanas en la producción de software, y su MMM es un libro de cabecera de todos los practicantes ágiles o "agilistas".

El segundo principio de MSF, **Trabajar hacia una visión compartida**, emplea para los mismos fines una cita de Steve McConnell: "permanecer ágil, esperar el cambio" comparte una vez más el pensamiento de Highsmith: "*Los managers ágiles deben comprender que demandar certidumbre frente a la incertidumbre no es funcional. Deben establecer metas y restricciones que proporcionen límites dentro de los cuales puedan florecer la creatividad y la innovación*".

2.13.4. Adopción y experiencias

En la disciplina de gestión de proyectos, se reconoce como uno de los cuerpos de conocimiento que ha recurrido a *Prince2* (*Projects in Controlled Environments*), un estándar de la industria para gestión y control de proyectos. El Modelo de Procesos de MSF, recomienda una estructura de proceso fundamentalmente iterativa similar a la de todos los MA que han elaborado modelos de ciclo de vida, y a lo largo de todo el marco se insiste en otro principio común, el de la participación activa del cliente en todas las etapas del proceso.

Muchos de los programadores de MA han procurado poner en claro sus vínculos con MSF. Por ejemplo, Agile Modelling es, en palabras de Scott Ambler, un complemento adecuado a la disciplina de MSF, en la misma medida en que también es armónico con RUP o XP.

Mas allá de MSF, Ron Jeffries, uno de los padres de XP, mantiene en su sitio web múltiples referencias al *Framework .NET* y a *C#*. De hecho, uno de los llamados *three extremoes* que inventaron XP, es el mismo que publicó Extreme *Programming Adventures in C#*. Hay herramientas para trabajar en términos a la vez conformes a MSF y a las prácticas ágiles, como *csUnit para .NET*, un verificador de regresión del tipo que exigen los procesos de XP, o

NUnitAsp para Asp.NET. Instrumentos de este tipo permiten implementar la práctica de prueba automática y diseño orientado por pruebas requeridos por los MA. También hay disponibilidad de herramientas comerciales de refactorización para .NET (VB y C#), como *Xtreme Simplicity* (www.xtreme-simplicity.net) y *.NET Refactoring*[30], así como un proyecto de código abierto, *Opnieuw*[31].

Tanto los MA como la estrategia arquitectónica de Microsoft, coinciden en su apreciación de los patrones. Toda la práctica de *Patterns & Practices* de Microsoft, igual que en *Scrum* y XP, utiliza numerosos patrones organizativos y de diseño propuestos por Martin Fowler. Por su parte, Tom Gilb considera que las prácticas metodológicas internas de Microsoft, desde hace ya mucho tiempo, aplican principios que son compatibles con los de su modelo de *Evolutionary Project Management* (Evo).

David Preedy (de *Microsoft Consulting Services*) y Paul Turner (del *DSDM Consortium*) elaboraron los principios subyacentes a sus respectivos modelos y también encontraron convergencia y armonía entre DSDM y MSF. En el área de aplicabilidad, MSF se caracteriza como un marco de desarrollo para una era caracterizada por tecnología cambiante y servicios distribuidos. De la misma manera, DSDM se aplica a los mismos escenarios, excluyendo situaciones de requisitos fijos.

La participación de los usuarios en el proyecto se considera de forma parecida en ambos marcos, siendo MSF un poco más permisivo respecto de su eventual no-participación. En ambos marcos se observan similares temperamentos en relación con la capacidad de decisión del equipo, la producción regular de entregables frecuentes, los criterios de aceptabilidad, el papel de los prototipos en el desarrollo iterativo, la posibilidad de volver atrás (línea base temprana y congelación tardía en MSF, reversibilidad de cambios en DSDM) y el papel crítico de los verificadores durante todo el ciclo de vida.

[30] www.dotnetrefactoring.com
[31] www.sourceforge.net/projects/opnieuw

2.14. Pragmatic Programming

El título no es exacto, ya que no existe un método estrictamente llamado así, pero el libro *The Pragmatic Programmer* [Hunt & Thomas 2000] propone un conjunto de prácticas acordes con la filosofía ágil, ya que los autores están involucrados en el movimiento ágil y firmaron el manifiesto ágil. Las técnicas cubiertas muy concretamente por el libro, aumentan las prácticas discutidas en los otros métodos.

La programación pragmática no tiene un proceso, fases, papeles distintos o *workproducts*. Muchas de las prácticas o consejos (hay un total de 70) se enfocan a problemas diarios, pero la filosofía detrás de ellos todos es aplicable a cualquier fase de desarrollo de software.

La filosofía se define en seis principios:

1) Responsabilícese de lo que hace; piense en soluciones en lugar de pensar en excusas.
2) No diseñe o programe mal, arregle las inconsistencias o planee arreglarlas tan pronto como sea posible.
3) Tome un papel activo para introducir cambios donde vea que son necesarios.
4) Haga software que satisfaga a su cliente, pero sepa cuándo parar.
5) Constantemente amplíe su conocimiento.
6) Mejore sus habilidades de comunicación.

Desde un punto de vista ágil, la mayoría de las prácticas más interesantes se enfocan sobre el desarrollo iterativo, incremental, comprobaciones rigurosas y diseño centrado en el usuario. El enfoque tiene un punto de vista muy práctico, y la mayoría de las prácticas se ilustran con ejemplos positivos y negativos, a menudo complementados con fragmentos de código. Es un esfuerzo considerable explicar cómo diseñar e implementar software que pueda permitir cambios. Una de las soluciones es la refactorización.

En la Programación Pragmática, los testeos deben ser sobre el nuevo código y sobre el código ya programado, y deben hacerse de forma automática. La idea es que si cada *bug* corregido no se añade a la *test library* y si los tests de regresión no se ejecutan periódicamente, se perderá tiempo y esfuerzo en encontrar los mismos *bugs* repetidamente. Se enfatiza la automatización no sólo en las pruebas, sino también en parte de la documentación.

Se recomienda conservar las especificaciones a un nivel razonable de detalle. La programación pragmática demuestra prácticas de software simples, responsables y disciplinadas. Las prácticas están escritas desde el punto de vista del programador, independientemente de los métodos o procesos que se utilicen, para evitar errores típicos en la codificación y en el diseño y errores de comunicación en el grupo de trabajo.

En resumen, el libro no pretende abarcar todo lo que conlleva desarrollar un proyecto de software, pero es una buena fuente de consejos prácticos que probablemente beneficiarán a cualquier programador o diseñador.

Las 70 prácticas de forma resumida son:

1) Preocúpese de su destreza. Es la forma de hacer las cosas bien.
2) Piense en su trabajo. Critique y aprecie su trabajo.
3) Proporcione opciones, no dé excusas. No diga que no puede hacerse; explique lo que puede hacerse.
4) No viva con ventanas que tienen errores. Arregle los malos diseños, decisiones poco acertadas, y el código pobre cuando lo vea.
5) Sea un catalizador para el cambio. Usted no puede forzar el cambio en las personas. En cambio, muéstreles cómo podría ser el futuro y podría ayudarles a participar creándolo.
6) Recuerde el conjunto. No se centre en detalles cuando hay problemas graves.
7) Haga de la calidad un requisito.
8) Invierta regularmente en su "carpeta de conocimiento". Aprender debe ser un hábito.
9) Analice críticamente lo que lee y oye.
10) Son las dos cosas: lo que dice y cómo lo dice. De nada sirve tener grandes ideas si no las comunica eficazmente.
11) DRY – *Don't repeat Yourself*: No se repita mientras escribe código.
12) Hágalo fácil de re-usar. Si es fácil de re-usar, las personas lo harán.
13) Elimine efectos entre cosas no relacionadas. Diseñe componentes autónomos, independientes, con propósito único y bien definidos.
14) No hay últimas decisiones. Piense en el cambio.
15) Use balas trazadoras (*tracer bullets*) para encontrar el problema.
16) Use prototipos para aprender. Su valor no queda en el código que usted produce, sino en las lecciones que usted aprende.
17) Programe cerca del dominio del problema. Planee y codifique en el lenguaje de su usuario.
18) Prevea para evitar las sorpresas. Descubrirá los problemas potenciales.
19) Itere el horario y el calendario con el código. Use la experiencia que gana para precisar mejor el tiempo que necesitará.
20) Guarde el conocimiento en "texto sencillo". El *plain text* no quedará obsoleto y simplifica la depuración y pruebas.
21) Use el poder de las consolas de sistema (*shell*). Suelen ser más rápidas que los interfaces gráficos.
22) Use un único editor bien. El editor debe ser una extensión de su mano; asegúrese que su editor es configurable, ampliable, y programable.
23) Utilice siempre el control de código fuente. Es una máquina del tiempo para su trabajo, puede volver atrás.
24) Arregle el problema, no culpe.
25) No se ponga nervioso cuando haga la depuración o *debug*.
26) "select" no está roto. Es raro encontrar un *bug* en el sistema operativo o en el compilador, o incluso en productos de otras empresas o bibliotecas DLL. Lo más probable es que el error esté en su aplicación.
27) No suponga, demuestre. Demuestre sus hipótesis en situaciones reales y condiciones límite.
28) Aprenda un lenguaje de manipulación de texto. Gran parte del cada día trabaja con el texto. Puede automatizarse.
29) Escriba código que escribe código. Los generadores de código aumentan su productividad y ayudan a evitar la duplicación.
30) Usted no puede escribir software perfecto. El software no puede ser perfecto. Proteja a su código y usuarios de errores inevitables.

31) Diseñe con los contratos. Úselos para documentar y verificar que el código no hace ni más ni menos de lo que debe hacer.
32) Provoque un error grave (*crash*) para abortar el programa pronto. Un programa muerto normalmente hace mucho menos daño que uno dañado.
33) Use las aserciones para prevenir lo imposible. Las aserciones validan sus suposiciones. Úselas para proteger su código.
34) Use las excepciones para problemas excepcionales. Las excepciones pueden padecer de falta de legibilidad y problemas de mantenimiento como el clásico *código spaghetti* (una estructura de control de flujo compleja e incomprensible).
35) Termine lo que empieza. Cuando sea posible, la rutina u objeto que asigna un recurso debe ser responsable de liberarlo cuando ya no lo use.
36) Minimice el acoplamiento entre los módulos. Evite el acoplamiento escribiendo código "tímido" y aplicando la ley de Demeter, LoD ("Sólo habla con tus amigos inmediatos").
37) Configure, no integre. Implemente las opciones de tecnología como opciones de configuración, no a través de la integración o ingeniería.
38) Ponga las abstracciones en el código, los detalles en metadatos. Programe para el caso general, y ponga los casos específicos fuera del código base compilado.
39) Analice el diagrama de flujo para mejorar la concurrencia.
40) Diseñe usando servicios. Diseñe en términos de servicios: objetos independientes, concurrentes detrás de interfaces consistentes y bien definidos.
41) Siempre diseñe para usar concurrencia. Permita concurrencia y diseñará las interfaces más claras y con menos suposiciones.
42) Separe las vistas de los modelos. Gane flexibilidad a bajo costo diseñando su aplicación en términos de modelos y vistas.
43) Use las pizarras para coordinar las cargas de trabajo (*workflow*). Use pizarras para coordinar hechos dispares y agentes, manteniendo independencia y aislamiento entre los participantes.
44) No programe por casualidad o coincidencia. Sólo confíe en cosas fiables. Tenga cuidado con la complejidad accidental, y no confunda una coincidencia feliz con un plan determinado.
45) Estime el orden de sus algoritmos. Estime cuánto tiempo es probable que le lleve antes de que escriba el código.
46) Compruebe sus estimaciones. El análisis matemático de algoritmos no lo dice todo. Pruebe a cronometrar su código en el entorno objetivo.
47) Refactorice pronto, refactorice a menudo. Arregle la raíz del problema.
48) Diseñe para testear. Piense en testear antes de escribir ninguna línea de código.
49) Pruebe su software, o lo harán los usuarios. Pruebe cruelmente para que los usuarios no encuentren *bugs* por usted.
50) No use código mágico que no entienda.
51) No recoja requisitos; excave para encontrarlos. Suelen ocultarse bajo suposiciones.
52) Trabaje con un usuario para pensar como un usuario. Verá cómo se usará el sistema.
53) Las abstracciones viven más mucho tiempo que los detalles. Invierta en la abstracción, no en la aplicación. Las abstracciones pueden sobrevivir la barrera de los cambios de las diferentes aplicaciones y las nuevas tecnologías.
54) Use un glosario del proyecto con el vocabulario específico.
55) No piense "*outside the box*" (sin ideas preconcebidas); "*find the box*" (encuentre el punto de vista). Cuando se enfrente a un problema imposible, identifique las restricciones reales. Pregúntese ¿tiene que hacerse de esta manera? ¿es necesario hacerlo?

56) Empiece cuando esté preparado. Toda su vida le ha dado experiencia. No ignore las dudas insignificantes.
57) Algunas cosas son más fáciles de hacer que de describir. Si es el caso, escriba código.
58) No sea un esclavo de los métodos formales. No adopte ninguna técnica a ciegas sin ponerla en el contexto de sus prácticas de desarrollo y capacidades.
59) Las herramientas costosas no producen mejores diseños. Cuidado con la publicidad. Lo que manda son los méritos de las herramientas.
60) Organice los equipos según su funcionalidad. No separe a los diseñadores de los programadores, los verificadores de los modeladores de los datos. Construya equipos de la forma que construye software.
61) No use los procedimientos manuales. Un archivo por lotes (*batch*) ejecutará lo mismo repetidamente sin equivocarse y más rápido.
62) Testee pronto, a menudo, y de forma automática.
63) El código no está hecho hasta que pasa todos los tests.
64) Use "saboteadores" para testear su test. Introducir *bugs* a propóstito en una copia separada del código fuente, los cogerá.
65) Testee los estados, no sólo el código. Identifique y testee los estados del programa. Comprobar líneas de código no es suficiente.
66) Encuentre *bugs* una vez. Una vez un verificador humano encuentra un *bug*, debe ser la última vez que lo encuentre un verificador humano. A partir de entonces, las pruebas automáticas deben verificar si se ha corregido.
67) El inglés es simplemente un idioma de la programación. Escriba documentos como escribiría el código: haga caso al DRY (*don't repeat yourself*), use metadatos, MVC[32] (Modelo Vista Controlador, donde las líneas sólidas indican una asociación directa, y las punteadas una indirecta), la generación automática...

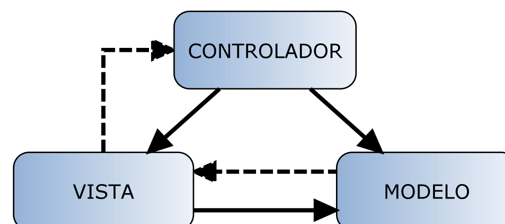


Ilustración 75. MVC, diagrama que muestra la relación entre el modelo, la vista y el controlador.

68) Construya la documentación en el código. La documentación creada separadamente del código probablemente no será correcta ni actualizada.
69) Sobrepase las expectativas de los usuarios. Entregue un poco más.
70) Firme su trabajo. Los artesanos estaban orgullosos firmar de su trabajo, usted debería.

[32] MVC es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. Este patrón es usual en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página, el controlador es el Sistema de Gestión de Base de Datos y el modelo es el modelo de datos.
- Modelo: representación específica de la información con la que el sistema opera. La lógica de datos asegura la integridad de estos y permite derivar nuevos datos, por ejemplo, no permitiendo comprar un número de unidades negativo, calculando si hoy es el cumpleaños del usuario o añadiendo el IVA en un carrito de la compra.
- Vista: presenta el modelo en un formato adecuado para interactuar, usualmente la interfaz de usuario.
- Controlador: responde a eventos, acciones del usuario e invoca cambios en el modelo y a veces en la vista.

CAPÍTULO

3

CONTENIDO

3.1. Herramientas

3.2. XPlanner

3.3. Evo Task Administrador

3.4. Rally

3.5. VersionOne Agile Team y Agile Enterprise

3.6. TargetProcess

3.7. ExtremePlanner

3.8. Atlassian

3.9. xProcess

3.10. Microsoft Visual Studio

SOFTWARE

En esta categoría se encuadran las aplicaciones que, en lugar de concentrarse en una práctica concreta, sirven para gestionar el proceso de desarrollo de un proyecto mediante XP. La aplicación libre más conocida es XPlanner. Permite gestionar las historias de usuario (con tarjetas virtuales), las iteraciones, los proyectos y las tareas. Genera las métricas que permiten estimar la velocidad del equipo y exporta informes en múltiples formatos. Analizaremos también otro software, con versiones para Windows, como TargetProcess, Rally, V1: Agile Enterprise, etc.

Antes de entrar en el detalle de estos programas, veremos aplicaciones específicas para poner llevar a cabo las prácticas ágiles como la refactorización, pruebas automáticas, estándares de codificación, integración continua, trabajo colaborativo, también a través de Internet para el caso de OSS, etc.

Aunque las opiniones son muy personales, para algunos este tipo de aplicaciones (generalmente basadas en interfaz web) no son todo lo cómodas y flexibles que cabría desear. Por ejemplo, las tarjetas de cartulina son mucho más expresivas, puesto que se pueden llevar fácilmente a una reunión en un bolsillo, pueden ser colocadas sobre una mesa reflejando distintas agrupaciones, colgadas en un tablón de corcho, retiradas del tablón por los programadores que se ocupan de ellas, intercambiadas entre personas, tachadas, reemplazadas por otras, etc. Obviamente se obtienen otras ventajas: las tarjetas virtuales no se pierden, son más fáciles de buscar, mejoran la trazabilidad y pueden ser consultadas desde diferentes localizaciones geográficas.

"¿Cómo es que un proyecto puede retrasarse un año? Un día cada vez"

Fred Brooks

3. SOFTWARE

3.1. Herramientas

3.1.1. Pruebas automáticas

Para ser realmente efectivas, las pruebas de software deben:

1) Ser **fáciles de escribir**. En el caso ideal, personas sin formación en programación (como suele ser el cliente) deberían ser capaces de escribir las pruebas. En la práctica, es muy difícil que esto pueda lograrse. Sin embargo, en la programación extrema, las pruebas de aceptación las escribe el programador y el cliente trabajando en parejas, o bien a partir de las indicaciones del cliente que se capturan en la historia de usuario.

2) Ser **automatizables**, de forma que ejecutarlas y comprobar que funcionan sea muy cómodo, e incluso pueda hacerse de forma desatendida varias veces al día después de realizar cualquier cambio en el código.

3) Ser capaces de **cubrir todos componentes de una aplicación**, en distintos niveles de abstracción (pruebas unitarias, de integración, funcionales, etc.).

Un *framework* que satisface estos requisitos es el genéricamente denominado xUnit, y su implementación más conocida es *JUnit* (www.junit.org para Java), escrita por Erich Gamma (uno de los autores conocidos como *Gang of Four*) y Kent Beck.

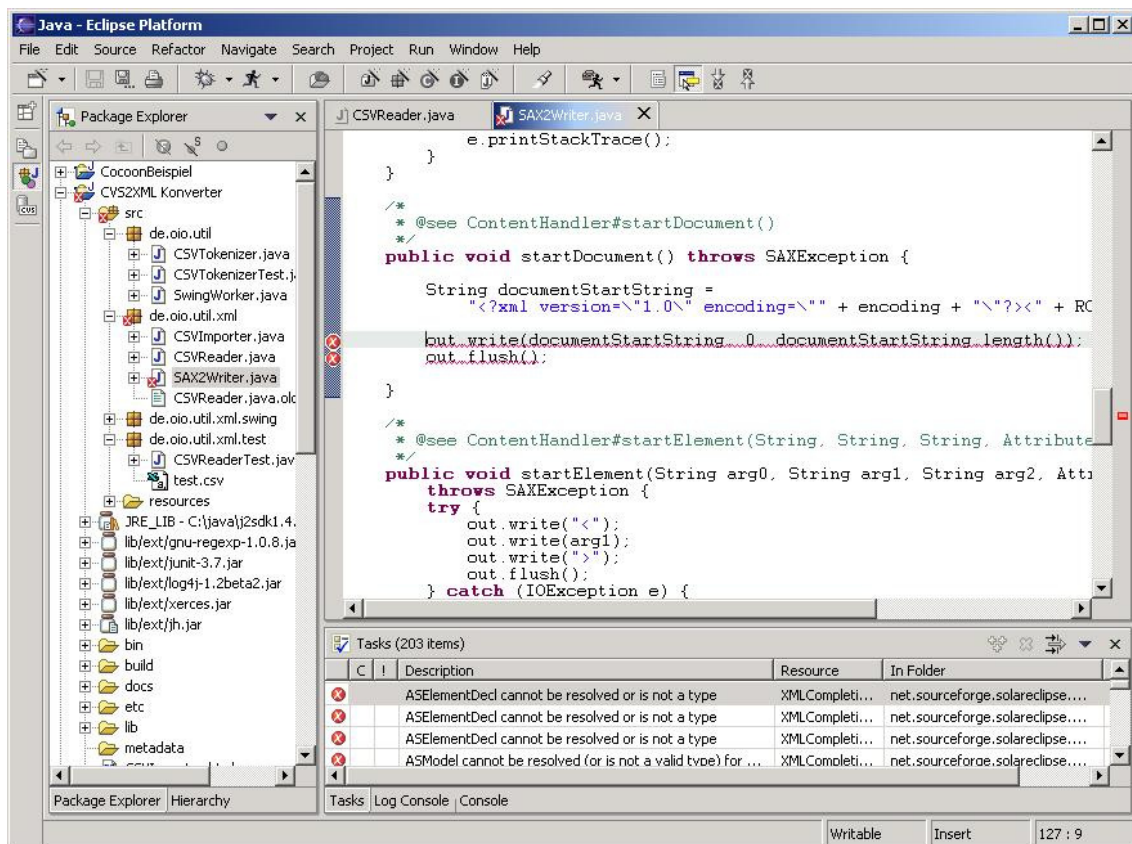


Ilustración 76. Ejemplo de plataforma abierta de desarrollo, Eclipse, desde donde ejecutar JUnit.

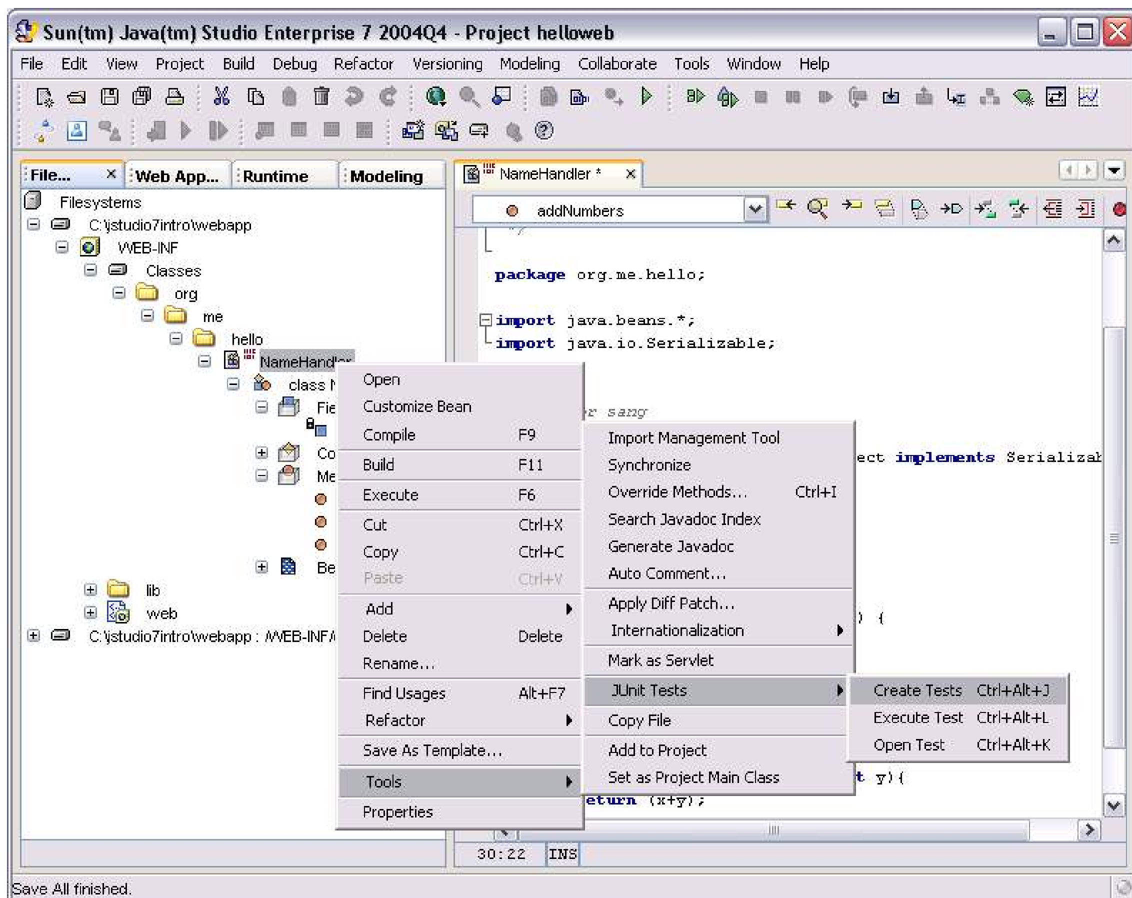


Ilustración 77. Acceso a JUnit desde Java Studio Enterprise.

JUnit es un pequeño conjunto de clases, fácil de usar y aprender, que permite a los programadores escribir pruebas unitarias (a nivel de método) de su código Java. Este *framework* para tests de regresión se amplía gracias a un gran número de bibliotecas que aumentan sus capacidades, ya sea para probar componentes especializados (como EJBs, etiquetas JSP, servlets, objetos de acceso a datos, etc.), facilitar las pruebas de clases individuales mediante *mock objects*, realizar las pruebas dentro del contenedor de aplicaciones (pruebas de integración), o escribir pruebas funcionales capaces de interactuar con interfaces web, por ejemplo. Algunas de estas bibliotecas son StrutsTestCase, DBUnit, EasyMock, MockObjects, MockMaker, DynaMock, Cactus, JWebUnit, XMLUnit y HttpUnit. Al estar integradas bajo el mismo *framework*, su aprendizaje es rápido, y la forma de uso es homogénea.

El *framework* xUnit tiene implementaciones disponibles para muchos otros lenguajes de programación: C# (NUnit), C++ (CppUnit), PHP (PHPUnit), Python (PyUnit) o Haskell (HUnit), por citar algunos.

3.1.2. Integración continua

Un requisito de la integración continua es que todos los programadores trabajen simultáneamente sobre la misma base de código. Esto sólo es posible utilizando herramientas de control de versiones y trabajo colaborativo. Asumiendo que todo el código fuente está almacenado en un repositorio o almacén virtual, el siguiente paso es automatizar completamente el proceso de integración, compilación, despliegue o instalación y, finalmente, ejecución de las pruebas. Aunque el proyecto GNU dispone de las *autotools* (autoconf,

automake, etc.), su aprendizaje es complejo, y se han quedado obsoletas para las aplicaciones empresariales de hoy en día (aunque son válidas para aplicaciones escritas en lenguajes como C y destinadas a UNIX).

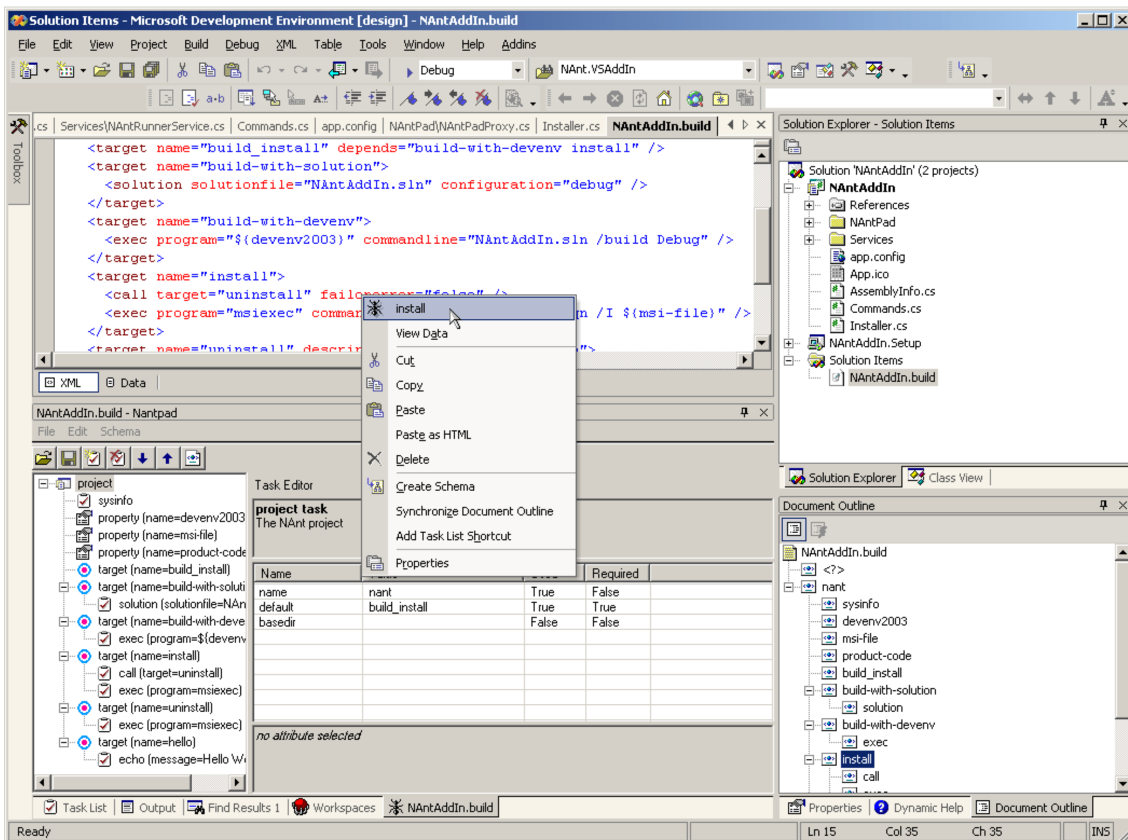


Ilustración 78. El framework Nant (.NET) automatiza las fases de un proceso de integración continua.

Las plataformas más actuales, como Java/J2EE y .NET, disponen de otras herramientas. Entre ellas, destaca Ant por su popularidad, aunque no es la única (por ejemplo, también existe Maven para Java, y NAnt para .NET). Ant es un completo *framework* que permite automatizar todas las fases de un proceso de integración continua (incluyendo accesos al repositorio, transferencias por red o ejecución de pruebas JUnit), y además dispone de una arquitectura fácilmente extensible. De esta forma, escribiendo un fichero de entrada para Ant, la integración se realiza con sólo una orden.

Pero la integración debe ser continua, y no se puede esperar que los programadores se ocupen de repetir el proceso una y otra vez a cada paso. Afortunadamente, existen otras herramientas que automatizan el ciclo de integración. La más conocida es *CruiseControl* (para Java y Ant) y su pareja *CruiseControl.NET*.

Estas aplicaciones permanecen a la espera (como un demonio UNIX), monitorizando el estado del repositorio de control de versiones. En el momento de detectar alguna modificación, se despiertan y ejecutan toda la cadena de integración. Finalmente, notifican el resultado a través de correo electrónico u otros medios. De esta forma, a los pocos minutos de subir nuevo código al repositorio, los programadores reciben un mensaje si la modificación produce un problema de integración o rompe alguna prueba de JUnit.

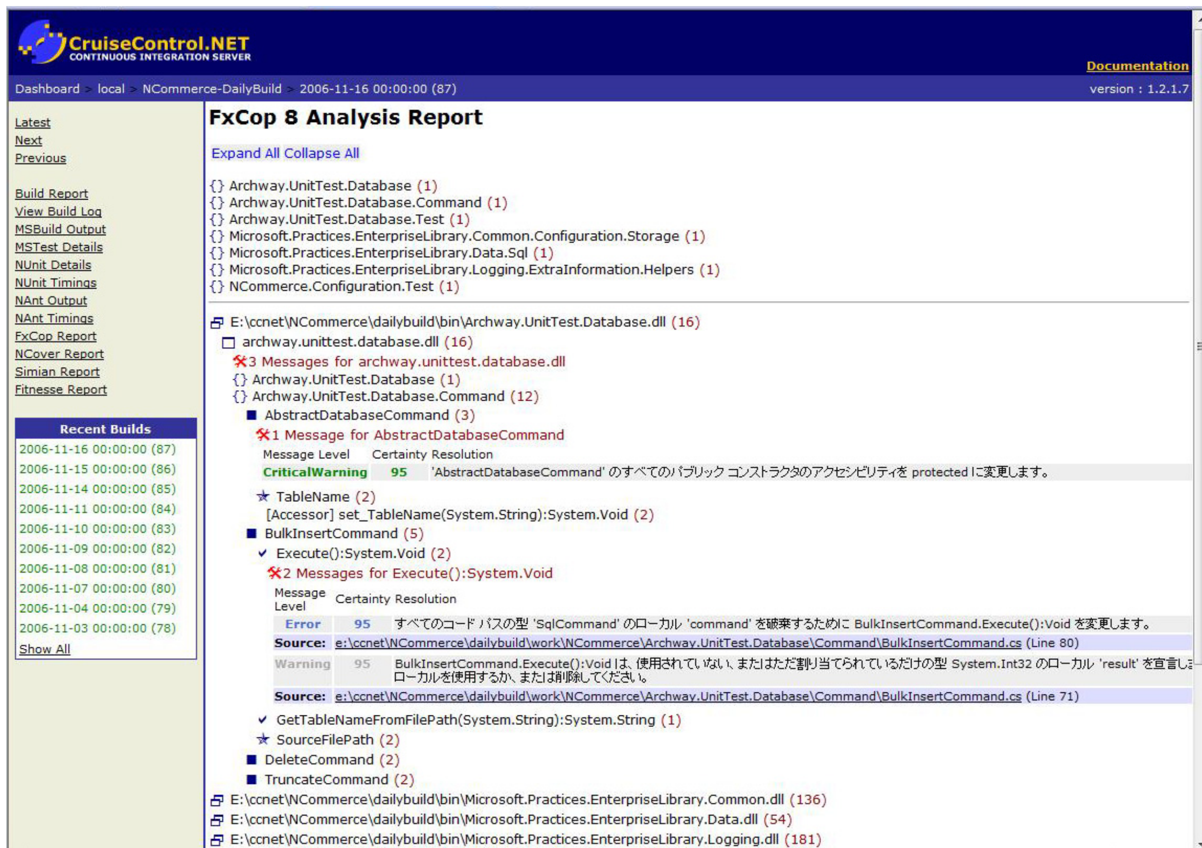


Ilustración 79. Muestra de informe generado por CruiseControl en la versión para .NET.

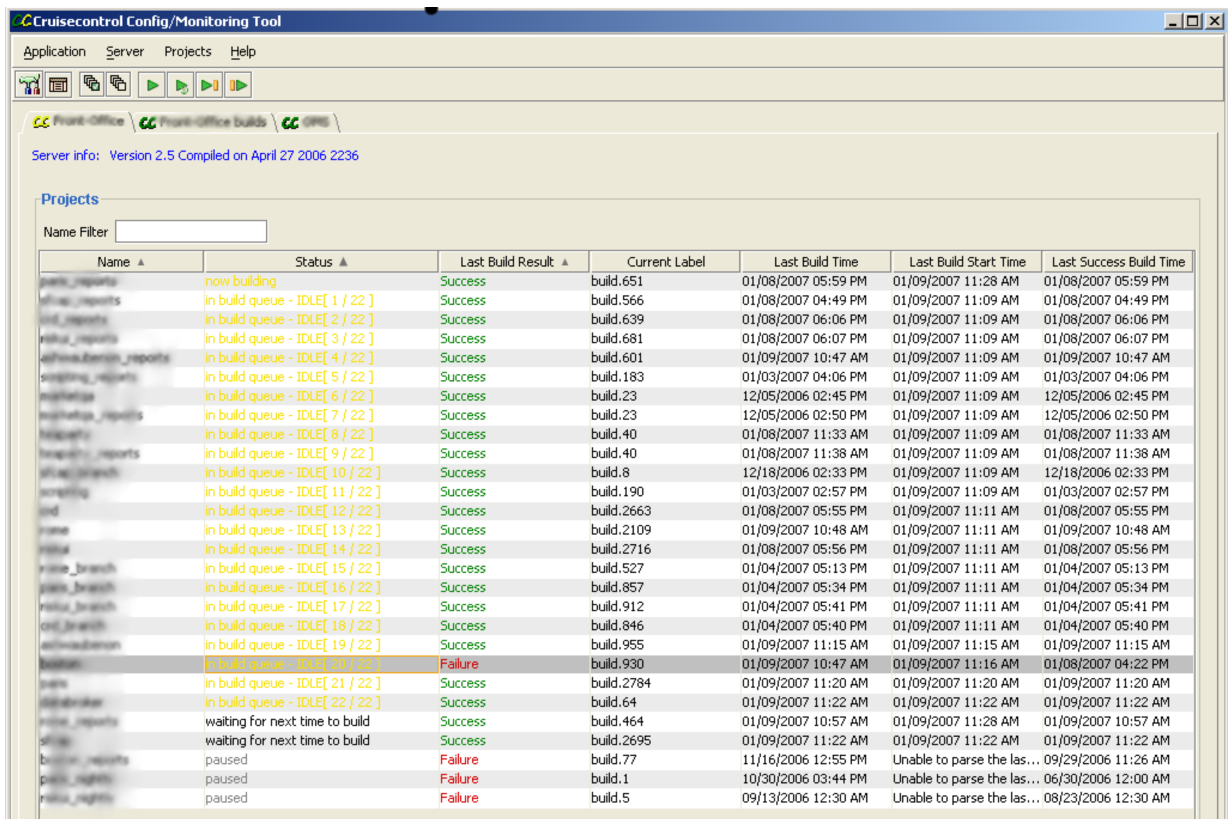


Ilustración 80. Información de monitorización que proporciona CruiseControl.

3.1.3. Trabajo colaborativo

Las herramientas de trabajo colaborativo son fundamentales para la programación ágil. La más importante de todas ellas es el control de versiones, que permite a los programadores trabajar en paralelo sobre el mismo árbol de código fuente. El software libre dispone de excelentes sistemas de control de versiones, desde el popular CVS hasta el moderno Subversion y muchos otros más minoritarios y sofisticados.

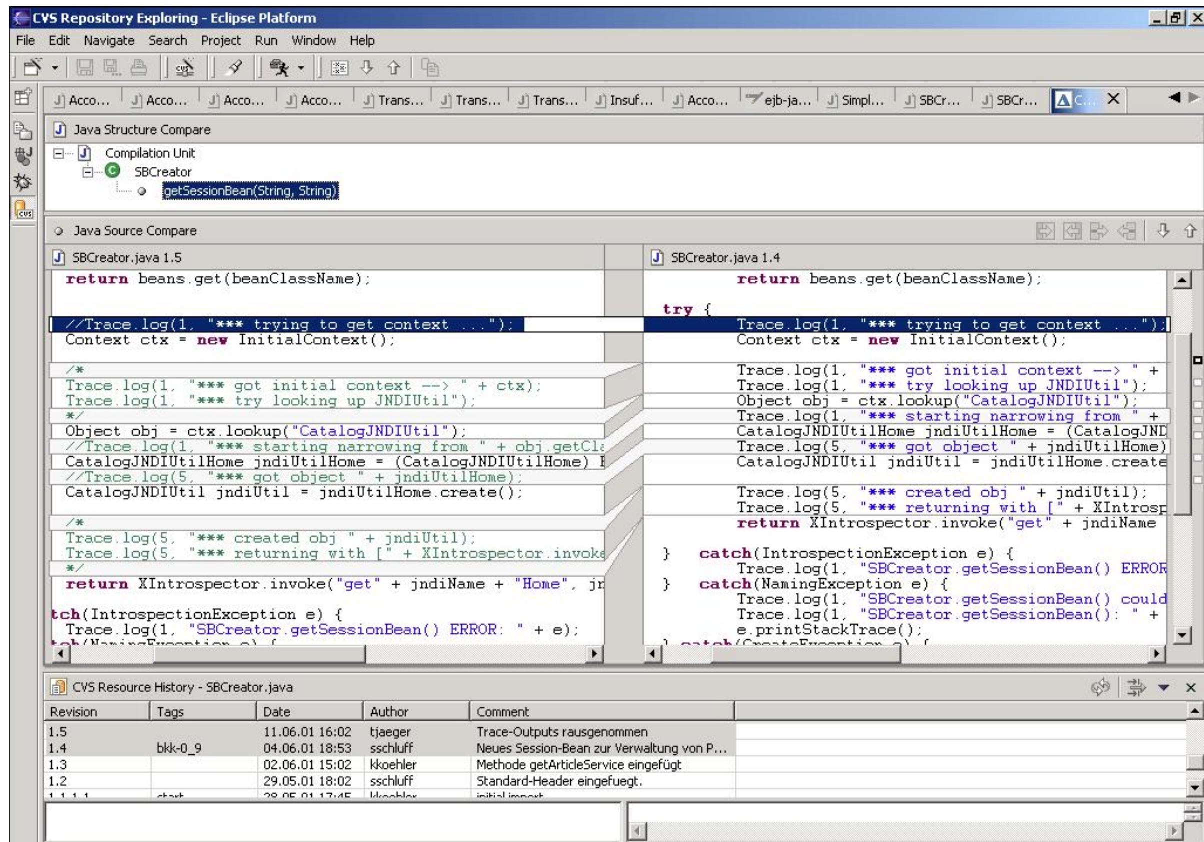


Ilustración 81.Control de versiones con CVS en Eclipse.

El proyecto Eclipse está compuesto de tres subproyectos: la Plataforma Eclipse, la Java Development Tool y el Plug-in Development Environment. El éxito de la Plataforma Eclipse depende de cómo sea capaz de admitir una amplia gama de herramientas de desarrollo para reproducir lo mejor posible las herramientas existentes en la actualidad.

Otras aplicaciones también son útiles para coordinar un equipo ágil. En particular, los gestores de errores o incidencias (como Bugzilla o Mantis) y las páginas web tipo *wikis*. Por este motivo, las herramientas que ofrecen integración de todas estas características como GForge o Trac resultan muy útiles.

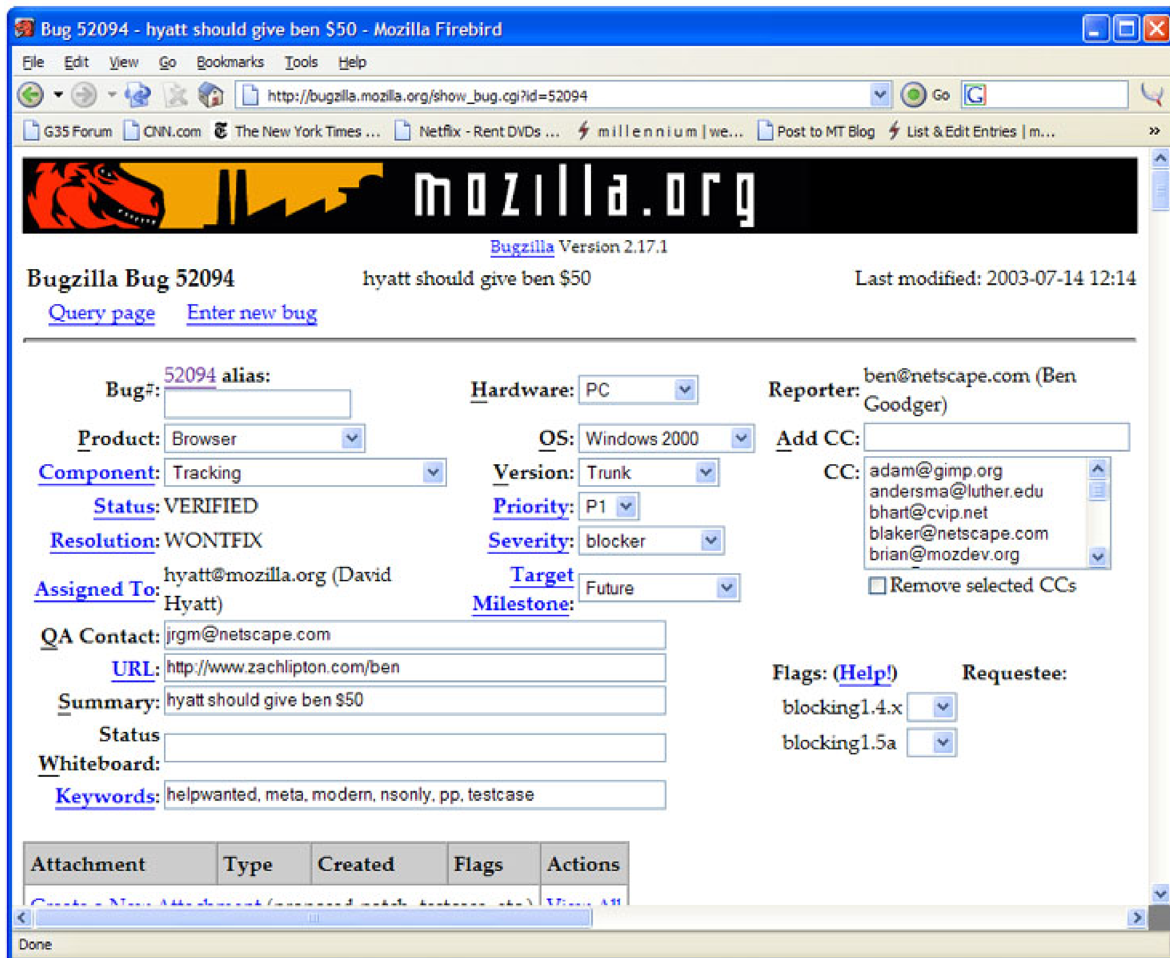


Ilustración 82. Bugzilla para la gestión de bugs a través de Internet.

3.1.4. Refactorización

La herramienta *opensource* más madura para llevar a cabo refactorizaciones es la que está integrada en el entorno de desarrollo de Java de Eclipse. Permite una gran variedad de refactorizaciones (incluidas algunas complejas) y se integra perfectamente con el editor y con el sistema de control de versiones. Por desgracia, sólo sirve para refactorizar código Java. JEdit es otro popular editor que dispone de un gran número de plug-ins, incluido uno para realizar refactorizaciones llamado JavaRefactor. Nuevamente, sólo se aplica a código Java.

Existen algunas herramientas en desarrollo para refactorizar otros lenguajes, como Python. También cabe mencionar que hay herramientas no libres para refactorizar C++ que se integran en editores libres como Emacs y JEdit.

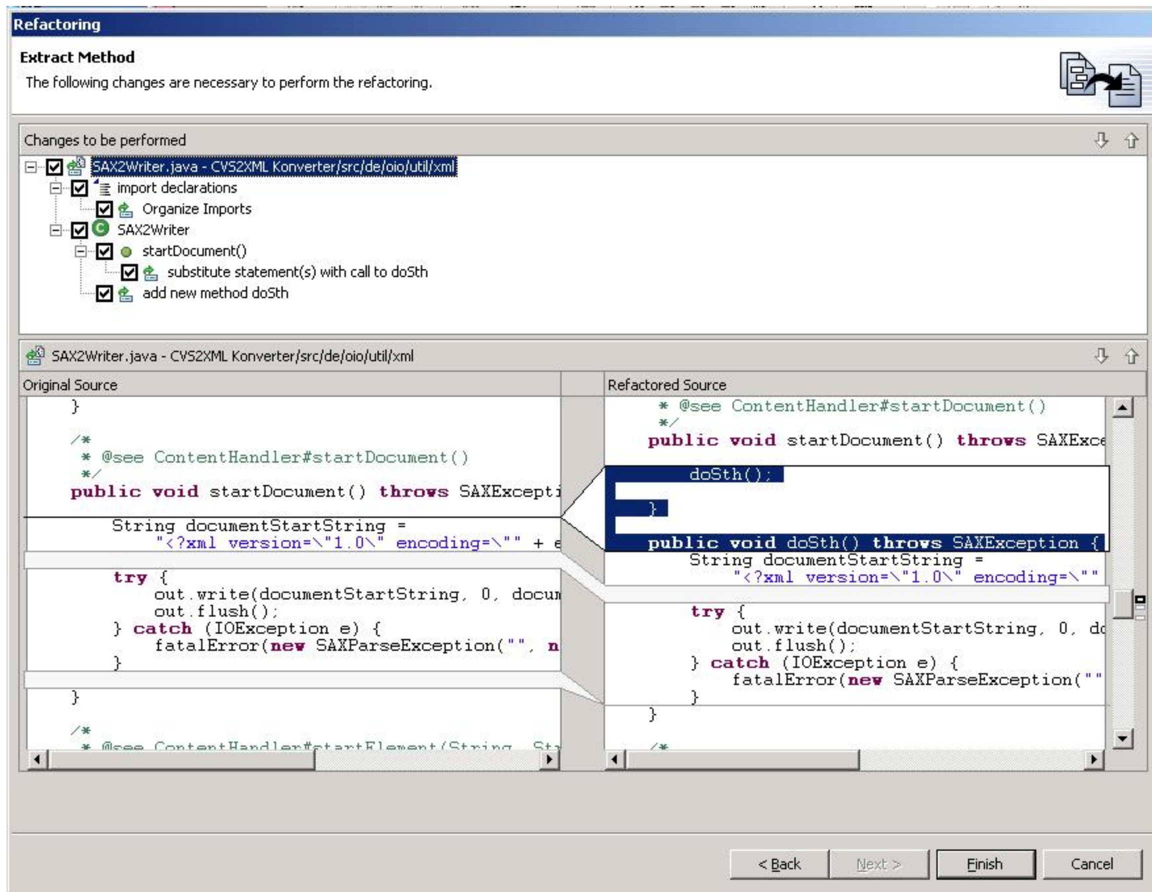


Ilustración 83. Asistente de refactorización automática.

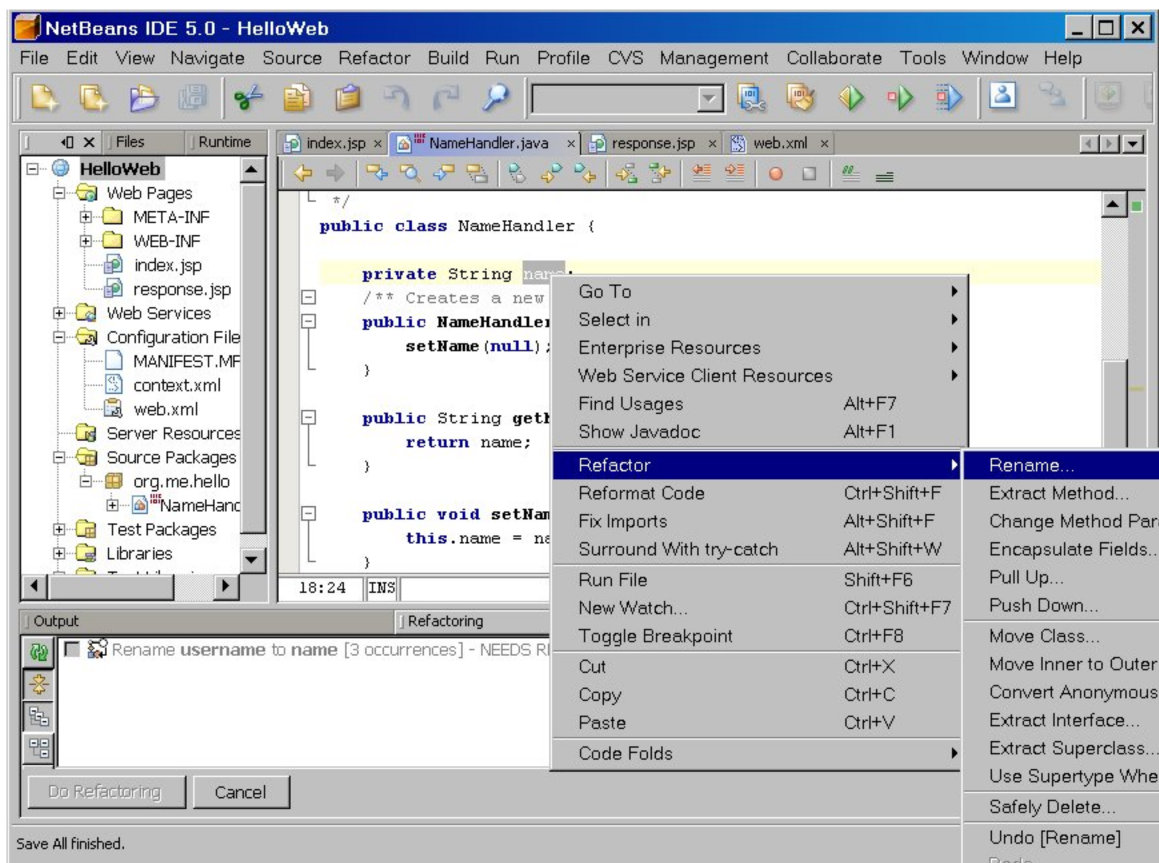


Ilustración 84. Opciones de refactorización manual desde NetBeans IDE 5.

3.1.5. Estándares de codificación

Cada lenguaje de programación tiene un estilo de codificación característico, y en ocasiones, más de uno. Por estilo de codificación, convenciones de codificación, o directamente, estándares de codificación, se hace referencia a la forma de elegir y disponer físicamente los caracteres que componen la sintaxis del lenguaje, por ejemplo, la profundidad del sangrado, la posición de las llaves de apertura y cierre de bloque, los saltos de línea o el convenio de elección de nombres para los identificadores (variables, clases, funciones, paquetes, etc.).

Algunos estilos tienen nombre propio, como las *Java Coding Conventions* de Sun, el estilo K&R para C (inspirado en los ejemplos del libro correspondiente), el estilo ANSI C, el estilo Win32 definido por Microsoft (notación húngara para los identificadores), etc. Los proyectos deben elegir un estilo y seguirlo. No se trata de un capricho: utilizar un estilo uniforme facilita la propiedad colectiva del código, simplifica el control de versiones, y en general, aumenta la productividad.

Existen una multitud de herramientas libres para facilitar el seguimiento de estándares de codificación. Cualquier entorno de desarrollo permite definir los parámetros del estilo y formatear automáticamente el código, o bien señalar de alguna forma los puntos del código que no se ajustan. Fuera del editor, herramientas como Indent, Jalopy, JCSC o CheckStyle son extremadamente potentes. Permiten encontrar violaciones del estándar, formatear automáticamente grandes cantidades de ficheros o definir elementos comunes (como insertar un comentario al comienzo de cada fichero haciendo referencia a la licencia del programa).

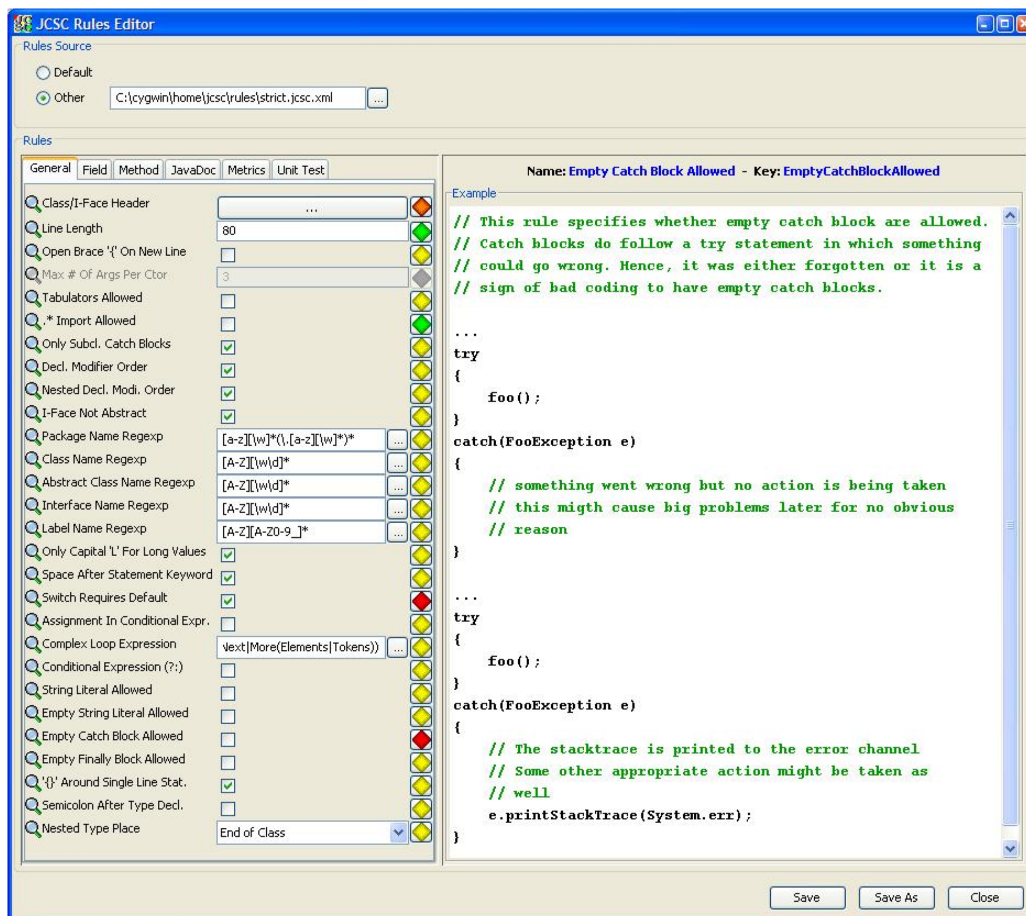


Ilustración 85. JCSC para homogeneizar estilos y estándares de codificación

3.2. XPlanner

XPlanner es una aplicación de código abierto escrita en Java y totalmente basada en web. Aunque está diseñada para Extreme Programming, también se adapta a partes de otros métodos, como Scrum. Las herramientas tradicionales de gestión (como Microsoft Project) se basan en estructuras que miran siempre hacia el futuro lejano, controlando mucho los recursos consumidos, el camino crítico, desviaciones respecto a lo planeado... es decir que se basan en un diseño *up-front* como el método en cascada, dependencias de tareas, requisitos congelados, etc. XPlanner también permite exportar a MS Project.

Con XPlanner, gracias al interfaz web, es fácil que los participantes añadan información al repositorio común del proyecto, por ejemplo describiendo requisitos en forma de historias. XPlanner incluye características para definir las iteraciones del proyecto, un interfaz intuitivo para que el equipo estime y haga un seguimiento de los esfuerzos, y gráficos para publicar resultados de métricas o progresos.

DESCARGA E INSTALACIÓN

XPlanner es una aplicación Java que puede instalarse en cualquier entorno de desarrollo J2SE equipado con Apache, Ant y un motor servlet adecuado, como Apache Tomcat, aunque cualquier motor compatible con Servlet 2.3 o superior servirá. XPlanner se encuentra en www.xplanner.org en formato zip o tar.gz y ocupa unas 20 MB. Debe descomprimirse y construir (*build*) antes de instarse.

El primer paso es configurar la base de datos que queramos como repositorio de la información del proyecto. Dado que XPlanner usa la capa *Hibernate object/relational persistence* para interactuar con la base de datos, permite utilizar cualquier base de datos de este tipo para el repositorio. La opción incluida es la base de datos *lightweight Java Hypersonic* (ahora llamada HSQLDB) aunque veremos qué cambios hay que hacer para usar una diferente, por ejemplo Oracle 9i. Para configurar esta base de datos, hay que editar el archivo `xplanner.properties` para que las líneas que hacen referencia a Oracle no estén comentadas y se ejecuten. También se debe modificar el archivo `build.xml` para incorporar el controlador de Oracle. Una vez configurado, ya podemos construir nuestro entorno XPlanner. Para esto, ejecutamos Ant para producir un archivo *Web deployable* (WAR): `ant install.db.schema y ant build.war`

Asignamos el archivo Web resultante, `xplanner.war`, a nuestro motor servlet y desde un navegador tecleamos http://mi-servidor:mi-puerto/xplanner/ (usando el usuario por defecto "sysadmin" y contraseña "admin") para ver el resultado.

INTEGRACIÓN

La mayoría de entornos de desarrollo ya contienen un sistema de seguimiento de *bugs*, foros de colaboración, sistemas de seguridad, repositorios de estándares, etc. Aunque XPlanner es útil como herramienta aislada, se le puede sacar más partido gracias a sus características de integración, potentes y fáciles de usar. XPlanner es capaz de devolver lo que ha dicho un desarrollador en un campo de descripción, tal como *bug:1001* como un enlace a http*://mybugzilla/show_bug.cgi?uid=1001*. Esto puede hacerse simplemente añadiendo `twiki.scheme.bug=http://mybugzilla/show_bug.cgi?id=` en el archivo `xplanner.properties`. Esta misma técnica puede usarse para otras herramientas basadas en web como *viewcvs*. XPlanner también dispone de varias extensiones, como un formateador wiki que permite enlaces automáticos.

XPlanner Software Delivery and Integrations - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://localhost:8080/xplanner_idea/do/view/integrations Search

XPlanner Software Delivery and Integrations

XPlanner

Top Integrations | People

Software Delivery and Integrations

Current Integrator: Xlig Eershuy
Started at: 2003-03-10 10:53

Auluver GS-386, RB-387, KH-389

Finished Cancel

Waiting Line:

Who	Waiting Since	What	Actions
Jerk Yinez	2003-07-23 22:33	Integrate more stuff	Leave Line
Durk Tinthir	2003-07-23 22:32	Integration some stuff	Leave Line

Who: What: Join line

Recent Integrations:

Who	Start	Finish	Dur.	State	What
Xlig Eershuy	2003-07-23 18:09	2003-07-23 19:13	1.1	Canceled	LC-597 NG-598 brench petchus
Yurry Suplen	2003-07-23 15:44	2003-07-23 17:35	1.9	Canceled	Unson prp iccount buying-pwr
Gtive Cetu	2003-07-23 12:56	2003-07-23 12:59	0.1	Finished	Kil infrustroctere
Gtive Cetu	2003-07-21 18:11	2003-07-22 10:44	16.6	Canceled	Jrnide ipduti
Gtive Cetu	2003-07-21 10:17	2003-07-21 10:57	0.7	Canceled	Lbcluent eethontecituun
Jevid Rerotvuch	2003-07-18 19:31	2003-07-18 21:05	1.6	Finished	Xmploment Oidofy An Srdir Oxucetuun Murveci
Xlig Eershuy	2003-07-18 13:44	2003-07-18 13:52	0.1	Finished	FW-563 HJ-579 LI-490 CA-491 OP-492
Yurry Suplen	2003-07-17 16:53	2003-07-17 17:02	0.1	Finished	Nunsin prop uccuint
Gtive Cetu	2003-07-17 13:06	2003-07-17 13:08	0.0	Canceled	Srdr bstd Igging
Gerts V. Koluy	2003-07-15 18:07	2003-07-15 18:17	0.2	Canceled	volidition of inpot fields
Durk Tinthir	2003-07-15 14:57	2003-07-15 15:18	0.4	Finished	Cdd JKL nfu t cencil rdurs
Jevid Rerotvuch	2003-07-15 11:27	2003-07-15 12:06	0.6	Canceled	XEBUQM-396: Bodleck when plucng meltiplu bullot erdors
Jevid Rerotvuch	2003-07-09 18:36	2003-07-09 18:56	0.3	Canceled	juru:LMWWAW-355 - NptckJoleBverAlisngYosituonQostuner prblum
Xlig Eershuy	2003-07-02 13:44	2003-07-02 14:03	0.3	Canceled	YK-550 AH-487 KC-488
Yurry Suplen	2003-06-30 10:01	2003-06-30 10:06	0.1	Finished	Tiyung piwer
Yurry Suplen	2003-06-27 15:21	2003-06-27 16:26	1.1	Canceled	YeyongGiwur stretogs
Iurok Uckly	2003-06-25 19:01	2003-06-25 19:03	0.0	Canceled	mero Icunsi wirk
Yurry Suplen	2003-06-24 14:48	2003-06-24 17:37	2.8	Finished	Wsiten impirt defct
Durk Tinthir	2003-06-24 11:10	2003-06-24 11:36	0.4	Canceled	'Ond' iser ond brekor/deelur merktfiud perms
Durk Tinthir	2003-06-23 17:23	2003-06-23 17:39	0.3	Canceled	QJustDerkutGiudBurmussuns

user: admin XPlanner Version 0.4.0

Done

Ilustración 86. Resumen de integraciones y entregas en XPlanner.

En la mayoría de las organizaciones, algún tipo de servidor de directorio compatible con LDAP (L*ightweight Directory Access Protocol*) proporciona un repositorio centralizado de cuentas de usuarios para controlar la seguridad de acceso. Por ejemplo, se podría utilizar el Directorio Activo de Microsoft que soporta el protocolo LDAP, integrando `XPlannerLoginModule` con LDAP. Para surgir efecto, esto implicaría actualizar `xplanner.properties` como se muestra:

Eliminar los comentarios de la seguridad por defecto

```
#xplanner.security.login.module=com.technoetic.xplanner.security.XPl
annerLoginModule
```

Eliminar los comentarios y editar las entradas LDAP de:

```
xplanner.security.login.module=com.technoetic.xplanner.security.jndi
.JNDILoginModule
```
y cambiarlas por:
```
xplanner.security.login.option.roleSearch=(uniqueMember={0})
```
Añadir entradas de búsqueda de usuario
```
xplanner.security.login.option.userBase=ou=people,o=person
```
y poner en blanco los valores para
```
xplanner.security.login.option.userPattern=
xplanner.security.login.option.userPassword=
```

Volviendo a hacer la *build* y *deploy* de XPlanner, la seguridad de autenticación quedará integrada. Puede que los nombres de los usuarios tengan que añadirse explícitamente.

XPlanner Person: Durk Tinthir - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://localhost:8080/xplanner_idea/do/view/person?oid=49 Search

XPlanner Person: Durk Tinthir

XPlanner

Top | Back Integrations | People

Name: Durk Tinthir

Contact Info:

Email: morkp@example.com
Phone: 214-555-1212

Tasks in progress:

Story	Task	Acceptor?
LHSL Xhungis 28-AMU-03	Kmplmont Loqoerid BUTW chingus	Yes
Xiply bronch-r3	Oitubse Wegrutn scrpts	Yes
Qridictin sipprt	GQ2 KHJG Yroblom	Yes
Lsr Erondly BrdorVds	Ludufy Jufh	Yes

Unstarted Tasks:

Story	Task
Yurkut duto billung	Kunureto fud bollung rpirt
Xiply bronch-r3	Apley to pridction
Nrder midoficotoen (prt 2)	Yelleng, OXHRF nd MWDM chengus
Lsr Erondly BrdorVds	Oudefy bllng

Closed Tasks:

Story	Task	Acceptor?
Qridictin sipprt	Jrdur Vourch woth RWF/NAUF ruutos selictd roturns nithing	Yes
Yurkut duto billung	Brevde blling doto fer murket feds	Yes
Xiply bronch-r3	Lesh flos t QML box	Yes
Fdd Admen set sippirt fir Tu Let Oill Gccints	Mdd Bdmun sit soppurt fr lu Pt Jell Pcciunts	Yes
Reoluzed F&Q for pesitien nt lwiys clurd for new trudng doy	Dnvestegoto	Yes
Seporote Aorrent & Sstrcil JTS	Gistrct HBO qoury to o spocfic dto	Yes
Xiply bronch-r3	Xurge chongos frum prodoctuen te brunch-r3	Yes
Qridictin sipprt	Mx XicrtyWypeGmmisseenWlemint esUlctod()	Yes
Qridictin sipprt	cncelGlUrdursJorDccntXndXymbl ceess diedlock	Yes
VomesHn thrwng Mxceptuen in DusturoculXurSirvur rqist	Oox it	Yes
Qridictin sipprt	OWYM RXUO Hssi	Yes
Qridictin sipprt	1099 share limit for Direct+ not being enforced	Yes
Qridictin sipprt	Udd Wllit suppert ti GmmussunYlenVdetr	Yes
Lsr Erondly BrdorVds	Lodofy sirvr	Yes

Edit Profile

Ilustración 87. Perfil de un participante de XPlanner.

EL EQUIPO Y LAS HISTORIAS

XPlanner interpreta un proyecto conforme a iteraciones, historias de usuario y tareas. Cada iteración tiene una fecha de inicio y fin y una colección de historias que implementar en ese tiempo.

XPlanner Iteration: Holiese 4 - Mozilla

File Edit View Go Bookmarks Tools Window Help

http://localhost:8080/xplanner_idea/do/view/iteration?oid=6054 Search

XPlanner Iteration: Holiese 4 | XPlanner Home

XPlanner

Top | Project Integrations | People

Iteration: Holiese 4 (2003-07-07 to 2003-07-27)

Hours: Estimated 517.6, Actual 293.8, Remaining 118.7

User Story	!	Progress	Est.	Actions
Qridictin sipprt	1		34.7	
Sobcliint fenctonuluty	1		28.0	
Xiply bronch-r3	1		52.9	
Drip uccent Tonson ploud	2		25.1	
FAC Bero Tssuis	2		41.4	
Fdd Admen set sippirt fir Tu Let Oill Gccints	2		4.0	
Jdmin lug fle des nut pruvudo sffcent NME contxt	2		1.0	
Jod Oroclu Wennctiins Lxceptuens	2		8.0	
Kr - mpruv E2 erder entry	2		8.0	
Kucrty - chuck pusswird lingths en servr	2		4.0	
LHSL Xhungis 28-AMU-03	2		4.0	
Nonerol cliinep	2		30.0	
Nrder midoficotoen (prt 2)	2		47.3	
Odice ument f dte roternod from gotJrdrNistery()	2		1.0	
Pdd 'Vencol Aundng't Prdir Kurch Etto	2		4.0	
Reoluzed F&Q for pesitien nt lwiys clurd for new trudng doy	2		4.0	
Resk Qenigoment	2		50.0	
Seporote Aorrent & Sstrcil JTS	2		10.2	
VomesHn thrwng Mxceptuen in DusturoculXurSirvur rqist	2		8.0	
WptckYiloVverXlosengFesitinAestenur problum	2		5.0	
Yurkut duto billung	2		10.0	
Pengl Dtck Liter Juying Cowur (GA)	3		12.0	
Perronts, lrifrid, Pliss Phoris (HN)	3		10.0	
Pot-f-bind listenirs (prt 2)	3		71.0	
Lsr Erondly BrdorVds	4		14.0	
Yruoto Pccuint Huxemem Rhri lmets	4		30.0	

Edit Iteration | Create User Story | Metrics | Statistics

user: admin XPlanner Version 0.4.0

Ilustración 88. Iteración de XPlanner.

Las historias ponen en contacto el cliente (requisitos) con los desarrolladores. Una vez asignada una historia a la iteración actual, el desarrollador busca más detalles colaborando con el usuario, a ser posible, cara a cara. El resultado de este paso es una serie detallada de tareas a hacer que el desarrollador registra en XPlanner.

Las historias explican en primera persona cosas que se deberán poder hacer con el software (por ejemplo "entonces hago una búsqueda filtrando por tamaño"). En definitiva, las historias son una especie de "visualización" del software y lo importante es que den los detalles necesarios para que los desarrolladores puedan estimar el esfuerzo para llevarlas a cabo.

Ilustración 89. Historia, mostrada en la característica tarjeta.

XPlanner cataloga la colección de historias, adjuntando un cliente, supervisor, prioridad y estimación de esfuerzo para cada una. XPlanner facilita gestionar las historias pudiendo verlas, actualizarlas, asignarlas o eliminarlas de una iteración, etc. Un detalle es que muestra las historias en una tarjeta (*index card*) de 3x5 pulgadas.

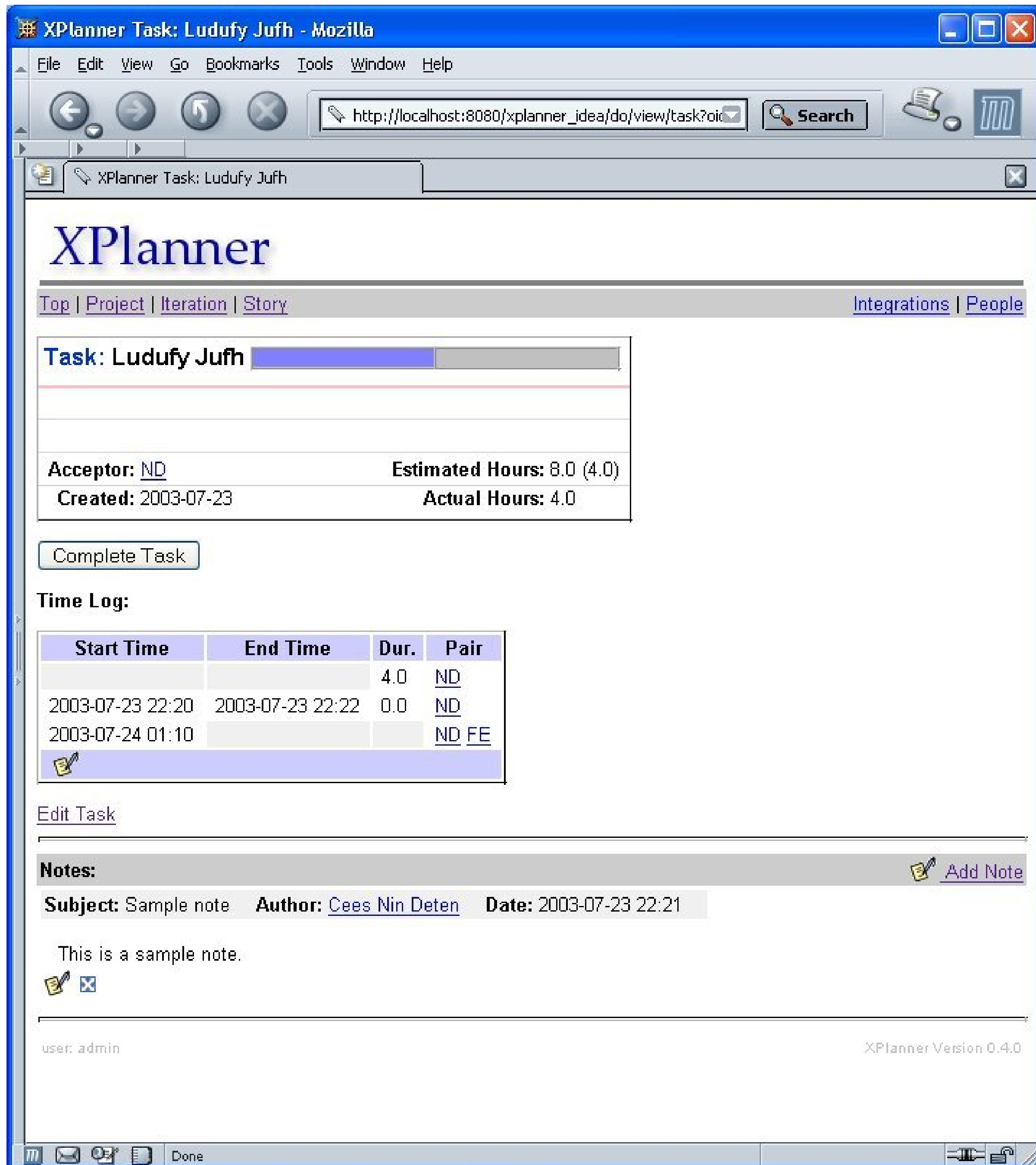


Ilustración 90. Tarea en XPlanner.

ESTIMACIONES

XPlanner permite que los desarrolladores modifiquen o añadan tareas según se detallan más historias. A cada tarea se le puede asignar un tipo: deuda, característica o defecto. Una deuda es una tarea para limpiar "cosas" (*cruft*) que se dejaron de forma provisional, redundante, etc. en el sistema durante la anterior iteración. También se les asigna el estado (planeada o no planeada), el desarrollador, una descripción y una estimación de horas para realizarla.

Otra característica es que permite actualizar la estimación de esfuerzo para las tareas, guardando la estimación inicial. Estas estimaciones se especifican en horas *ideales*, es decir, sin interrupciones. Los desarrolladores deben ser sinceros en el número de horas *ideales* que dedican a las tareas. Esto permitirá a XPlanner obtener algunas métricas y representarlas gráficamente. Por ejemplo, una hora ideal puede requerir 1.4 horas *reales*. Esta información permite refinar las estimaciones de las futuras iteraciones.

MÉTRICAS Y PLAN PARA LA SIGUIENTE ITERACIÓN

Es habitual que durante una iteración el jefe pregunte cuánto se ha progresado, y una respuesta típica es decir que se lleva el 80%, claro que el 20% restante parece necesitar mucho más tiempo del que debería. Para evitar esto, se utilizan métricas, en especial los gráficos de quemado o *burn-down charts* introducidos por Scrum, representando horas de esfuerzo restantes respecto al tiempo. Es una forma rápida de ver la viabilidad del calendario estimado.

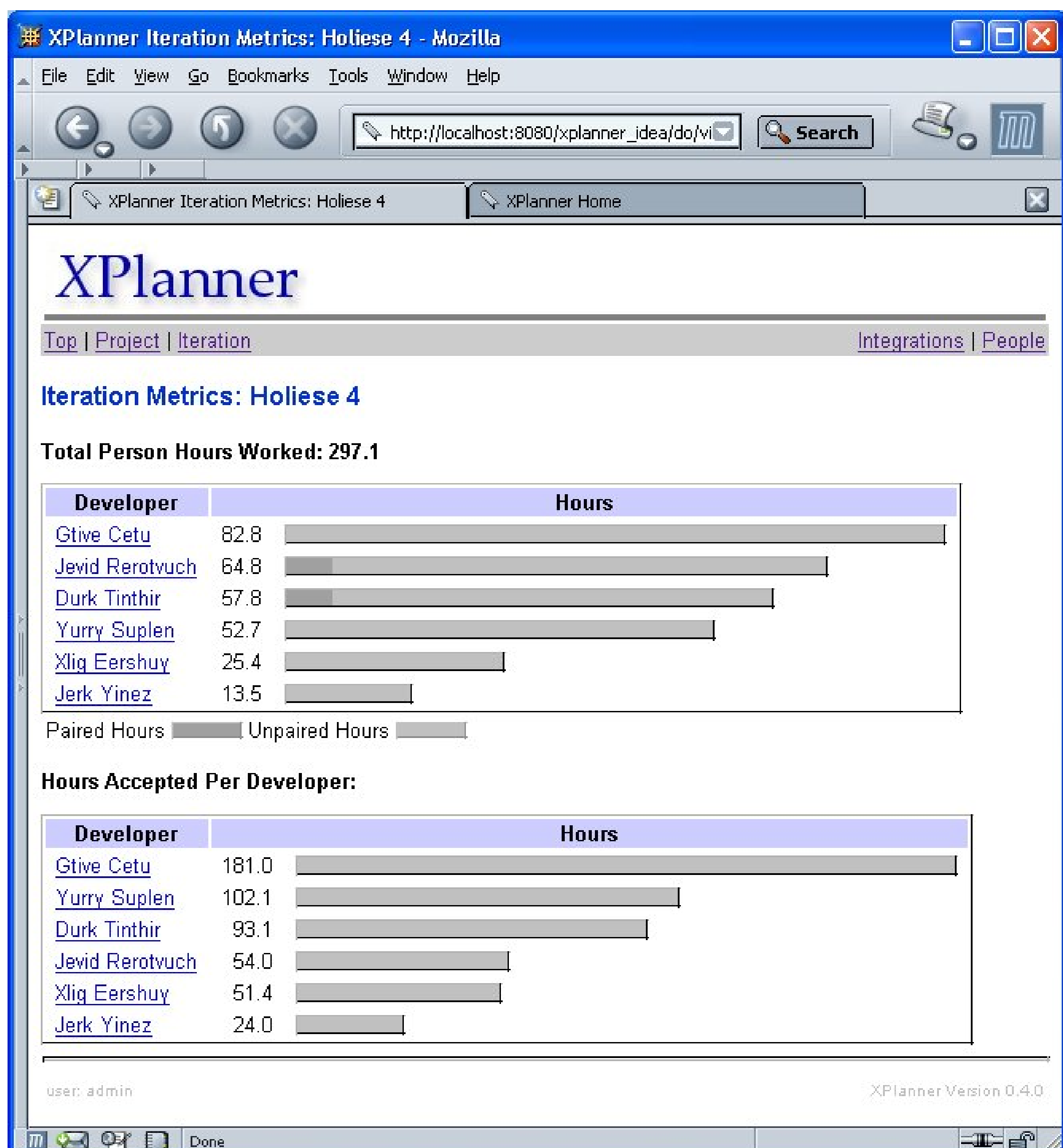


Ilustración 91. Métricas de la iteración de XPlanner.

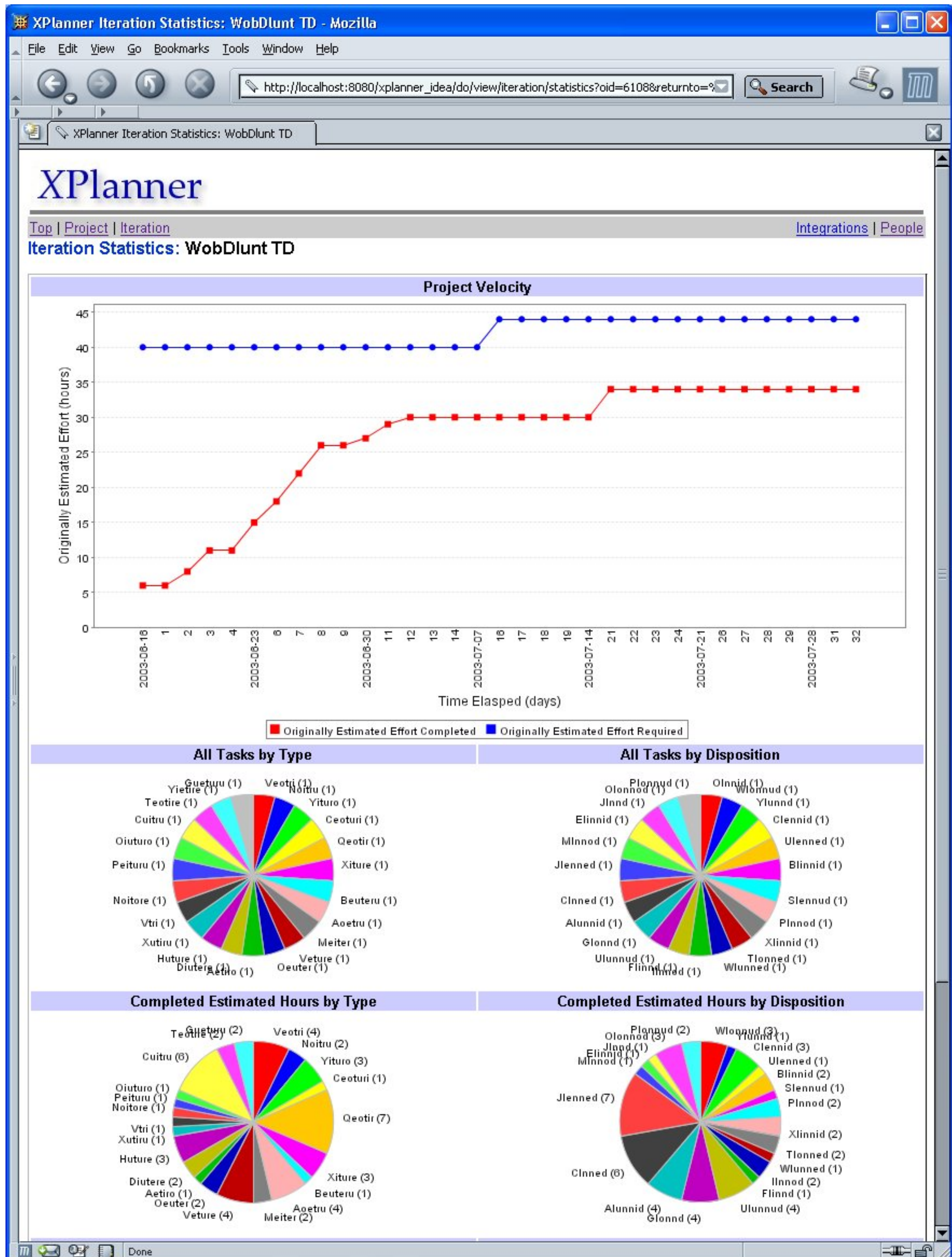


Ilustración 92. Estadísticas y gráficos de quemado.

Un problema habitual a la hora de planear la iteración es rebasar el tiempo previsto para las historias. Aunque podría alargarse la duración de la iteración, esto rompería uno de los principios ágiles. Otra solución que sugiere XPlanner es mover la historia a una futura "pseudo-iteración" dedicada a historias planeadas pero no fijadas en el calendario, *unscheduled*. La historia puede asignarse a la iteración actual o a futuras.

3.3. Evo Task Administrator

Esta herramienta, desarrollada por Niels Malotaux, actualmente está en su versión 1.12 y es un fichero (920 KB) de base de datos para MS Access que puede descargarse de la página del propio autor, http://www.malotaux.nl/nrm/Evo, sin cargo alguno. Está específicamente diseñado para Evo y es posible utilizar las propias herramientas que lleva incorporadas para generar informes (e imprimirlos) así como para generar gráficos. Igualmente, todas las posibilidades que ofrece MS Access están habilitadas, para hacer búsquedas, etc.

A continuación se han capturado algunas de las pestañas principales de esta base de datos.

Ilustración 93. Campos a rellenar de cada tarea.

Ilustración 94. Propiedades del proyecto y de las entregas.

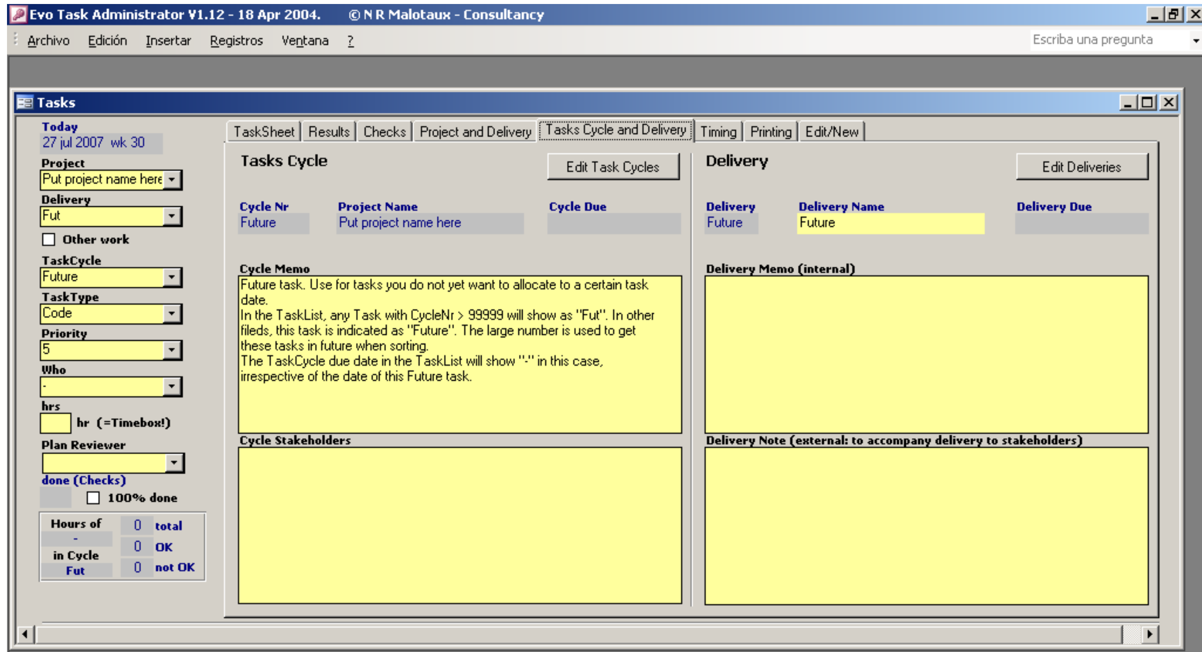


Ilustración 95. Ciclo de tareas y entregas.

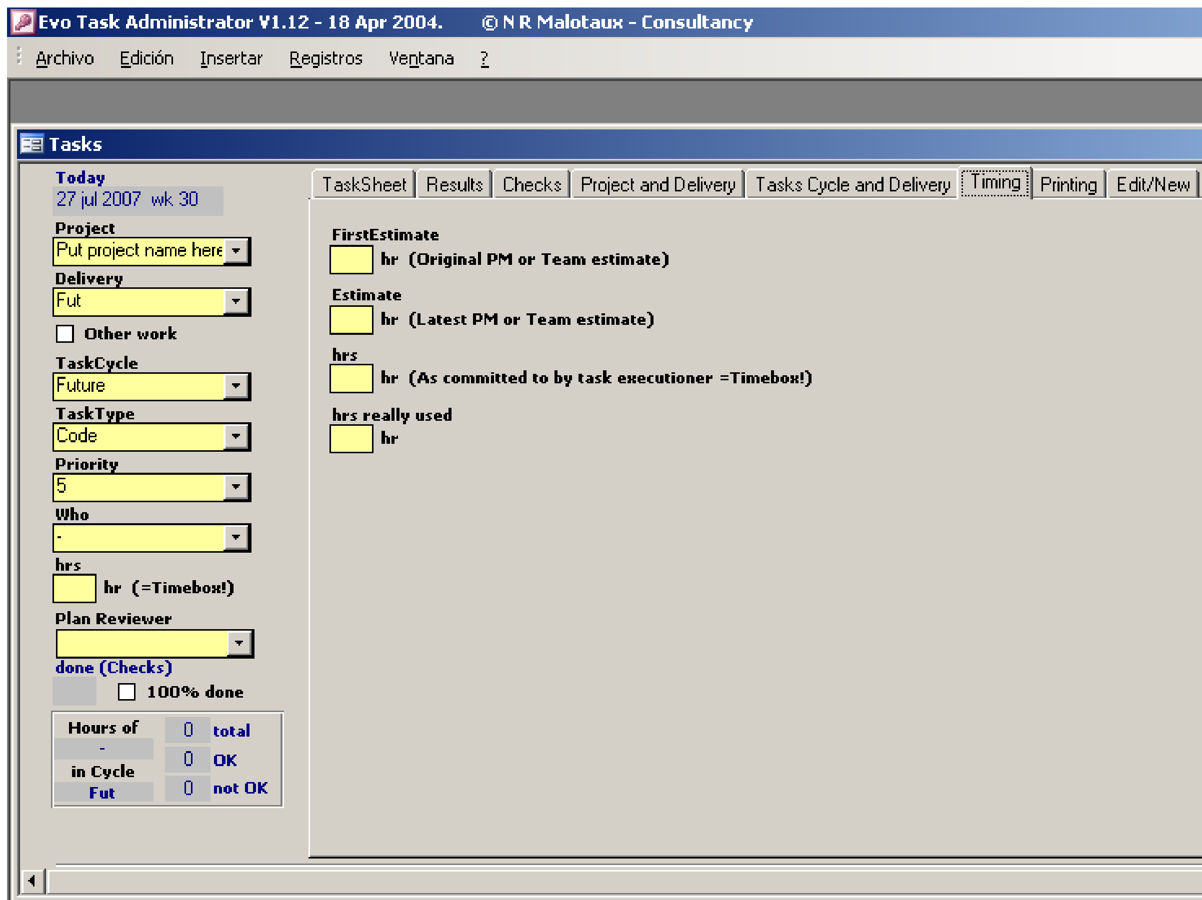


Ilustración 96. Calendario y estimaciones de duración de ciclo.

3.4. Rally

Rally Web-Services, www.rallydev.com, es un puente entre las herramientas de diseño, codificación, testeo, SCM o control de versiones con la gestión de ciclo de vida, gestión ágil de proyectos (plan y seguimiento de iteraciones) y gestión de programa. Con *Rally Agile Product Manager,* se conecta CRM (*Customer Relationship Management*) con la gestión ágil.

Rally usa los principios adaptativos y de *Lean Development* defendidos por Toyota y Dell para dar valor a los productos. Permite trabajar con múltiples equipos, ser accesible desde Internet, etc. Su precio es de 19, 39 u 85 dólares por mes y usuario, según la versión.

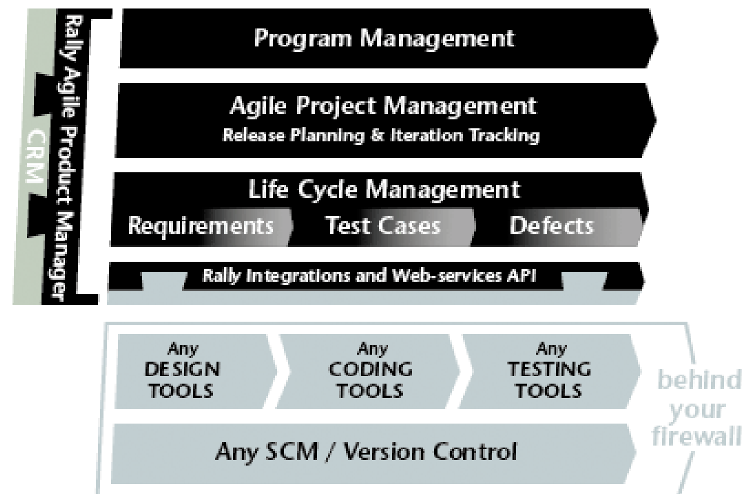


Ilustración 97. Relación entre los programas de Rally, CRM y las herramientas de desarrollo

GESTIÓN DE REQUISITOS

Rally ayuda a abrazar el cambio: seguir dependencias, decomponer características, *backlog*...

Ilustración 98. Detalle de una característica.

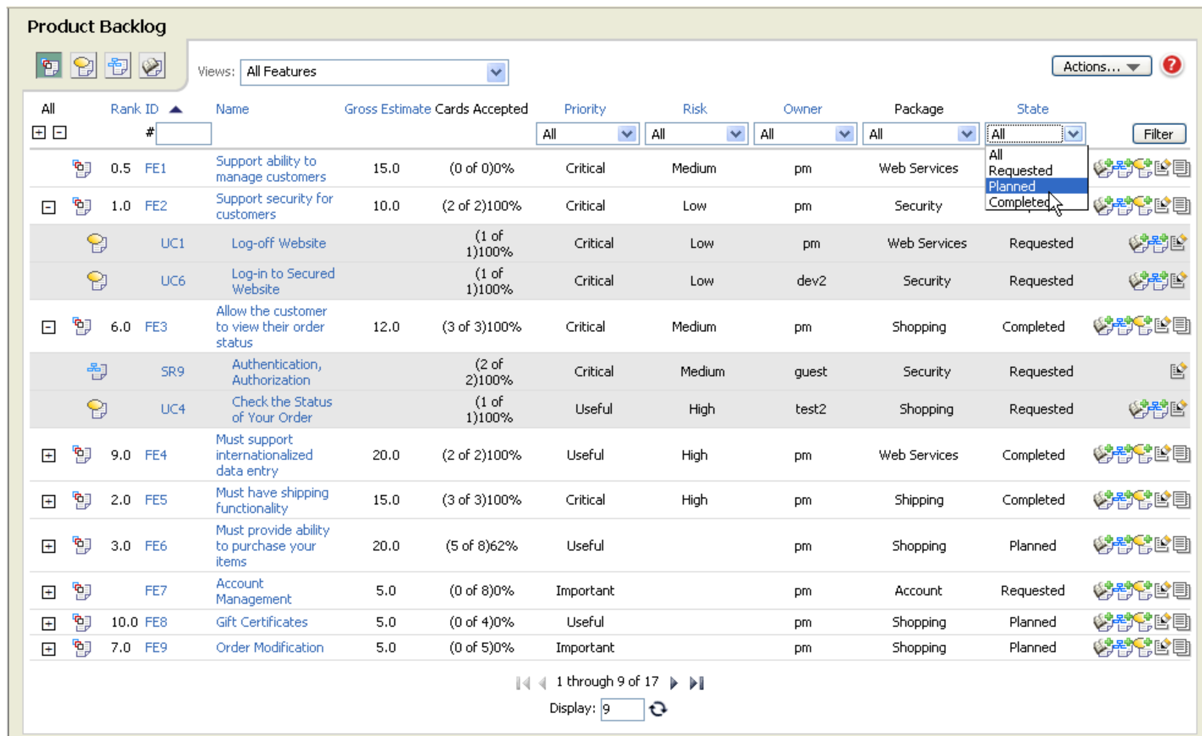


Ilustración 99. Product backlog.

Se puede enlazar con sistemas CRM para capturar *support cases* e informar a los empleados del avance del proyecto.

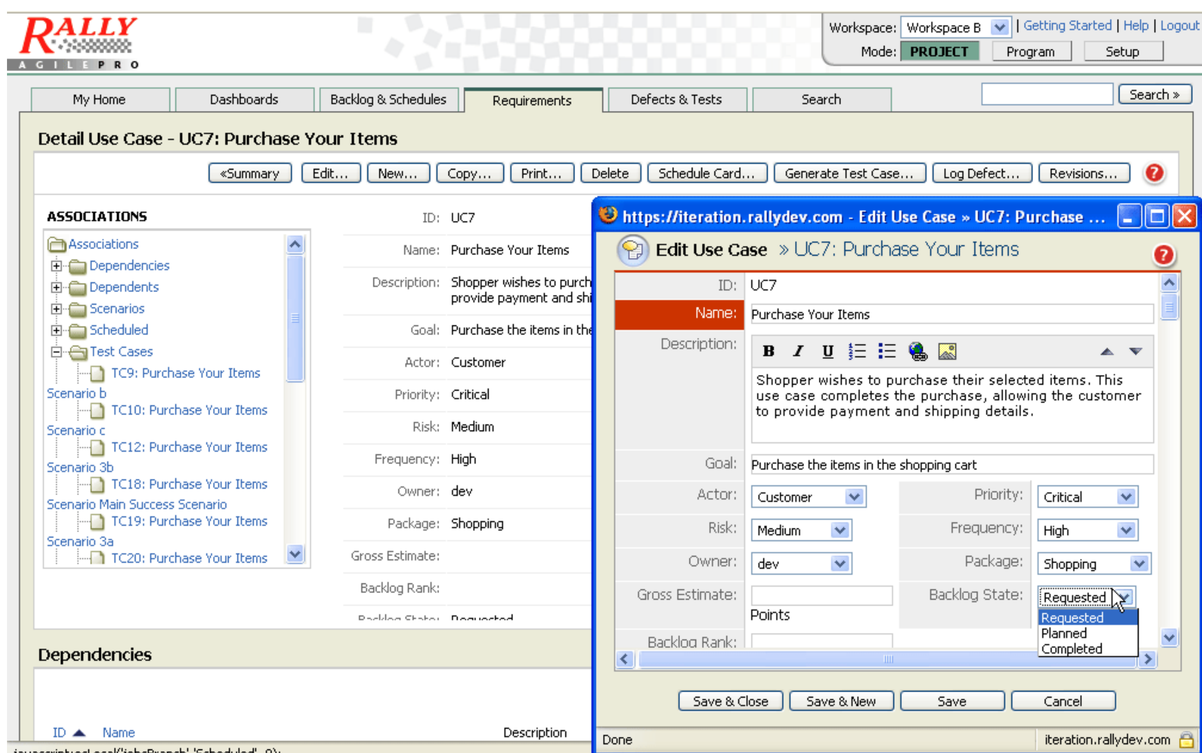


Ilustración 100. Editor de casos de uso para especificar requisitos.

TESTS Y CALIDAD

Rally permite generar *test cases* directamente desde los requisitos para facilitar el seguimiento.

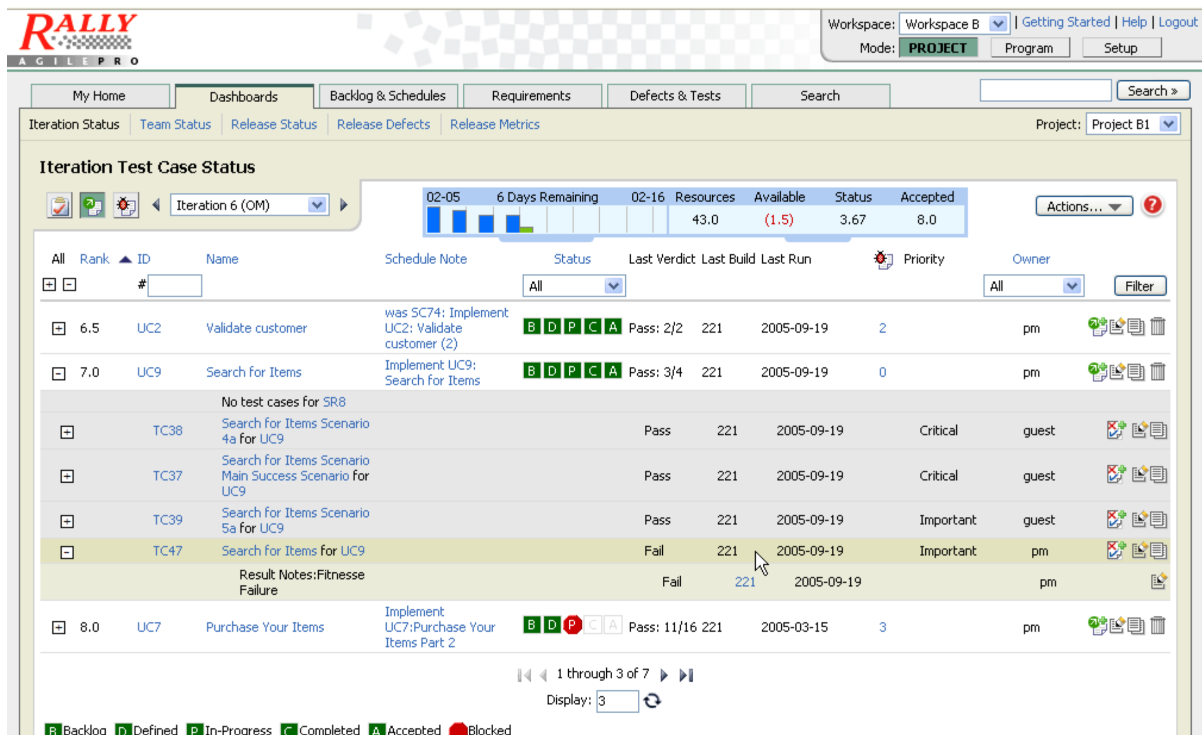


Ilustración 101. Estado de una iteración de un test case.

DISEÑO Y GESTIÓN DE TEST CASE

Rally automatiza la gestión de todos los tipos de test: aceptación, regresión, funcional, de sistema... Con los servicios web de Rally, es posible que entornos de test y desarrollo de software incluyan datos directamente en Rally.

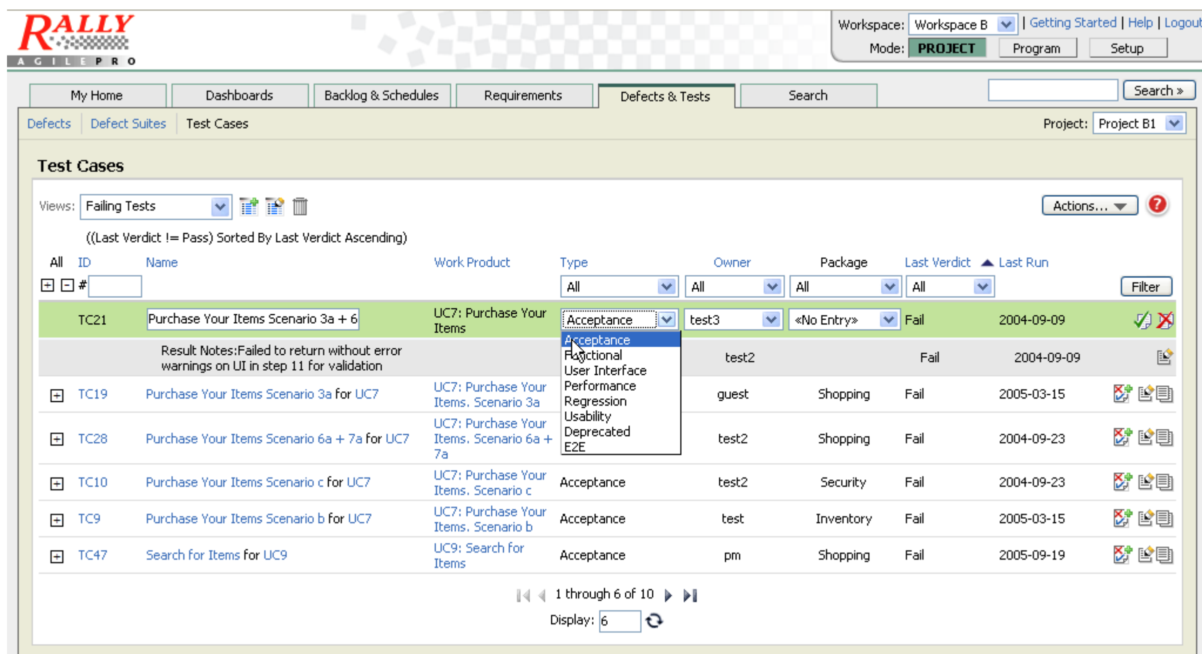


Ilustración 102. Resumen de test cases.

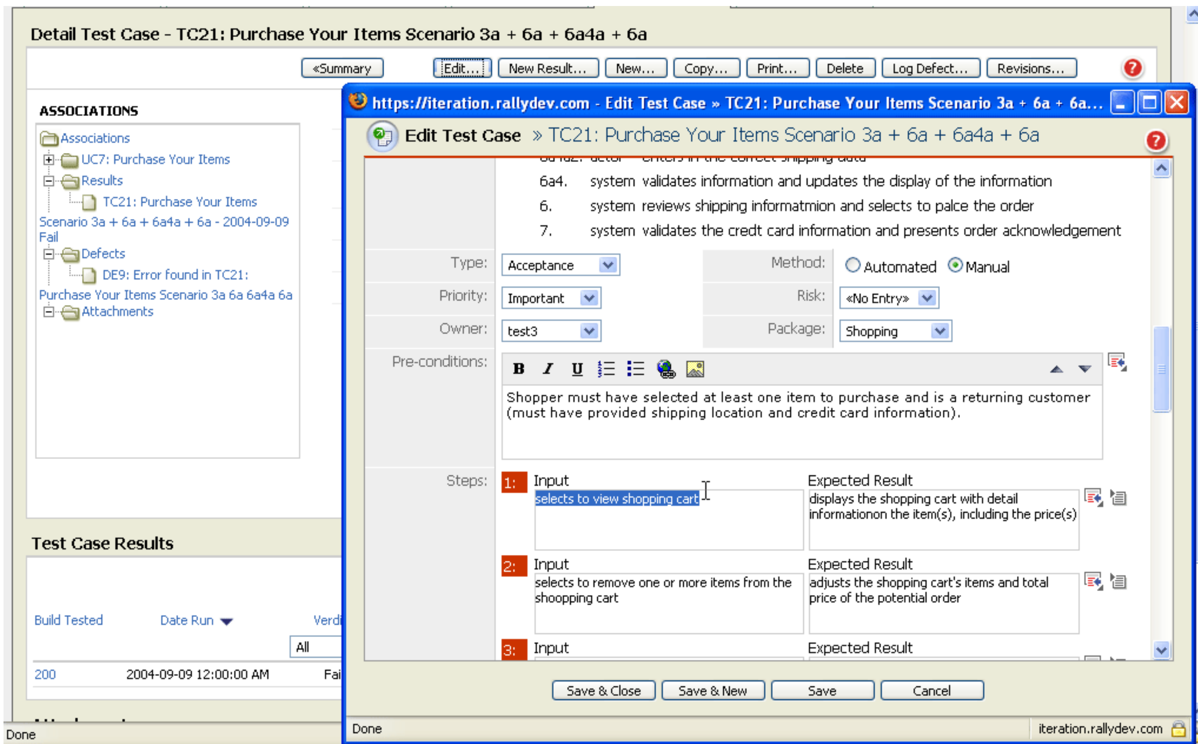


Ilustración 103. Editor de test cases.

GESTIÓN DE DEFECTOS

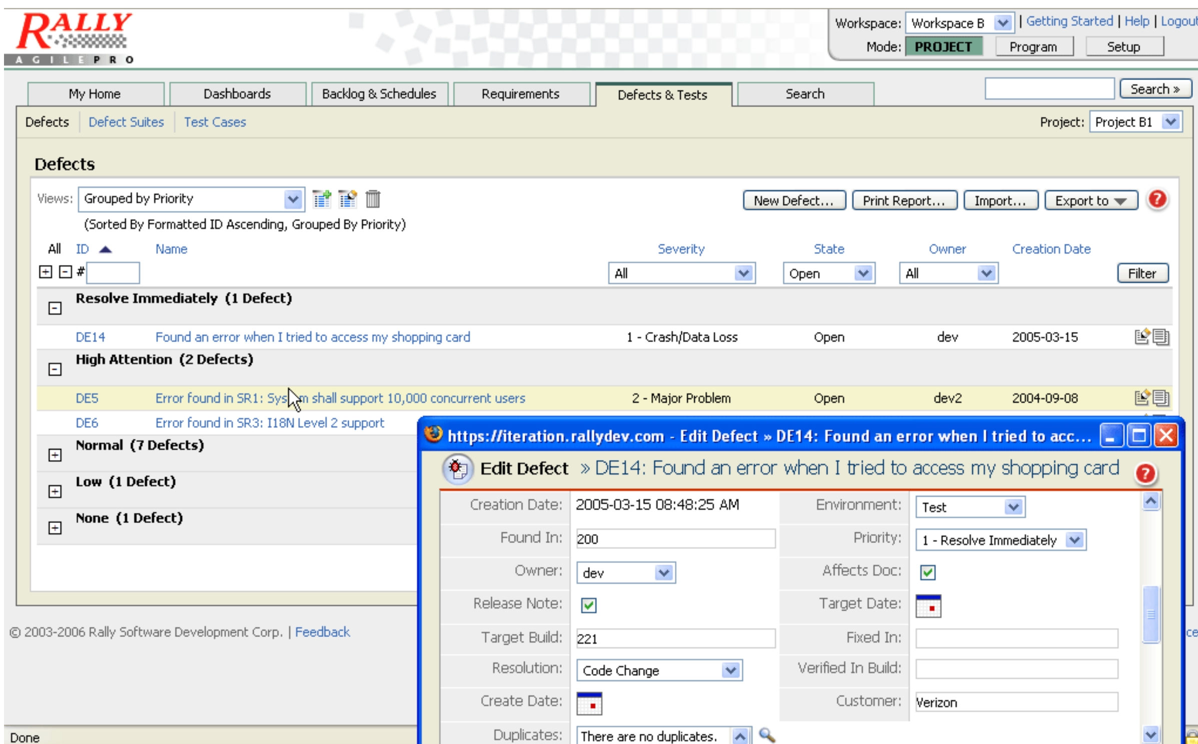


Ilustración 104. Resumen del estado de los defectos.

Ilustración 105. Resumen del estado de los defectos de toda la iteración.

Ilustración 106. Añadir un nuevo defecto o bug.

PERSONALIZAR RALLY

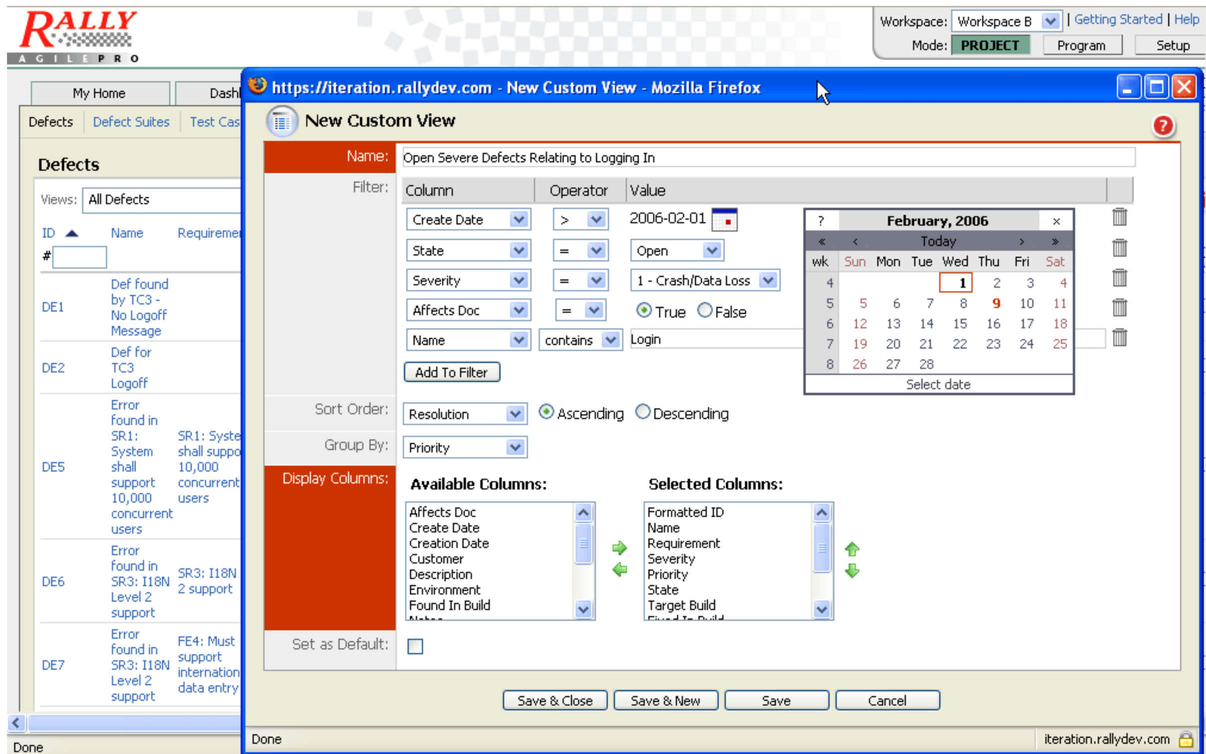


Ilustración 107. Vista personalizada de defectos.

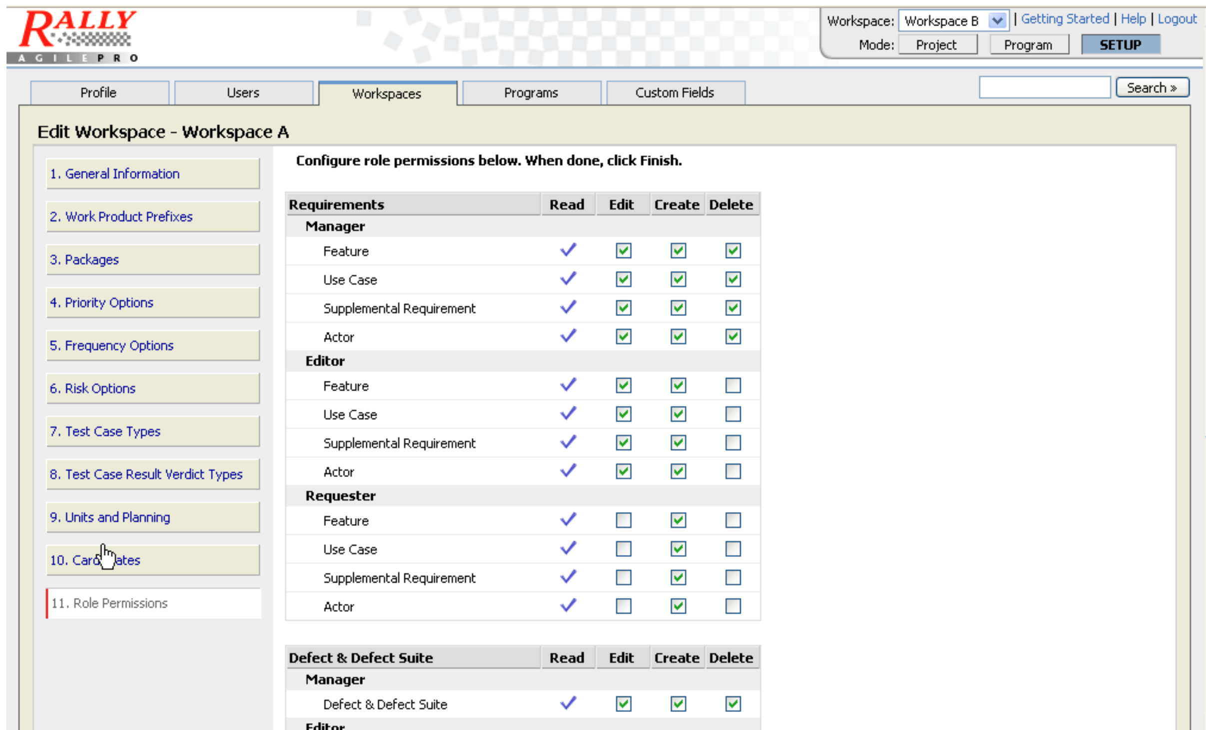


Ilustración 108. Configurar permisos para crear, editar y borrar, según el grupo de pertenencia.

COORDINAR PROGRAMAS DE VARIOS EQUIPOS

Rally es capaz de hacer el seguimiento de varios equipos trabajando para un mismo proyecto, con entregas de cada uno de ellos, calendarios de iteraciones, etc.

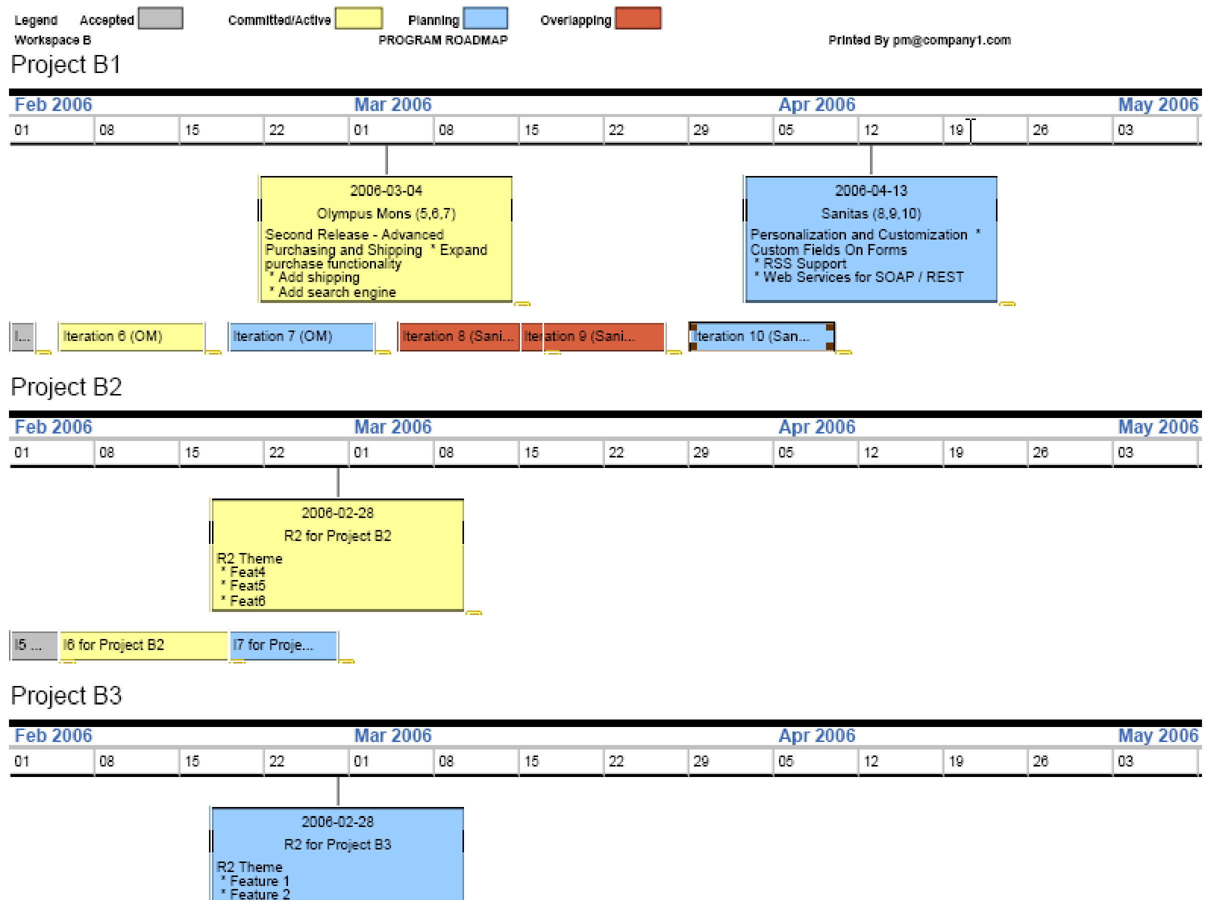


Ilustración 109. Seguimiento de varios equipos con Program roadmap.

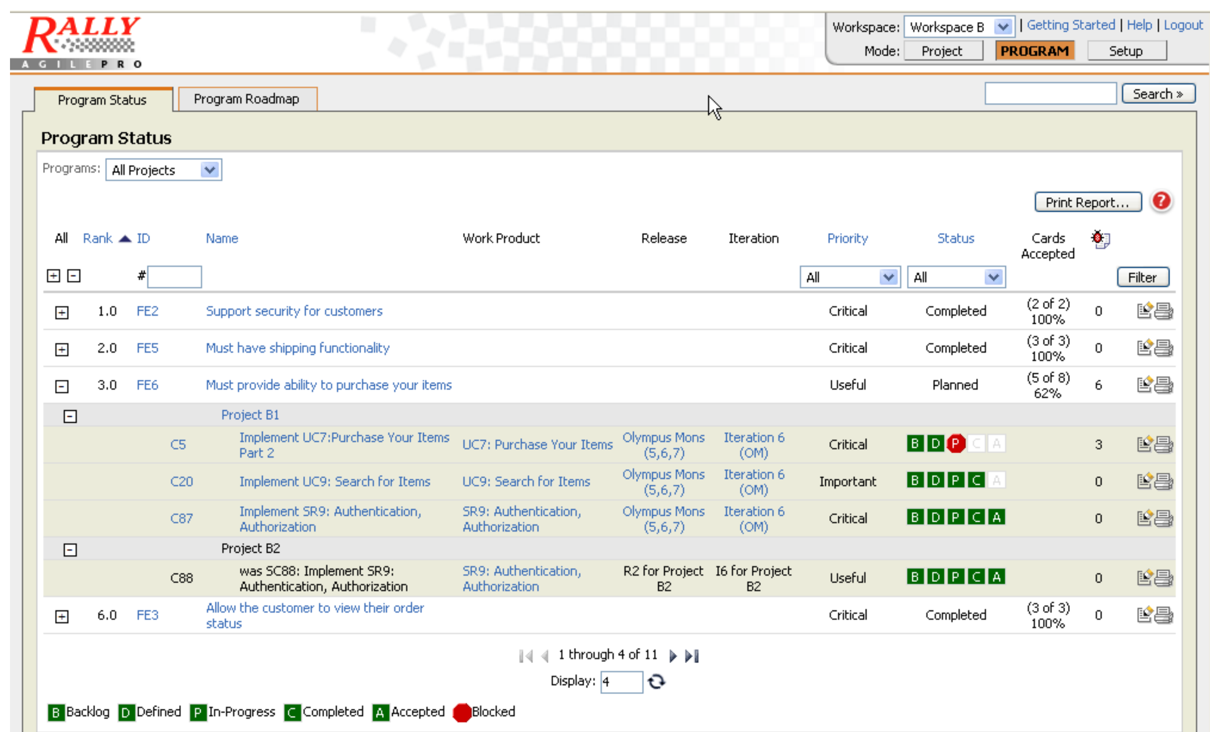


Ilustración 110. Estado global del programa.

PLANNING Y SEGUIMIENTO DE VERSIONES

Ilustración 111. Plan de la versión.

Ilustración 112. Estado de la versión.

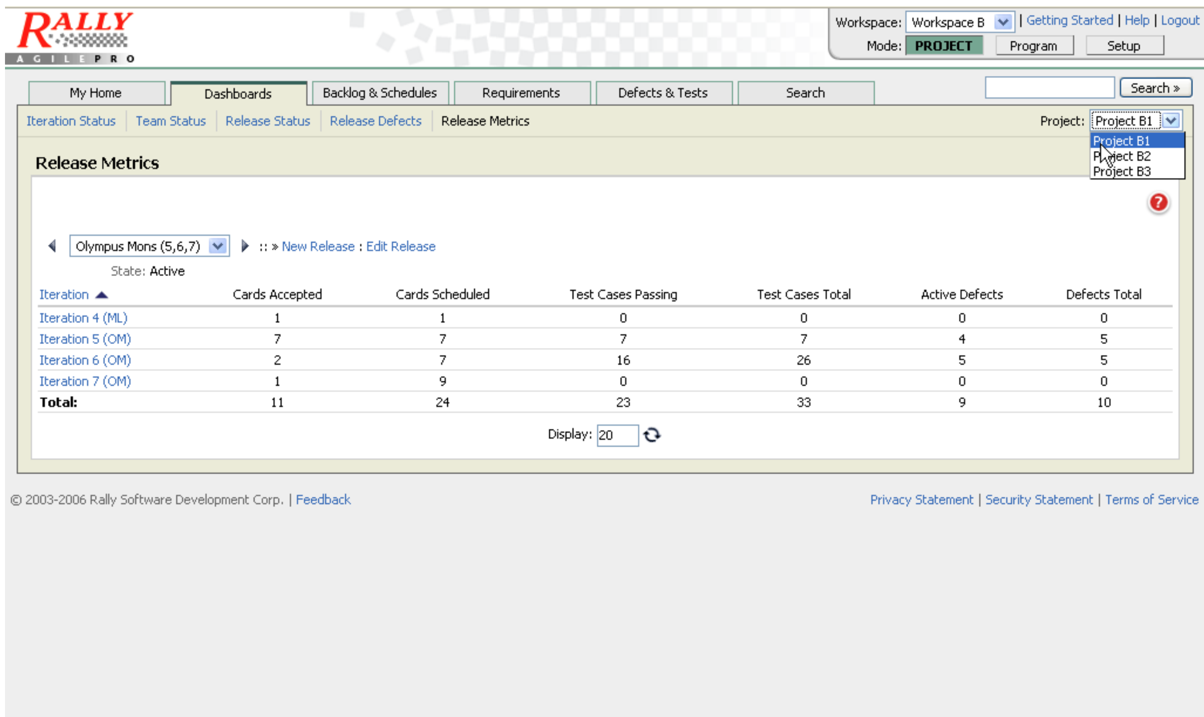


Ilustración 113. Métricas de la iteración.

PLAN DE ITERACIÓN Y ESTADO EN TIEMPO REAL

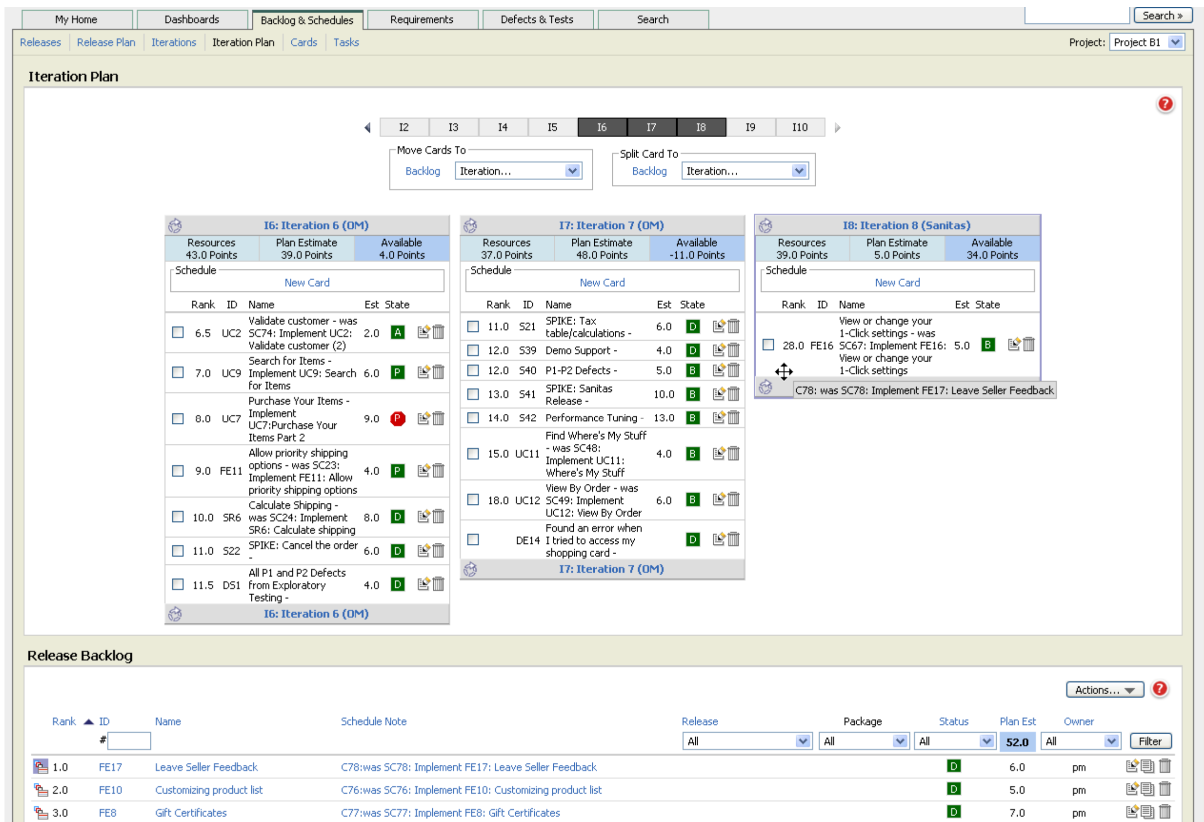


Ilustración 114. Plan de la iteración.

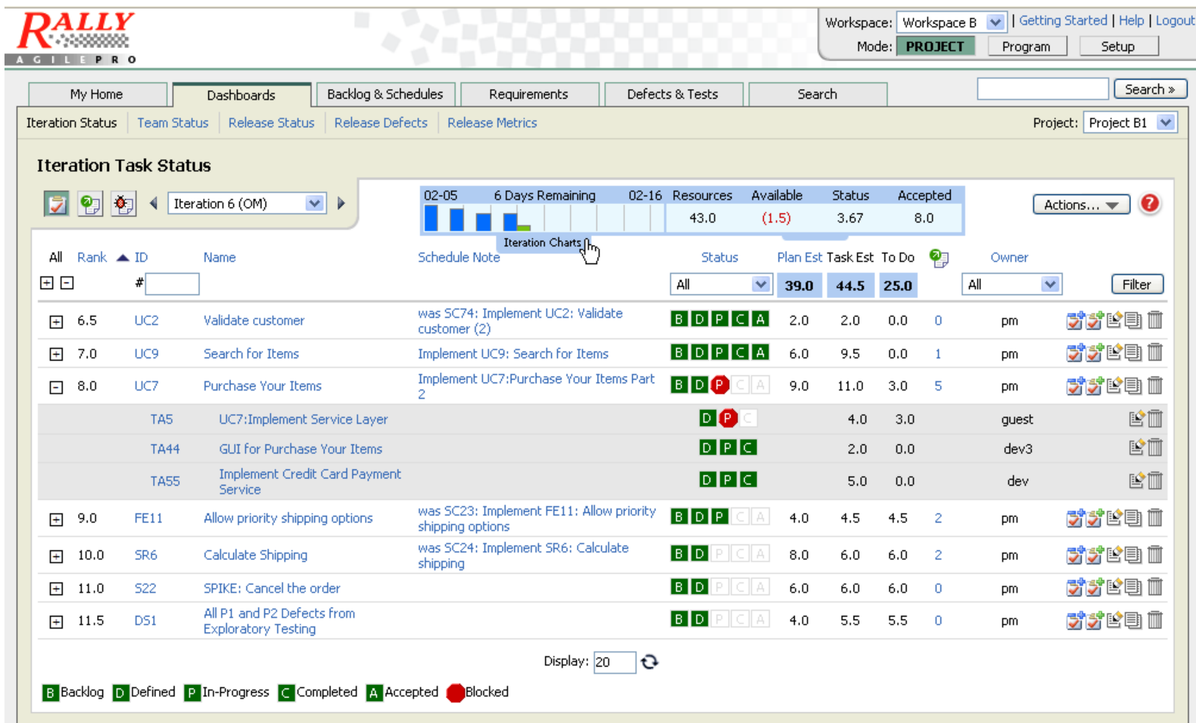


Ilustración 115. Estado de las tareas de una iteración.

Rally crea automáticamente varios gráficos de quemado a partir de los estados de las características, fechas, estimaciones, etc. El gráfico de quemado de la iteración representa las partes aceptadas y las restantes, mientras que el gráfico de quemado acumulativo representa los 5 estados: aceptado, completado, en progreso, definido y *backlog*.

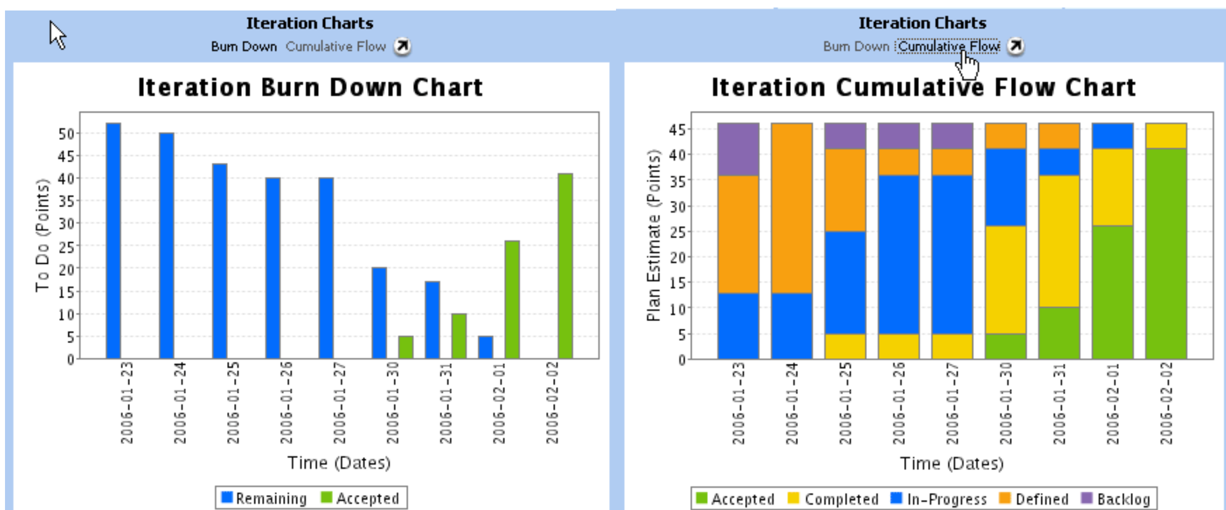


Ilustración 116. Gráficos de quemado de la iteración y acumulativo.

ENTORNO DE TRABAJO (WORKBENCH) PERSONALIZADO

Rally permite ver de forma rápida las notificaciones recibidas y filtrarlas: desde la última vez que se conectó, las de las últimas 24 horas, de la última semana o del último mes, filtrar por persona, etc. También puede enviar las notificaciones por e-mail o RSS, un formato de la familia de XML, especialmente diseñado para páginas web con constantes actualizaciones como es el caso.

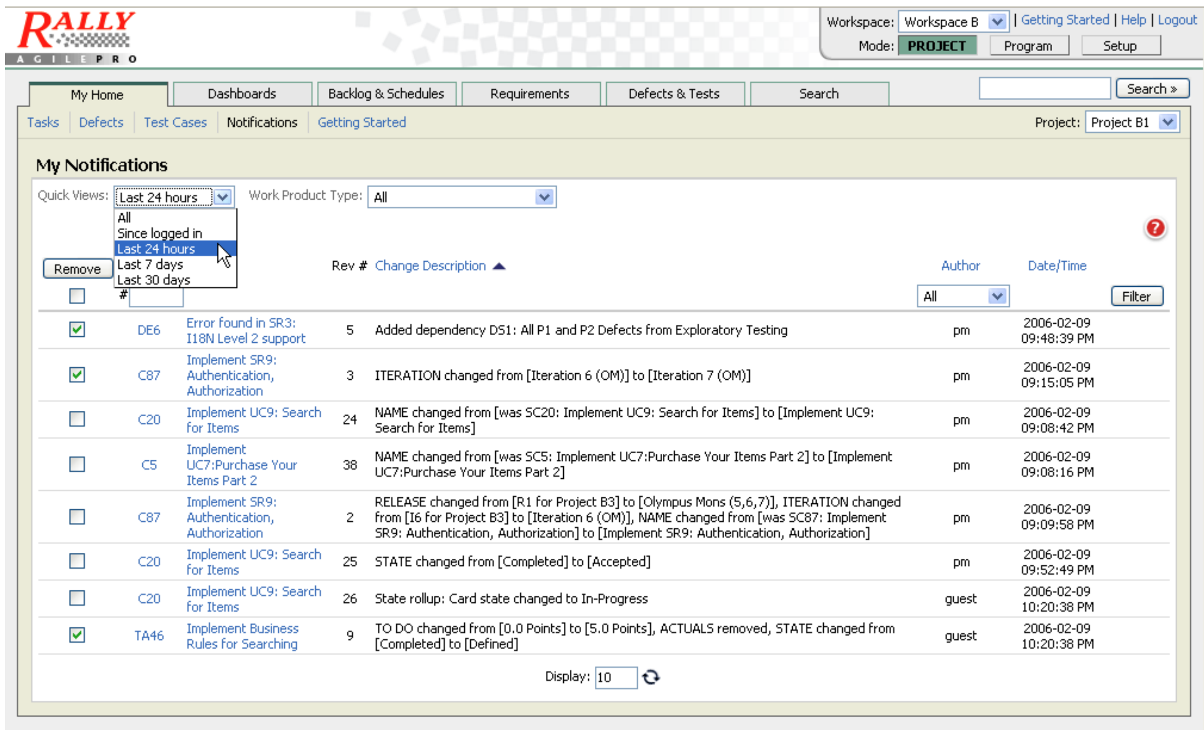


Ilustración 117. Ejemplo de notificaciones recibidas.

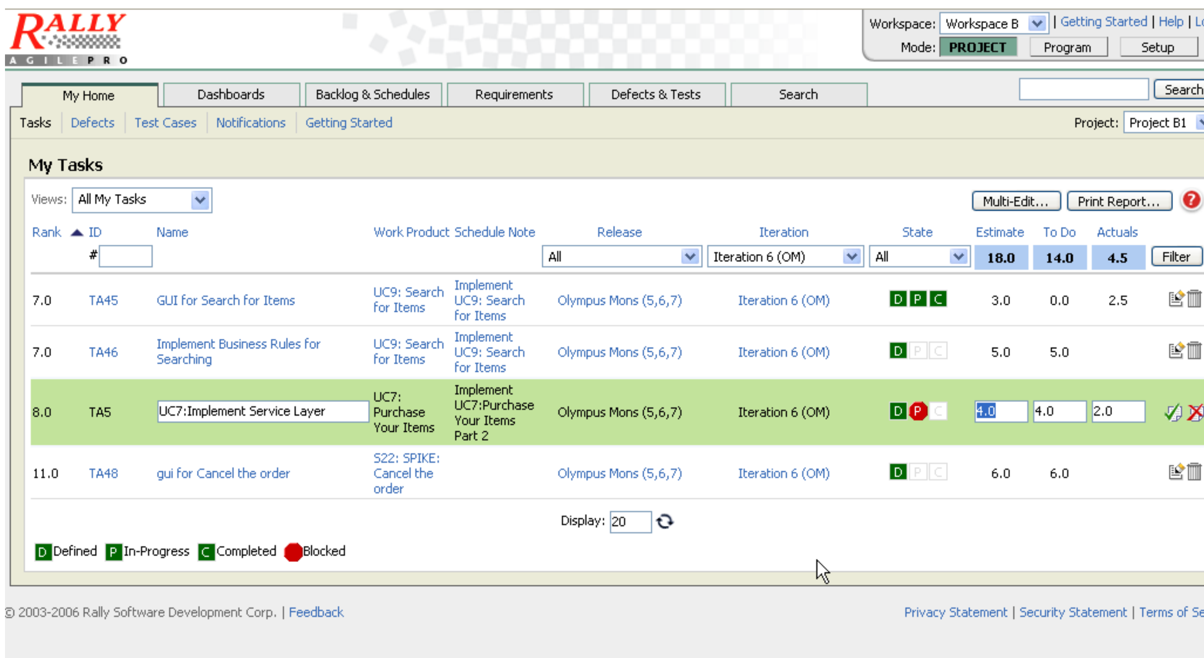


Ilustración 118. Vista de las tareas asignadas a una persona.

HERRAMIENTAS DE MARKETING

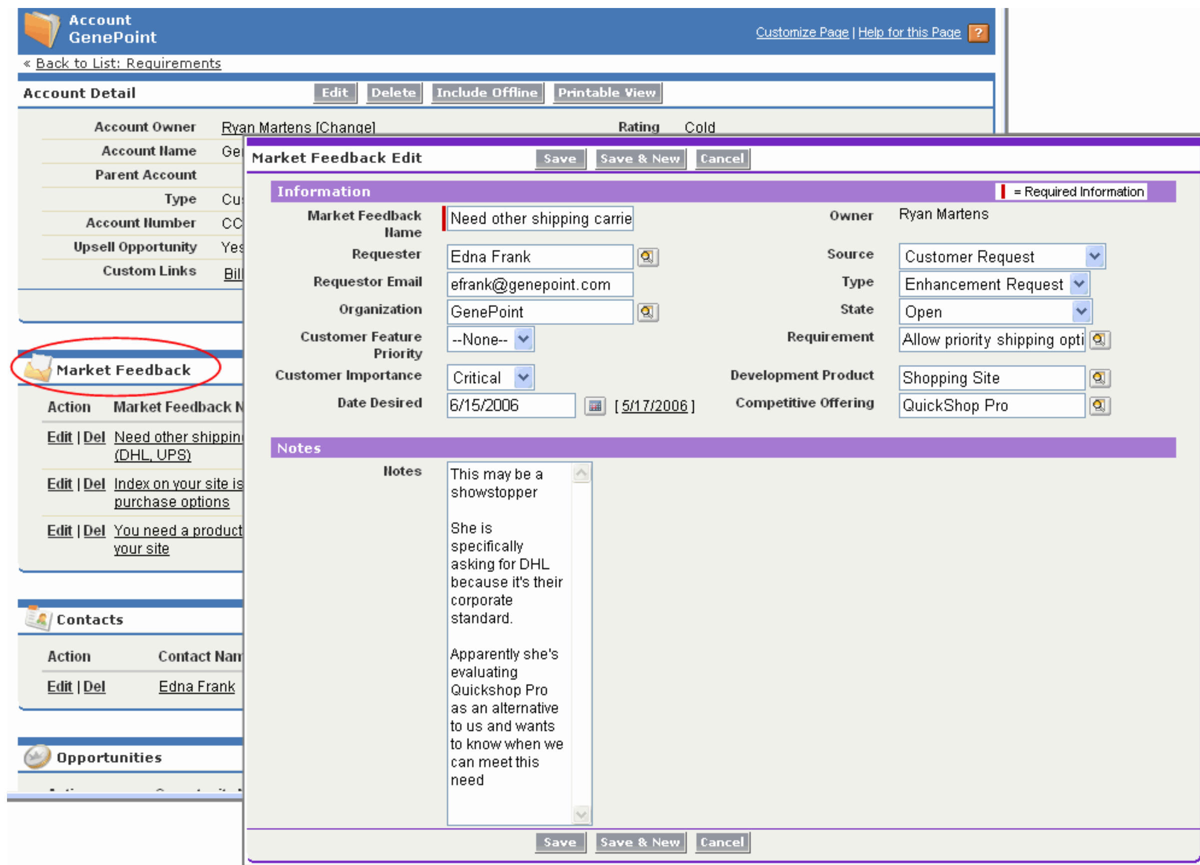


Ilustración 119. Recoger el feedback.

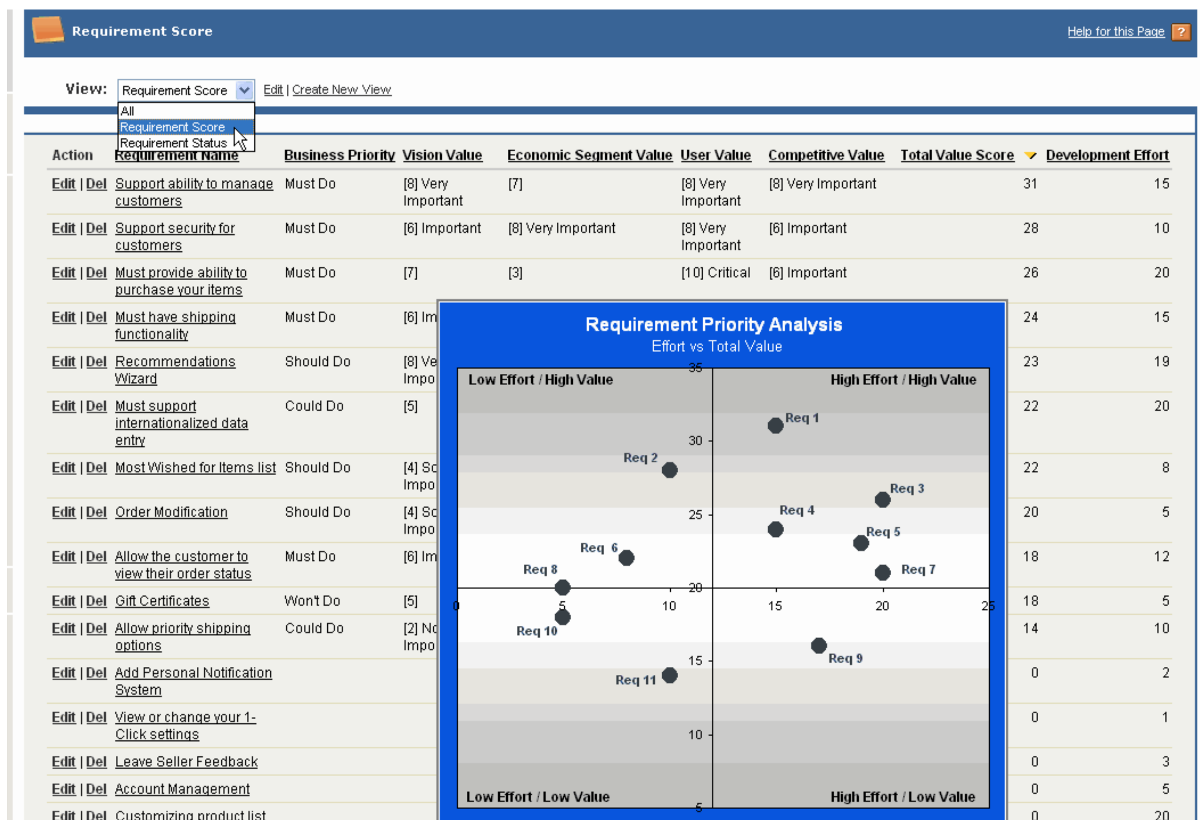


Ilustración 120. Valoración de los requisitos.

Ilustración 121. Detalle de requisito.

GESTIÓN DE VERSIONES Y ESTADO DE DESARROLLO

Ilustración 122. Página principal del "dashboard" (cuadro de mandos) de la versión.

Setup · Help · Logout AppExchange Agile Product Manager
POWERED BY SALESFORCE.COM

Product Releases | Requirements | Market Feedback | Product Families | Accounts | Reports | Dashboards | Rally Help & Support

Product Release
Online Shopping Site Release 1

Customize Page | Help for this Page

« Back to List: Market Feedback

Product Release Detail Edit Delete Clone Printable View

Release Name	Online Shopping Site Release 1	Owner	Ryan Martens [Change]
Theme			
Rally Release	https://demo.rallydev.com/slm/desktops/project....		
Release Date			
Development Product			
Custom Links	Associate Rally Release		
Created By	Ryan Martens, 5/16/2006 2:30 PM	Last Modified By	Ryan Martens, 5/16/2006 3:34 PM

Edit Delete Clone Printable View

Issues New — Issues Help

Action	Description	Owner Alias	Due Date	Priority
Edit | Del	International Tax calculation location	RMart	5/31/2006	
Edit | Del	Support Training of 24X7 partner delayed 10 days	RMart	6/8/2006	Low
Edit | Del	Lead Developer - out for week	RMart	5/25/2006	Medium
Edit | Del	Catalog Feed very limited for Europe	RMart	5/30/2006	High

Open Activities New Task New Event — Open Activities Help

Action	Subject	Name	Task	Date	Status	Priority	Assigned To
Edit | Cls	Marketing campaign plan		✓	5/24/2006	In Progress	High	Jami Yannett
Edit | Del	Weekly status meeting			5/24/2006 9:00 AM			Ryan Martens
Edit | Del	Shopping cart review with team			5/24/2006 3:00 PM			Ryan Martens
Edit | Cls	Sales collateral plan		✓	5/25/2006	In Progress	Normal	Jami Yannett
Edit | Cls	Online advertising plan		✓	5/25/2006	In Progress	Normal	Jami Yannett

View More »

Activity History Log A Call Mail Merge Send An Email View All — Activity History Help

No records to display

Notes & Attachments New Note Attach File View All — Notes & Attachments Help

Action	Type	Title	Last Modified
Edit | Del | View	Attachment	Implementing Agile Dev Practices Workshop.pdf	5/16/2006 3:35 PM
Edit | Del | View	Attachment	Rally Features Datasheet.pdf	5/16/2006 3:35 PM

Requirements New — Requirements Help

Action	Requirement Name	Description	Dev Product	Development Status
Edit | Del	Support ability to manage customers	The system must have the ability to manage customers, create, update, retire, re-instate		Complete
Edit | Del	Support security for customers	Need the ability to prevent unauthorized users from accessing secured areas. Only registered customers will be allowed to do certain things.		Complete
Edit | Del	Must have shipping functionality	The system must have the capability of communicating with the shipping components of the order.		Complete
Edit | Del	Must provide ability to purchase your items	Use a shopping cart type metaphor		In-Progress
Edit | Del	Allow the customer to view their order status	The customer should have the ability to view the detail of their order and the status of the order such as shipping state, etc.		Complete

View More (9) »

Show me ▾ more records per related list

Home | Release Dashboard | Product Releases | Requirements | Market Feedback | Product Families | Accounts | Reports | Dashboards | Rally Help & Support | All Tabs

Ilustración 123. Detalle de la versión.

3.5. VersionOne Agile Team y Agile Enterprise

V1: Agile Enterprise, www.versionone.net, es una herramienta basada en web para planear y gestionar proyectos ágiles, con plantillas para Scrum, XP, DSDM y Agile UP. Gestiona requisitos, características, tareas, problemas, defectos y tests. Soporta muchas prácticas ágiles: planear por características, planning de versiones, de iteraciones, supervisión, gestión de tareas y test, informes de velocidad y gráficos de quemado, etc. Es capaz de gestionar proyectos donde interoperen varios equipos.

Su funcionamiento, es muy intuitivo: pestañas, arrastrar y soltar, menús y árboles desplegables, por ejemplo para ver todas las tareas de una historia o las historias de una característica...

V1: Agile Team (hasta 5 usuarios) cuesta 995 dólares/año (hasta 17 usuarios) y *V1: Agile Enterprise* (sin límite de usuarios) cuesta 30 dólares por usuario y mes o 500 dólares por usuario.

PLANNING

Planear, estimar, y priorizar características. Dispone de un entorno parecido a MS Excel, y además permite realizar operaciones en modo consola (*batch*) para ser más rápido. Arrastrando podemos asignar o mover características o defectos a las iteraciones.

Ilustración 124. Vista general del plan de una iteración.

SEGUIMIENTO

Se puede realizar por usuario, por característica, test, proyecto, prioridad, estado, etc.

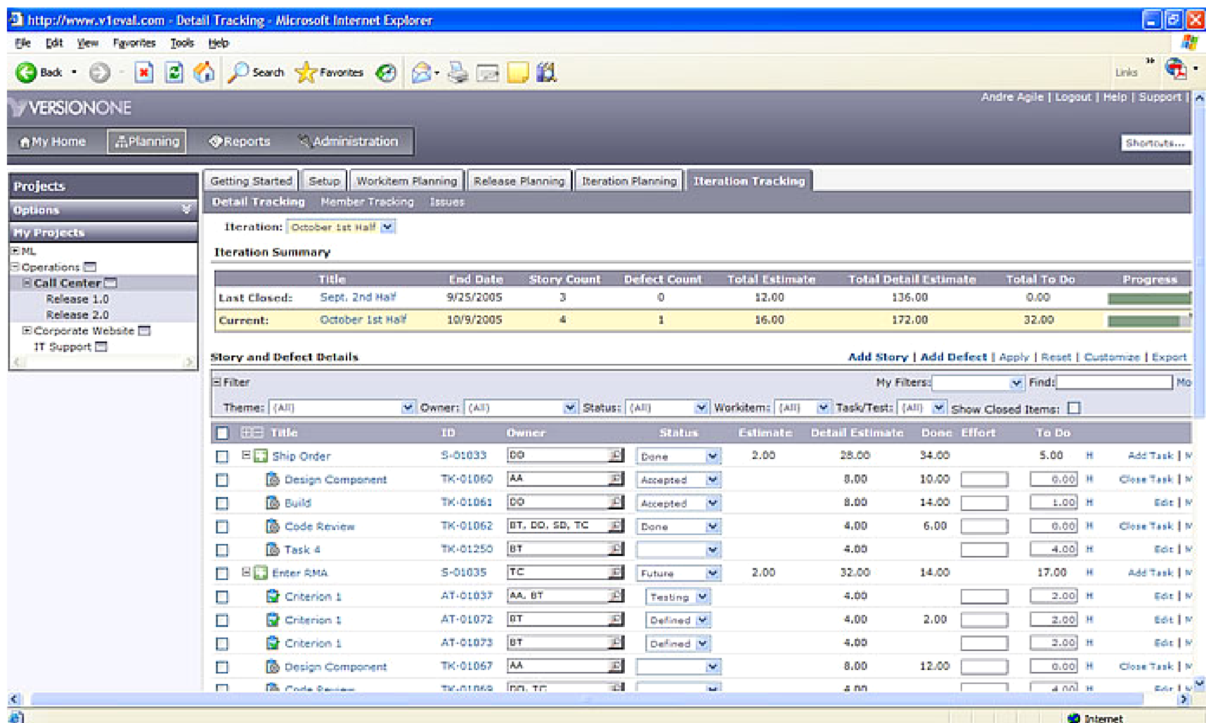


Ilustración 125. Seguimiento de la iteración.

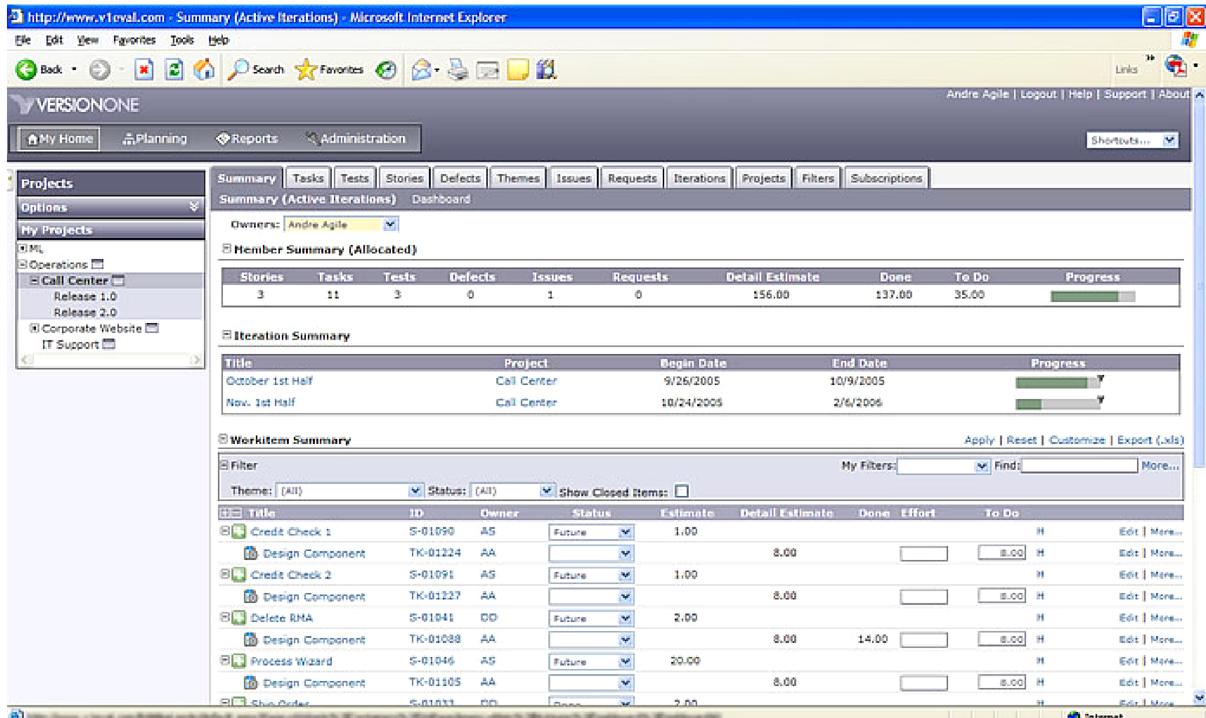


Ilustración 126. Resumen de la iteración.

INFORMES Y ANÁLISIS

V1: Agile Enterprise proporciona informes y análisis en todos los niveles: ejecutivo, programa, versión, proyecto, iteración (sprint en Scrum) y miembro.

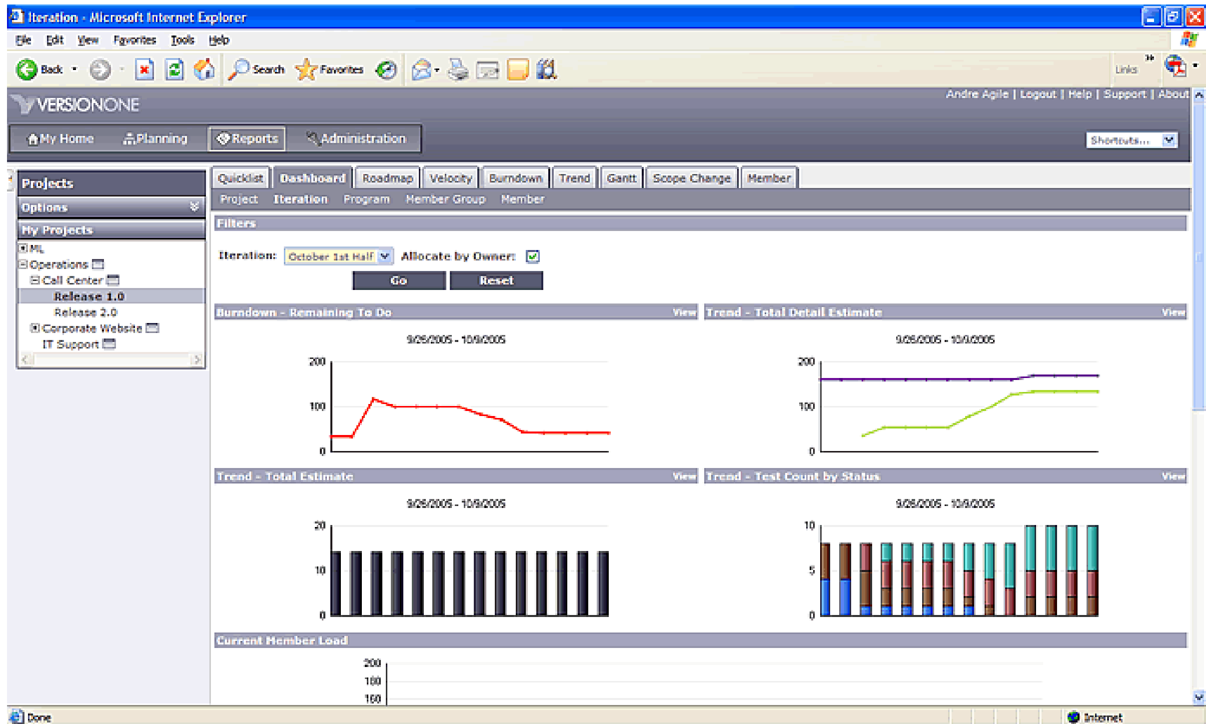


Ilustración 127. Agile Enterprise dispone de más de 50 gráficos predefinidos.

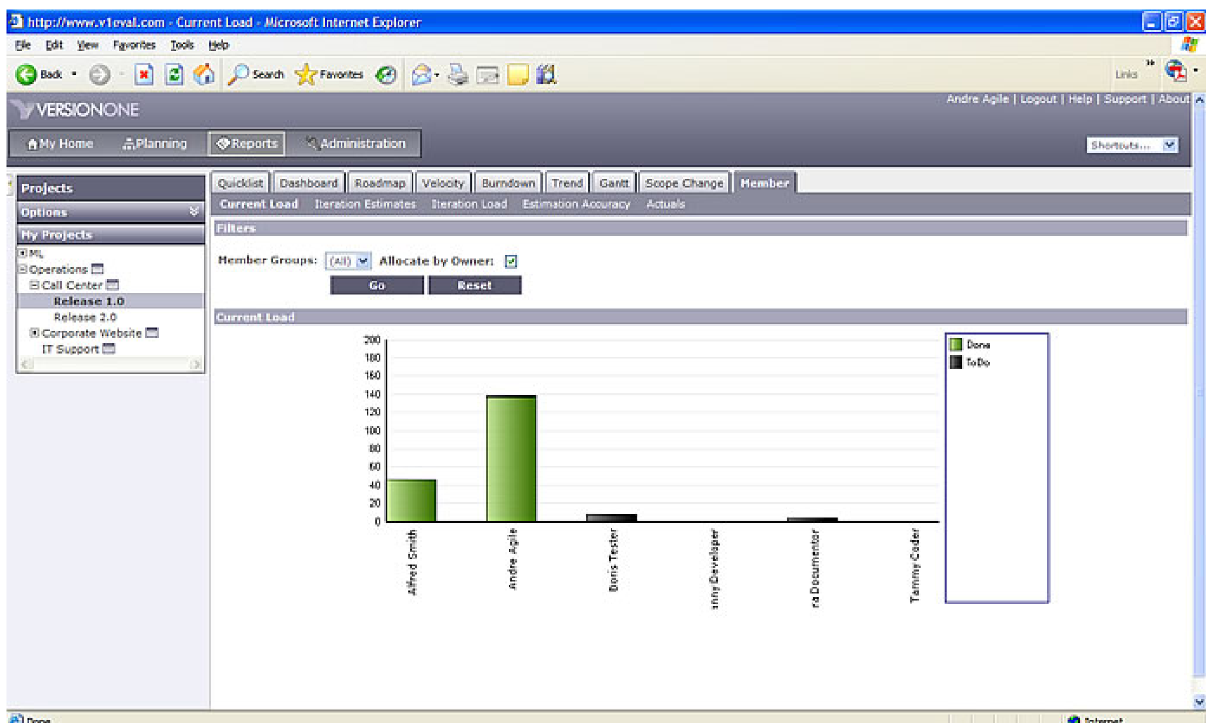


Ilustración 128. Horas de las tareas asignadas a cada persona.

3.6. TargetProcess

TargetProcess, www.targetprocess.com, simplifica las tareas de planear, hacer el seguimiento y control de calidad de los proyectos de software. Su precio es 249 dólares por licencia (una por usuario).

CUADRO DE MANDOS PERSONALIZABLE

El cuadro de mandos (*dashboard*) es la página de inicio de TargetProcess. Proporciona información accesible sobre el estado del proyecto, últimas actividades, tiempo total invertido, lista de tareas, etc. Se pueden elegir los componentes necesarios para hacer un *dashboard* totalmente a medida.

Ilustración 129. La página de inicio de TargetProcess es personalizable.

Los posibles componentes en el *dashboard* o cuadro de mandos son: lista de tareas, lista de nuevas entidades, tiempo por usuario, resumen de progreso, gráfico de quemado y progreso de iteración diaria.

DESARROLLO ITERATIVO

TargetProcess soporta el desarrollo iterativo de cualquier tamaño: se pueden usar sólo releases, iteraciones e historias de usuario o añadir características y tareas para proyectos mayores.

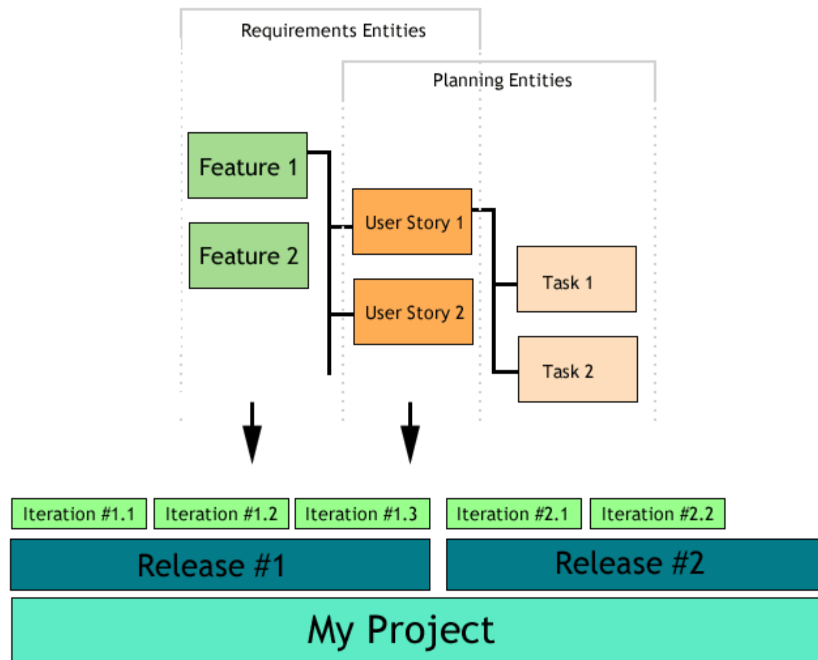


Ilustración 130. Descomposición de un proyecto en características, historias y tareas.

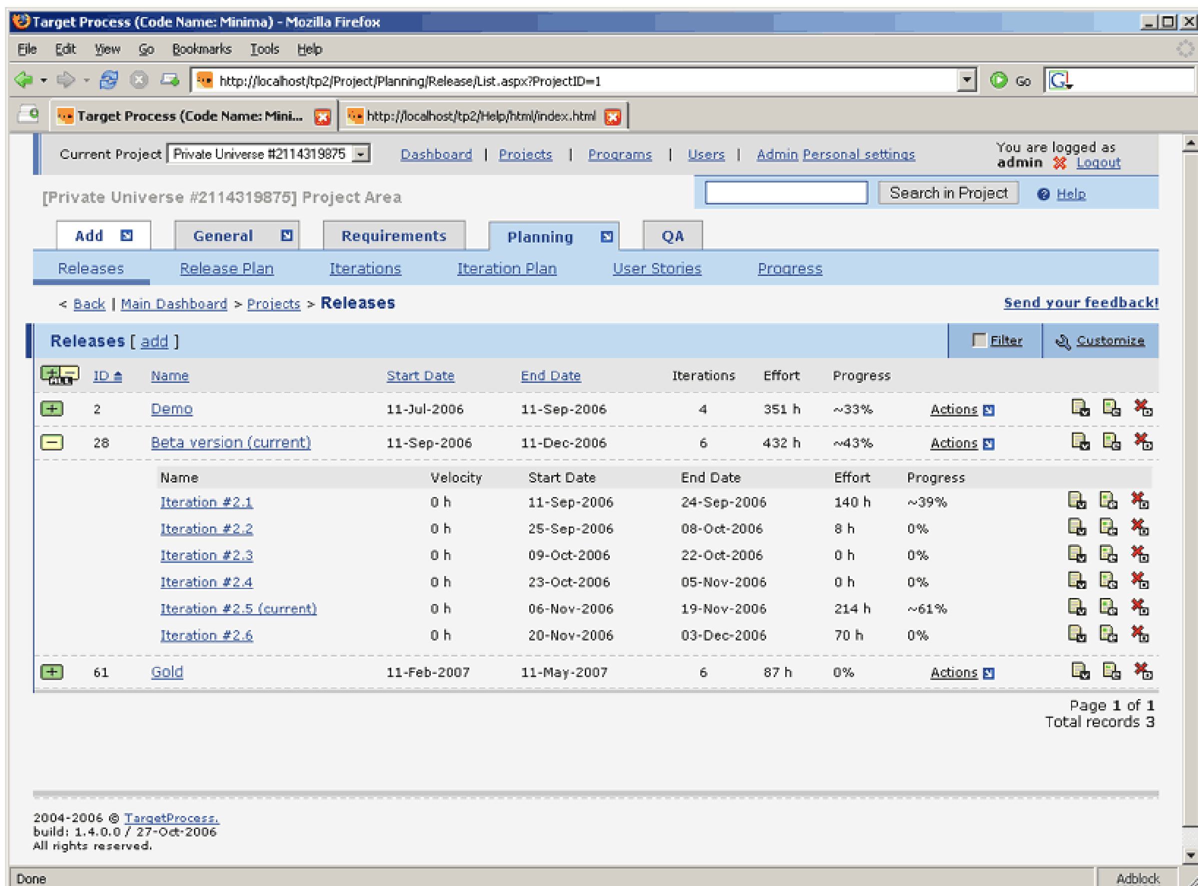


Ilustración 131. Control de releases: fechas, horas, progreso, estado de las iteraciones, etc.

RELEASES Y PLAN DE ITERACIONES

Arrastrando y soltando podemos crear planes y el plan de cada iteración *(sprint)*. TargetProcess controla el esfuerzo total asignado y el estado de las historias de usuario.

Ilustración 132. Para mover cualquier elemento, sólo hay que arrastrar y soltar.

Se pueden aplicar historias o *bugs* a cada iteración, así como después filtrar según prioridades, por ejemplo. El plan de iteración está totalmente basado en AJAX: todas las acciones se ejecutan en la máquina del cliente para mayor velocidad.

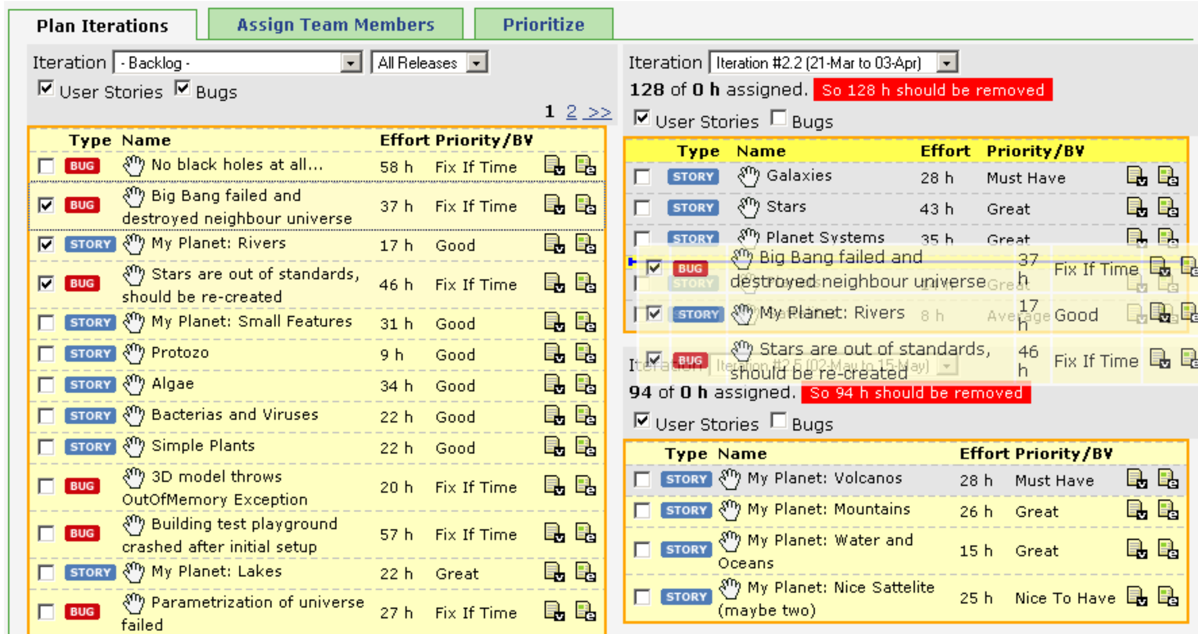


Ilustración 133. Basta con arrastrar y soltar para mover historias o bugs a las iteraciones.

PERSONALIZACIÓN DEL PROCESO DE DESARROLLO

TargetProcess permite crear procesos personalizados para cada proyecto. Cada proceso consta de prácticas como Planning, seguimiento temporal, seguimiento de *bugs*, gestión a alto nivel, *test cases* o *source control*.

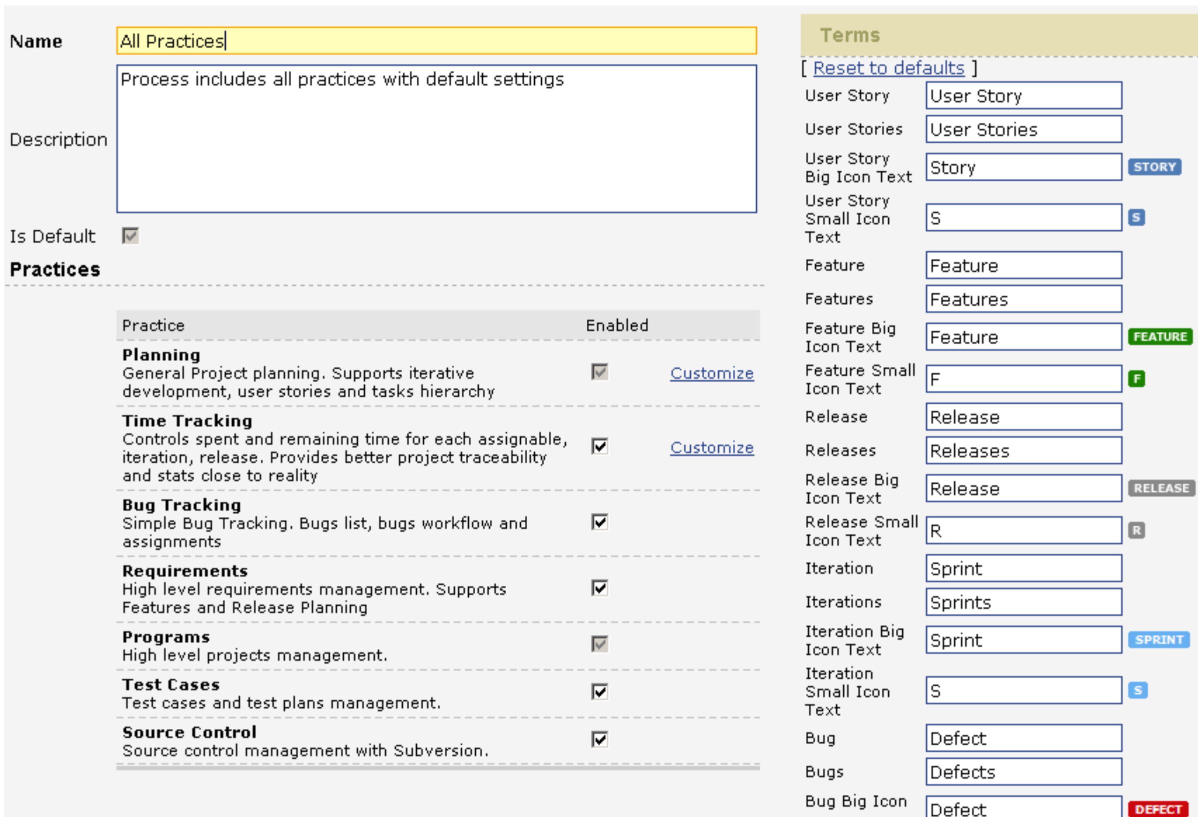


Ilustración 134. Es posible personalizar las partes de cada proceso.

REAL TIME PROGRESS TRACKING

TargetProcess lleva el control de historias asignadas y completadas, número de bugs, y relación de horas de esfuerzo estimadas y completadas.

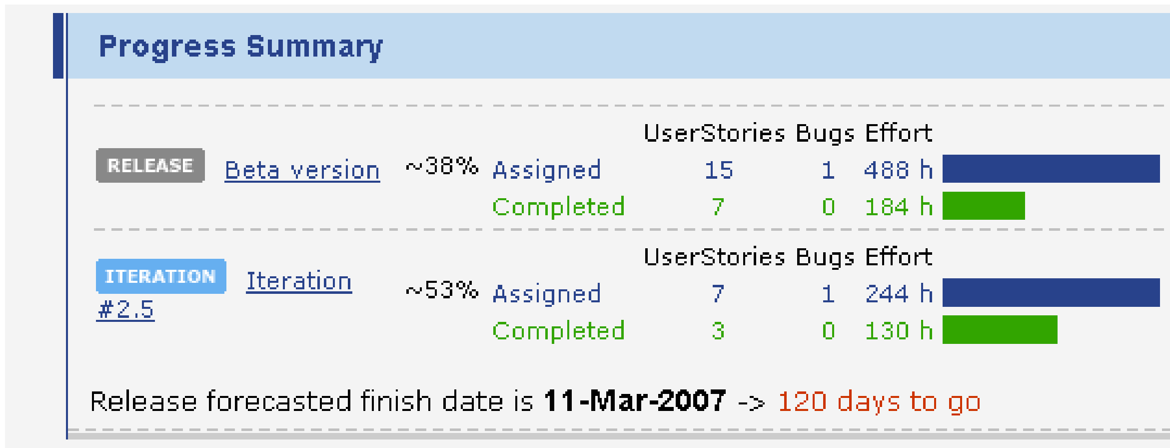


Ilustración 135. Time Tracking facilita un seguimiento preciso y completo de la iteración.

Progress for current iteration Iteration #2.5 / from 06-Nov-06 to 19-Nov-06

Show Info

Type	Name	Team	Effort	06-Nov	07-Nov	08-Nov	09-Nov	10-Nov	Today
STORY	Galaxies	Developer(s) Unassigned	28				26		14
TASK	Put Galaxy points	Developer(s) Teddy Bear	5						
TASK	Control and Correct process	Developer(s) Administrator	14				12		0
STORY	Stars	Developer(s) Unassigned	43						
TASK	Put Stars Points	Developer(s) Administrator	15						
TASK	Control and Correct process	Developer(s) Tom Cat	19						
STORY	Planet Systems	Developer(s) Teddy Bear	35	14	20		6	0	
STORY	Planets	Developer(s) Jerry Mouse	14		6			0	
STORY	My Planet: Nice Sattelite (maybe two)	Developer(s) Teddy Bear	25			22		8	
STORY	My Planet: Water and Oceans	Developer(s) Tom Cat	15				20	12	2
STORY	My Planet: Volcanos	Developer(s) Tom Cat	28			14	7		0
BUG	Frustrated problems with life setup	Developer(s) Administrator Verifier Unassigned	56						

Red remaining time increased
Yellow remaining time decreased, but spent effort already exceeds the estimate
Green remaining time decreased and spent effort does not exceed the estimate

Ilustración 136. Descomposición de historias en tareas, mostrando si están retrasadas.

Los gráficos de quemado ayudan a saber cómo ha cambiado la velocidad durante las iteraciones, cuándo se completará la versión, porcentajes ya hechos...

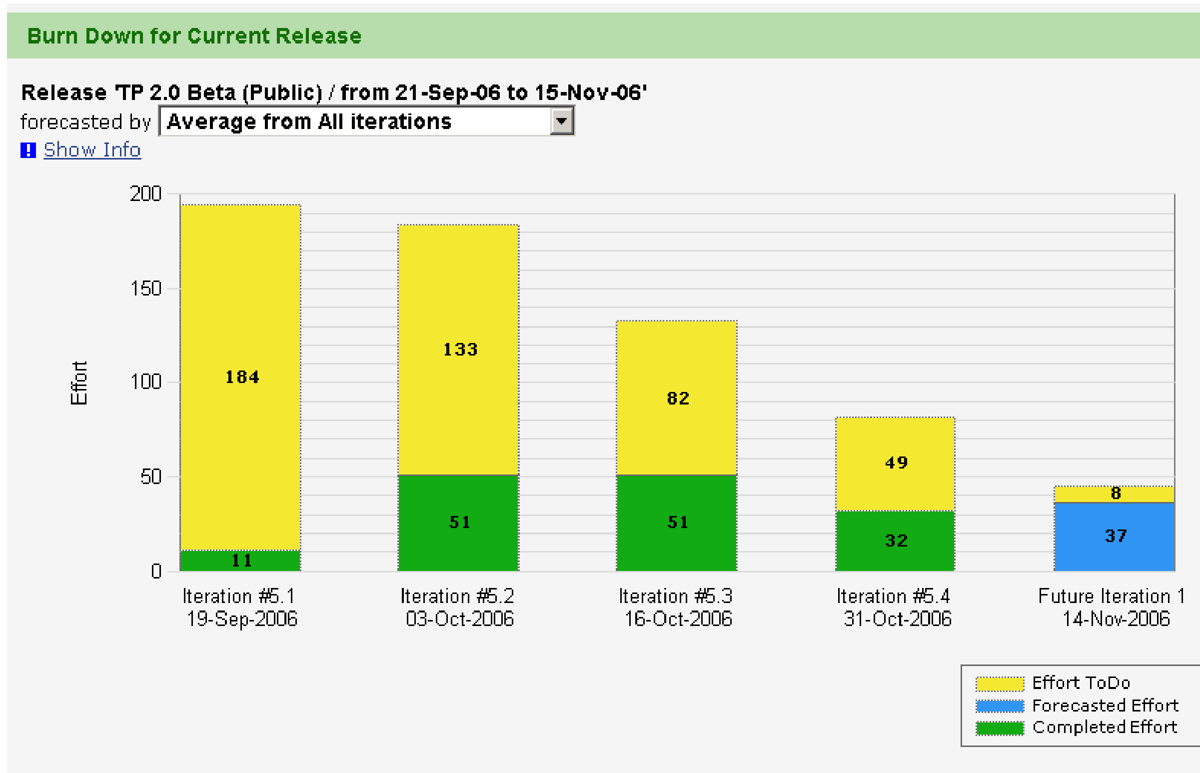


Ilustración 137. Gráfico de quemado: esfuerzo total, estimado y completado.

ASIGNACIÓN O REASIGNACIÓN DE USUARIOS

Las asignaciones de los desarrolladores se pueden hacer desde cualquier sitio.

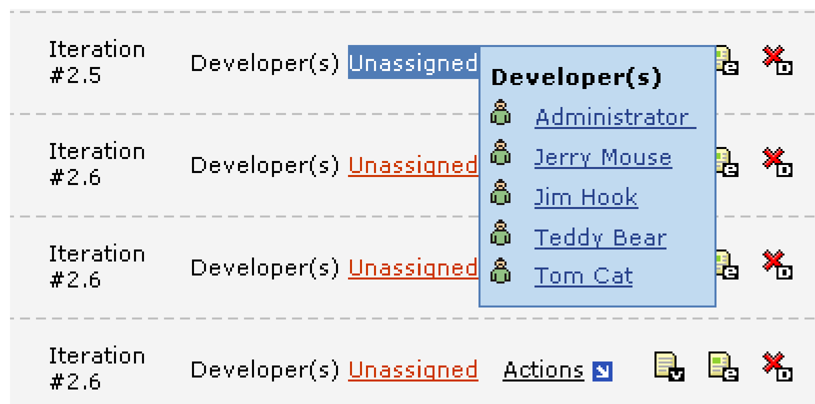


Ilustración 138. Asignación de usuarios.

SEGUIMIENTO DE BUGS INTEGRADO

No es necesaria ninguna aplicación externa para controlar los *bugs*. Dispone de todas las opciones de búsqueda, notificaciones y gráficos habituales.

Ilustración 139. Gestión de bugs integrada.

TEST CASES INTEGRADOS

Los casos de test junto con el seguimiento de *bugs* dan información completa de la calidad de cada versión. *Test Runner* proporcionando todos los enlaces necesarios desde una misma página y también crea gráficos.

TEST CASE #104. Planets Test: distribution

Steps	Success
1. Check planet with mass > standard mass	1. ~40%
2. Check planet with mass < standard mass	2. ~55%
3. Check planet with mass = standard mass	3. ~5%

Attachments

Passed Failed Skip Cancel

current test case

all test cases for this test plan run

TESTS RUN #127. All Test Cases Tests Run

Passed: 0 Failed: 0 Not Run: 14

Test Cases Grouped By User Stories

- STORY Planets Passed: 0 Failed: 0 Not Run: 1
  - TEST CASE Planets Test: distribution
- STORY Simplified 3D Demo Passed: 0 Failed: 0 Not Run: 3
  - TEST CASE Planets Test: mass
  - TEST CASE Planets Test: life probability
  - TEST CASE Planets Test: composition
- STORY Technical Specification Passed: 0 Failed: 0 Not Run: 4
  - TEST CASE Stars Test: spectrum
  - TEST CASE Stars Test: potential life-cycle
  - TEST CASE Stars Test: planet systems
  - TEST CASE Planets Test: quantity
- STORY Universe Design Specification Passed: 0 Failed: 0 Not Run: 6
  - TEST CASE Galaxies Smoke Test
  - TEST CASE Stars Smoke Test
  - TEST CASE Planets Smoke Test
  - TEST CASE Space Smoke Test
  - TEST CASE Stars Test: range
  - TEST CASE Stars Test: mass

Ilustración 140. Test Runner simplifica las pruebas manuales.

Add General Requirements Planning QA

Bugs Test Cases Test Plans Test Runs Reports

< Back | Main Dashboard > Projects > **Test Runs** Send your feedback!

Show Info

Test Runs [add] Filter Customize

Release	Iteration	State	Passed/Failed/NotRun		Assignments	
Demo	Iteration #1.1	Open Done	6 / 8 / 0	New Run Change Last Run	Executor Administrator	Actions
Demo	Iteration #1.2	Open Done	8 / 6 / 0	New Run Change Last Run	Executor Administrator	Actions
Demo	Iteration #1.3	Open Done	9 / 5 / 0	New Run Change Last Run	Executor Unassigned	Actions
Demo	Iteration #1.4	Open Done	8 / 6 / 0	New Run Change Last Run	Executor Unassigned	Actions
Beta version	Iteration #2.1	Open Done	9 / 5 / 0	New Run Change Last Run	Executor Unassigned	Actions
Beta version	Iteration #2.3	Open Done	10 / 4 / 0	New Run Change Last Run	Executor Unassigned	Actions
Beta version	Iteration #2.4	Open Done	10 / 4 / 0	New Run Change Last Run	Executor Unassigned	Actions
Beta version	Iteration #2.5	Open Done	12 / 2 / 0	New Run Change Last Run	Executor Unassigned	Actions
Beta version	Iteration #2.6	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Beta version	Iteration #2.7	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.1	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.2	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.3	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.4	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.5	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.6	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
Gold	Iteration #3.7	Open Done	0 / 0 / 14	Run	Executor Unassigned	Actions
		Open: 17				

Ilustración 141. TargetProcess da toda la información para asegurar la calidad.

Bug Submission Tool (Tp.Tray) es una herramienta para aumentar la productividad. Esta aplicación facilita enviar *bugs*: basta con capturar la pantalla y añadir un nuevo *bug* a TargetProcess.

Ilustración 142. La Bug Submission Tool permite adjuntar capturas de pantalla para describir bugs.

INTEGRACIÓN DE SUBVERSIONES

Mediante la *Subversion integration,* se puede comparar las diferencias entre versiones, automatizar operaciones repetitivas que actualizan varios campos (introduciendo por ejemplo #75 status: fixed time: 2 comm: había un problema en el constructor) y ligar requisitos/defectos al código.

Ilustración 143. Es posible añadir información muy rápido introduciendo los datos en modo texto.

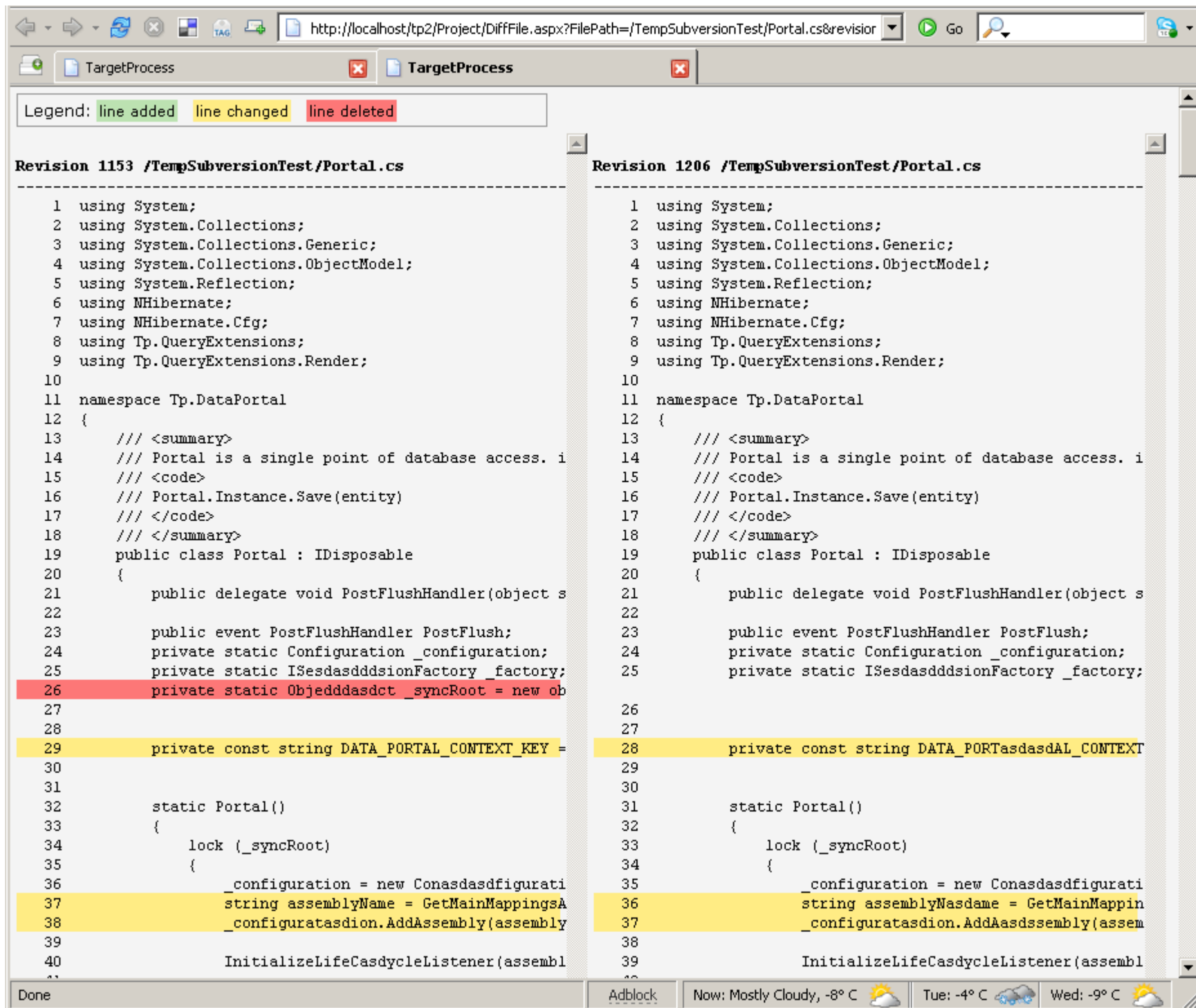


Ilustración 144. Se pueden ver las diferencias entre versiones.

ASIGNACIONES POR PERSONA

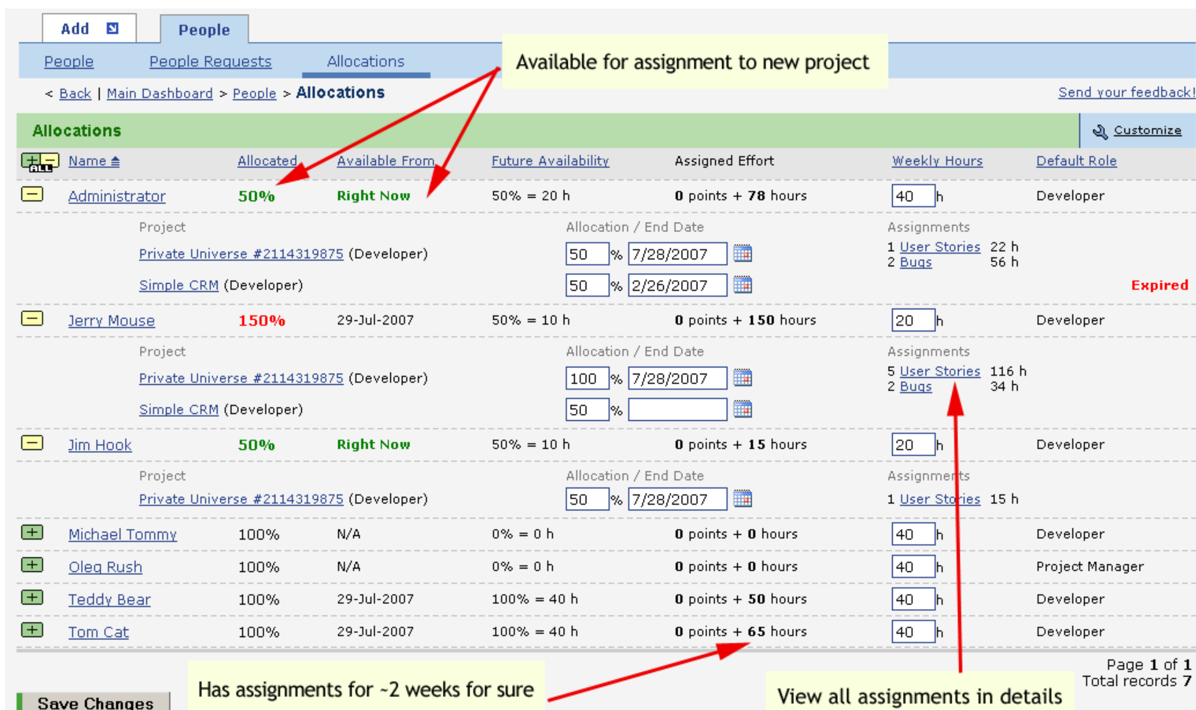


Ilustración 145. TargetProcess facilita información sobre todo el equipo y la carga que llevan.

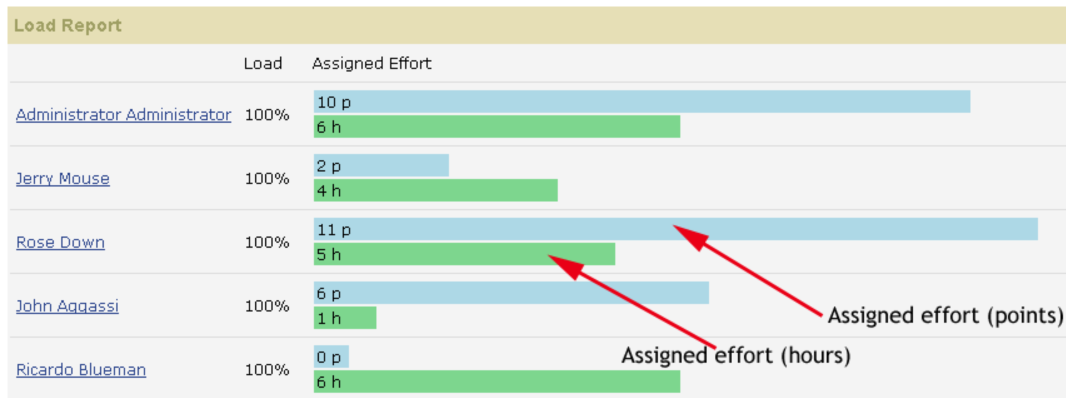


Ilustración 146. Carga de trabajo para cada desarrollador.

TargetProcess permite todas las opciones imaginables para personalizar el entorno con el logo de la empresa, personalizar los menús, enviar notificaciones por e-mail, etc.

3.7. ExtremePlanner

ExtremePlanner, www.extremeplanner.com, es otro software basado en web que permite las características principales para gestionar un proyecto *extremo*. Su coste es de 200 dólares por licencia.

PÁGINA PRINCIPAL

Desde la página de inicio podemos ver todas las versiones planeadas para el proyecto, iteraciones y qué está haciendo cada miembro del equipo. Para obtener más detalles, basta con hacer clic en historias, tareas, versión, persona...

Ilustración 147. Página de inicio de ExtremePlanner.

INFORMES Y NOTIFICACIONES

Resulta útil poder ver las notificaciones nuevas desde ayer o desde cualquier fecha. También permite enviar las notificaciones por e-mail.

Ilustración 148. Informe de actividades.

PLANNING DE ITERACIONES Y VERSIONES

Todo el funcionamiento está basado en "arrastrar y soltar". ExtremePlanner hace el seguimiento de esfuerzo total y capacidad (horas/persona) y avisa si se sobrepasan los límites.

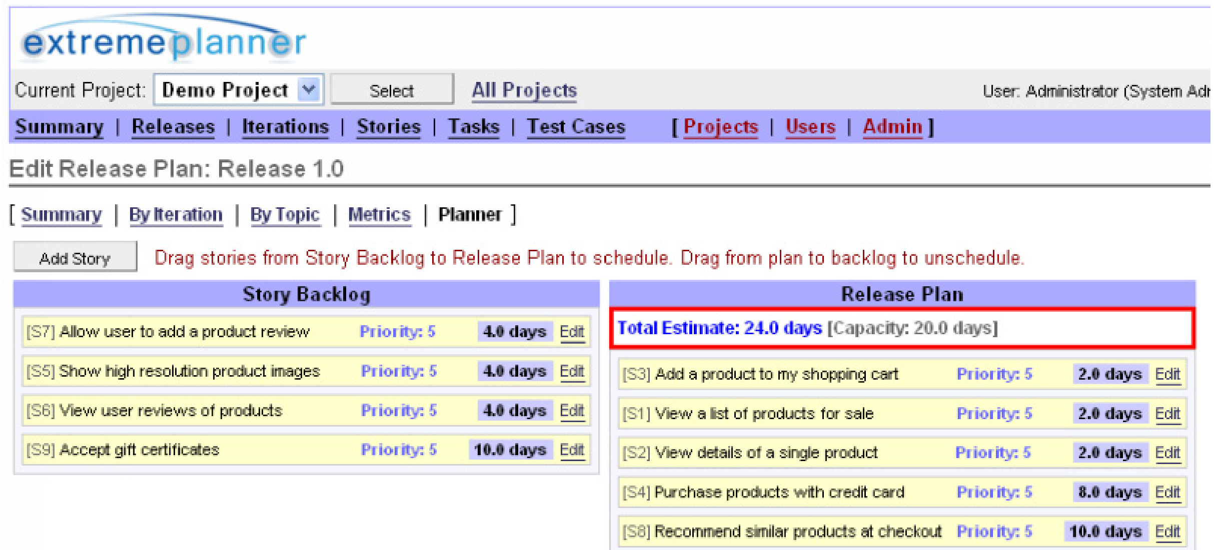


Ilustración 149. Gestión de versión, historias y tareas.

MANEJAR HISTORIAS, TAREAS, BUGS, TESTS...

ExtremePlanner controla todo lo necesario para gestionar un proyecto de Extreme Programming, permitiendo incluso adjuntar archivos y enlaces a recursos internos o externos.

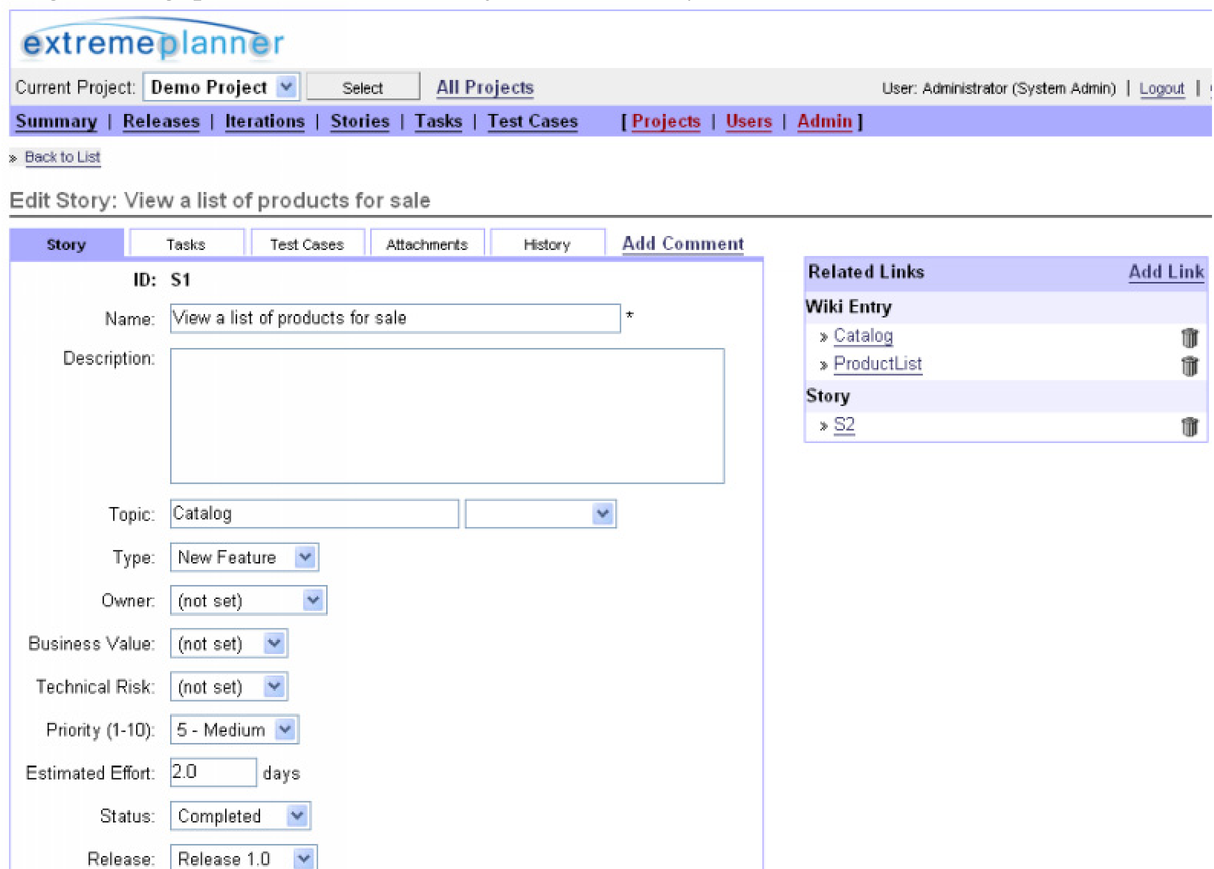


Ilustración 150. Datos para especificar una historia.

SEGUIMIENTO DEL AVANCE

Seguimiento de tareas activas, sin empezar o completadas y gráficos de quemado.

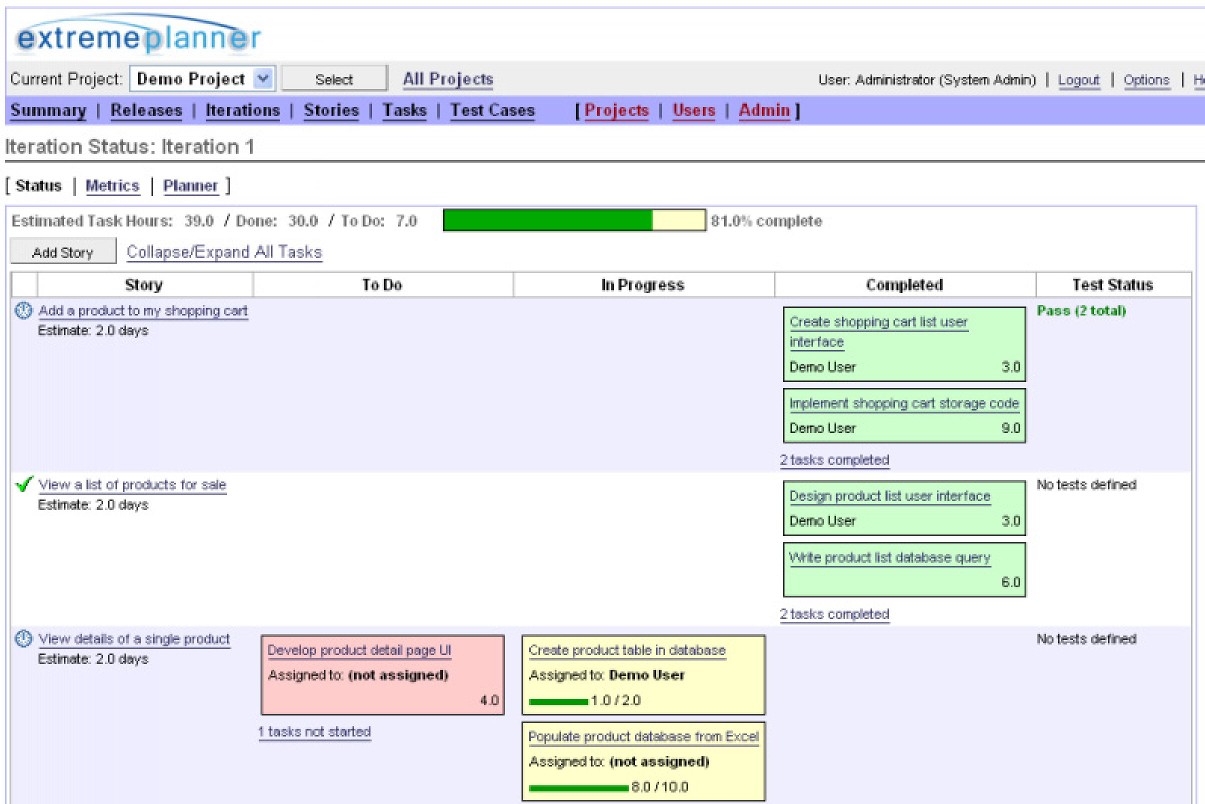


Ilustración 151. Seguimiento de las tareas de cada historia para formar una versión.

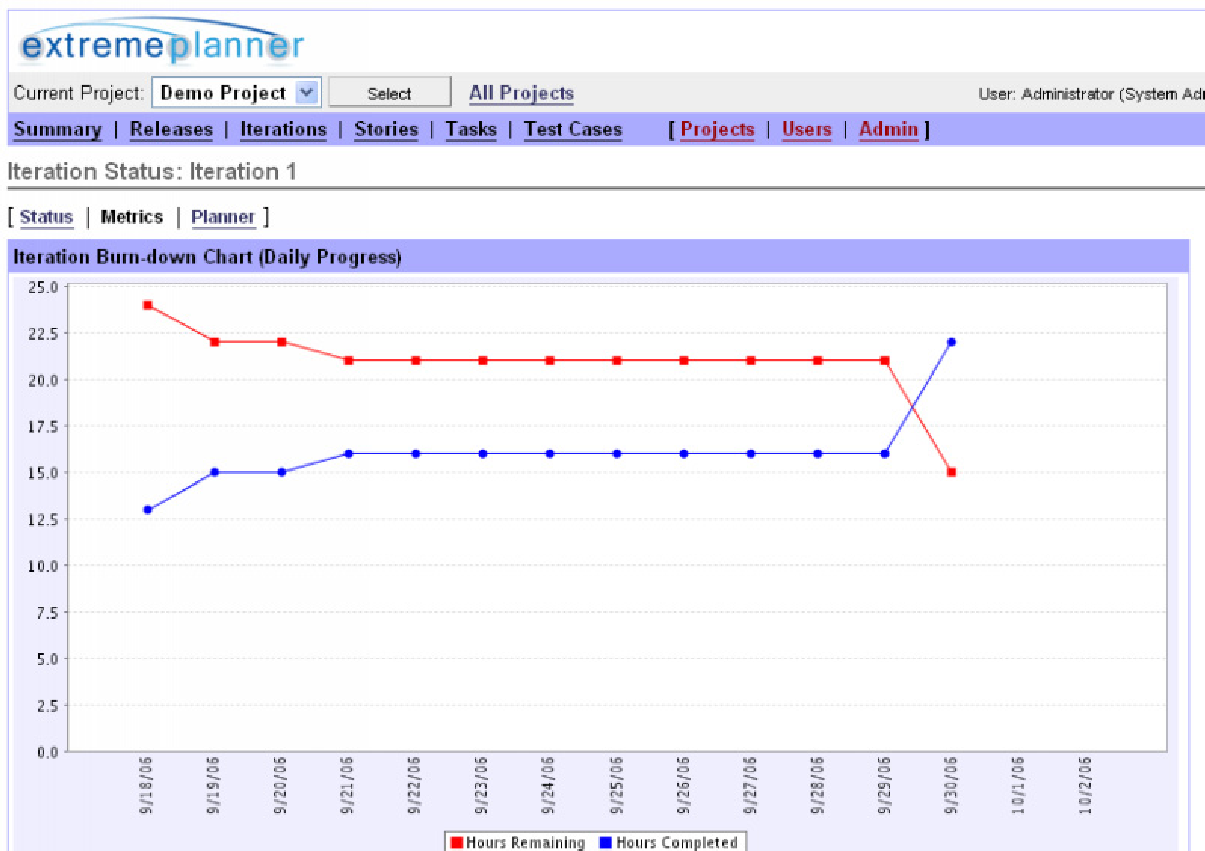


Ilustración 152. Gráfico de quemado.

GENERACIÓN Y EXPORTACIÓN DE INFORMES

Es posible clasificar, buscar, filtrar historias, tareas, test cases, importar listas de características o tareas desde MS Excel y exportar informes a MS Excel, Word, o formato XML.

extremeplanner

Current Project: Demo Project | Select | All Projects — User: Administrator (System Admin) | Logout | Options | Help

Summary | Releases | Iterations | Stories | Tasks | Test Cases [Projects | Users | Admin]

Stories

Iteration: (any) Release: (any) Type: (any) Status: (any) Search: Go Save settings

Estimated Effort: 46.0 days / Completed: 6.0 days / Remaining: 40.0 days 13.0% complete

Add Story | Delete | Schedule... | Import Stories from Excel

ID	Story Name	Topic	Type	Pri.	Estimate	Task Est	Release	Iteration	Tasks	Last Updated	Status
S3	Add a product to my shopping cart	Shopping Cart	New Feature	5	2	14	Release 1.0	Iteration 1	2 of 2	2/8/07 11:52 AM	Active
S1	View a list of products for sale	Catalog	New Feature	5	2	9	Release 1.0	Iteration 1	2 of 2	2/7/07 10:14 AM	Completed
S2	View details of a single product	Catalog	New Feature	5	2	16	Release 1.0	Iteration 1	0 of 3	2/7/07 10:14 AM	Proposed
S7	Allow user to add a product review	Reviews	New Feature	5	4	0		Iteration 1	0 of 0	2/7/07 10:12 AM	Proposed
S5	Show high resolution product images	Catalog	New Feature	5	4	20		Iteration 2	1 of 1	2/12/07 12:37 PM	Completed
S6	View user reviews of products	Reviews	New Feature	5	4	0		Iteration 2	0 of 0	2/7/07 9:19 AM	Proposed
S4	Purchase products with credit card	Shopping Cart	New Feature	5	8	0	Release 1.0	Iteration 1	0 of 0	2/7/07 10:12 AM	Proposed
S9	Accept gift certificates		New Feature	5	10	14			0 of 2	2/12/07 12:30 PM	Proposed
S8	Recommend similar products at checkout	Shopping Cart	New Feature	5	10	0	Release 1.0	Iteration 2	0 of 0	2/7/07 10:09 AM	Proposed

9 items found, displaying all items.

Export options: Excel RTF

Customize View

Ilustración 153. Informe de las historias con sus respectivas tareas, avances, estados, autores...

extremeplanner

Current Project: Demo Project | Select | All Projects — User: Administrator

Summary | Releases | Iterations | Stories | Tasks | Test Cases [Projects | Users | Admin]

Release Plan: Release 1.0

[Summary | By Iteration | By Topic | Metrics | Planner]

Iteration	Story	ID	Estimated days	Done
Iteration 1	Add a product to my shopping cart	S3	2	
	View a list of products for sale	S1	2	✓
	View details of a single product	S2	2	
	Purchase products with credit card	S4	8	
Iteration 1 Total:			**14**	
Iteration 2	Recommend similar products at checkout	S8	10	
Iteration 2 Total:			**10**	
Total Estimated Effort:			**24**	

WARNING: Estimated effort is greater than the defined release capacity of 20.0.

Export options: Excel RTF

Ilustración 154. Plan de una versión y posibilidad de exportar informes a Excel, Word y XML.

CONTROL DE ACCESO Y AUDITORÍA

Ilustración 155. Un control de acceso permite seguir un proyecto y mantener partes reservadas.

INTEGRACIÓN IDE

Los desarrolladores pueden ver y actualizar información de las tareas directamente desde sus IDE (*Integrated Development Environment*) como Eclipse o Microsoft Visual Studio.

Ilustración 156. Integración para actualizar información de tareas desde Eclipse o MS Visual Studio.

3.8. Atlassian

Atlassian, www.atlassian.com, propone 7 módulos de software utilizados por más de 7.000 usuarios en 87 países que abarcan todo lo necesario para gestionar proyectos de software. Los precios de las versiones de JIRA van desde los 1.200 a los 4.800 dólares. Veamos una breve descripción de su software:

JIRA es una aplicación para el seguimiento de *bugs*, *issues*, características, tareas, versiones, y la gestión de proyectos.

CONFLUENCE es una *wiki* de empresa que facilita colaborar y compartir conocimiento.

BAMBOO es un servidor continuo para la integración. Bamboo aporta medidas en tiempo real, proporciona *feedback* mediante notificaciones o e-mail, métricas, patrones... y se integra con otras herramientas de desarrollo.

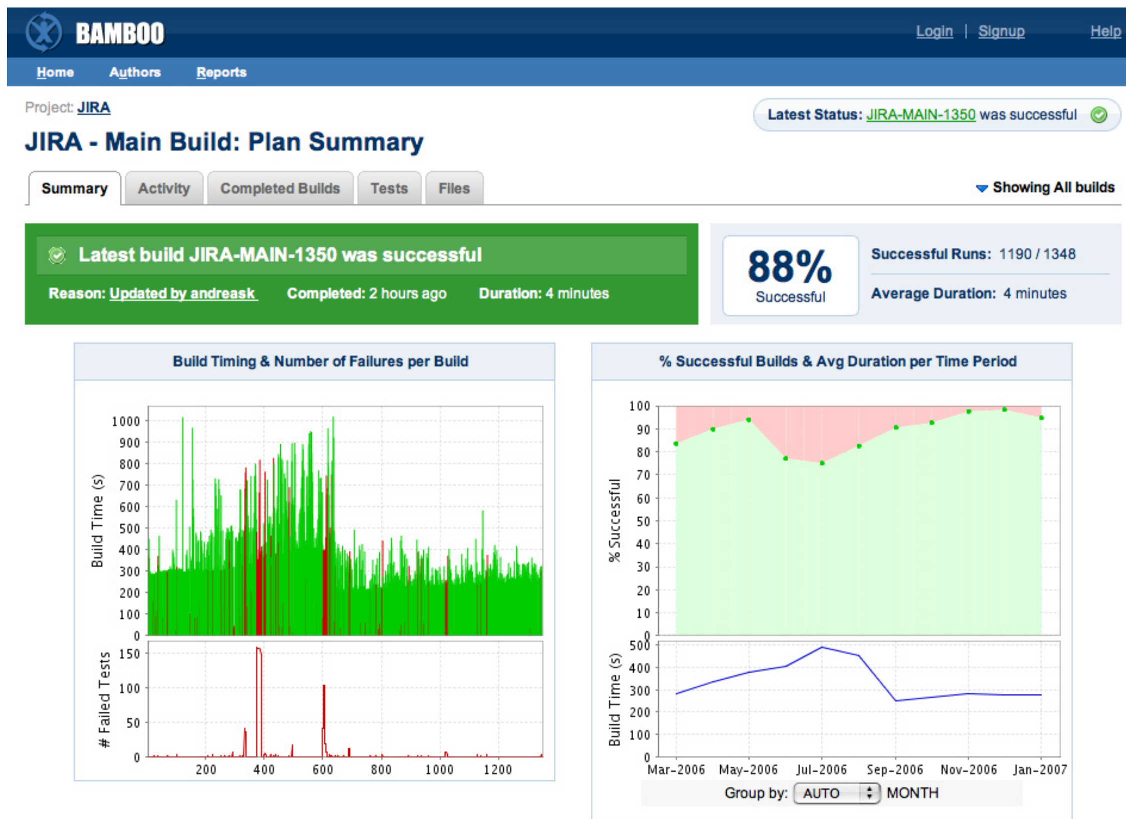


Ilustración 157.Bamboo facilita la integración continua.

CLOVER es un entorno para garantizar la calidad de los tests. Mientras que los tests unitarios miran la calidad del código, Clover mira la calidad del test unitario. Permite encontrar partes de código no testeadas, generar informes, encontrar "código muerto" (código que ya no se usa en una aplicación y puede eliminarse de forma segura), integrarse con otras plataformas de test (JUnit, TestNG, JTiger), *plug-ins* (para Eclipse e IntelliJ IDEA), etc.

CROWD es una herramienta basada en web que simplifica la gestión centralizada de identidades (pertenencia a grupos) y el aprovisionamiento de aplicaciones (relaciones entre la

intranet, CRM, la web oficial pública, el supervisor de *bugs* etc. con el personal (*staff*, base de datos LDAP), clientes, *partners*, etc.)

CRUCIBLE facilita revisar los cambios al código, añadir comentarios y guardar resultados.

Ilustración 158. Crucible para la revisión de código.

FISHEYE es un almacén o repositorio con información útil difícil de extraer, comprender o mantener actualizada. Facilita trabajar con código que cambia constantemente, supervisar cualquier elemento, buscar archivos, compartir mediante enlaces (directorios, diferencias, revisiones, líneas de código fuente, anotaciones) y analizar resultados.

Ilustración 159. FishEye o repositorio de Atlassian.

3.9. xProcess

La empresa Ivis Technologies, www.ivis.com propone un completo programa con todas las características deseables para gestionar completamente un proyecto. El precio de xProcess Enterprise es de 55 dólares por año y usuario.

Ilustración 160. Vista del entorno que ve un participante con sus tareas asignadas, calendarios, etc.

Acceso vía web de los participantes

- Asignar todos los participantes en el plan.
- Los participantes gestionan sus calendarios.
- Utilizar artefactos gestionados con el plan.

Definición de proceso

- Especificar patrones para tareas y documentos.
- Usar la experiencia del propio proyecto para mejorar.
- Usar procesos para crear planes de proyecto.
- Modelar visualmente restricciones y volumen de trabajo (*workflow*).

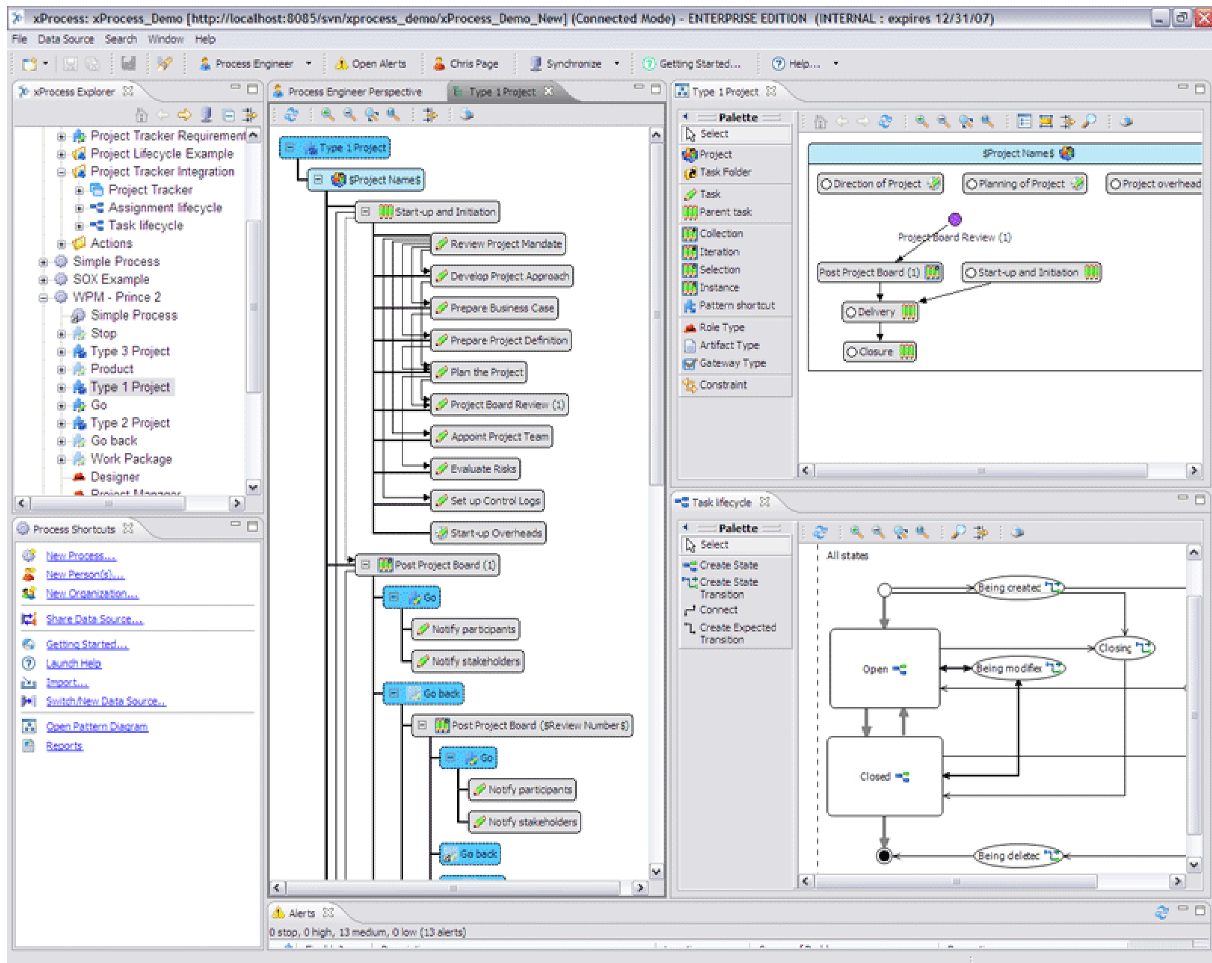


Ilustración 161. Diagrama de proceso.

Gestión de proyecto

- Reflejar los cambios de prioridades.
- Ver el impacto de cambiar los recursos.
- Hacer todos los aspectos del plan accesibles.
- Crear tareas basadas en el proceso.

Gestión de recursos

- Examinar todos los recursos cuando cambian.
- Previsión de tareas basadas en la disponibilidad.
- Optimizar calendarios para cada papel y tiempo.
- Gestionar la utilización de recursos de empresa.

Gestión de artefactos

- Incorporar los recursos (*assets*) dentro de sus proyectos.
- Gestionar archivos internos o externos.
- Integrar los recursos con el proceso.
- Control de versiones constante.

Entorno ejecutivo e informes

- Uso de recursos en tiempo real.
- Valor y eficiencia ganada por el proyecto.
- Estado del proyecto y gráficos de quemado.
- Entorno configurable y gráficos.

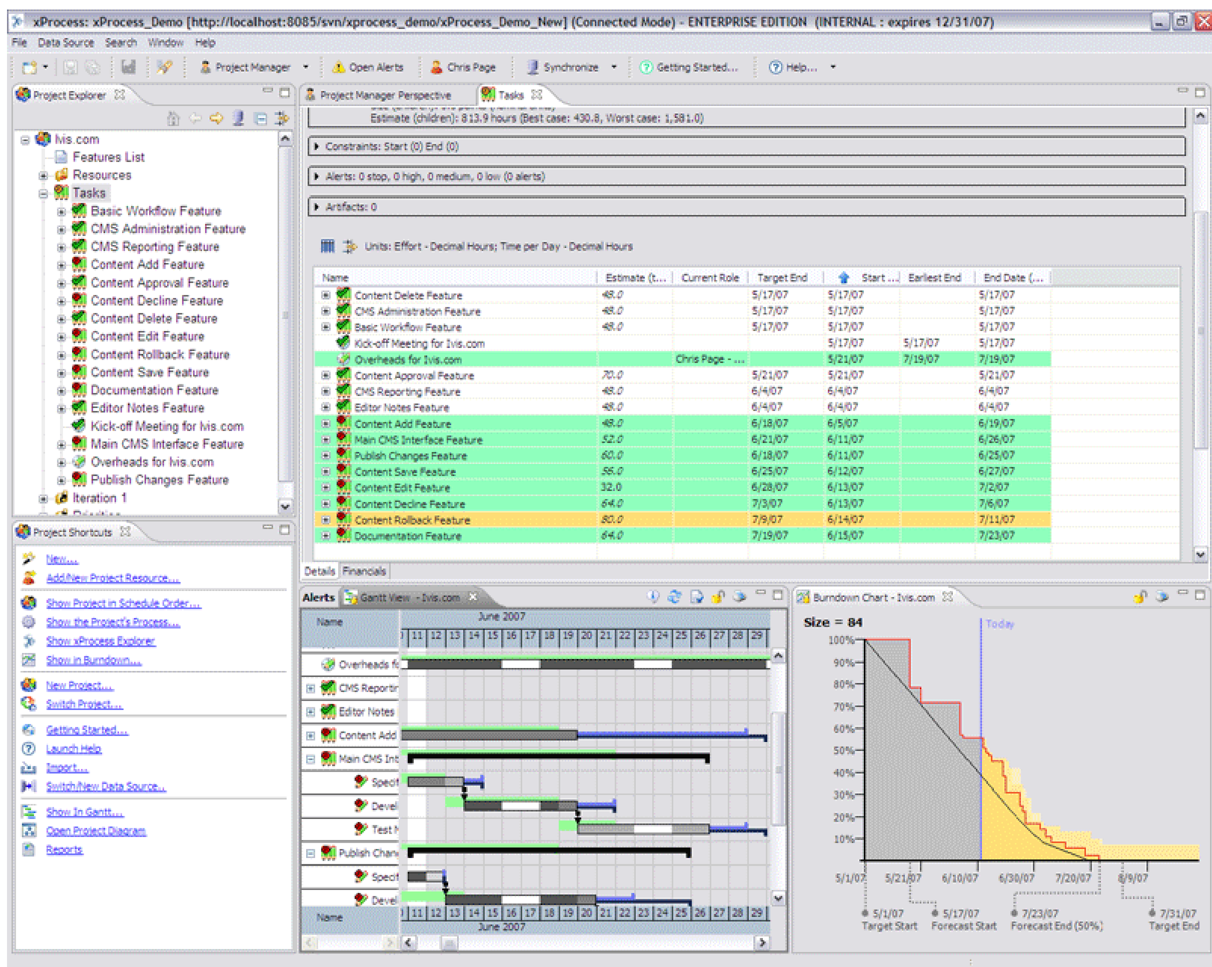


Ilustración 162. Vista del calendario, gráfico de quemado, tareas... de un proyecto en xProcess.

3.10. Microsoft Visual Studio

Visual Studio 2005 es una herramienta de desarrollo, creada a partir de sus predecesores: Visual Basic, Visual C++ 1.0 (1993), MS Visual Studio .NET 2002 (englobó a los dos anteriores) y Visual Studio 2003 que fue utilizado para crear aplicaciones .NET Framework 1.1. Actualmente se dispone de la versión 2008 de Visual Studio.

Visual Studio 2005 utiliza el framework .NET 2.0 como biblioteca de clases y rutinas de los lenguajes compatibles con este IDE (Integrated Development Environment): MS Visual Basic 5, MS Visual C# 2005, MS Visual C++ 2005 y MS Visual J# 2005.

Uno de los objetivos del framework .NET es simplificar el desarrollo y despliegue de aplicaciones. Esto afecta a aplicaciones que se ejecutan localmente o aplicaciones remotas que están distribuidas a través de Internet. La simplificación se logra con un CLR (*Common Language Runtime*) que proporciona un entorno de ejecución gestionado que permite cualquier lenguaje de programación que permitan las rutinas. Los lenguajes que utilizan CLR se compilan al *Microsoft Intermediate Languaje* (MSIL) que después se traduce a código nativo.

Existen dos tipos principales de ventanas en Visual Studio: las "*Document windows*" que contienen editores, páginas web o ayuda, y las ventanas de herramientas que incluyen Solution Explorer, Class View, las "*Properties windows*".

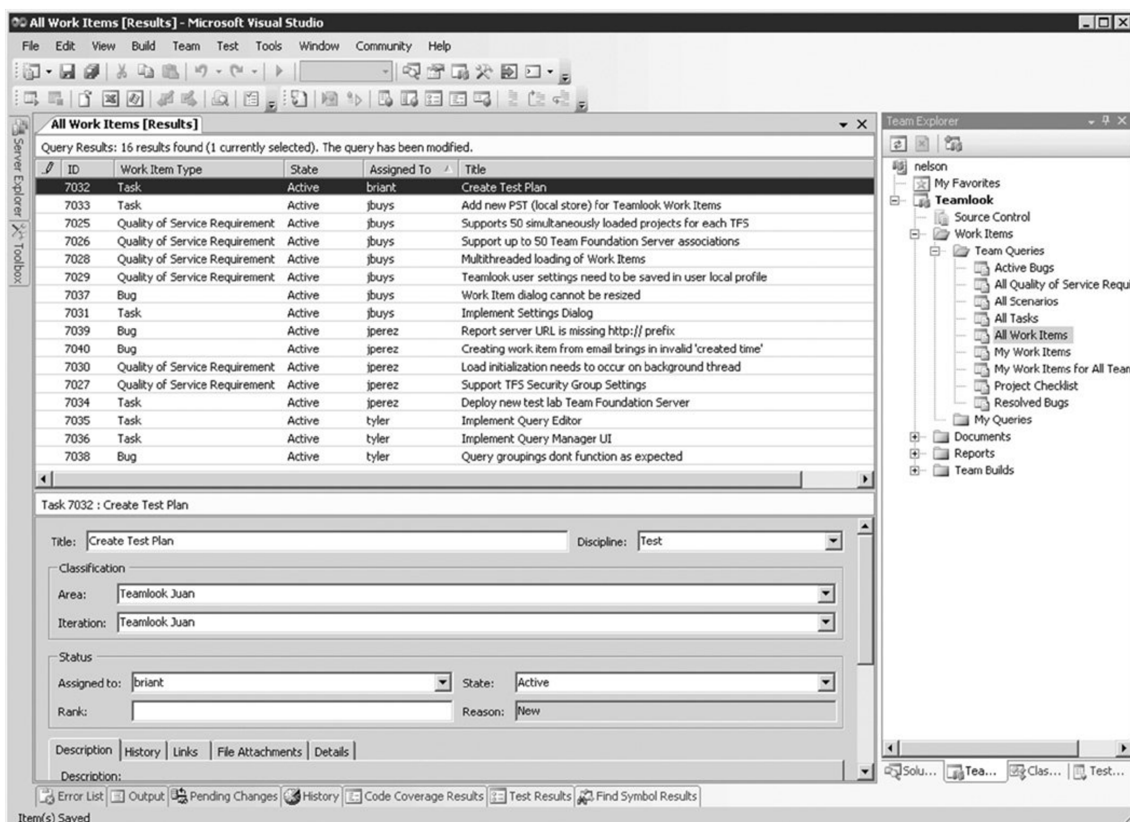


Ilustración 163. En el Team Explorer de Visual Studio Team System, podemos ver en una misma ventana los requisitos, las tareas y los bugs.

También permite ayudarse de MS Excel para agregar datos, ya que todo se guarda en la misma base de datos. Si fuera necesario, es posible incluso exportar a MS Project y configurar

Outlook como cliente del servidor de la base de datos para recibir avisos, correos, etc. Para una visión más profunda, *Working with Microsoft Visual Studio 2005* de Microsoft Press, por C. Skibo, M. Young y B. Johnson, explora a fondo todas las posibilidades de Visual Studio, plantillas, macros, *add-ins*, etc.

Función	Descripción	Valor
Analizador estático de código	Examinar el código fuente para detectar malas prácticas de programación y riesgos comunes de seguridad.	Evita tener que introducir costosas vulnerabilidades de seguridad en las aplicaciones; mejore la legibilidad y el mantenimiento del código.
Analizador dinámico de código	Diseñado específicamente para detectar y ayudar a depurar daños de memoria y vulnerabilidades críticas de seguridad en tiempo de ejecución, incluyendo, entre otros, problemas con heaps, identificadores y bloqueos.	Encuentra rápidamente y de forma más eficaz errores en tiempo de ejecución difíciles de detectar.
Perfilador de código, Code Profiler	Analizar las características de rendimiento y el espacio de memoria de su aplicación en un formato organizado y fácil de entender.	Optimiza el rendimiento de la aplicación y reduzca el uso de memoria.
Tests de unidad integrados	Visual Studio Team System convierte la comprobación en una parte importantísima de la experiencia de Visual Studio. En ningún lado esto es más evidente como en el soporte a pruebas unitarias, creado justo en el IDE.	La facilidad de realizar pruebas unitarias de aplicaciones aumenta la probabilidad de que dichas pruebas unitarias se utilicen, mejorando así la solidez de la aplicación.
Cobertura de código, Code Coverage	Mostrar visualmente qué rutas de código han sido ejecutadas.	Aumenta la efectividad de las pruebas unitarias y la solidez de la aplicación incrementando la cobertura de código.
Políticas de Check-in	Respetar las políticas de revisión de Team Foundation Server, garantizando que todas ellas se cumplan antes de que el código se compruebe en el árbol de fuentes.	Ayuda a reducir el trabajo de reelaboración proporcionando comprobaciones en una fase temprana del proceso de desarrollo.
Entorno de desarrollo multiuso	Visual Studio Team Edition para desarrolladores se basa en Visual Studio Professional Edition e incluye, entre otros, soporte para todos los lenguajes .NET, ClickOnce, desarrollo de 64 bits y móvil, y depuración entre equipos.	Utiliza una única herramienta potente para todas sus necesidades de desarrollo empresariales.

Tabla 23. Funciones de las diferentes partes de Visual Studio.

Tarea	Ahorro	Comentarios
Generación automatizada	1.7%	Se reduce el tiempo del proceso de generación. Los responsables de desarrollo dedican menos tiempo al proceso de generación.
Control del código fuente	14.5%	Administrar el sistema de control del código fuente requiere menos tiempo. Las funciones avanzadas y un sistema con mejor rendimiento aumentan la productividad.
Resolución de defectos	11.6%	El seguimiento de elementos de trabajo integrados, el seguimiento de defectos y el control del código fuente, así como la posibilidad de realizar consultas personalizadas, hacen que la resolución de problemas sea más rápida y sencilla, lo que supone un ahorro de tiempo considerable para el desarrollador durante la fase de prueba.

Reducción del trabajo repetido	6.7%	Mejor gestión del control del código fuente y de la generación, así como una resolución de defectos más rápida, mejoran la calidad y reducen la repetición del trabajo, lo que supone un ahorro de tiempo considerable para el responsable de desarrollo durante la fase de revisión y examen.

Tabla 24. Visual Team System: ahorros en la fase de desarrollo.

Tarea	Ahorro	Comentarios
Pruebas de regresión	25%	Se reduce el tiempo del proceso de generación Revisar el conjunto de pruebas de regresión requiere menos tiempo. Analizar y optimizar los escenarios de prueba necesarios para futuros conjuntos de regresión requiere menos tiempo.
Ausencia de transferencia de documentos	6%	No tener que transferir los documentos de Microsoft Word a la herramienta de seguimiento de defectos supone un ahorro de tiempo.
Revisar defectos reactivados	6%	Ya no es necesario realizar una consulta SQL manual durante el proceso de revisión de defectos reactivados.
Producir informes estadísticos	25%	Crear los informes estadísticos sobre las pruebas requiere mucho menos tiempo.

Tabla 25. Visual Team System: ahorros en la fase de prueba.

Existen varias versiones de Visual Studio 2005, especializadas en diferentes aspectos: Team Foundation Server, Team suite, Team Edition para arquitectos, para desarrolladores, para profesionales de comprobación, para profesionales de bases de datos, y Team Test Load.

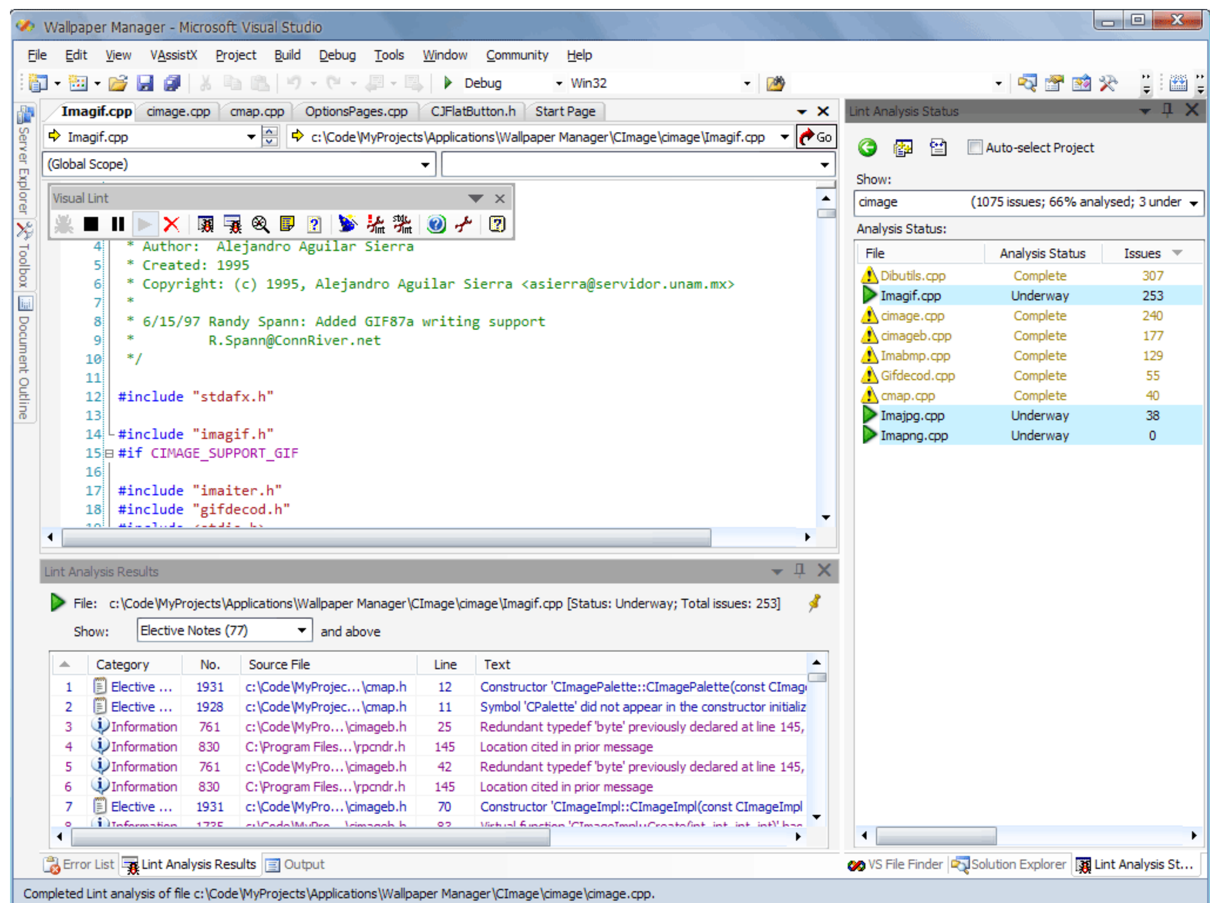


Ilustración 164. Entorno de Microsoft Visual Studio.

CAPÍTULO

4

CONTENIDO

4.1. Resumen

4.2. Crítica a las prácticas ágiles

4.3. Métodos ágiles y desarrollo de software libre

4.4 Otros factores influyentes

4.5. Complementariedad y similitudes

4.6. Estadísticas

4.7. Problemas comunes a los métodos ágiles

4.8. Limitaciones de los métodos ágiles

CONCLUSIONES Y LÍNEAS DE FUTURO

Los métodos que hemos examinado no son fáciles de comparar entre sí conforme a un pequeño conjunto de criterios. Algunos, como XP, han definido claramente sus procesos y prácticas, mientras que otros, como Scrum, son bastante más difusos en ese sentido, limitándose a un conjunto de principios y valores. Lean Development también presenta más principios que prácticas. Algunos, como FDD, no cubren todos los pasos del ciclo de vida, sino unos pocos de ellos. Varios métodos como ASD dejan toda la práctica concreta a algún otro método.

Existen multitud de métodos ágiles híbridos como se han citado en el texto. También se podrían mencionar Grizzly de Ron Crocker (2004) o Dispersed eXtreme Programming (DXP), creado por Michael Kircher en Siemens.

A continuación, haremos un resumen comparativo, veremos diferentes críticas a los métodos y prácticas ágiles, contrastaremos estas críticas con estadísticas reales y por último citaremos limitaciones y problemas comunes de los métodos ágiles.

Por último comentaremos posibles líneas de futuro como continuación de este proyecto.

"Plan the work, and work the plan"

Tom DeMarco

4. Conclusiones y líneas de futuro

Ilustración 165. Ojalá pudiera parar y pensar un momento.

"Aquí está el oso Edward,
bajando las escaleras,
detrás de Christopher Robin.

¡Bump! ¡Bump! ¡Bump!
golpeándose la nuca.

Para él, es la única manera de
bajarlas, pero está seguro que
debe haber otra forma mejor,
ojalá pudiera parar de golpearse
un momento y pensar en ello."

Winnie - the Pooh
Alan Alexander Milne, 1926

4.1. Resumen

La ilustración refleja el cambio de concepción. XP no sólo es iterativo sino que hace todas las fases simultáneamente (las filas se convierten en columnas).

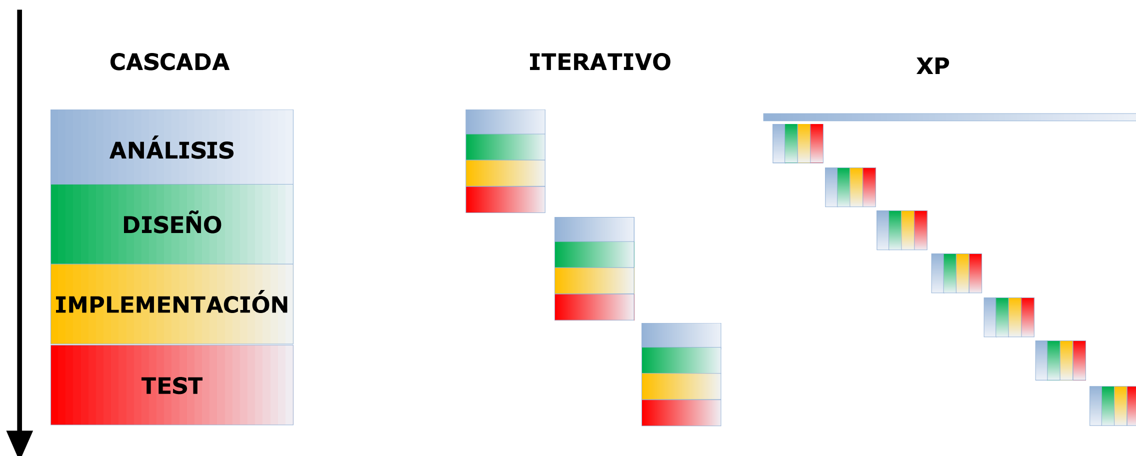


Ilustración 166. Cambio en la concepción: Cascada-Iterativo-XP.

Prácticamente todos los métodos ágiles adoptan buenas prácticas como:

- la comunicación entre programadores, y entre clientes y desarrolladores,
- el hincapié en la refactorización,
- el hincapié en las pruebas que debe pasar el software, y
- la importancia que dan a mantener estándares de escritura del código.

Y todos los métodos ágiles enfatizan los siguientes aspectos:

- entregar algo útil,
- favorecer la colaboración y la confianza en la gente,
- promover la excelencia técnica,
- hacer la cosa más sencilla posible, y
- adaptarse continuamente.

Estas ideas no son nada nuevas; algunas son evidentes incluso para los no programadores. Son reglas que derivan del mero sentido común.

En la siguiente ilustración se aprecia la historia de los métodos ágiles, con fechas y los vínculos entre unos y otros métodos.

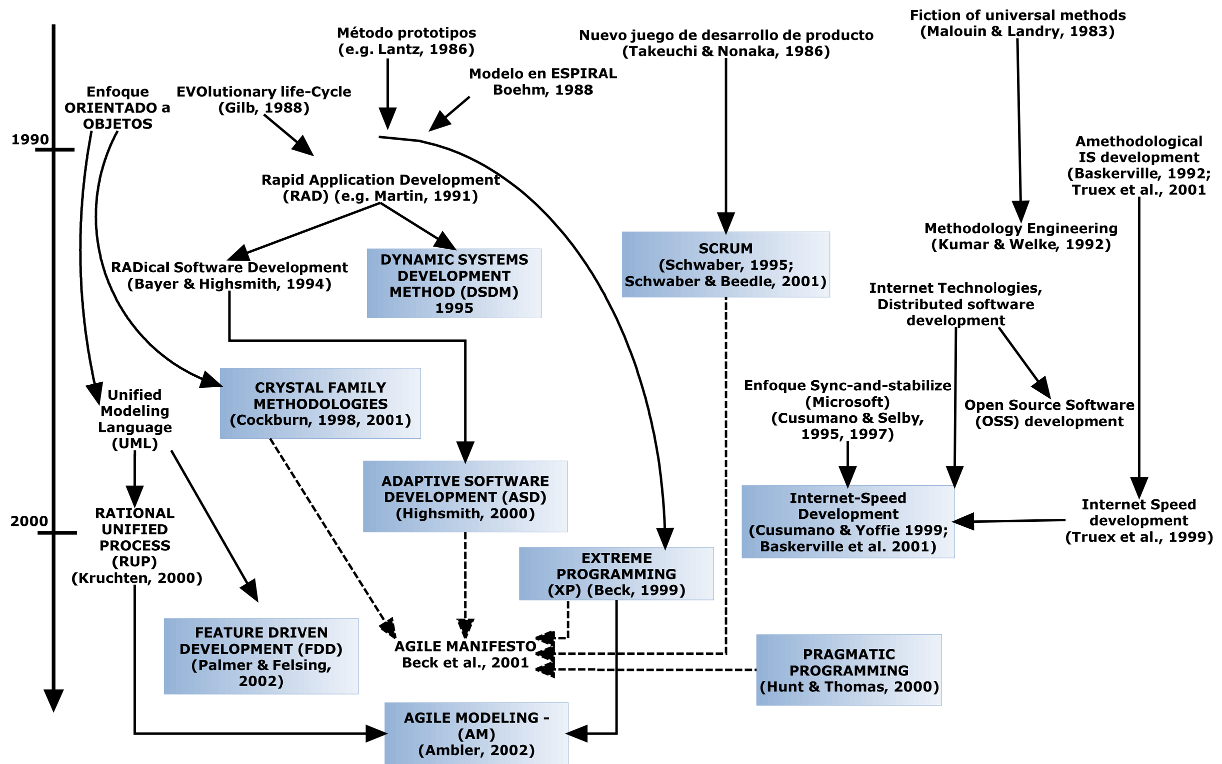


Ilustración 167. Árbol genealógico de los métodos ágiles. Extraído del artículo New Directions on Agile Methods: A Comparative Analysis (Abrahamsson et al.).

Veamos las diferencias generales entre el enfoque metodológico, lo que demuestra la práctica, y el enfoque ágil:

	Enfoque metodológico	**Desarrollo de SW en la práctica**	**Visión de los métodos ágiles**
Actividades	Tareas independientes	Proyectos personales interrelacionados	Proyectos personales interrelacionados
	Duración predecible	Finalización impredecible	Finalización de la siguiente release predecible
	Repetible	Dependiente del contexto	A menudo dependiente del contexto
Ejecución del proceso	Fiable	Dependiente de las condiciones del entorno	Ejecución ligada a pequeñas versiones, por tanto, fiable
	Interacciones especificables	Inherentemente interactivo	Iteraciones especificadas fomentando interactividad
	Tareas en secuencia	Muchas tareas están entrelazadas	Las tareas muchas veces no se mandan; dependen del contexto

Esfuerzo de los desarrolladores	Dedicados a proyectos de software	Comunes a todas las actividades (proyecto, no proyecto, personal, rutinas...)	Los desarrolladores estiman el esfuerzo requerido
	No diferenciados	Específico a individuos	Específico a individuos
	Totalmente disponibles	Totalmente utilizados	Totalmente utilizados
Control de trabajo	Regularidad	Oportunismo, improvisación e interrupción	Sólo existe control mutuo acordado respecto al trabajo de otros (algunos MA definen controles: Crystal o DSDM)
	Milestones, planning, gestión de control	Preferencia individual y negociación mutua	Preferencia individual y negociación mutua

Tabla 26. La teoría del enfoque metodológico, la práctica, y el enfoque ágil.

Existen multitud de frases que resumen los problemas y las filosofías de los métodos (en los anexos se encuentran los epigramas de la programación según Alan J. Perlis):

- *Una solución al 80% hoy, en vez de una al 100% mañana.* Bob Charette (LD)
- *El cliente recibe lo que necesita hoy, no lo que necesitaba ayer.* Darrell Norton (LSD)
- *Build the right product before you build it right*". DSDM Consortium
- *Si funciona bien, arréglelo de todos modos*. Refactoring, XP
- *También yo creo en el modelado; sólo que lo llamo por su nombre propio, 'mentir', y trato de convertirlo en arte.* Kent Beck
- *Los XPertos no hacen diagramas*. Martin Fowler
- *La diferencia entre un atracador de bancos y un teórico del método CMM es que con un atracador se puede negociar.* Ken Orr, Cutter Consortium
- *Haga un plan para tirarlo, lo hará de todos modos.* Fred Brooks
- *¿Cómo es que un proyecto puede retrasarse un año? Un día cada vez.* Fred Brooks
- *Plan the work, and work the plan.* Tom DeMarco
- *En teoría, no hay diferencia entre teoría y práctica. En la práctica, la hay.* John McMillan
- *La documentación es como un seguro: satisface porque casi nadie que lo contrata depende de sus beneficios.* Alan Perlis
- *Uno podría entender la cultura OSS como un tipo de fusión de colectivismo e individualismo: dar a la comunidad es lo que hace un héroe al individuo en los ojos de los otros.*

A pesar de que todos comparten los principios ágiles, existen diferencias entre ellos, por ejemplo:

- XP defiende la propiedad colectiva de código, mientras que en FDD la propiedad es individual.
- XP ataca "*lo más difícil primero*" mientras que Crystal "*lo más fácil primero, lo más difícil segundo*" para no desanimar.
- XP, Scrum o AM están más pensados para pequeños equipos (10-20), mientras que OSS, ASD o DSDM pueden con equipos de más de 100 desarrolladores.

	Características	*Papeles*	*Fases*
Adaptive Software Development *ASD*	(Highsmith 2000) Cultura adaptativa, colaboración, desarrollo iterativo basado en componentes; *mission-driven*. Son más conceptos y cultura que pautas para la práctica.	El patrocinador ejecutivo tiene la responsabilidad global. Utiliza sesiones JAD (ayudante para planear y moderar, secretario para tomar actas, el jefe de proyecto, representantes del cliente, y representantes de los programadores	Especular-Colaborar-Aprender. Inicio, plan de ciclo adaptativo, ingeniería concurrente componentes (iterativa), revisión de calidad (estas 3 forman el bucle de aprendizaje), control de calidad final y entrega.
Agile Modelling *AM*	(Ambler, 2002) Se basa en 13 principios. Utiliza 13 + 5 prácticas. Aplica los principios ágiles al modelado: cultura ágil, comunicación, simplicidad. Los principios ágiles también aplican al modelado. Es una buena filosofía, pero depende de otros métodos para ponerse en práctica.	Al requerir de otros métodos, utiliza los papeles de esos métodos.	Modelado inicial de requisitos, Modelado inicial de arquitectura, model storming y desarrollo test-driven (estas dos iterativas). Revisión opcional en todas las iteraciones.
Agile Unified Process *AUP*	(Ambler, 2005). Simplificación ágil de RUP; iterativo, proporciona versiones incrementales. Disciplinas: Modelo, Implementación, Test, Despliegue, Gestión de configuración, Gestión de proyecto y Entorno. Lista todos los deliverables.	Administrador base datos ágil, Modelador ágil, Manager de configuración, Desplegador, Desarrollador, Ingeniero de proceso, Manager de proyecto, Revisor, Participante (stakeholder), Escritor técnico, Manager de tests, Tester, Especialista de herramientas.	Concepción, Elaboración, Construcción y Transición. 4 milestones antes de pasar de pase: Objetivos del ciclo de vida, Arquitectura del ciclo de vida, Capacidad de operación inicial y Versión del producto.
CRYSTAL	(Cockburn, 2002) Familia de métodos, cada uno con requisitos principios subyacentes. Técnicas, papeles, herramientas varían. Selección el método a partir de la criticidad y su tamaño. Basado en 7 valores. Sólo existen 2 de los métodos propuestos. Sugiere principios diferentes según tamaño y criticidad.	Patrocinador, usuario experto, diseñador principal, diseñador-programador, experto comercial, coordinador, tester, escritor técnico.	Se definen prácticas, herramientas a utilizar y workproducts necesarios, pero no fases. Énfasis en el modelo de ciclos anidados.
Dynamic Systems Development Method *DSDM*	(DSDM, 1994) Aplicación de controles a RAD, uso de cajas de tiempo, equipos autorizados, el DSDM Consortium lo mantiene y mejora. Primer método ágil para desarrollo de SW. Uso de prototipos. Creado por 16 expertos en RAD. Sólo miembros del DSDM Consortium tienen acceso a la documentación.	Patrocinador, visionario, usuario embajador, usuario asesor, jefe de proyecto, coordinador técnico, líder de equipo, desarrolladores, tester, escriba.	Estudio de viabilidad, estudio comercial, iteración modelo funcional, iteración diseñar y construir e implementación (iterativa).
Enterprise Unified Process *EUP*	(Ambler, 2004) Refleja todo el ciclo de vida del software. Extensión de RUP/AUP.	Usa los papeles de RUP/AUP ya que suele adoptarse primero RUP o AUP y luego incorporar las extensiones de EUP.	Añade Producción y Retiro a las de RUP (Concepción, elaboración, construcción, transición). Añade disciplinas: Soporte y Operaciones, y 7 disciplinas de empresa.

	Características	*Papeles*	*Fases*
Evolutionary Project Management *EVO*	(Gilb, 1988). Utiliza la "Herramienta-?" y el Planguage. 10 principios. Modelo de 5 pilares: metas y valores, soluciones, estimación de impacto, plan evolutivo y Funciones, sub-funciones y definiciones. Es un método de gestión. La entrega evolutiva es estándar del DoD y del IEEE. Usa tablas de impacto.	No se definen los papeles de desarrollo ya que es más un método de gestión que de proceso. Mapa de flujo de valor para ver cuándo no se aporta valor al cliente.	Basado en el Planear-Hacer-Estudiar-Actuar de Shewhart. Utiliza entrega evolutiva (McConnell 1996). Valores: aprendizaje, incrementos temprano, pequeños y simplicidad.
Feature Driven Development *FDD*	(Palmer & Felsing, 2002). Proceso en 5 pasos, 2 últimos iterativos. Desarrollo basado en componentes orientados a objetos. Iteraciones cortas: 2 horas-2 semanas. Simplicidad, diseñar e implementar el sistema por características, modelado de objetos. Sólo se centra en diseño e implementación. Necesita otros enfoques.	Principales: Manager del proyecto, arquitecto jefe, jefe de desarrollo, programador jefe, propietarios de las clases, experto del dominio. Apoyo: Gestor de versiones, Experto del lenguaje, Ingeniero de las build, Toolsmith, Administrador de Sistemas	Elaborar modelo general, Construir lista de características, planear por característica, diseñar por característica y construir por característica (las dos últimas iterativas)
Internet-Speed Development *ISD*	Combinación de cascada y espiral. MSF es un ejemplo de ISD. Da importancia a dividir los proyectos en varios equipos y a la calidad. MSF es un ejemplo de ISD.	Los que tenga el método en concreto, como MSF	Previsión, planning, desarrollo, estabilización, despliegue. Build diaria. Importancia en la calidad y la gestión de riesgos.
Lean Software Development *LSD*	(Poppendieck, 2003). Utiliza 12 estrategias de gestión. Norton propuso 7 principios (eliminar desperdicios, amplificar conocimiento, decidir tan tarde como sea posible, entregar tan pronto como sea posible, otorgar poder al equipo, integridad incorporada y ver la totalidad) y 22 herramientas para llevarlos a cabo.	No se definen. Es un modelo logístico, un método basado en procesos productivos.	La práctica consiste en cumplir los 7 principios en todo lo que se haga
Microsoft Solutions Framework *MSF*	(Microsoft, 1994) MSF tiene una perspectiva de entrega de soluciones; MOF de servicio de gestión. MSF pone énfasis en los proyectos y MOF en hacer funcionar el entorno de producción. Tiene 8 principios: Comunicación abierta, visión compartida, equipo con poder, responsabilidad clara y aceptada, aportar valor, esperar el cambio, calidad y aprender de las experiencias.	Jefe de producto, de programa, de logística, desarrollador, tester y Encargado de la Experiencia con los Usuarios.	Prever, planear, desarrollar, estabilizar, desplegar. Muchos métodos son complementos adecuados para MSF: AM, RUP, XP, DSDM.
Open Source Software *OSS*	Basado en voluntarios, desarrollo distribuido, a veces, los problemas son más retos que necesidades comerciales. Licencias; código fuente disponible sin cargo. Es más una filosofía que un método, pero sus éxitos son bien palpables. Posibilidad para transformar software OSS en comercial.	Líderes del proyecto, Programadores voluntarios (Miembros Senior, Programadores Periféricos, Contribuyentes Ocasionales, Ayudantes), otros individuos y Posters.	Descubrir problema, encontrar voluntarios, identificar solución, escribir código y testear, revisar cambios de código, entregar código y documentación, gestionar versiones.

	Características	*Papeles*	*Fases*
Pragmatic Programming *PP*	(Hunt & Thomas, 2000) Énfasis en el pragmatismo, teoría de programar es menos importante, alto nivel de automatización en todos los aspectos. Consejos concretos y empíricamente validados. Se centra en prácticas individuales, pero no es un método con el cual pueda desarrollarse un sistema.	Sólo se centra en el papel de los programadores. Es importante que amplíen sus conocimientos. Filosofía de 6 principios.	Las 70 prácticas que lista son para el equipo de desarrolladores
Rational Unified Process *RUP*	(Kruchten, 2000). Modelo de desarrollo de software completo. Asignación de papeles según actividades. Método no demasiado ágil con modelado. Diseñado para complementar a UML. No tiene limitaciones de uso. Falta especificar cómo adaptarse a los cambios de requisitos. Prácticas: Desarrollo iterativo, Gestionar requisitos, Arquitectura basadas en componentes, Software de visualización de modelos, Verificar la calidad, Consideración de cambios.	Existen 9 categorías que dan lugar a los 30 papeles de RUP: Business Modeling (3), Requisitos (5), Análisis y diseño (6), Implementación (3), Test (2), despliegue (4), Configuración y Gestión de cambios (2), Gestión de proyecto (2) y entorno (3)	Concepción, elaboración, construcción, transición. Todas iterativas, de 2 semanas a 6 meses como máximo. Se especifican deliverables.
SCRUM	(Schwaber & Sutherland, 1995). Framework de gestión. Equipos independientes, pequeños, auto-organizativos. Ciclos de 30 días (*sprint*) para lanzar versión. Integración con muchos otros métodos, en especial XP.	Scrum master, propietario del producto, equipo de scrum, cliente y dirección.	Pregame (planning y diseño de arquitectura), game, post-game. Prácticas: product backlog, estimación de esfuerzo, sprint, reunión para planear sprint, sprint backlog, reunión diaria, reunión sprint review.
Extreme Programming *XP*	(Beck, 1999 y 2004) Desarrollo Costumer-driven, pequeños equipos, builds diarias. Basado en 5 valores: Comunicación, simplicidad, feedback, coraje y respeto. 12 prácticas, en especial refactorización. Se especifican más las prácticas individuales que la gestión global. Principios (1999): juego planteamiento, entregas frecuentes, metáfora, diseño simple, pruebas continuas, refactorizar, programación en parejas, propiedad colectiva, integración continua, semanas de 40 horas, cliente en casa, estándares de codificación. Extras: lugar de trabajo abierto y compartido, contratos alcance parcial. Tarjetas CRC	Programador, cliente en casa, tester, supervisor, instructor, consultor y director (big boss)	Exploración, planificación de las versiones, planificación de las iteraciones, Productionizing y muerte.

Tabla 27. Resumen comparativo de los métodos ágiles más importantes.

4.2. Crítica a las prácticas ágiles

1) COMPARTIR ESPACIO

Con respecto a la idea de mantener a todos los programadores en una habitación, existen maneras menos chocantes de conseguir que funcione el trabajo en equipo. El libro *Software Project Survival Guide* de S. McConnell (1997) es una referencia práctica y realista, aunque no haya tenido tanto éxito de ventas como *Extreme Programming Explained*.

2) REFACTORIZACIÓN

La refactorización es importante para mejorar el diseño de cualquier programa y para conseguir código más simple, aunque refactorizar a todas horas puede ser innecesario cuando se dispone de diseños previos. ¿No es más sencillo diseñar desde el principio, en lugar de refactorizar a todas horas? Recordemos que tras cada refactorización hay que volver a pasar las pruebas, gastando tiempo y dinero. XP insiste en que hay que tener coraje para refactorizar sin piedad y para desechar el código que no funcione tras una refactorización. Pero tirar código ya escrito es un desperdicio. Precisamente, el diseño pretende descartar código antes de que sea escrito.

Muchos programadores consideran diseñar sólo con la refactorización un proceso ineficiente. El uso correcto del polimorfismo y de la herencia implica un diseño previo. Aparte, la refactorización como diseño deja de lado características de los sistemas no expresables en lenguajes de programación (escalabilidad, velocidad, eficiencia, etc.). Una de las múltiples ventajas que otorga partir de un diseño previo es que se sabe cuándo un módulo, paquete o clase está acabado, y no se necesita perder más tiempo en refactorizar y ejecutar pruebas.

3) REQUISITOS

La falta de requisitos escritos detallados implica un claro desconocimiento de adónde se quiere ir. XP afirma que las historias de los usuarios y las tarjetas CRC ya proporcionan todos los requisitos necesarios. Puede que sea así, pero XP también afirma que se diseñó para gestionar "proyectos de alto riesgo con requisitos dinámicos". Para algunos resulta difícil que un proyecto de alto riesgo se aborde con unas historias, tarjetas y metáforas.

Frente al desprecio de los métodos ágiles hacia la recolección completa de requisitos, vemos una visión muy diferente en el artículo *Requirements Engineering as a Success Factor in Software Projects*, de H.F. Hofmann y Franz Lehner (IEEE Software, volumen 18, enero/febrero de 2001), donde se destaca la existencia e importancia de la ingeniería de requisitos. Este artículo empieza así: *"Los requisitos deficientes son la única y mayor causa de los fallos en los proyectos de software. A partir del estudio de cientos de organizaciones, Capers Jones descubrió que los requisitos de ingeniería son deficientes en más de 75% de todas las empresas. En otras palabras, tener los requisitos apropiados puede ser la única, más importante y difícil parte de un proyecto de software".*

La ingeniería de requisitos indica tanto el proceso de especificación de requisitos estudiando las necesidades de los contratantes como el proceso de análisis y purificación sistemática de aquellas especificaciones. El resultado principal es una especificación, un enunciado conciso de los requisitos que el software debe satisfacer; es decir, de las condiciones o capacidades que un usuario debe tener para lograr un objetivo o que un sistema posee para satisfacer un contrato o estándar. Idealmente, una especificación permite a los contratantes aprender

rápidamente sobre el software y desarrollos para entender exactamente qué es lo que los contratantes quieren.

A pesar de la terminología heterogénea en toda la extensión del texto, la ingeniería de requisitos debe incluir cuatro actividades relacionadas pero independientes: **extracción, modelado, validación y verificación**. En la práctica, la mayor parte de ellos variarán en tiempo e intensidad para diferentes proyectos. Por lo general, primero deducimos los requisitos a partir de cualquier fuente que esté disponible (expertos, depositarios, o el software actual) y luego los modelamos para especificar una solución. La extracción y modelado de requisitos están vinculados. El modelado describe una solución observada en el contexto del dominio de una aplicación utilizando apuntes informales, semiformales o formales. La normalización gradual de estos modelos desde el punto de vista de los requisitos lleva a una especificación de candidatos satisfactoria, que luego deben ser validados y verificados. Esto entrega a los contratantes la información de la interpretación de sus requisitos, de manera que puedan corregir los errores lo antes posible.

4) PROGRAMACIÓN EN PAREJAS

Respecto a la programación en parejas, recuerda el refrán "cuatro ojos ven más que dos". Quizá es una buena idea, pero a la mayoría de la gente no le gusta que la observen y controlen mientras trabajan. Muchas personas son susceptibles a las críticas o los "consejos", y prefieren encontrar ellas mismas sus propios errores.

Desde luego, la figura del programador solitario, dueño absoluto de su código y que almacena toda la información en su cabeza, lleva camino de la extinción. Existen, sin embargo, maneras más sencillas de fomentar el trabajo en equipo que la programación en parejas y el agrupamiento de todos los programadores en una misma sala.

La programación en parejas, desde luego, necesita pagar a dos programadores en lugar de a uno. Algunos seguidores de la programación en pareja afirman que una pareja de programadores escribe código tres veces más rápidamente que un solo programador. Sin embargo, algunos estudios concluyen lo contrario:

- Un estudio realizado en la Universidad de Utah apunta que los programadores en pareja escriben menos código que por separado (un 15% más de tiempo de trabajo), y que éste es ligeramente más claro que en la programación tradicional (un 15%) (http://collaboration.csc.ncsu.edu/laurie/Papers/WilliamsUpchurch.pdf).
- J. Nawrocki y A. Wojciechowski afirman en *Experimental Evalution of Pair Programming* (ESCOM 2001) que *"la programación es parejas es una tecnología bastante cara"* y que parece ser menos eficiente de lo anunciado en otros estudios anteriores.
- Matthias M. Müller y Frank Padberg alertaron en su *On the Economic Evaluation of XP Projects* (European Software Engineering Conference, 2003) de los riesgos económicos que conlleva la programación en parejas en algunas circunstancias.

La programación en parejas puede tener su sentido en actividades de enseñanza de la programación a alumnos o cuando se tratan problemas muy complicados. Más información en http://collaboration.csc.ncsu.edu/laurie/Papers/XPAUPairLearning.pdf.

Por otro lado, los proyectos de informática suelen contratarse a otras empresas, ajenas a la cultura empresarial de la empresa solicitante. A menudo, es difícil lograr que exista una

interacción social positiva entre los programadores de la empresa solicitante y los de la empresa encargada del proyecto. Es más: a veces, la interacción social resulta casi imposible. Basta con imaginar el caso de un programador subcontratado, con un contrato en prácticas, trabajando como programador externo en una gran empresa, rodeado de gente que cobra el doble o triple que él y que lo mira como un "externo", como alguien que está de paso. La integración en el grupo no siempre es fácil.

5) DOCUMENTACIÓN Y CÓDIGO FUENTE

Los métodos ágiles piensan que "refactorizando sin piedad" una y otra vez se obtiene código claro para todo el mundo. Sin embargo, cuanto más eficiente es un código en C++, más difícil se hace comprenderlo. Por otra parte, resulta difícil compaginar la claridad extrema con la idea de largos listados, de cientos y miles de páginas, por muy refactorizados que estén. Un sistema de software es más que el código fuente, del mismo modo que un montón de ladrillos no es un edificio. Incluso si las líneas fueran perfectamente inteligibles, entender un sistema real implicaría partir de bajo nivel: primero se entenderían líneas de código aisladas, luego métodos, luego clases completas, luego relaciones entre clases... llevaría mucho tiempo.

6) WORKPRODUCTS O ARTEFACTOS

Un miembro de Rational, John Smith, opina que la terminología de XP encubre una complejidad no reconocida: aunque las palabras "artefacto" y "producto de trabajo" no figuran en los índices de sus libros canónicos, Smith cuenta más de 30 artefactos encubiertos: Historias, Restricciones, Tareas, Tareas técnicas, Pruebas de aceptación, Código de software, Entregas, Metáforas, Diseños, Documentos de diseño, Estándares de codificación, Unidades de prueba, Espacio de trabajo, Plan de entrega, Plan de iteración, Informes y notas, Plan general y presupuesto, Informes de progreso, Estimaciones de historias, Estimaciones de tareas, Defectos, Documentación adicional, Datos de prueba, Herramientas de prueba, Herramientas de gestión de código, Resultados de pruebas, Spikes (soluciones), Registros de tiempo de trabajo, Datos métricos, Resultados de seguimiento. La lista podría ampliarse. En un proyecto pequeño, RUP demanda menos que eso. Al no tratar sus artefactos como tales, XP hace difícil operarlos de una manera disciplinada y pasa por ser más ligero de lo que en realidad es.

7) DISEÑO Y CÓDIGO FUENTE

XP afirma "El código es el diseño". Por una parte, diseñar y escribir código son actividades completamente distintas. El código fuente manifiesta el diseño a partir del cual fue creado, pero no es el diseño. Consideremos ahora que implementamos un determinado diseño en varios lenguajes de programación, ¿tenemos un diseño por cada lenguaje usado? No: en realidad tenemos un único diseño materializado de distintas formas.

Por otro lado, una manera de facilitar la resolución de problemas consiste en subir un peldaño en la escalera de la abstracción. Si en informática no se aceptara dicha estrategia, todavía se programaría en código máquina o en lenguaje ensamblador. El uso de diagramas UML, por caso, nos facilita la comprensión del diseño de un sistema mucho más que el código fuente. Por su naturaleza icónica, UML está mucho más cerca de nosotros que el código fuente. En un buen diseño se incluye una descomposición del sistema en subsistemas, así como una descripción de los servicios que cada subsistema proporciona a los restantes. Ojear diagramas UML es más fácil que leer páginas y páginas de código fuente. UML tiene sus problemas, y hay muchas situaciones que no pueden reflejarse exactamente con su notación, pero el código fuente aún está mucho más limitado.

Como se ha visto, los métodos ágiles dan una importancia capital al código (“*Código, código y código”*, “*¡Usa el código, Luke!”*), aunque no muestran mucho respeto hacia él. En XP, por ejemplo, si un fragmento de código no funciona bien se deshecha (*“Hay que tener valentía para tirarlo a la basura”*).

El planteamiento de “escribir código, comprobar, refactorizar” lleva a mezclar errores del código y errores de diseño. Una vez superados los errores del código, hay que solucionar los problemas de diseño (por ejemplo, las clases no se relacionan entre sí como deberían o los métodos están bien escritos, pero no dan los resultados esperados). Todo eso implica cambiar código ya escrito o tirarlo a la basura, desperdiciándose así el trabajo invertido en escribirlo y probarlo. Por tanto, la regla “Haga la cosa más simple que posiblemente podría funcionar” debe seguirse con cierto sentido común y ciertas matizaciones para no acabar despreciando mucho código ya escrito y probado.

8) PRUEBAS

Hacer hincapié en las pruebas es uno de los aspectos más positivos y realistas de los métodos ágiles. En el fondo, usan las pruebas y la refactorización para poner un poco de orden y de razón en el código, escrito a golpes de impulsos, de intuiciones y -a veces- de errores. Sin embargo, quizá centrarse sólo en las pruebas, ya sean unitarias o funcionales, no resulte buena idea. Una clase puede pasar todas las pruebas unitarias que se quiera y, sin embargo, puede estar mal diseñada. Nada nos dicen las pruebas unitarias sobre los fallos de diseño, los cuales se pueden traducir en un mal rendimiento del sistema o en que sea difícil de mantener.

Consideremos, por ejemplo, una jerarquía de clases. Las clases pueden funcionar correctamente y pasar todas las pruebas que se han pensando para ellas. No obstante, la jerarquía puede estar mal diseñada. Puede ser que haya métodos que estén a una altura incorrecta de la jerarquía o que hagan demasiadas llamadas innecesarias a métodos de otras clases de las jerarquías. En estos casos, refactorizar puede no ser la solución para los problemas.

Suponer que un sistema funciona bien porque pasa todas las pruebas es ingenuo: las pruebas pueden tener errores. Si se dispone de una especificación previa -y escrita- de los requisitos, se puede saber qué resultados reales se esperan del sistema y juzgar su comportamiento. Sin una especificación exacta y consensuada del sistema, el programador puede alegar que el sistema funciona bien según sus pruebas; y el cliente puede decir, resignadamente, que el sistema no cumple sus expectativas. No escribir los requisitos se traduce, con el tiempo, en que cada uno los recuerda a su manera.

La programación orientada por pruebas, *test-driven development*, en su modalidad convencional, se resiste a ciertos ambientes académicos. Dijkstra afirmaba que ***“la prueba de programas puede ser una manera muy efectiva para mostrar la presencia de bugs, pero es irremediablemente inadecuada para mostrar su ausencia”***. En la enseñanza se prefieren los métodos formales de demostración, a los que en el movimiento ágil rara vez se hace referencia.

9) CLIENTE “EN CASA”

La idea del cliente “en casa”, en el sentido definido originalmente por XP, plantea bastantes problemas prácticos. Por un lado, la empresa promotora debe prescindir de un trabajador durante el tiempo del proyecto (que pueden ser meses o años). Eso exige que no se elija para ese puesto a una persona con mucha experiencia o buena conocedora del negocio (directivos,

socios, etc.), porque las personas con esas cualidades son necesarias en la empresa, pero a su vez, implica que esa persona no conocerá a fondo los requisitos exactos.

10) CONTRATO DE ALCANCE PARCIAL

Pongámonos en la situación de un responsable de informática de una empresa que firma un contrato de alcance optativo o parcial cuya primera iteración durará tres semanas y costará 100.000 €. Pasadas las tres semanas, puede encontrarse con una de estas opciones:

a) El sistema se corresponde bastante bien a sus necesidades.
b) El sistema corresponde en parte a sus necesidades.
c) El sistema no guarda relación con lo solicitado.

En el caso c), la empresa habrá perdido 100.000 €, más los gastos ocasionados por no tener el sistema aún en marcha. La empresa ni siquiera podrá reclamar legalmente por incumplimiento de contrato, pues han firmado un contrato aceptando lo que salga al final de la primera iteración. Mirado así, nos llevaría a la necesidad de establecer unos requisitos y unos contratos más convencionales.

4.3. Métodos ágiles y desarrollo de software libre

Antes de trazar paralelismos entre la programación extrema y el desarrollo del software libre, debe plantearse una cuestión. ¿Cuál es la metodología, si es que existe alguna, que siguen los proyectos de software libre? Obviamente, no es posible dar una respuesta en términos absolutos, teniendo en cuenta la infinidad de proyectos de software libre que existen. No obstante, observando los proyectos más representativos, se pueden identificar algunos rasgos comunes.

La mejor representación del proceso de desarrollo de software libre es el llamado **modelo bazar** (en oposición al **modelo catedral**, más habitual en el mundo de software propietario). Se trata de una analogía entre el proceso de desarrollo de software y el funcionamiento de un bazar.

En un bazar, los tenderos acuden, plantan sus puestos y dialogan con los clientes. No existe una autoridad central que organice el funcionamiento, pero eso no impide que tanto los clientes como los proveedores consigan sus objetivos. En el otro lado, la catedral es diseñada por un arquitecto, ejecutada siguiendo un plan minuciosamente elaborado, completada y entregada al cliente.

Algunas de las prácticas de la programación extrema son totalmente compartidas por el modelo bazar:

El software libre aplica la máxima "*release soon, release often*", por la cual se compilan nuevas versiones muy frecuentemente, incluso desde el mismo comienzo del proyecto. No es extraño ver números de versiones como 0.0.1 ó linux-2.6.15-rc2 (RC significa *Release Candidate*). Esta práctica encaja perfectamente con las entregas frecuentes de la programación extrema.

También se produce integración continua, gracias a que el código se almacena en repositorios de control de versiones (CVS, Subversion, etc.). En cualquier momento, un nuevo usuario o desarrollador puede descargarse el código del repositorio y compilarlo (y si no lo consigue, se pondrá en contacto con los desarrolladores para notificar el problema). Muchos proyectos ofrecen también versiones empaquetadas automáticamente (*nightly builds*) para los que no

quieren acceder directamente al repositorio. Es también muy común encontrar proyectos que mantienen dos ramas de desarrollo, una considerada *stable* y otra denominada *unstable* o *devel*.

Práctica XP	**Modelo bazar**	**Herramientas**
Juego de la planificación		
Entregas frecuentes	Release soon, release often	
Diseño simple		
Pruebas automáticas		JUnit, HttpUnit, DbUnit...
Integración continua	Repositorios, nightly builds, unstable	CVS, SVN, Ant, CruiseControl, Tinderbox
Refactorización		Eclipse
Programación por parejas		
Propiedad colectiva del código	GPL y otras licencias libres	CVS, SVN
Semana de 40 horas		
Cliente en el equipo	Feedback de bugs y feature requests	Bugzilla, listas de correo, foros
Metáfora		
Estándares de codificación	Guías de estilo	Jalopy, Indent, JCSC...

Tabla 28. Relación entre las prácticas de la programación extrema, las del modelo bazar y las herramientas libres que dan soporte a las prácticas.

La rama *unstable* suele ser el foco de la integración continua. A veces, esta distinción se hace mediante un convenio de numeración de versiones (números impares indican versiones en desarrollo, números pares indican versiones estables); otras veces se mantienen explícitamente los nombres de las ramas y se da al usuario la posibilidad de elegir, como por ejemplo en Linux Debian. Precisamente en esta distribución se produce uno de los casos más representativos de integración continua, puesto que los contenidos de la rama *unstable* se están integrando permanentemente de forma automática, usando el demonio *buildd*.

La **propiedad colectiva** del código está en la propia naturaleza del software libre, por lo que esta práctica es perfectamente compatible con el modelo bazar.

Respecto a los **estándares de codificación**, algunos grandes proyectos de software libre disponen de guías de estilo explícitas para la codificación. Incluso en aquellos proyectos que no disponen de ellas, el estilo suele estar implícitamente determinado por el código escrito por los líderes del proyecto.

La práctica de **introducir al cliente** en el equipo no es perfectamente asimilable, puesto que los proyectos de software libre se caracterizan por una gran dispersión geográfica y tampoco es sencillo identificar quién es el cliente. Es frecuente que el desarrollador sea el propio cliente, puesto que la motivación para desarrollar software libre es, en muchos casos, egoísta (resolver una necesidad propia). Para facilitar la participación del cliente, la comunidad del software libre ha creado herramientas colaborativas como los gestores de errores (*bugtrackers*), los *wikis* (webs donde todo el mundo colabora), las solicitudes de funcionalidades (*feature requests*), los foros, las listas de correo, los canales de IRC, etc.

Utilizando estas herramientas, los programadores y los clientes (usuarios) se comunican de forma ágil, salvando la dispersión geográfica.

Por último, hay un conjunto de prácticas de la programación extrema para las que difícilmente se puede encontrar equivalencia en el modelo bazar:

El **juego de la planificación** no encaja en el modelo bazar, que se caracteriza por una escasa planificación: habitualmente no se planea qué funcionalidades incluirá la próxima versión, ni cuándo estará lista (de hecho, cuando se pregunta cuándo saldrá la próxima versión, se suele responder que "*saldrá cuando esté lista"*). Dada la creciente repercusión comercial del software libre, recientemente algunos proyectos, como GNOME o Ubuntu, han comenzado a introducir cierta planificación en sus ciclos de desarrollo, fijando las fechas de entrega de nuevas versiones, las funcionalidades que incorporarán, la duración de los procesos de congelación, el tiempo durante el que se mantendrá el soporte técnico y de seguridad de las versiones antiguas, etc.

El **diseño simple** tampoco es una característica que sea explícitamente promovida por el modelo bazar. Sin embargo, el diseño simple suele manifestarse en los proyectos de software libre como un efecto lateral del entorno: en un ámbito de alta competencia (para cualquier área, pueden encontrarse varios proyectos de software libre que la cubren) se establece un proceso de selección natural. Como los proyectos con un diseño simple son más flexibles y escalables, consiguen atraer más fácilmente a nuevos desarrolladores y sobreviven.

Al tratarse de equipos muy heterogéneos, la práctica de la **refactorización** queda frecuentemente a cargo de cada programador. La programación por parejas no puede ser llevada a cabo porque los programadores se encuentran geográficamente distantes y no comparten un lugar y horario de trabajo.

4.4. Otros factores influyentes

4.4.1. La programación extrema y Smalltalk

Los creadores de la programación extrema (Cunningham y Beck) proceden de la programación en Smalltalk. Consciente o inconscientemente, han llevado a XP algunas prácticas específicas de ese lenguaje, que no son generalizables con facilidad a otros como C++, Java o lenguajes .Net. Smalltalk es un lenguaje puro orientado a objetos. Es de tipos dinámicos, con ligadura dinámica pura. En Smalltalk, por consiguiente, no se declara que una variable es de un cierto tipo concreto.

Este planteamiento presenta ciertas desventajas:

- Sólo se conoce si el mensaje enviado a un objeto forma parte de su protocolo en tiempo de ejecución. Como puede esperarse, esto conlleva en tiempo de compilación una gran cantidad de errores del tipo "no coinciden los tipos" o la necesidad de escribir código que compruebe los tipos en tiempo de ejecución o que procese las excepciones generadas.
- Todas las operaciones, incluso las más simples (sumas de números, etc.), se ligan dinámicamente. Esto redunda en un tiempo de procesado adicional en tiempo de ejecución. En jerarquías muy profundas de clases y con muchos métodos, el tiempo necesario para localizar el método adecuado puede ser significativo.

Sin embargo, los tipos dinámicos y el enlace dinámico puro permiten que Smalltalk sea un lenguaje muy sencillo: sólo hay cinco palabras reservadas y la sintaxis es muy simple. Lenguajes como C++, Java y C# tienen más de cincuenta palabras reservadas, además de contar con una sintaxis más compleja que la de Smalltalk.

El entorno de desarrollo de Smalltalk forma una unidad con lo que es el lenguaje en sí, y permite modificar o añadir código durante la depuración, sin que sea necesario parar el programa y volver a compilar. Con los modernos entornos de desarrollo integrado, quizá esto parezca lo habitual, pero en la década de los ochenta, e incluso cuando se desarrolló XP, el entorno de Smalltalk era muy superior al resto.

La idea de escribir pruebas unitarias antes de escribir el código que se desea probar tiene sentido en Smalltalk, pues no existe impedimento a referirse a código que aún no existe. En cambio, en lenguajes como C++, Java o C# no pueden usarse referencias a objetos que aún no existen. Para poder seguir este principio de XP, se debe recurrir a *frameworks* específicos o a objetos *mock*. Los objetos *mock* son objetos que se usan para comprobar el comportamiento de otros objetos. Los podemos imaginar como implementaciones de una clase (o interfaz) que simulan el comportamiento real que tendría una verdadera implementación de ésta. No deben confundirse los objetos *mock* con los objetos *stub*. Los *stubs* sólo proporcionan la implementación de una clase mediante la implementación de su interfaz (es decir, del conjunto de métodos públicos de la clase).

Un objeto *mock*, en cambio, incluye además la capacidad de comprobar cómo interaccionan sus métodos con el resto de los objetos del sistema, y permite avisar al programador de cualquier discrepancia con respecto al comportamiento esperado. Para ello permite, antes de ejecutar una prueba, cargar objetos con datos, los cuales pueden llamarse durante el transcurso de una prueba o de varias. Un uso habitual de los objetos *mock* es simular conexiones JDBC (Java Database Connectivity), lo cual permite realizar pruebas continuas de conexión y desconexión. La escritura de objetos *mock*, innecesarios en Smalltalk, conlleva un coste que debe ser tenido en cuenta al abordar proyectos “extremos” escritos en otros lenguajes. Por otro lado, la escritura de código sencillo de leer es mucho más difícil en lenguajes del estilo de C que en Smalltalk.

4.4.2. La curva de Boehm y XP

El matemático Barry Boehm estableció en 1987, tras estudiar los datos de sesenta y siete proyectos de software, que encontrar y solucionar un problema de software después de que haya sido entregado al cliente cuesta hasta cien veces más que encontrar y arreglar el problema en las etapas iniciales de diseño. En 2001, Boehm postuló que para sistemas pequeños, el factor de proporcionalidad se halla más próximo a cinco que a cien.

En el libro *Extreme Programming Explained* de Kent Beck, aparece una gráfica que se conoce ahora como la curva de Boehm. En ella podemos ver cómo aumenta el coste de los cambios en una u otra etapa del desarrollo de software.

Beck habla del **costo del cambio**, mientras que Boehm habla del **costo de solucionar errores**. Si se acepta la afirmación de Beck, se pueden saltar las etapas de análisis y de diseño: lo mismo cuesta cambiar algo o arreglar algún error en esas etapas que en las de implementación, pruebas y distribución.

Según Beck, hoy día no es válida la curva exponencial de Boehm, pues el uso de lenguajes orientados a objetos, de las modernas herramientas de desarrollo y de ciertas prácticas (usadas en la programación extrema) transforma la curva de Boehm en una curva plana. En consecuencia, con XP puede conseguirse un coste fijo para todas las etapas del proceso de desarrollo del software. Tal y como escribe: *"El cambio es barato. El coste del cambio no se eleva exponencialmente cuando el sistema crece. Contrariamente a la creencia popular, el aumento en el coste del cambio disminuye gradualmente"*.

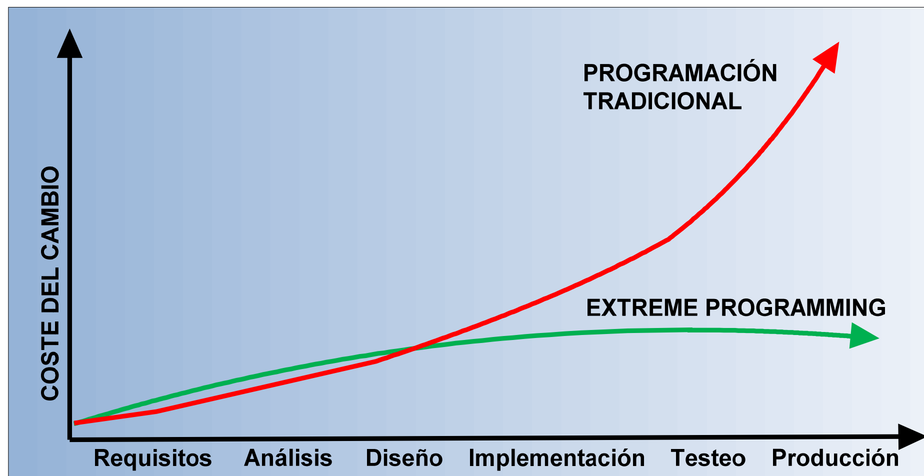


Ilustración 168. Gastos derivados del cambio en XP y en los métodos tradicionales según Beck.

Existe, sin embargo, un problema fundamental: la curva plana de Beck no está justificada por datos empíricos actuales. Pese a todos los avances en software, la curva sigue siendo exponencial (y nada invita a pensar que acabará siendo plana). Los defensores más realistas de XP afirman que la curva de Boehm sigue siendo exponencial, y que XP es sólo un método para atacar los costes que aparecen en la curva de Boehm.

Aceptar eso resulta muy restrictivo para XP, pues lo reduce a emplearlo cuando el coste de implementar las funciones del sistema no crece demasiado rápido (o sea, para proyectos pequeños o muy pequeños). Para otro tipo de proyectos, todos los errores e inexactitudes procedentes de las historias de usuarios, etc., se arreglarán en la etapa de implementación y en la de pruebas (lo que costará mucho más tiempo y dinero que haberlos solucionado desde el principio en las etapas de análisis y diseño).

XP argumenta que, aun cuando no pueda conseguirse una curva plana, esta metodología sí permite controlar el coste del proyecto, mediante iteraciones de corta duración y lanzamiento de versiones. Obviamente eso es cierto, pero ya lo hacían los métodos basados en procesos iterativos e incrementales.

¿Cómo atraer el interés de la industria? Se postula que la curva de Boehm se vuelve plana con XP. Como eso no es exactamente cierto, se dice que, pese a todo, permite controlar los gastos del proyecto y se recuerda que XP tiene algunas buenas prácticas (conocidas y usadas desde hace más de treinta años aunque quizá sin bautizarlas). Muchos consideran que no hay razón para pensar que XP controlará los gastos mejor que otro proceso iterativo e incremental.

4.4.3. El proyecto C3

En la bibliografía de XP, aparece el proyecto C3. Podemos dar por ciertos estos hechos:

- El proyecto C3 comenzó en enero de 1995 como un proyecto de precio fijado realizado en Smalltalk (el lenguaje ideal para XP, por su tipado dinámico).
- En mayo de 1996, la empresa encargada del proyecto no consiguió entregar un sistema que funcionara. En ese momento, Beck entró en el proyecto y lo volvió "extremo".
- Durante las treinta semanas posteriores a la llegada de Beck, la productividad del equipo aumentó de manera significativa.
- En agosto de 1998, el C3 pagaba las nóminas a un grupo piloto de unas 10.000 personas (en pruebas probó ser capaz de pagar a otras 20.000 más).
- En febrero de 2000, el proyecto fue cancelado. Daimler-Chrysler abandonó XP como método de desarrollo.

Como vemos, el proyecto estuvo cuatro años siguiendo las prácticas extremas y bajo la batuta del creador de XP, pero el resultado no fue ningún éxito. Pero hasta los fracasos se pueden vender y rentabilizar. Veamos, por ejemplo, lo que apareció publicado en el artículo *Extreme measures* de la revista semanal *The Economist* (7 de diciembre de 2000, ocho meses después de que el proyecto C3 fuera desechado):

> *La programación extrema (XP) fue inventada en 1996, cuando Kent Beck, desarrollador de software, fue llamado por un fabricante americano de coches, Chrysler, para rescatar un proyecto que había demostrado ser tan frustrante que había sido desechado. Cuando el señor Beck trabajó en esta empresa sumida en la ignorancia, conocida como Chrysler Comprehensive Compensation (C3), formuló una serie de directrices para mantener el código "elegantemente escrito". El sistema C3 proporciona ahora la información correcta de las nóminas mensuales para más de 86.000 empleados.*

Resulta difícil creer que *The Economist* dejara que transcurrieran ocho meses entre la redacción del artículo y su publicación. Quizá Beck no reflejó "exactamente" en la entrevista cuál había sido el final del proyecto: el sistema jamás pagó a más de 10.000 empleados; ni siquiera en fase de pruebas demostró poder pagar a más de 30.000 personas. Esta falsa información, publicada en una revista muy influyente en los ambientes económicos, animó a muchos empresarios a probar suerte con XP. No olvidemos que el artículo también decía:

> *Enseñe a un programador alguna disciplina y programará lógica y limpiamente mientras dure el día; déle libertad absoluta y programará toda su vida a su modo idiosincrásico. Pero en el mundo rápidamente cambiante del diseño de software, nadie quiere pasar meses estudiando minuciosamente los rollos de código indescifrable cuando se necesitan reparaciones urgentes. Por eso, los informáticos tras el movimiento llamado "programación extrema" (XP) argumentan que más método igual a menos locura, sobre todo cuando los equipos de programadores afrontan presupuestos apretados y fechas límites estrictas.*

Kent Beck y sus colaboradores no entendieron por qué se canceló el proyecto C3. Según ellos, *"era un éxito"* y *"formaban el mejor equipo de desarrollo sobre la faz de la tierra"*. Quizá el proyecto fue divertido para los desarrolladores, pero no para los clientes que pagaban.

Según Beck, "*el problema fundamental fue que el Gold Owner (promotor) y el Goal Donor no eran el mismo. El cliente que suministraba historias no se preocupaba de las mismas cosas que los gestores que evaluaban el rendimiento del equipo"*. El *Goal Donor* es el representante de los clientes que se sienta en la habitación con los programadores, que explica

que todo está bien para cambiar los requisitos en medio del proceso, mientras que el *Gold Owner* es el promotor del proyecto. En el caso del C3, el *Gold Owner* canceló el proyecto (febrero 2000), después de cuatro años de trabajo, porque el programa sólo estaba pagando a un tercio de los empleados. Para ver las esperanzas puestas en XP, se puede consultar el artículo *Chrysler Goes to "Extremes",* publicado dos años antes de la cancelación del proyecto en www.xprogramming.com/publications/dc9810cs.pdf.

Las desventuras iniciales de XP no acaban en el proyecto C3. El segundo proyecto (*Vehicle Cost and Profit System: VCAPS*) para Ford tampoco acabó bien, se canceló antes de finalizar. El VCAPS pretendía jubilar a otro sistema antiguo, desarrollado por el método en cascada. Algunos creen que el proyecto "murió con las botas puestas", otros que fue cancelado por motivos políticos, y otros creen que "la operación fue un éxito, pero el paciente murió".

4.4.4. Marketing. El poder de la palabra

Posiblemente, el mayor éxito de los métodos ágiles sea la elección del nombre. La palabra ágil suena a dinámico, sencillo, flexible, rápido. Resulta atractiva para los desarrolladores y jefes. Y no digamos eXtreme Programming: un buen nombre comercial. Quiere dar sensación de que con XP, usted no se sienta delante de un terminal y escribe código. No, usted realiza prácticas extremas, arriesgadas... aunque realmente nos limitemos a compilar "Hola, mundo" en C++. También artículos como el que apareció en *The economist* ayudaron.

4.5. Complementariedad y similitudes

Hemos visto métodos que no son excluyentes, algunos casan y están pensados para complementar a otros. Por ejemplo, LSD y Extreme Programming se acoplan muy bien:

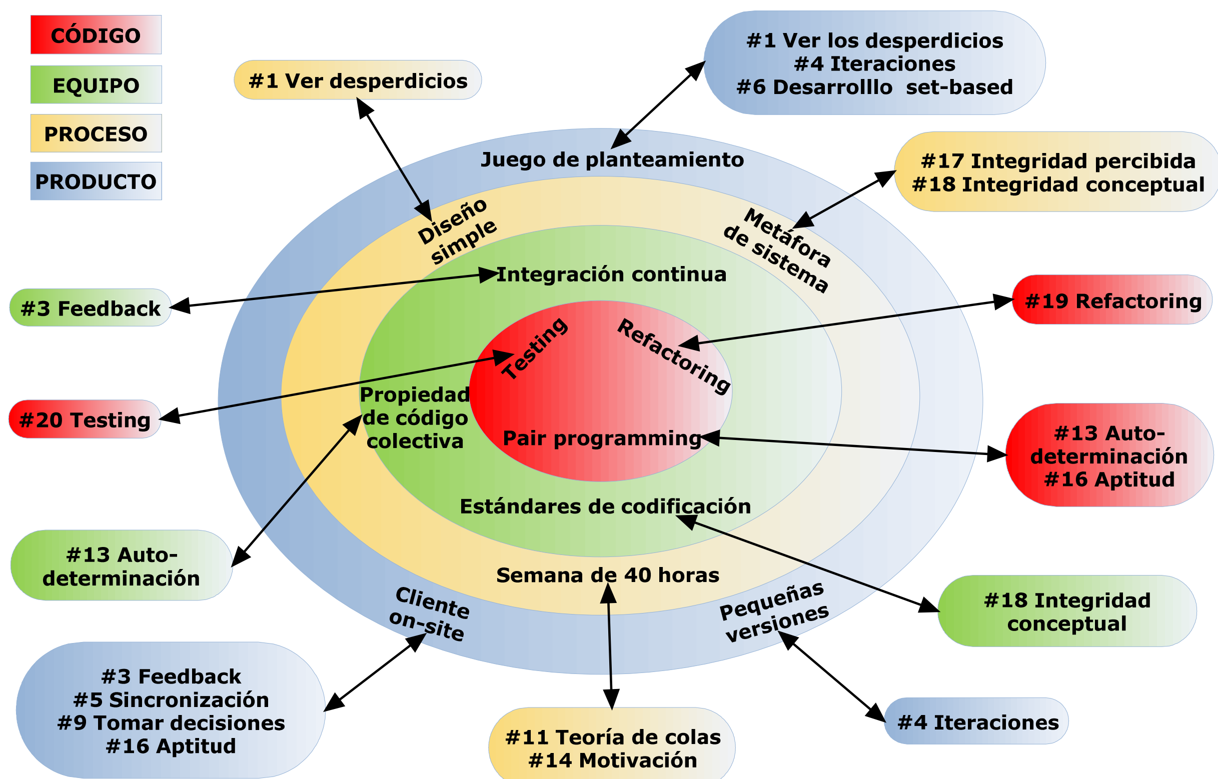


Ilustración 169**. Comparación-integración entre XP y LSD.**

Existen numerosas muestras de complementariedad entre los métodos:

- En las herramientas de RUP ahora hay plug-ins para XP.
- *Crystal* permite la adopción de prácticas de otras metodologías como XP y *Scrum* para reemplazar algunas de sus propias prácticas, como los talleres de reflexión.
- Se ha elaborado en particular la combinación de DSDM con XP, llamada EnterpriseXP. También hay combinaciones entre DSDM y RUP o MSF.
- Para las técnicas concretas de programación, LD promueve el uso de otros MA que sean consistentes con su visión, como XP o sobre todo *Scrum.*
- Pragmatic Programming sólo da pautas sobre la forma mejor forma de programar, dando cabida a cualquier otro método en lo que se refiere a gestión, artefactos, etc.
- Muchos métodos son complementos adecuados de MSF: AM, RUP, XP, DSDM...

Eliminando diferencias terminológicas, se podrían considerar varias combinaciones con las disciplinas de MSF. Se podría proponer que MSF se utilice como marco general, Planguage como lenguaje de especificación de requisitos, Scrum (con sus patrones organizacionales) como método de gestión, XP (con patrones de diseño, programación guiada por pruebas y refactorización) como metodología de desarrollo, RUP como abastecedor de artefactos, ASD como cultura empresarial y quizá hasta CMM como método de evaluación de madurez.

4.6. Estadísticas

4.6.1. Estadísticas del grupo Standish (1994-2004)

En el gráfico siguiente se aprecia el uso de las funciones del software, según un estudio de Standish Group. Prácticamente la mitad de las funciones no se usa nunca. Esto es un claro 80/20: el 80% del valor proviene del 20% de las características. Concentrarse en el 20% útil es una aplicación del mismo principio que subyace a la idea de YAGNI o "no vas a necesitarlo" de XP. El artículo completo se encuentra en los anexos.

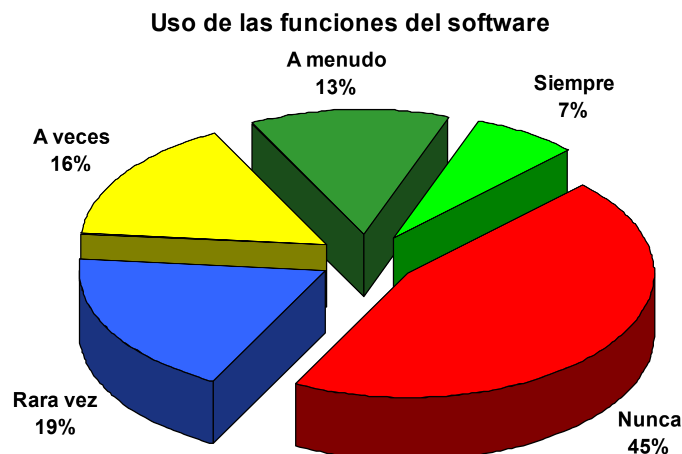


Ilustración 170. Estadística sobre el uso de las funciones del software.

Si comparamos la estadística del grupo Standish, CHAOS Database, entre los años 1994 y 2004 vemos que la adopción de prácticas ágiles o métodos ágiles, ha mejorado el panorama:

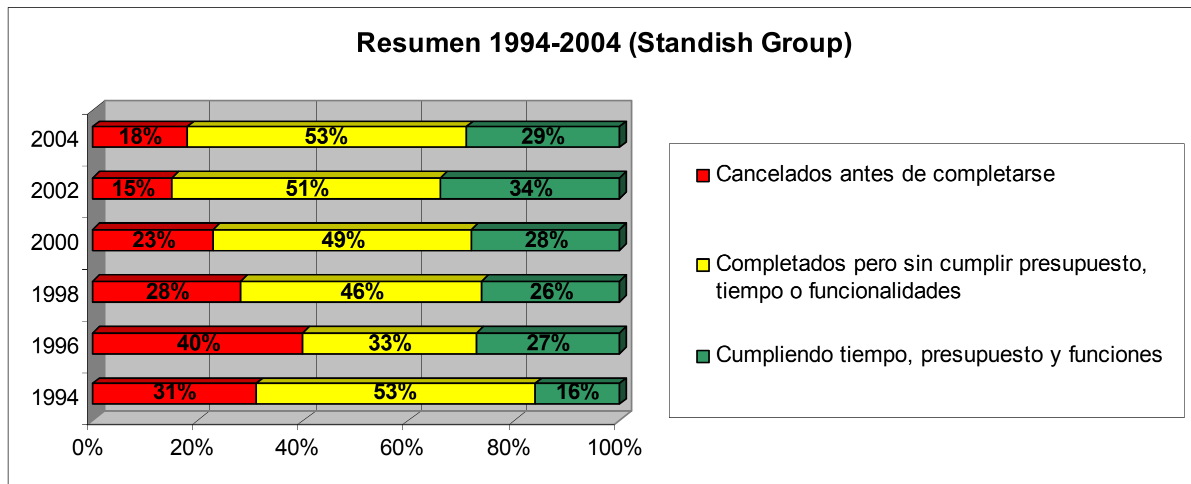


Ilustración 171. Mejoras de 1994 a 2004 al aplicar conceptos ágiles.

También podemos apreciar las mejoras en el coste extra (fuera de presupuesto según el presupuesto original, en porcentaje) y el tiempo de más que se necesitó para finalizarlo (en porcentaje respecto al estimado inicialmente):

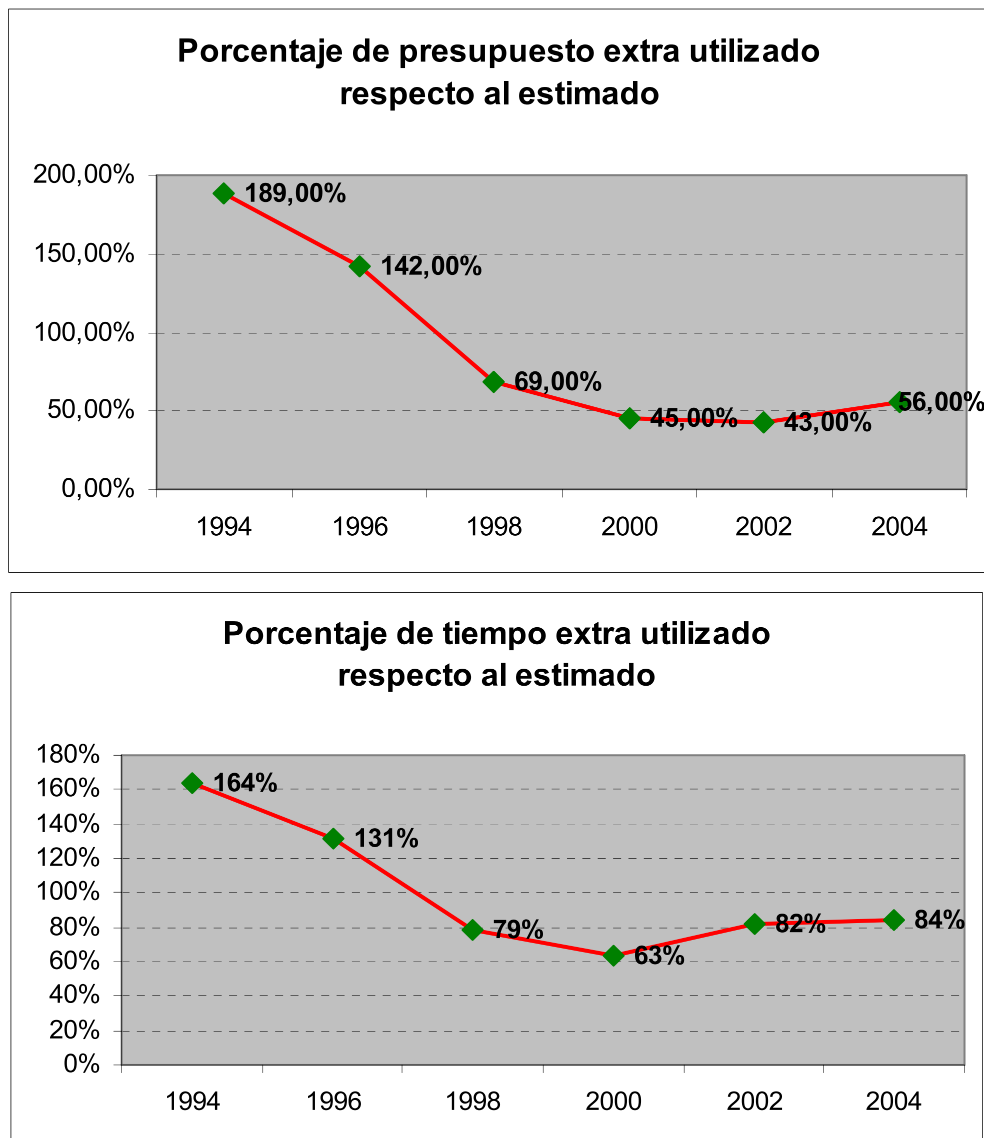


Ilustración 172. Sobrecosto económico y de tiempo respecto a las estimaciones iniciales.

4.6.2. Estadísticas de Ambysoft (2006)

Una encuesta realizada a 4.232 profesionales de las tecnologías de la información en marzo de 2006, llevada a cabo por Scott Ambler en Ambysoft (Dr. Dobb's Portal - The world software development - www.ddj.com/dept/architect/191800169) se resume en:

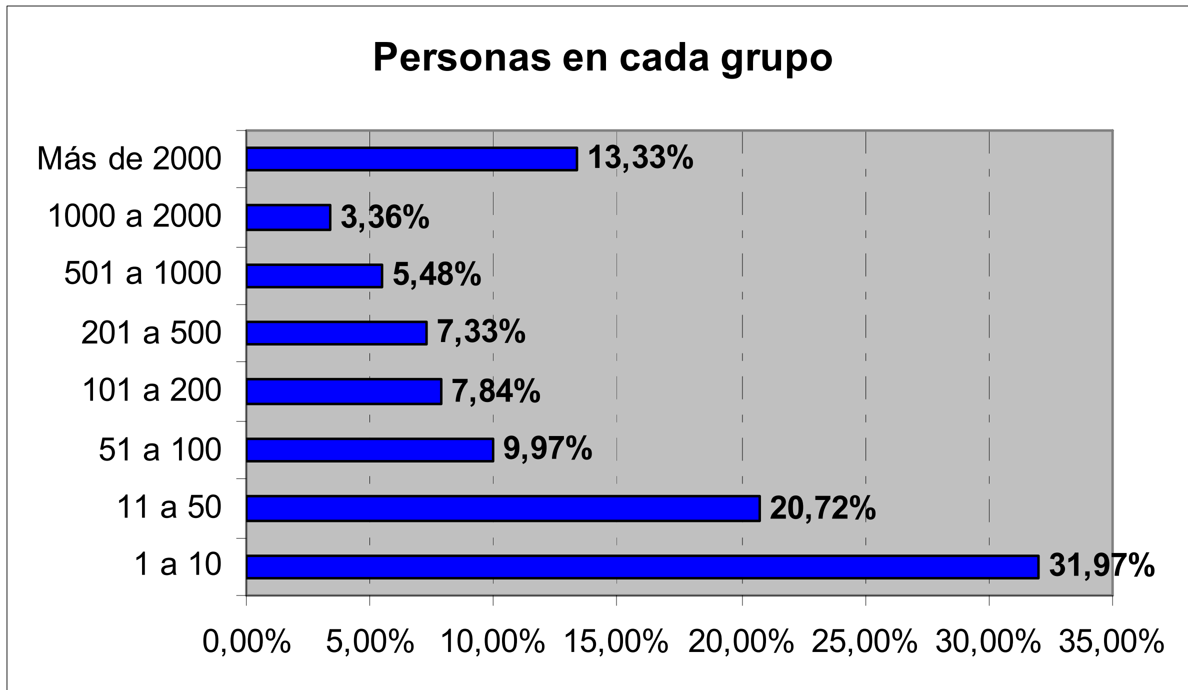


Ilustración 173. Tamaño de los grupos encuestados.

Sobre el colectivo de la muestra, sus conocimientos respecto a métodos ágiles, así como el nivel que considera que tienen (sobre 5) en las diferentes áreas:

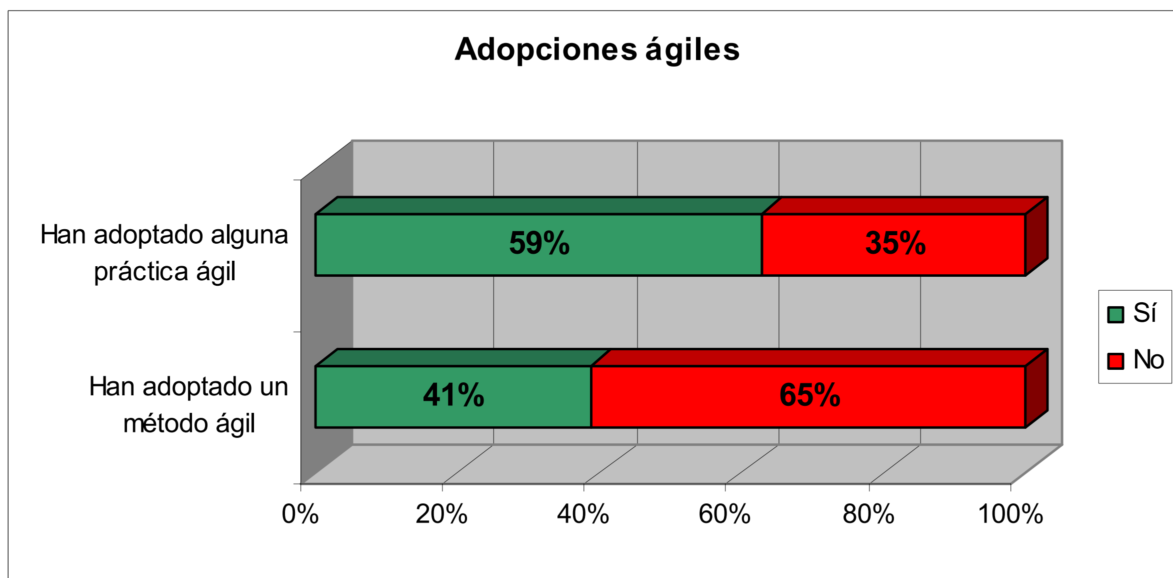


Ilustración 174. Porcentajes de proyectos que utilizaron prácticas o métodos ágiles.

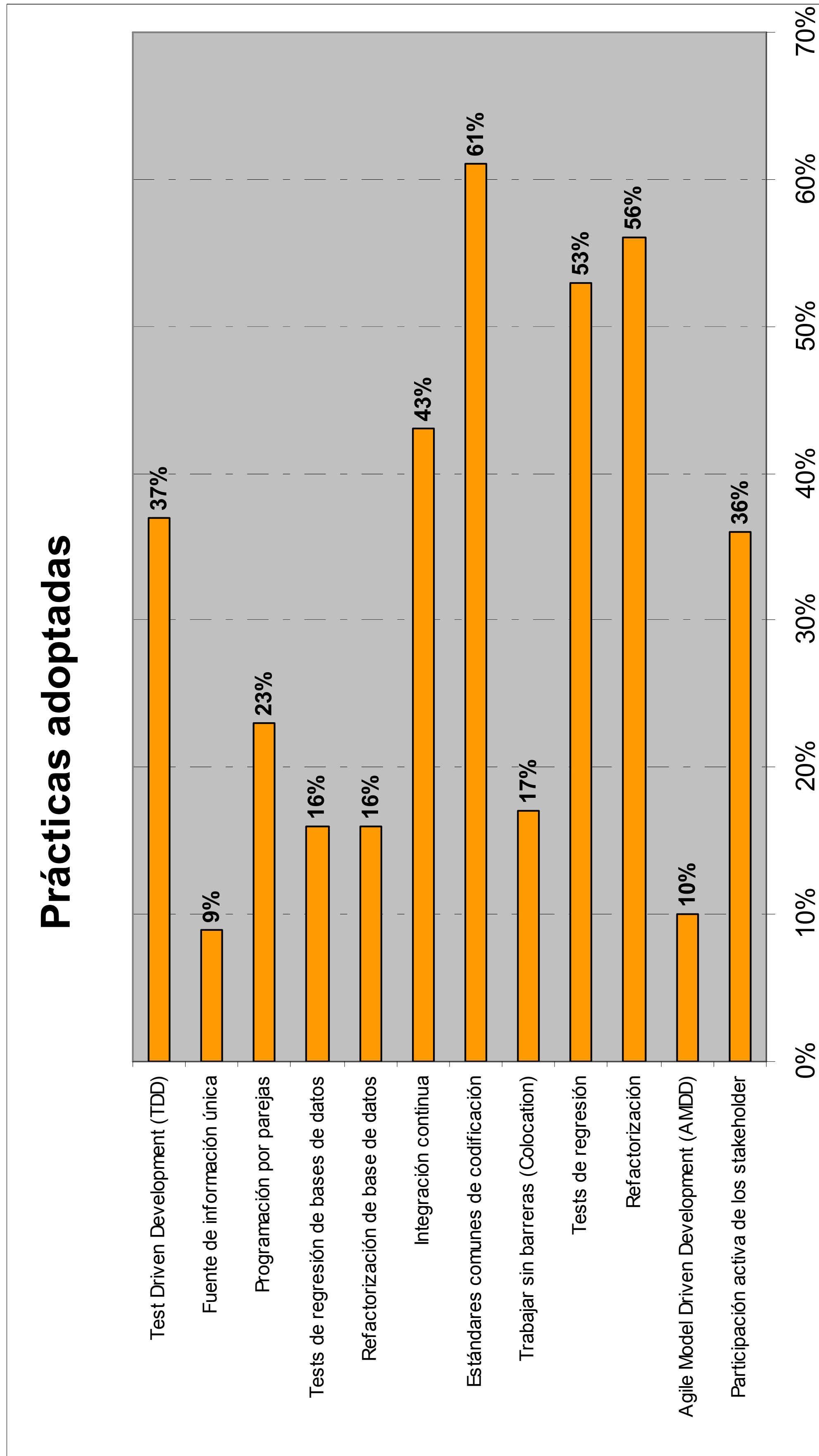


Ilustración 175. Prácticas ágiles concretas utilizadas.

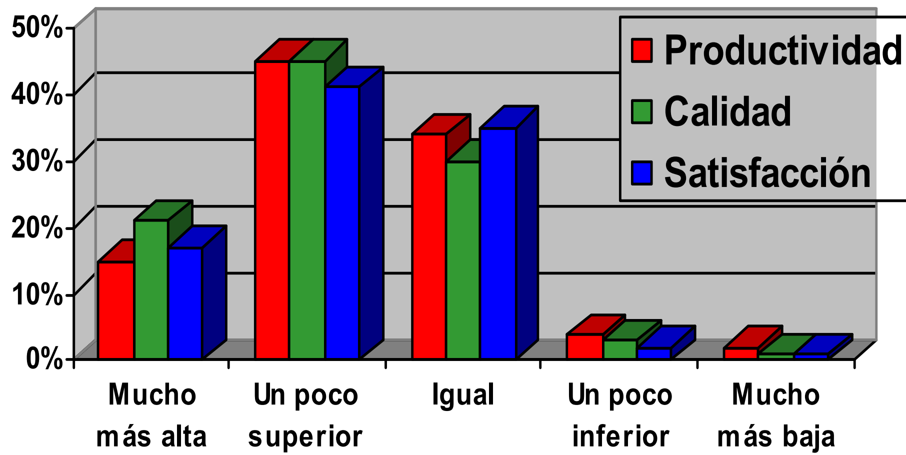


Ilustración 176. Cambios en la productividad, calidad y satisfacción al aplicar prácticas ágiles

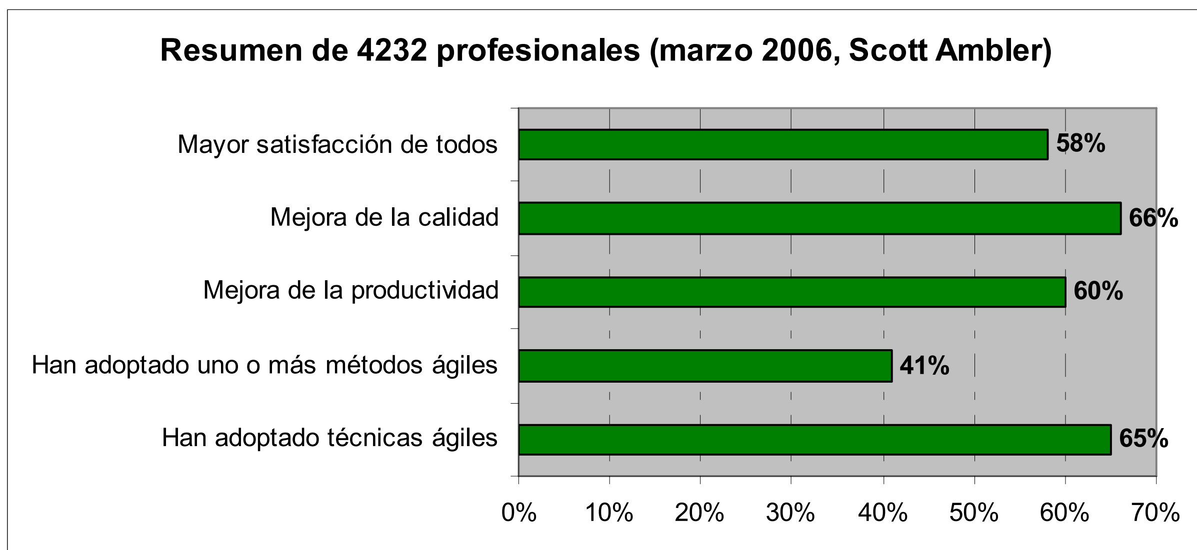


Ilustración 177. Resumen de beneficios al aplicar conceptos ágiles.

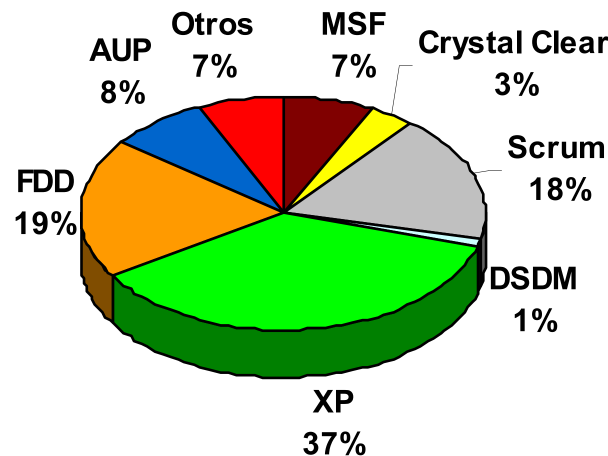


Ilustración 178. Métodos utilizados en base a un estudio donde el 41% de 2611 proyectos que afirmaron usar métodos ágiles.

Los 10 factores para el éxito según el grupo Standish son:

1) Implicación del usuario/cliente
2) Apoyo de la dirección ejecutiva
3) Objetivos comerciales claros
4) Optimizar el alcance/posibilidades
5) Proceso ágil
6) Jefe de proyecto competente
7) Gestión financiera
8) Recursos cualificados
9) Metodología formal
10) Herramientas e infraestructura estándares

4.7. Problemas comunes a los métodos ágiles

Entre los métodos ágiles son comunes estos problemas:

1) Falta de documentación de diseño (en el sentido convencional)
Las tarjetas CRC, la metáfora, la refactorización y el código no proporcionan una documentación perdurable sin ambigüedad. El código no puede tomarse como una documentación. Primero, porque ningún lenguaje actual proporciona código legible directamente, y segundo, porque en sistemas de tamaño medio o grande se necesitaría leer los cientos o miles de páginas del listado de código fuente.

Desde el punto de vista del mantenimiento y uso del sistema, proponer poca o ninguna documentación constituye una mala práctica. Las tarjetas CRC no son recomendables para utilizar en sistemas grandes (con más de 30-40 clases). Incluso en sistemas pequeños, estas tarjetas no capturan las interacciones (envío de mensajes) entre objetos.

2) PROBLEMAS DERIVADOS DE LA COMUNICACIÓN ORAL

Este tipo de comunicación resulta difícil de preservar cuando pasa el tiempo y está sujeta a muchas ambigüedades.

3) FALTA DE CALIDAD

Probar el código de forma constante no genera productos de calidad, sólo revela falta de análisis y diseño. Por otro lado, ningún método ágil cuenta con algún sistema para asegurar la calidad de los sistemas. Métodos como el TSP (Team Software Process) o el PSP (Personal Software Process)[33] sí miden y gestionan la calidad del producto final.

4) FUERTE DEPENDENCIA DE LAS PERSONAS

Como se evita en lo posible la documentación y el diseño convencional, los proyectos ágiles dependen críticamente de las personas. La substitución de los clientes que actúan como interlocutores ante los desarrolladores puede provocar cambios importantes en el desarrollo de los proyectos, dependiendo de los intereses de los nuevos interlocutores (esto no ocurriría si se hubieran consensuado antes los requisitos mínimos exigibles). El proyecto C3 fracasó, entre otros motivos, porque se marchó la persona clave que representaba los intereses de Daimler-Chrysler. Por otro lado, la desaparición e incorporación de personas en el equipo de trabajo puede dirigir el proyecto a rumbos inciertos.

La fuerte dependencia de las personas puede ocasionar problemas cuando éstas no son tan diligentes como cabría esperar: un programador que deje de refactorizar o que no haga que su código pase las pruebas al cien por ciento, puede poner en dificultades el proyecto.

5) DISCREPANCIAS ENTRE LAS PRÁCTICAS ÁGILES Y EL MODELO ACTUAL DE CONTROL DE CALIDAD

Compaginar estas prácticas con la documentación necesaria para certificar un sistema con la norma ISO 9001 resulta difícil. Resulta significativo que Scott W. Ambler (creador de Agile Modeling) reconociera, en el turno de preguntas de la conferencia Agile Modeling and Software Quality (12ª International Conference on Software Quality, 2002), que no podía afirmar nada acerca de si "*XP u otros métodos ágiles tenían en cuenta la consistencia, la seguridad, la integridad o la privacidad según reglas o regulaciones nacionales o internacionales*".

6) FALTA DE PROCESOS DE REVISIÓN DEL CÓDIGO

Con métodos como el PSP o el TSP se han conseguido reducciones de errores que oscilan entre el 60 y el 80%. La programación en parejas tiene resultados del 20-40%, que no es mucho frente al 10-25% de un programador convencional.

7) FALTA DE REUSABILIDAD

La regla YAGNI (No vas a necesitarlo) y la falta de documentación convencional hacen difícil que pueda reutilizarse el código ágil.

8) SOBRECOSTOS Y RETRASOS DERIVADOS DE LA REFACTORIZACIÓN CONTINUA

Para un sistema de ciertas proporciones, los costes y retrasos derivados de la refactorización no pueden despreciarse.

---

[33] PSP y TSP son versiones reducidas de CMM realizadas por el SEI, Software Engineering Institute, http://www.sei.cmu.edu/tsp/main.html

9) RESTRICCIONES EN CUANTO A TAMAÑO DE LOS PROYECTOS ABORDABLES. OLVIDO DE CIERTAS CARACTERÍSTICAS DE LOS SISTEMAS
Por ejemplo: eficacia, escalabilidad, seguridad y facilidad de mantenimiento. Ninguna de estas propiedades puede conseguirse sin un diseño explícito e inicial.

10) RIGIDEZ
Algunos métodos ágiles son muy rígidos: deben cumplirse muchas reglas de una forma estricta para garantizar el éxito del proyecto. XP es un buen ejemplo de método ágil que fomenta "la diversión de los programadores", pero que exige en realidad mucho esfuerzo, concentración y orden.

11) ABRAZAR EL CAMBIO
Los métodos ágiles abrazan el cambio, incluso lo fomentan. Esta actitud puede ser peligrosa: abrazar el cambio a todas horas implica cambiar constantemente el código, y no siempre se puede actuar así. Los modelos de datos son "pesados" y no pueden cambiarse así como así sólo porque el cliente quiera incorporar más funciones al sistema. Las interfaces de los sistemas o de sus componentes tampoco pueden cambiarse alegremente, aun cuando sea para mejorarlas, en aplicaciones que exceden el alcance de una empresa u organismo.

Responder exageradamente al cambio ha sido la fuente de muchos desastres en el desarrollo de software. El más famoso es el sobrecosto de 3 millones de dólares del *US Federal Aviation Administration's Advanced Automation System*, desarrollado para el control del tráfico aéreo en los Estados Unidos.

12) PROBLEMAS DERIVADOS DEL FRACASO DE LOS PROYECTOS ÁGILES
Estos problemas jamás se describen en los métodos ágiles, si bien deberían tenerse en cuenta. Si un proyecto ágil fracasa, no hay documentación o hay muy poca; lo mismo ocurre con el diseño. La comprensión del sistema se queda en las mentes de los desarrolladores. Quizá ahí esté bien cuidada y preservada para las generaciones futuras, pero puede no satisfacer a los promotores del proyecto.

4.8. Limitaciones de los métodos ágiles

Desde una perspectiva más metodológica, cuando los proyectos están relacionados con las necesidades empresariales y la interoperalidad del software, es habitual usar también normas como ENV 12204 (*Constructs for Enterprise Modeling*), ENV 13550 (*Enterprise Model Execution and Integration Services)*, ISO 14258 (*Concepts and rules for enterprise models*) e ISO 15704 (*Requirements for enterprise reference architecture and methodologies*). Para revisar el código, aparte de las pruebas, las técnicas extraídas de métodos como el TSP (*Team Software Process*) y el PSP (*Personal Software Process*) gozan de buena aceptación a la hora de detectar fallos en el código.

En el artículo *New Directions on Agile Methods: A Comparative Analysis* (*25ª International Conference on Sofware Engineering, 2003*) de Pekka Abrahamsson et al., se concluye que no hay evidencia empírica sólida e inequívoca de que los métodos ágiles sean útiles o beneficiosos. En el mejor de los casos no se puede hablar más que de indicios de su eficacia frente a los métodos tradicionales. La mayoría de las reglas ágiles se basa en experiencias subjetivas y en reglas prácticas, difíciles de extender a situaciones generales. Casi toda la bibliografía se basa en experiencias concretas, descritas por practicantes de las metodologías o por sus creadores.

A esto, algunos añaden el efecto Hawthorne, bien conocido en psicología industrial y que dice así: *"Cualquier cambio del entorno de trabajo de los trabajadores aumenta su productividad, independientemente del cambio concreto"*.

Gerold Keefer, de AVOCA GmbH, ha publicado estudios respecto XP y el desarrollo fiable de software. Algunos de los hechos tenidos en cuenta son:

- no sólo el primero, sino también el segundo proyecto de referencia de XP fueron cancelados.
- la viabilidad de la programación orientada por pruebas, cuyo proceso puede consumir hasta el 30% o 40% de los recursos de un proyecto.
- problemas de escalabilidad, altos costos y pobres resultados de la programación en parejas.
- las malas prácticas resultantes de la negación a documentar.
- al retorno a la "*programación de garaje*".
- no hay proyectos con alcances, precios y fechas fijas, ni requisitos no funcionales como rendimiento y seguridad, ni (a pesar de su insistencia en reutilización y patrones) sobre integración de componentes listos para usar (COTS).
- Tampoco le convence la premisa de "*hacer la cosa más simple que pueda funcionar*"; prefiere la postura de Einstein: "*Que sea lo más simple posible, pero no más simple que eso*".

Las propias prácticas de los métodos ágiles limitan o descartan su uso en algunos proyectos. A continuación se detallan algunos casos donde no convendría usar métodos ágiles (al menos en su formulación actual).

A) APLICACIONES DISTRIBUIDAS

Las pruebas unitarias son complicadas de aplicar entre componentes. Sería necesario construir una arquitectura de pruebas para probar directamente los componentes, que podría ser tan complicada como el sistema que se desea construir. Por otro lado, el entorno de ejecución de las pruebas debería ejecutarse en el entorno (arquitecturas cliente-servidor, por ejemplo) para poder comprobar por separado los componentes. En Java, eso implicaría que el entorno de pruebas se construyera con servlets o páginas JSP.

B) APLICACIONES BASADAS FUNDAMENTALMENTE EN INTERFACES GRÁFICAS DE USUARIO

No es fácil aplicar pruebas unitarias a las interfaces gráficas. Aun cuando se consiga, eso no asegura que sean cómodas para los usuarios.

C) APLICACIONES QUE REQUIEREN SEGUIR UN DISEÑO ESTRICTO

Por ejemplo: sistemas operativos, software de telecomunicaciones, etc.

D) BIBLIOTECAS DE CLASES

El "carpe diem" que propugna XP no es adecuado para construir diseñar bibliotecas de clases. Es casi imposible conseguir bibliotecas reutilizables escribiendo a trozos el código de las clases.

E) APLICACIONES QUE REQUIEREN UNA DOCUMENTACIÓN EXHAUSTIVA

Por ejemplo: sistemas militares, médicos o industriales.

F) APLICACIONES CON CÓDIGO HEREDADO

Habría que reescribir todo el código heredado siguiendo los principios ágiles.

G) PROYECTOS MUY GRANDES
En estos casos, la comunicación cara a cara entre los miembros del equipo es difícil de conseguir o es imposible.

H) PROYECTOS ESCRITOS EN LENGUAJES NO ORIENTADOS A OBJETOS
Lenguajes como C, Pascal, Cobol o Fortran hacen imposible técnicas como la refactorización. No hay que olvidar que las prácticas de XP se desarrollaron con Smalltalk. Un lenguaje híbrido como C++ también plantea problemas para las metodologías ágiles: el código más simple que ejecuta una tarea suele ser de difícil lectura directa por el uso de punteros.

I) APLICACIONES DONDE LA ESCALABILIDAD O LA EFICACIA SEAN IMPORTANTES
La escalabilidad o la eficacia no son características que se puedan añadir durante el proceso de desarrollo de software o que puedan obtenerse refactorizando: deben considerarse desde un principio.

J) APLICACIONES DESTINADAS AL MERCADO
Es difícil lanzar un producto de software comercial sin haber decidido detenidamente qué funciones se espera del producto y a qué segmento del mercado se dirige.

4.9. Líneas de futuro

Existen diferentes líneas para continuar este proyecto. Las principales podrían ser:

a) Profundizar en cualquiera de los catorce métodos, ya que aquí se ha pretendido justificar por qué se hace necesario aplicar ideas ágiles y hacer una introducción breve pero completa de todos ellos, no entrar de forma exhaustiva en un método en concreto. Algunos de ellos tienen mucha literatura dedicada y tienen fundamentos suficientes como para dedicarles un proyecto entero.

b) Realizar un proyecto de desarrollo de software real, en un pequeño grupo, utilizando un método ágil en particular, o algunas de las técnicas ágiles que mejor se adapten al proyecto. Lo ideal sería hacer el proyecto de las dos maneras, tradicional como el método en cascada y la forma ágil, para poder comparar los resultados finales de calidad, tiempo, satisfacción del cliente, etc. Como esto supondría un trabajo extra, también sería posible comparar los resultados del proyecto que se hiciera de forma ágil, con los resultados de proyectos anteriores que hicieron el mismo grupo de desarrolladores. Ellos sabrán sopesar si, a pesar de que cada proyecto es diferente, la cantidad de tiempo que han perdido reescribiendo código que habían escrito pero luego resultó que no era lo que el cliente quería o no cumplía los requisitos. Eso nos puede dar una idea de la mejora.

c) Analizar alguno de los programas utilizados para el desarrollo ágil. Este software permite muchas funciones avanzadas como la compilación automática, es decir, cuando se trabaja en un grupo geográficamente distribuido (incluso a nivel mundial como pasa con el software Open Source tipo Linux) cada vez que algún componente del grupo de desarrolladores pone en el servidor (repositorio) una nueva versión de un archivo, el sistema automáticamente lo recompila todo y genera los errores que ha provocado. Esta información se comparte con todos los demás desarrolladores y se

genera automáticamente. Existen frameworks, como Microsoft Solutions Framework, MSF, que incluyen muchas más herramientas como éstas y un análisis de todas las facilidades que supone también sería suficiente para dedicarle un proyecto entero.

d) Dado que el manifiesto ágil y los 12 principios ágiles (que comparten todos los métodos ágiles) no son estrictos sino que dejan al propio autor que realice el método como quiera basándose en esas ideas, es posible, como ya se ha visto, que aparezcan nuevas mezclas de métodos, o combinaciones de métodos para crear otro nuevo. Podrían analizarse estos nuevos métodos o combinaciones.

e) Una posible manera de ver las mejoras reales gracias a los métodos ágiles, sería conseguir información de cualquier proyecto de gran envergadura que se hubiera realizado aplicando algún método ágil o algunas prácticas ágiles y que se pudiera comparar con proyectos anteriores (realizados de forma "pesada") del mismo equipo de desarrolladores. Las empresas guardan esta información de forma muy celosa, ya que sería dar información a la competencia sobre su funcionamiento, qué usan para hacer mejores productos, etc. En el anexo "Xp + Scrum, ejemplo de aplicación" se ve un ejemplo de este tipo, a partir de un proyecto de la empresa SirsiDynix.

CAPÍTULO

5

CONTENIDO

A. Acrónimos y glosario

B. The Agile Manifesto, The 12 principles & The Agile Project Manifesto

C. CASE

D. CMM

E. UML

F. The CHAOS Report, 1994 (Standish Group)

G. Casos de uso

H. RAD

I. EUP

J. Scrum + XP, caso práctico

K. Dilbert y los métodos ágiles

L. Epigramas sobre la programación

M. Resumen en transparencias

ANEXOS

"En teoría, no hay diferencia entre teoría y práctica. En la práctica, la hay."

John McMillan

5. ANEXOS

Anexo A. Acrónimos y glosario

80/20: El 80% del valor proviene del 20% de las características o funcionalidades que no se hacen funcionar nunca.

CAÓRDICO: Traducción de *chaordic*, una combinación de caos y orden, palabra inventada por Dee Hock, fundador y anterior CEO de Visa International.

CASO DE USO (USE CASE): Es una pieza de funcionalidad bien delimitada y reutilizable que da valor a N Actores que interactúan con el sistema.

CONFIGURATION ITEM: Un conjunto de hardware, software o ambos que se diseña para *configuration management* y se trata como una sola entidad en el proceso de gestión de configuración. (IEEE-STD-610). Ejemplos de *configuration items* son: Un fichero de código fuente; un programa con varios archivos fuente que incluya otros; un incremento completo consistente en varios programas y sus respectivas especificaciones; un sistema finalizado formado por software operativo, documentación de usuario y especificaciones.

COTS: COMMERCIAL OFF-THE-SHELF: Productos de software o hardware que ya están disponibles comercialmente para vender, comprar, adquirir licencias, etc.

CRC, TARJETAS. Las tarjetas *Clase-Responsabilidad-Colaborador* son simples tarjetas de papel, de 4x6 o 3x5 pulgadas, y es una técnica que reemplaza a los diagramas en la representación de modelos.

ERD – ENTITY-RELATION DIAGRAM: Diagrama o dibujo esquemático que muestra todas las tablas y columnas que existen en una base de datos relacional y usa líneas y notaciones para indicar relaciones entre ellas.

ERP – ENTERPRISE RESOURCE PLANNING: Los sistemas de planificación de recursos de la empresa, son sistemas de gestión de información que integran y automatizan muchas de las prácticas asociadas con los aspectos operativos o productivos de una empresa. Están formados por diferentes partes integradas en una única aplicación: producción, ventas, compras, logística, contabilidad, gestión de proyectos, sistema de información geográfica, inventarios y control de almacenes, pedidos, nóminas, etc.

EXCELENCIA TÉCNICA: Decisiones adecuadas y oportunidad en la toma de las mismas, habilidad en el manejo de algunas técnicas y buen juicio para proceder. En otras palabras, "hacer lo correcto, correctamente".

EXPERIENCIA, NIVEL DE: En los métodos ágiles se definen tres niveles de experiencia. Un programador de **Nivel 1** es capaz de "seguir los procedimientos"; uno de **Nivel 2** es capaz de "apartarse de los procedimientos específicos y encontrar otros distintos" y uno de **Nivel 3** es capaz de "manejar con fluidez, mezclar e inventar procedimientos".

FRAMEWORK: Es un armazón, una estructura de apoyo para organizar y puede desarrollarse software utilizándola. Puede incluir programas de apoyo, librerías de código, un lenguaje de

script, u otro software para ayudar a desarrollar y unir los diferentes componentes de un proyecto de software. Un ejemplo, el .NET de Microsoft.

GESTIÓN DE CONFIGURACIÓN: Una disciplina que aplica la dirección técnica y administrativa y la vigilancia para identificar y documentar las características físicas y funcionales de un *configuration item*, controla los cambios de esos características, registra e informa cómo se procesa el cambio y el estado de la ejecución, y verifica la que se acaten los requisitos especificados. (IEEE-STD-610)

HOLÍSTICA: Es la idea de que todas las propiedades de un sistema no pueden ser determinadas o explicadas como la suma de sus componentes. El sistema completo se comporta de un modo distinto que la suma de sus partes.

HORA EXTREMA: Inventada por Peter Merel para introducir a la gente en el método XP en 60 minutos y proporciona pautas canónicas para utilizar XP.

IDE: Entorno Integrado de Desarrollo, como MS Visual Studio, Netbeans, Eclipse...

IV&V: Verificación y validación independiente, realizada por externos del equipo.

JAD (JOINT APPLICATION DEVELOPMENT): Una sesión de JAD es esencialmente un taller donde diseñadores y representantes del cliente discuten las características del producto.

LOD: LAW OF DEMETER: "*Sólo habla con tus amigos inmediatos*" Una de las 70 practicas de la Programación pragmática y consejo "agíl" en general: evitar que los objetos o funciones utilicen muchos módulos para aislar más fácilmente un fallo.

MILESTONE: Dentro de la gestión de proyectos, un *milestone* es un elemento que marca la finalización de una fase, a menudo acompañado de algún signo a alto nivel como la finalización de un *deliverable* o una reunión.

MOSCOW: Método para establecer prioridades. Significa: M - MUST, Debe tener esto. S - SHOULD Debería tener esto si fuera posible. C - COULD Podría tener esto si no afecta a ninguna función. W - WON'T No tendrá esto ahora, pero quizá sí en un futuro.

MOCK, OBJETO: Los objetos *mock* se usan para comprobar el comportamiento de otros objetos. Los podemos imaginar como implementaciones de una clase (o interfaz) que simulan el comportamiento real que tendría una verdadera implementación de ésta. No deben confundirse con los objetos *stub* que sólo proporcionan la implementación de una clase mediante la implementación de su interfaz (conjunto de métodos públicos de la clase).

MVC – MODELO VISTA CONTROLADOR: Es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. El patrón MVC se ve frecuentemente en aplicaciones web, donde la vista es la página HTML y el código que provee de datos dinámicos a la página, el controlador es el Sistema de Gestión de Base de Datos y el modelo es el modelo de datos.

PMBOK (PROJECT MANAGEMENT BODY OF KNOWLEDGE): La Guía del PMBOK es un estándar en la gestión de proyectos desarrollado por el Project Management Institute, PMI. Su

intención es documentar y estandarizar información y prácticas generalmente aceptadas en la gestión de proyectos.

PRUEBA (TEST) UNITARIA O DE UNIDAD: Verifica una clase a nivel de método, o un pequeño conjunto de clases y son responsabilidad del programador.

PRUEBA (TEST) FUNCIONAL O DE ACEPTACIÓN: Verifica todo el sistema, o una gran parte, y son propuestas por el cliente.

QA (QUALITY ASSESSMENT O QUALITY ASSURANCE): El Control de calidad está formado por la evaluación, y mediciones de los procesos de diseño, desarrollo, producción, instalación, servicio y documentación.

RAD: **RAPID APPLICATIONS DEVELOPMENT:** Método iterativo de desarrollo. El libro de referencia es el de Steve McConnell (Microsoft) de 1996.

REFACTORIZACIÓN: Un cambio hecho a la estructura interna del software para hacerlo más fácil de entender y más barato de modificar, sin cambiar su comportamiento observable.

REQUISITOS NO FUNCIONALES: Se refieren a cuestiones como usabilidad, seguridad y rendimiento.

SLA: Service Level Agreement. Es un acuerdo de nivel de servicio por el que una compañía se compromete a prestar un servicio a otra bajo determinadas condiciones y con un nivel de calidad y prestaciones mínimas.

SMALLTALK: Lenguaje de programación considerado el primero en utilizar el paradigma orientado a objetos. En Smalltalk absolutamente todo es un objeto, incluso el propio entorno Smalltalk. Se caracteriza por su orientación a objetos pura, tipado dinámico, herencia simple, interactúa entre objetos mediante el envío de mensajes y posee un recolector de basura. Es multiplataforma y puede compilar en tiempo de ejecución o interpretado. Smalltalk tuvo gran influencia en la creación de otros lenguajes como Java o Ruby.

SMOKE TEST: test simple para asegurar que el software funciona hasta un punto mínimo. Proviene de prácticas de hardware, donde se dejaba el circuito a ver si se quemaba.

SPAGHETTI CODE: Código con una estructura de control de flujo compleja e incomprensible.

SPIKE: Una púa es un trozo desechable de código, usado para comprender cómo podría resolverse un problema de programación, para saber si se está en la dirección correcta. Es diferente del esqueleto ambulante de *Crystal*. Es la versión ágil de la idea de prototipo. Se lo llama así porque "va de punta a punta, pero es muy fino".

STAKEHOLDER: Cada uno de los participantes en un proyecto como el usuario final, el contratista, el comprador, etc.

TESTEAR: Ver Verificación y Validación. Los métodos ágiles usan este término para referirse a cualquier actividad V&V, no sólo a las pruebas manuales o automáticas para cómo funciona el programa.

TEST DE REGRESIÓN: Test realizado después de realizar un cambio (arreglo o mejora) para asegurarse de que el sistema no ha sufrido un retroceso (la funcionalidad que antes funcionaba debe seguir funcionando).

TEST-FIRST: Cuando se programa por parejas, antes de escribir código, se escriben los tests automatizados para verificarlo.

THREEE EXTREMOES: Kent Beck, Ron Reffries y Ward Cunningham; impulsores de Extreme Programming.

TIME-BOXING: Forma de gestionar proyectos que fija fechas de inicio y fin para el proyecto o sus iteraciones y permite cambiar las funcionalidades previstas para cumplir con el calendario.

UML: Unified Modelling Language. Notación estándar que permite modelar visualmente todos los procesos implicados en el análisis, diseño y desarrollo orientado a objetos de un sistema.

V&V: Validación y verificación.

VALIDACIÓN: Comprobar (testeando o repasando) para asegurarse que el sistema tal y como está construido o especificado, cumple las necesidades.

VERIFICACIÓN: Comprobar (testeando o repasando) para asegurarse de que el sistema se ha construido o especificado de forma precisa y con integridad.

WIKI: Introducida por Ward Cunningham, *wiki* significa rápido en hawaiano. Una wiki es una página web que facilita que todo el mundo contribuya a ampliarla y a crear enlaces entre sus páginas. En general, Wikis son un tipo de software que ayuda a comunicarse en línea, donde editar es tan fácil como leer, siendo la herramienta perfecta para colaborar en grupo.

WORKPRODUCT: Cualquier documento que requiere utilizar un método. Se suele traducir al castellano como artefacto.

Anexo B. The Agile Manifesto, Principles & Agile Project Manifesto

THE AGILE MANIFESTO

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools.
Working software over comprehensive documentation.
Customer collaboration over contract negotiation.
Responding to change over following a plan.

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Principles behind the Agile Manifesto

1) Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2) Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3) Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4) Business people and developers must work together daily throughout the project.
5) Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6) The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7) Working software is the primary measure of progress.
8) Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9) Continuous attention to technical excellence and good design enhances agility.
10) Simplicity –the art of maximizing the amount of work not done– is essential.
11) The best architectures, requirements, and designs emerge from self-organizing teams.
12) At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

AGILE PROJECT MANIFESTO

Declaration of Interdependence

Agile and adaptive approaches for linking people, projects and value.

We are a community of project leaders that are highly successful at delivering results. To achieve these results:

- We **increase return on investment** by making continuous flow of value our focus.
- We **deliver reliable results** by engaging customers in frequent interactions and shared ownership.
- We **expect uncertainty** and manage for it through iterations, anticipation, and adaptation.
- We **unleash creativity and innovation** by recognizing that individuals are the ultimate source of value, and creating an environment where they can make a difference.
- We **boost performance** through group accountability for results and shared responsibility for team effectiveness.
- We **improve effectiveness and reliability** through situationally specific strategies, processes and practices.

[©2005 David Anderson, Sanjiv Augustine, Christopher Avery, Alistair Cockburn, Mike Cohn, Doug DeCarlo, Donna Fitzgerald, Jim Highsmith, Ole Jepsen, Lowell Lindstrom, Todd Little, Kent McDonald, Pollyanna Pixton, Preston Smith and Robert Wysocki.]

The title "Declaration of Interdependence" has multiple meanings. It means that project team members are part of an interdependent whole and not a group of unconnected individuals. It means that project teams, their customers, and their stakeholders are also interdependent. Project teams who do not recognize this interdependence will rarely be successful.

These values also form an interdependent set. While each is important independent of the others, the six form a system of values that provides a modern view of managing projects, particularly the complex, uncertain ones. The six statements -- value, uncertainty, customers, individuals, teams, and context (situationally specific) -- define an inseparable whole. For example: It's hard to deliver value without a customer who values something. It's hard to have viable teams without recognizing individual contributions. It's hard to manage uncertainty without applying situational specific strategies.

Each of the value statements has a distinct form -- why the item is important precedes the description of the value. So, "increasing return on investment" is why focusing on continuous flow of value is important. The value statements emphasize the importance of delivering reliable (not the same as repeatable) results, managing uncertainty, unleashing creativity and innovation, boosting performance, and improving effectiveness.

Each of the means statements conveys what this group thinks are the most important aspects of modern project management, and they also attempt to differentiate an agile-adaptive style of project management. For example, in the last value statement, the phrase "situationally specific strategies, processes, and practices," indicates that these items should not be overly standardized and static, but dynamic to fit the needs of projects and teams. Other styles of project management are more prone to standardization and prescriptive processes.

If you are interested in more information on this organization, please visit our web site www.apln.org or follow the discussion on www.groups.yahoo.com/group/agileprojectmanagement

Jim Highsmith, 17 February 2005.

Anexo C. CASE - Computer Aided Software Engineering

CASE es una filosofía que busca la mejor comprensión de los modelos de empresa, sus actividades y el desarrollo de los sistemas de información. Esta filosofía involucra además el uso de programas que permiten:

- Construir los modelos que describen la empresa,
- Describir el medio en el que se realizan las actividades,
- Llevar a cabo la planificación,
- El desarrollo del Sistema Informático, desde la planificación, pasando por el análisis y diseño de sistemas, hasta la generación del código de los programas y la documentación.

CASE es la creación de sistemas software utilizando técnicas de diseño y metodologías de desarrollo bien definidas, soportadas por herramientas automatizadas operativas en el ordenador.

OBJETIVOS DEL CASE

1) Aumentar la productividad de las áreas de desarrollo y mantenimiento de los sistemas informáticos.
2) Mejorar la calidad del software desarrollado.
3) Reducir tiempos y costes de desarrollo y mantenimiento del software.
4) Mejorar la gestión y dominio sobre el proyecto en cuanto a su planificación, ejecución y control.
5) Mejorar el archivo de datos (*enciclopedia*) de conocimientos (*know-how*) y sus facilidades de uso, reduciendo la dependencia de analistas y programadores.
6) Automatizar :
   - El desarrollo del software,
   - la documentación,
   - la generación del código,
   - el chequeo de errores, y
   - la gestión del proyecto.

7) Permitir:
   - La reutilización (reuso) del software,
   - la portabilidad del software, y
   - la estandarización de la documentación.

8) Integrar las fases de desarrollo (ingeniería del software) con las herramientas CASE
9) Facilitar la utilización de las distintas metodologías que desarrollan la propia ingeniería del software.

ENCICLOPEDIA (REPOSITORY)

En el contexto CASE se entiende por enciclopedia a la base de datos que contiene todas las informaciones relacionadas con las especificaciones, análisis y diseño del software. En está base de datos se incluyen las informaciones de:

1) **DATOS**: Elementos atributos (campos), asociaciones (relaciones), entidades (registros), almacenes de datos, estructuras, etc.
2) **PROCESOS**: Procesos, Funciones, módulos, etc.

3) **GRÁFICOS**: DFD (Diagrama de flujo de datos), DER (Diagrama Entidad Relación) DFD (Diagrama de Descomposición Funcional), ED (Diagrama de Estructura), Diagrama de Clases, etc.
4) **REGLAS**: de gestión, de métodos, etc.

CLASIFICACIÓN DE LAS HERRAMIENTAS CASE

CASE es una combinación de herramientas software (aplicaciones) y de metodologías de desarrollo:

- Las herramientas permiten automatizar el proceso de desarrollo del software.
- Las metodologías definen los procesos a automatizar.

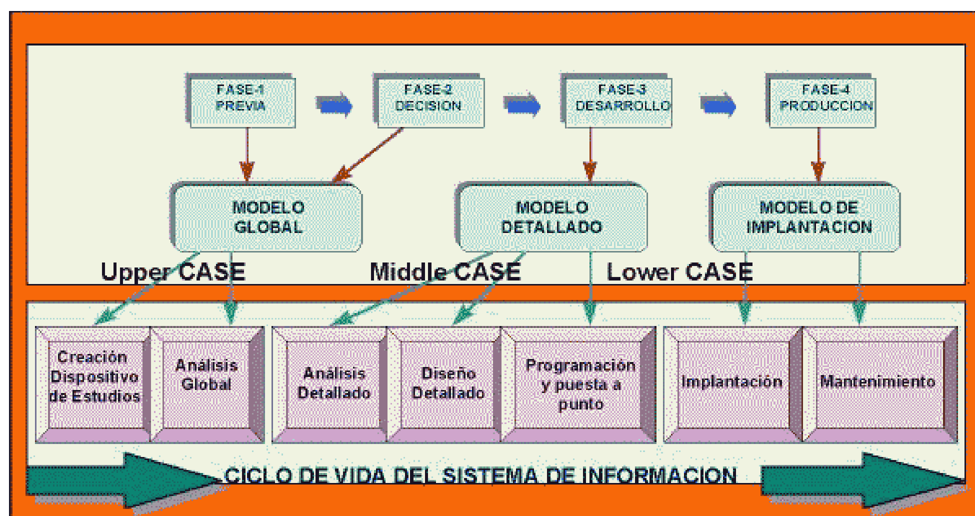
Una primera clasificación del CASE es considerar su amplitud:

TOOLKIT: es una colección de herramientas integradas que permiten automatizar un conjunto de tareas de algunas de las fases del ciclo de vida del sistema informático: Planificación estratégica, Análisis, Diseño, Generación de programas.

WORKBENCH: Son conjuntos integrados de herramientas que dan soporte a la automatización del proceso completo de desarrollo del sistema informático. Permiten cubrir el ciclo de vida completo. El producto final que aportan es un sistema en código ejecutable y su documentación. Una segunda clasificación teniendo en cuenta las fases (y/o tareas) del ciclo de vida que automatizan:

- UPPER CASE: Planificación estratégica, Requerimientos de Desarrollo Funcional de Planes Corporativos.
- MIDDLE CASE: Análisis y Diseño.
- LOWER CASE: Generación de código, test e implantación.

CASE EN EL "CICLO DE VIDA DEL SISTEMA" DURANTE LA ESTAPA ANÁLISIS Y DISEÑO.



CASE se considera por las Direcciones de Informática como un amplio espectro de tecnologías que permiten mejoras radicales en la productividad y en la calidad en todos los aspectos del desarrollo de aplicaciones moderno.

A esta realidad con la que hoy nos encontramos, debemos añadir dos aspectos de máximo interés para las organizaciones:

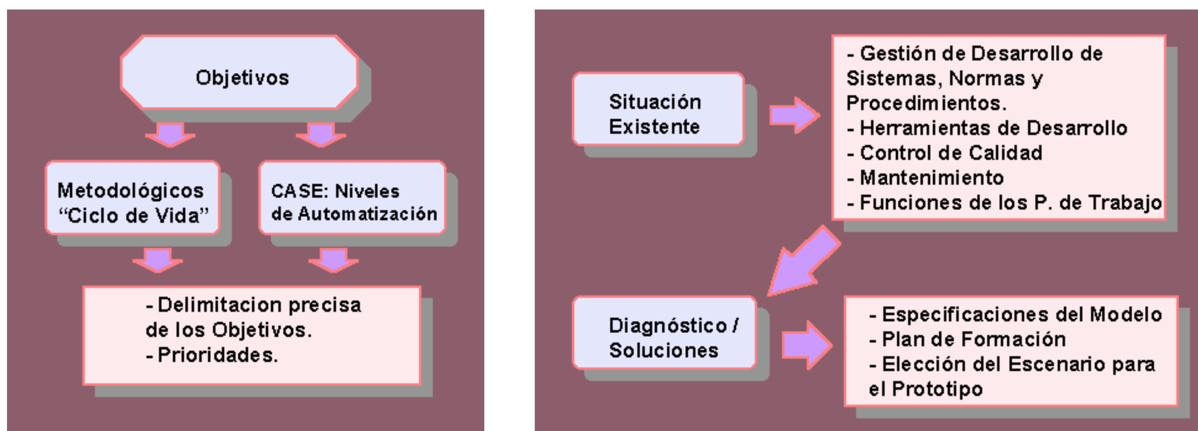
- Las nuevas capacidades de importación / exportación de datos entre las distintas herramientas.
- El descenso permanente del precio de las licencias.

ETAPAS EN UN PROYECTO DE INTRODUCCION DEL CASE

Para llevar a cabo con éxito el proyecto de introducción del CASE en el Área de Desarrollo, recomendamos que como mínimo se tengan en cuenta cinco etapas:

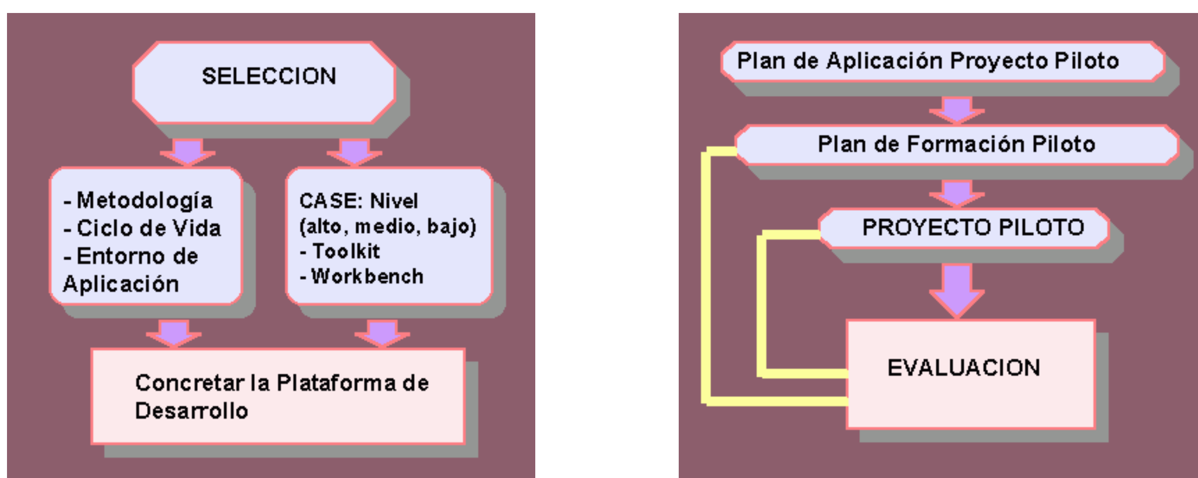
ETAPA 1 (izquierda): Descripción de Objetivos - Grupo de Trabajo - Planificación provisional del proyecto.

ETAPA 2 (derecha): Análisis del Área de Desarrollo

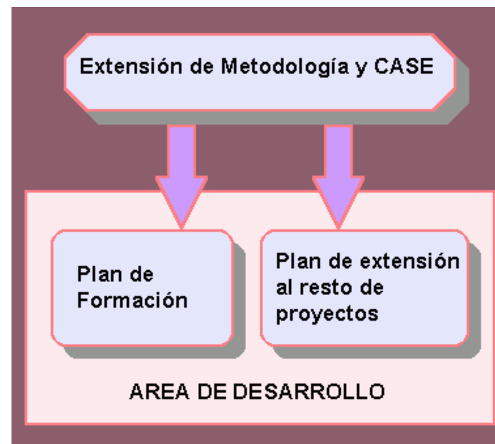


ETAPA 3 (izquierda): Selección de Metodología y Herramientas CASE

ETAPA 4 (derecha): Aplicación en Escenarios y Evaluación (es muy importante que el proyecto de evaluación NO sea crítico y su tamaño pequeño)



ETAPA 5: Extensión de la Metodología y CASE en la Organización



CAUSAS POR LAS QUE FRACASAN ALGUNOS PROYECTOS DE INTRODUCCIÓN DEL CASE

No siempre han tenido éxito los proyectos de introducción del CASE. Dado que los nuevos programas de formación de analistas ya tienen en cuenta tanto la Metodología como el uso y prácticas con sistemas CASE, están permitiendo reducir los riegos de fracaso.

No obstante en muchas organizaciones actuales no se dispone de analistas formados, ni de experiencias CASE. Son estas organizaciones las que deben poner especial atención en las causas más frecuentes por las que puede fracasar el proyecto:

- No se tienen en cuenta las tres primeras etapas,
- No se concreta ninguna Metodología,
- El proyecto de evaluación es demasiado ambicioso ó crítico,
- En la etapa quinta no se lleva a cabo la Formación que se precisa,
- Los Usuarios (Área de Desarrollo), no están motivados.

Un proyecto de introducción de CASE es siempre un proyecto estratégico para el Área de Desarrollo y como tal, "no tiene vuelta atrás". Cuando la decisión ya ha sido tomada, siga con pasos firmes todas las etapas, teniendo muy en cuenta que los tiempos y esfuerzos para cubrirlas dependerán de las personas que integran el Área de Desarrollo.
En organizaciones muy preparadas, su introducción ha requerido un año.

BIBLIOGRAFÍA:

http://ceds.nauta.es/Program/case.htm
http://www.revistaespacios.com/a00v21n01/32002101.html#uncaso
http://delta.cs.cinvestav.mx/~pmejia/softeng/trans.html
http://www.inei.gob.pe/cpi/bancopub/libfree/lib615/cap101.htm

Anexo D. CMM: Capability Maturity Model

RESUMEN:
Dentro de la competitividad actual, conseguir productos software excelentes a buen precio en márgenes de tiempo breves es el sueño de miles de empresas. Esto se puede conseguir concentrando esfuerzos en torno a dos pilares fundamentales: Las personas por un lado, y los métodos y procedimientos por otro. El Modelo de Madurez de Capacidad del Software (CMM) hace hincapié en la mejora del proceso de software en base a los procedimientos internos y sin descuidar a las personas.

INTRODUCCIÓN

En un entorno donde lo que prevalece es la maximización de los beneficios se busca como consecuencia inmediata la máxima eficiencia y efectividad en las tareas que generan el beneficio. Al mismo tiempo, la globalización a todos sus efectos, el nuevo orden social, las nuevas formas de hacer negocios, las nuevas necesidades, los nuevos descubrimientos en todos los ámbitos, las nuevas relaciones sociales y, en definitiva, la propia evolución de la sociedad, elevan día a día la complejidad no sólo de los procesos económicos sino también de las relaciones personales.

Una de las formas más importantes de enfrentarse a estos retos, es a través de las distintas herramientas informáticas eficaces y eficientes que facilitan la ejecución y control del trabajo diario. Sin embargo, la historia de la ingeniería del software o de la producción de software está repleta de grandes fracasos y decepciones. Proyectos de miles de millones de dólares que no han cumplido sus objetivos y a menor nivel pero de forma mucho más abundante, millones de usuarios decepcionados con el software que manejan como principal elemento de su trabajo. Los problemas más frecuentes en la producción de software son:

1) La entrega del material fuera del plazo estipulado.
2) El material no cumple los requisitos estipulados, ya sea en eficiencia o funcionalidad.

Todos los problemas reflejados en el párrafo anterior originan pérdidas económicas en el mejor de los casos, ya que si es un sistema crítico podemos llegar a hablar incluso de pérdidas de vidas humanas.

LA MEJORA DEL PROCESO DE GESTIÓN DEL SOFTWARE

Para conseguir tener un proceso de producción de software sin fallos, adecuado a las necesidades estipuladas en un principio y entregado a tiempo, está claro que la producción de software debe convertirse en un proceso disciplinado y aceptado por todos.

Hay varias razones por las que puede fallar el proceso de software; las principales son:
1) El personal no se involucra lo suficiente en el control de calidad del trabajo.
2) La alta dirección no ha adquirido conciencia de la importancia de un buen proceso de software para su compañía. La principal consecuencia de esto es que el proceso de software no tiene los recursos adecuados ya sea en forma de tiempo, dinero, tecnología, personal y su formación, etc.
3) Las prácticas establecidas no son las adecuadas.

Hasta ahora hemos empleado el termino "*proceso de software*", pero ¿qué queremos decir con este término?: "Un proceso es un conjunto de pasos definidos para lograr una tarea", mientras

que *"un proceso definido es aquel que está escrito a tal detalle que permite que los ingenieros lo usen constantemente"*. Los procesos definidos ayudan a la planificación y desarrollo de un trabajo. El proceso que establezcamos debe ser flexible y debe facilitar el cambio y la innovación. Al mismo tiempo, el proceso debe poder aprenderse.

Aquí es donde entra el CMM o "Modelo de Madurez de Capacidad del Software". El CMM está destinado a la evaluación y mejora de procesos. Se debe evaluar a la organización para conocerla ya que sin conocerla no se puede mejorar. *El propósito de CMM es guiar a las organizaciones en la selección de estrategias de mejora determinando la madurez del proceso actual e identificando los puntos importantes que se deben estudiar y trabajar para mejorar tanto el proceso como la calidad del software*. Dicho en palabras de Dymon [Dymon 1997] ayudar a las personas a identificar aquellas actividades críticas que indican la capacidad para realizar de la organización.

Hay dos razones fundamentales para creer en la efectividad de este modelo:

1) El modelo CMM está construido en base a prácticas reales.
2) Cada nueva (y correcta) implementación del CMM es un nuevo éxito.

EL MODELO DE MADUREZ DE CAPACIDAD DEL SOFTWARE (CMM)

Acabamos de ver una pequeña introducción al significado del CMM. También consideramos brevemente las ventajas de su empleo en una organización. Ahora bien, ¿tan bueno es el CMM que no tiene ningún inconveniente? Por supuesto que el CMM tiene inconvenientes, aunque mejor deberíamos llamarlo riesgos. El CMM puede ser mal interpretado y para evitarlo es conveniente que las personas que lo utilicen comprendan el modelo y sus implicaciones a la hora de aplicarlo a la organización.

Este modelo es fruto del trabajo de SEI (Software Engineering Institute) que desde 1986 centra sus esfuerzos en mejorar la práctica del proceso del Software. En 1991 consiguieron estabilizar la primera versión del CMM. Desde entonces este modelo se ha empleado en organizaciones tales como el Departamento de Defensa de los EE.UU., sedes que necesitaban controlar de manera exhaustiva el proceso de producción de software.

El CMM es una forma de comprender la propia gestión de procesos dentro de la organización. Es cierto que el CMM evalúa a la organización ya que para mejorar es preciso, antes, evaluar. Pero no podemos cometer el error de reducir el CMM a una mera lista de comprobaciones; CMM es mucho más que eso, es una "institucionalización" del proceso para construir software con el objetivo de conseguir una mejora continua.

A la hora de aplicar el CMM debemos tener claro una serie de aspectos sobre la organización:

1) El tamaño de la organización.
2) Su nivel cultural.
3) Las tecnologías que emplea.

Conociendo estos tres puntos podemos acometer el conocimiento de los objetivos de la organización. Posteriormente deberemos decidir cómo vamos a medir. El CMM establece 5 posibles niveles de madurez en los que puede encontrarse una organización:

a) Nivel 1: el más básico.
b) Nivel 2: el proceso repetible.
c) Nivel 3: el proceso definido.
d) Nivel 4: el proceso gestionado.

e) Nivel 5: el proceso de optimización.

Nadie puede obtener el significado exclusive del CMM. El CMM no aporta una medida absoluta, no existe un nivel de 2'5, todos los resultados deben ser interpretados, ya que el CMM es flexible para adaptarse en su utilización a las peculiaridades de cada organización. Debido a este punto, el conocimiento del Modelo por parte de quien lo aplica se hace aún más importante.

Nos queda ahora abordar en profundidad el proceso del CMM, teniendo en cuenta las siguientes definiciones [Paulk 1994]:

a) **Institucionalizar**: Edificar una infraestructura y una cultura que soporte los métodos, las prácticas y los procesos para que éstos sean la forma real de hacer negocios. Será fundamental conocer cuál es el grado de conocimientos de todos sus trabajadores, así como el esquema cultural y social en que se ubica la empresa.
b) **Proceso de Software**: Conjunto de actividades, métodos, prácticas y transformaciones para desarrollar y mantener software y productos asociados.
c) **Capacidad de un proceso**: Rango de resultados esperados que se pueden obtener tras seguir un proceso.
d) **Madurez de un proceso de software**: Es el punto hasta el que un determinado proceso está explícitamente definido, administrado, medido, controlado y ejecutado de manera efectiva.
e) **Nivel de madurez**: Plataforma bien definida desde la que podemos obtener un proceso maduro de software.
f) **Procedimiento documentado**: La actividad o procedimiento es un proceso rutinario y que ha sido codificado.

CARACTERÍSTICAS DEL CMM

Comenzaremos este apartado dando una definición formal al CMM [Dymon 1997]: El CMM es un modelo que describe cómo las prácticas de la ingeniería del software de una organización evolucionan bajo ciertas condiciones:

1) El trabajo se organiza y se trata como un proceso.
2) La evolución del proceso se gestiona sistemáticamente.

El CMM guarda cierta relación con los estándares de calidad como ISO 9001, en palabras de Dymon, *"este estándar es efectivo para proporcionar una base de una buena práctica por debajo de la cual una organización no debería descender"*. Por el contrario, el CMM es un estándar progresivo con una dimensión dinámica que conduce a una organización a mejorar continuamente sus prácticas actuales de software. ***Según los estudios realizados por el SEI una organización que se encuentre en un nivel de madurez 3 podría obtener sin problemas la certificación ISO 9001***. Pero una organización que posea una certificación ISO 9001 podría quedar ubicada en un nivel de madurez 2 o 3, dependiendo del caso. Para obtener mayores detalles en esta comparación se puede consultar en [Paulk 1994].

Detallemos ahora un poco más los distintos niveles de madurez:

- **Nivel 1**: Nivel **inicial**, el proceso de software es impredecible y poco controlado. Esto no significa que una organización no produzca buen software, sino que el coste

(financiero, humano, temporal, etc.) es demasiado alto tanto para los productores como para los usuarios.

- **Nivel 2**: Nivel **repetible**, en este nivel existe una disciplina básica en la gestión de procesos basada en la repetición de tareas aprendidas previamente. Ya hay una planificación en términos de coste, calendario y requisitos.
- **Nivel 3**: Nivel **definido**, el proceso es estándar y consistente, se conoce lo que hace que el proceso de software tenga éxito y se aplica a toda la organización.
- **Nivel 4:** Nivel **gestionado**, el proceso del nivel 3 es medido y controlado cuantitativamente, está implementado en toda la organización.
- **Nivel 5**: Nivel **optimizado**, existe una evolución continua en la optimización del proceso.

El CMM se centra en los tres principales aspectos que influyen en una organización:

a) **Las personas**: Se trata por disciplinas como el desarrollo organizativo, gestión de los RRHH y la Gestión de la Calidad Total (TQM).
b) **La tecnología**: La tecnología cambia a su propio ritmo a lo largo del tiempo, se puede adquirir.
c) **El proceso**. Pero, ¿cómo se gestiona el proceso y cómo se mejora? ¿Se puede comprar? La gestión del proceso se puede aprender e institucionalizar, aquí es donde entra el CMM.

La complejidad aparente del CMM se simplifica en cuatro conceptos base:

1) La evolución es posible pero lleva tiempo.
2) Hay etapas distinguibles en la madurez del proceso.
3) La evolución implica que algunas cosas deben ser aplicadas antes que otras.
4) La madurez disminuirá a menos que se mantenga. *"Los cambios duraderos requieren un esfuerzo constante"* [Dymon 1997].

Para cambiar el proceso del software debemos:

- Gestionar las influencias.
- Gestionar las mejoras sistemáticas.

El cambio puede empezar a aplicarse a través del **ciclo de Deming: Planificar, Hacer, Verificar y Actuar** [Deming 1982]. Adaptado a nuestra situación, Iniciar es acordar el motivo y la estrategia para el cambio. Diagnosticar es acordar qué cambiar, posteriormente debemos Establecer la infraestructura (equipos y planes), Actuar (llevar a cabo los planes) e Institucionalizar (capturar y reutilizar las lecciones aprendidas).

Volvemos aquí a insistir en algo de crucial importancia: la aplicación del modelo requiere el compromiso de la alta dirección ya que está claro que durante un tiempo ciertos recursos deberían desviarse de las actividades de generar ingresos y dedicarse a la mejora del proceso.

APLICANDO EL CMM

Tras una más que suficiente introducción empecemos ya ha aplicar el modelo CMM. Primero describiremos las características generales de las acciones a realizar en todos los niveles, posteriormente iremos recorriendo nivel a nivel.

Cada nivel de madurez se compone de una serie de prácticas, "*las colecciones de prácticas de software y de gestión específicas de un nivel de madurez se denominan **Áreas Clave de Proceso (KPAs)***" [Dymon 1997]. En otros términos una KPA es un grupo de actividades relacionadas que cuando se llevan a cabo en conjunto alcanzan una serie de objetivos que se consideran importantes para aumentar la capacidad del proceso. El nivel 1 no recoge ninguna KPA, el resto acoge a 18 KPAs. Cada KPA tiene una serie de prácticas claves a realizar. Por lo tanto y según lo visto hasta ahora ya tenemos dos cosas; objetivos a cumplir y prácticas claves. Las prácticas claves se agrupan en cinco características comunes o características comunes de institucionalización. Cada KPA tiene los cinco tipos de características comunes y al menos una práctica clave bajo cada característica común. Las características comunes son:

- Compromiso para realizar (Co).
- Capacidad para realizar (Ab).
- Actividades realizadas (Ac).
- Medición y análisis (Me).
- Verificación de la implementación (Ve).

Por lo tanto, no nos debe extrañar que en cada KPA nos encontremos estos cinco tipos de actividades más o menos repetidas.

Con todo esto ya podemos afrontar la descripción esquemática de los cinco niveles de madurez. Destacar que empezaremos directamente con el nivel de madurez 2, si una organización no cumple los parámetros estipulados en este nivel, directamente se encontrará situada en el nivel de madurez 1. Pero antes de ello, un apunte más, todo este proceso generará datos, la existencia de un repositorio de datos facilita la labor de proyectos futuros y será parte fundamental para la mejora del proceso dentro de la organización. No estamos perdiendo tiempo si dedicamos personal a estudiar la forma en que se va a guardar y acceder a los datos obtenidos.

Si lo que pretendemos aplicar el modelo CMM a una organización de tipo medio y queremos que el CMM sea efectivo, para ello puede ser necesario depurar y eliminar ciertas acciones o condiciones que para este tipo de organización puede resultar excesivo y no hará sino saturar de trabajo al personal sin producir resultados. Por ello se verá que en muchas ocasiones nos bastará una sencilla lista de comprobación para estudiar una determinada KPA.

BIBLIOGRAFÍA

- Deming E. W., "Quality, Productivity, and Competitive Position". Massachusetts Institute of Technology, Cambridge, Mass., 1982.
- Dymon M., Kenneth. "Una Guia del CMM". Process Inc US. 1997
- Humphrey W., "Managing the Software Process". Addison-Wesley, Reading, Mass.,1989.
- Internacional Organization for Standardization, "Quality Systems Model for Quality Assurance in Desing/development, Production, Installation and Servicing.". ISO-9001. 15/03/1987.
- Paulk M. A., "A comparison of ISO 9001 and the Capability Maturity Model for Software. Technical report CMU/SEI-94-TR-012
- S. Pressman R., "Ingenieria del software, un enfoque practice". 5ª Edicion. McGrawHill. 2002.
- Paulk M., Curtis B., Chrissis MB., Weber C. V. "Capability Maturity Models for Software, Version l.l".Technical Report CMU/SEI-93-TR-024 ESC-TR-93-177. Febrero 1993.
- Paulk M., Garcia S., Chrissis MB., Weber C. V. Bush M. "Key Practices of the Capability
- Maturity Model SM, Version l.l".Technical Report CMU/SEI-93-TR-025 ESC-TR-93-178. Febrero 1993.

Anexo E. UML - Unified Modeling Language

UML es un lenguaje de modelado visual que se usa para especificar, visualizar, construir y documentar artefactos de un sistema de software. Se usa para entender, diseñar, configurar, mantener y controlar la información sobre los sistemas a construir. Es un estándar del OMG, Object Management Group www.omg.org, www.uml.org.

UML capta la información sobre la estructura estática y el comportamiento dinámico de un sistema. Un sistema se modela como una colección de objetos discretos que interactúan para realizar un trabajo que finalmente beneficia a un usuario externo. El lenguaje de modelado pretende unificar la experiencia en técnicas de modelado e incorporar las mejores prácticas actuales en un acercamiento estándar.

UML no es un lenguaje de programación. Las herramientas pueden ofrecer generadores de código de UML para una gran variedad de lenguaje de programación, así como construir modelos por ingeniería inversa a partir de programas existentes. Es un lenguaje de propósito general para el modelado orientado a objetos.

UML es también un lenguaje de modelado visual que permite una abstracción del sistema y sus componentes. Existían diversos métodos y técnicas Orientadas a Objetos, con muchos aspectos en común pero utilizando distintas notaciones, se presentaban inconvenientes para el aprendizaje, aplicación, construcción y uso de herramientas, etc., además de pugnas entre enfoques, lo que generó la creación del UML como estándar para el modelado de sistemas de software principalmente, pero con posibilidades de ser aplicado a todo tipo de proyectos.

UML tiene tres elementos fundamentales:

Bloques básicos de construcción	• *Elementos* • *Relaciones* • *Diagramas*
Reglas que dictan cómo se pueden combinar estos bloques básicos. UML tiene reglas para:	• *Nombres* • *Alcance* • *Visibilidad* • *Integridad* • *Ejecución*
Mecanismos comunes. Que se basen en algún patrón, al igual que en arquitectura se puede hablar del barroco, románico, etc.	• *Especificaciones* • *Adornos* • *Divisiones comunes* • *Mecanismos de extensibilidad*

Como dijo Grady Booch: *El 80% de la mayoría de los problemas pueden modelarse usando alrededor del 20% de UML.*

En todo proceso de software donde se utilice una metodología orientada a objetos y la notación UML no pueden faltar los diagramas, para representar las diferentes vistas del producto final.

Los diagramas de UML se pueden dividir en estáticos (aportan una visión estática del sistema) y dinámicos (aportan una visión dinámica del sistema).

Los diagramas estáticos:	• *Diagrama de casos de uso* • *Diagrama de clases* • *Diagrama de objetos* • *Diagrama de componentes* • *Diagrama de despliegue*
Los diagramas dinámicos:	• *Diagrama de estados* • *Diagrama de actividad* • *Diagramas de interacción* • *Diagrama de secuencia* • *Diagrama de colaboración*

OBJETIVOS DEL UML

- UML es un lenguaje de modelado de propósito general que pueden usar todos los modeladores. No tiene propietario y está basado en el común acuerdo de gran parte de la comunidad informática.
- UML no pretende ser un método de desarrollo completo. No incluye un proceso de desarrollo paso a paso. UML incluye todos los conceptos que se consideran necesarios para utilizar un proceso moderno iterativo, basado en construir una sólida arquitectura para resolver requisitos dirigidos por casos de uso.
- Ser tan simple como sea posible pero manteniendo la capacidad de modelar toda la gama de sistemas que se necesita construir. UML necesita ser lo suficientemente expresivo para manejar todos los conceptos que se originan en un sistema moderno, tales como la concurrencia y distribución, así como también los mecanismos de la ingeniería de software, como son la encapsulación y componentes.
- Debe ser un lenguaje universal, como cualquier lenguaje de propósito general.
- Imponer un estándar mundial.

ARQUITECTURA DEL UML

Arquitectura de cuatro capas, definida a fin de cumplir con la especificación Meta Object Facility del OMG:

- Meta-metamodelo: define el lenguaje para especificar metamodelos.
- Metamodelo: define el lenguaje para especificar modelos.
- Modelo: define el lenguaje para describir un dominio de información.
- Objetos de usuario: define un dominio de información específico.

ÁREAS CONCEPTUALES DE UML

Los conceptos y modelos de UML pueden agruparse en las siguientes áreas conceptuales:

Estructura estática:

Cualquier modelo preciso debe primero definir su universo, esto es, los conceptos clave de la aplicación, sus propiedades internas, y las relaciones entre cada una de ellas. Este conjunto de construcciones es la estructura estática. Los conceptos de la aplicación se modelan como clases, cada una de las cuales describe un conjunto de objetos que almacenan información y se comunican para implementar un comportamiento. La información que almacena se modela como atributos; La estructura estática se expresa con diagramas de clases y puede usarse para generar la mayoría de las declaraciones de estructuras de datos en un programa.

Comportamiento dinámico:
Hay dos formas de modelar el comportamiento, una es la historia de la vida de un objeto y la forma como interactúa con el resto del mundo, y la otra es por los patrones de comunicación de un conjunto de objetos conectados, es decir la forma en que interactúan entre sí. La visión de un objeto aislado es una máquina de estados, mostrando la forma en que el objeto responde a los eventos en función de su estado actual. La visión de la interacción de los objetos se representa con los enlaces entre objetos junto con el flujo de mensajes y los enlaces entre ellos. Este punto de vista unifica la estructura de los datos, el control de flujo y el flujo de datos.

Construcciones de implementación:
Los modelos UML tienen significado para el análisis lógico y para la implementación física. Un componente es una parte física reemplazable de un sistema y es capaz de responder a las peticiones descritas por un conjunto de interfaces. Un nodo es un recurso computacional que define una localización durante la ejecución de un sistema. Puede contener componentes y objetos.

Organización del modelo:
La información del modelo debe ser dividida en piezas coherentes, para que los equipos puedan trabajar en las diferentes partes de forma concurrente. El conocimiento humano requiere que se organice el contenido del modelo en paquetes de tamaño modesto. Los paquetes son unidades organizativas, jerárquicas y de propósito general de los modelos de UML. Pueden usarse para almacenamiento, control de acceso, gestión de la configuración y construcción de bibliotecas que contengan fragmentos de código reutilizable.

Mecanismos de extensión:
UML tiene una limitada capacidad de extensión pero que es suficiente para la mayoría de las extensiones que requiere el día a día sin la necesidad de un cambio en el lenguaje básico. Un estereotipo es una nueva clase de elemento de modelado con la misma estructura que un elemento existente pero con restricciones adicionales.

DIAGRAMAS

Un Modelo captura una vista de un sistema del mundo real. Es una abstracción de dicho sistema, considerando un cierto propósito. Así, el modelo describe completamente aquellos aspectos del sistema que son relevantes al propósito del modelo, y a un apropiado nivel de detalle.

Un proceso de desarrollo de software debe ofrecer un conjunto de modelos que permitan expresar el producto desde cada una de las perspectivas de interés. El código fuente del sistema es el modelo más detallado del sistema (y además es ejecutable). Sin embargo, se requieren otros modelos. Cada modelo es completo desde su punto de vista del sistema, sin embargo, existen relaciones de trazabilidad entre los diferentes modelos.

Un Diagrama es una representación gráfica de una colección de elementos de modelado, a menudo dibujada como un grafo conexo de arcos (relaciones) y vértices (otros elementos del modelo). Un diagrama no es un elemento semántico, un diagrama muestra representaciones de elementos semánticos del modelo, pero su significado no se ve afectado por la forma en que son representados. Un diagrama está contenido dentro de un paquete.

La mayoría de los diagramas de UML y algunos símbolos complejos son grafos que contienen formas conectadas por rutas. La información está sobre todo en la topología, no en el tamaño

o la colocación de los símbolos (hay algunas excepciones como el diagrama de secuencia con un eje métrico de tiempo). Hay tres clases importantes de relaciones visuales: conexión (generalmente de líneas a formas de dos dimensiones), contención (de símbolos por formas cerradas de dos dimensiones), y adhesión visual (un símbolo que está "cerca" de otro en un diagrama). Estas relaciones geométricas se reasignan a conexiones entre nodos en un gráfico en la forma analizada de la notación.

La notación de UML está pensada para ser dibujada en superficies bidimensionales. Algunas formas bidimensionales son proyecciones de formas tridimensionales tales como cubos, pero todavía se representan como iconos en una superficie bidimensional.

Hay cuatro clases de construcciones gráficas que se usan en la notación de UML:

1) Un **icono** es una figura gráfica con un tamaño y forma fijos. No se amplía para contener a su contenido. Los iconos pueden aparecer dentro de símbolos de área, como terminadores en las rutas o como símbolos independientes que puedan o no conectar con las rutas.

2) Los **símbolos de dos dimensiones** tienen altura y anchura variables, y pueden ampliarse para permitir otras cosas tales como listas de cadenas o de otros símbolos. Muchos de ellos están divididos en compartimientos similares o de tipos diferentes. Las rutas se conectan con los símbolos, el arrastrar o suprimir uno de ellos afecta a su contenido y las rutas conectadas.

3) Una **ruta** es una secuencia de segmentos de recta o de curva que se unen en sus puntos finales. Conceptualmente una ruta es una sola entidad topológica, aunque sus segmentos se pueden manipular gráficamente. Un segmento no debería existir separado de su ruta. Las rutas siempre van conectadas en ambos extremos.

4) Las **cadenas** presentan varias clases de información en una forma "no analizada", UML asume que cada uso de una cadena en la notación tiene una sintaxis por la cual pueda ser analizada la información del modelo subyacente. Las cadenas pueden existir como el contenido de un compartimiento, como elementos en las listas, como etiquetas unidas a los símbolos o a las rutas, o como elementos independientes en un diagrama.

UML ESTÁ COMPUESTO POR LOS SIGUIENTES DIAGRAMAS:

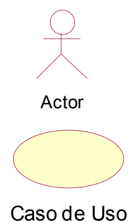
Área	Vista	Diagramas	Conceptos principales
Estructural	Vista Estática	Diagrama de Clases	Clase, asociación, generalización, dependencia, realización, interfaz.
	Vista de Casos de Uso	Diagramas de Casos de Uso	Caso de Uso, Actor, asociación, extensión, generalización.
	Vista de Implementación	Diagramas de Componentes	Componente, interfaz, dependencia, realización.
	Vista de Despliegue	Diagramas de Despliegue	Nodo, componente, dependencia, localización.

Dinámica	Vista de Estados de máquina	Diagramas de Estados	Estado, evento, transición, acción.
	Vista de actividad	Diagramas de Actividad	Estado, actividad, transición, determinación, división, unión.
	Vista de interacción	Diagramas de Secuencia	Interacción, objeto, mensaje, activación.
		Diagramas de Colaboración	Colaboración, interacción, rol de colaboración, mensaje.
Administración o Gestión de modelo	Vista de Gestión de modelo	Diagramas de Clases	Paquete, subsistema, modelo.
Extensión de UML	Todas	Todos	Restricción, estereotipo, valores, etiquetados.

DIAGRAMAS DE CASOS DE USO.

Los diagramas de casos de uso muestran la funcionalidad del sistema desde la perspectiva que tienen los usuarios y lo que el sistema debe de hacer para satisfacer los requisitos propuestos. Pueden mostrar el comportamiento de un sistema completo o de una parte.

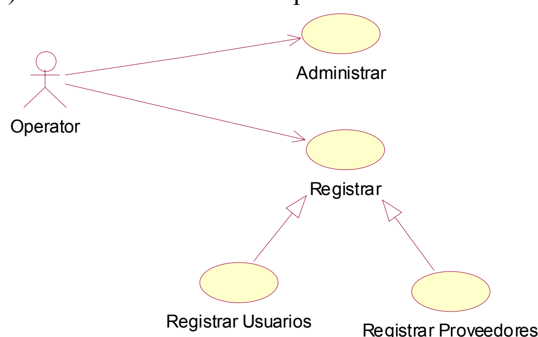
Los elementos básicos que se utilizan son:



1) **Actores**: Son los diferentes usuarios y el papel que representan dentro del sistema.
2) **Caso de uso**: Representan todo lo que el usuario puede realizar dentro del sistema.
3) **Relaciones**: Para asociar los elementos anteriores.

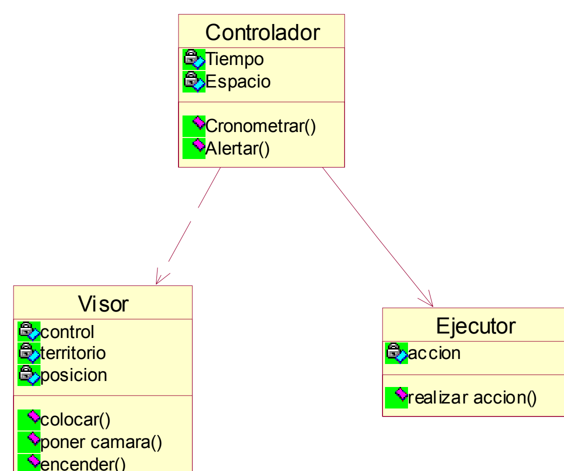
Comunicación	Generalización	Extensión(*)	Inclusión(*)

(*) En estas dos se deben de poner también las letras (include, o exclude).



DIAGRAMAS DE CLASES

Los diagramas de clases son estáticos porque no describen un comportamiento en función del tiempo. Tienen que ver con la implementación de la aplicación.

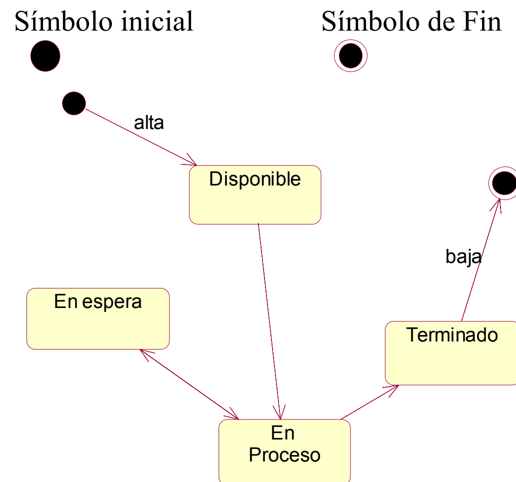


Los elementos son:

1) Clases: Se pueden definir como la descripción de un conjunto de objetos con las mismas propiedades. Puede ser un concepto del mundo real, se puede decir que es una plantilla para crear objetos.
2) Relaciones: Las relaciones pueden ser de distintos tipos asociación, agregación, herencia (generalización, especialización).

DIAGRAMA DE ESTADOS

El diagrama de estados es un gráfico compuesto de los estados del sistema y sus transiciones. Si se asocia a una clase describirá como una instancia de esta clase reacciona ante los eventos. Si se asocia a un caso de uso describirá como funciona ese caso de uso con el sistema funcionando.



Estos diagramas son muy útiles para mostrar el ciclo de vida de las clases con complejidad media o alta, pero no tiene mucho sentido para clases de una complejidad simple ni tampoco para clases con una complejidad bastante elevada.

DIAGRAMA DE COLABORACIÓN

Los diagramas de colaboración son útiles para mostrar los efectos que puede tener un objeto con los demás. Los elementos básicos son clases encerradas en rectángulos, enlaces entre clases y operaciones entre clases. La estructura estática viene dada por los enlaces; la dinámica por el envío de mensajes por los enlaces.

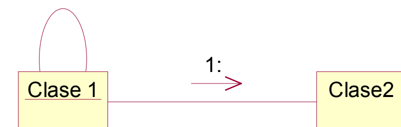
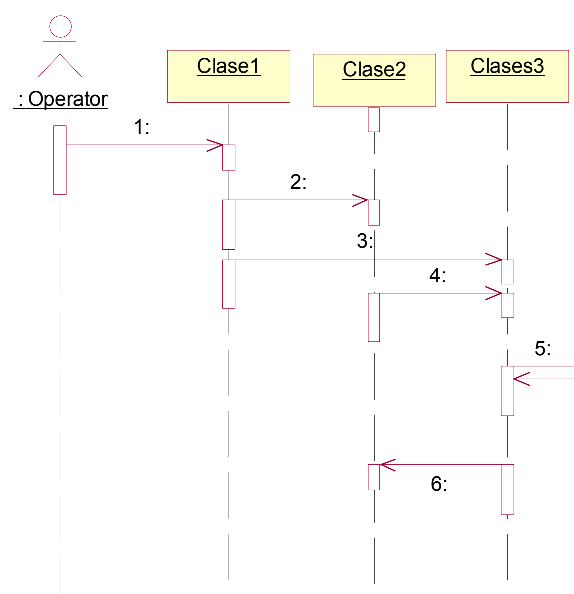


DIAGRAMA DE SECUENCIA

Estos diagramas representan la interacción entre clases, se leen de izquierda a derecha y de arriba abajo. Muestran el comportamiento del sistema de forma cronológica. Y por esto son muy útiles cuando se está trabajando con sistemas de tiempo real.



Los elementos básicos son las clases que se representan con rectángulos, los actores, las barras de sincronización temporal que se representan con líneas discontinuas y los mensajes que se representan con flechas.

Anexo F. The CHAOS Report, 1994 (Standish group)

INTRODUCTION

In 1986, Alfred Spector, president of Transarc Corporation, co-authored a paper comparing bridge building to software development. The premise: Bridges are normally built on-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always breaks down. (Nevertheless, bridge building did not always have such a stellar record. Many bridge building projects overshot their estimates, time frames, and some even fell down.)

One of the biggest reasons bridges come in on-time, on-budget and do not fall down is because of the extreme detail of design. The design is frozen and the contractor has little flexibility in changing the specifications. However, in today's fast moving business environment, a frozen design does not accommodate changes in the business practices. Therefore a more flexible model must be used. This could be and has been used as a rationale for development failure.

But there is another difference between software failures and bridge failures, beside 3,000 years of experience. When a bridge falls down, it is investigated and a report is written on the cause of the failure. This is not so in the computer industry where failures are covered up, ignored, and/or rationalized. As a result, we keep making the same mistakes over and over again.

Consequently the focus of this latest research project at The Standish Group has been to identify:

- The scope of software project failures
- The major factors that cause software projects to fail
- The key ingredients that can reduce project failures

FAILURE RECORD

In the United States, we spend more than $250 billion each year on IT application development of approximately 175,000 projects. The average cost of a development project for a large company is $2,322,000; for a medium company, it is $1,331,000; and for a small company, it is $434,000. A great many of these projects will fail. Software development projects are in chaos, and we can no longer imitate the three monkeys -- hear no failures, see no failures, speak no failures.

The Standish Group research shows a staggering 31.1% of projects will be canceled before they ever get completed. Further results indicate 52.7% of projects will cost 189% of their original estimates. The cost of these failures and overruns are just the tip of the proverbial iceberg. The lost opportunity costs are not measurable, but could easily be in the trillions of dollars. One just has to look to the City of Denver to realize the extent of this problem. The failure to produce reliable software to handle baggage at the new Denver airport is costing the city $1.1 million per day.

Based on this research, The Standish Group estimates that in 1995 American companies and government agencies will spend $81 billion for canceled software projects. These same organizations will pay an additional $59 billion for software projects that will be completed, but will exceed their original time estimates. Risk is always a factor when pushing the

technology envelope, but many of these projects were as mundane as a drivers license database, a new accounting package, or an order entry system.

On the success side, the average is only 16.2% for software projects that are completed on-time and on-budget. In the larger companies, the news is even worse: only 9% of their projects come in on-time and on-budget. And, even when these projects are completed, many are no more than a mere shadow of their original specification requirements. Projects completed by the largest American companies have only approximately 42% of the originally-proposed features and functions. Smaller companies do much better. A total of 78.4% of their software projects will get deployed with at least 74.2% of their original features and functions.

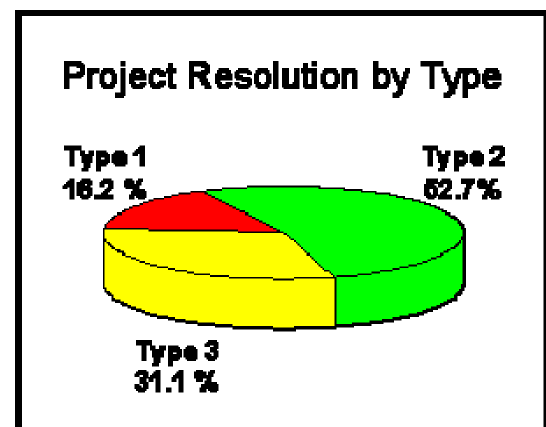
This data may seem disheartening, and in fact, 48% of the IT executives in our research sample feel that there are more failures currently than just five years ago. The good news is that over 50% feel there are fewer or the same number of failures today than there were five and ten years ago.

METHODOLOGY

The survey made by The Standish Group was as thorough as possible, short of the unreachable goal of surveying every company with MIS in the country. The results are based on what we at The Standish Group define as "key findings" from our research surveys and several personal interviews. The respondents were IT executive managers. The sample included large, medium, and small companies across major industry segments, e.g., banking, securities, manufacturing, retail, wholesale, heath care, insurance, services, and local, state, and federal organizations. The total sample size was 365 respondents and represented 8,380 applications. In addition, The Standish Group conducted four focus groups and numerous personal interviews to provide qualitative context for the survey results.

For purposes of the study, projects were classified into three resolution types:

- **Resolution Type 1, or project success**: The project is completed on-time and on-budget, with all features and functions as initially specified.
- **Resolution Type 2, or project challenged**: The project is completed and operational but over-budget, over the time estimate, and offers fewer features and functions than originally specified.
- **Resolution Type 3, or project impaired**: The project is canceled at some point during the development cycle.



Overall, the success rate was only 16.2%, while challenged projects accounted for 52.7%, and impaired (canceled) for 31.1%.

FAILURE STATISTICS

The Standish Group further segmented these results by large, medium and small companies. A large company is any company with greater than $500 million dollars in revenue per year, a medium company is defined as having $200 million to $500 million in yearly revenue, and a

small company is from $100 million to $200 million. The figures for failure were equally disheartening in companies of all sizes. Only 9% of projects in large companies were successful. At 16.2% and 28% respectively, medium and small companies were somewhat more successful. A whopping 61.5% of all large company projects were challenged (Resolution Type 2) compared to 46.7% for medium companies and 50.4% for small companies. The most projects, 37.1%, were impaired and subsequently canceled (Resolution Type 3) in medium companies, compared to 29.5% in large companies and 21.6% in small companies.

Restarts

One of the major causes of both cost and time overruns is restarts. For every 100 projects that start, there are 94 restarts. This does not mean that 94 of 100 will have one restart; some projects can have several restarts. For example, the California Department of Motor Vehicles project, a failure scenario summarized later in this article, had many restarts.

Cost Overruns

Equally telling were the results for cost overruns, time overruns, and failure of the applications to provide expected features. For combined Type 2 and Type 3 projects, almost a third experienced cost overruns of 150 to 200%. The average across all companies is 189% of the original cost estimate. The average cost overrun is 178% for large companies, 182% for medium companies, and 214% for small companies.

Cost overruns	% of responses
Under 20%	*15.5%*
21 - 50%	*31.5%*
51 - 100%	*29.6%*
101 - 200%	*10.2%*
201 - 400%	*8.8%*
Over 400%	*4.4%*

Time Overruns

For the same combined challenged and impaired projects, over one-third also experienced time overruns of 200 to 300%. The average overrun is 222% of the original time estimate. For large companies, the average is 230%; for medium companies, the average is 202%; and for small companies, the average is 239%.

Time overruns	% of responses
Under 20%	*13.9%*
21 - 50%	*18.3%*
51 - 100%	*20.0%*
101 - 200%	*35.5%*
201 - 400%	*11.2%*
Over 400%	*1.1%*

Content Deficiencies

For challenged projects, more than a quarter were completed with only 25% to 49% of originally-specified features and functions.

On average, only 61% of originally specified features and functions were available on these projects. Large companies have the worst record with only 42% of the features and functions in the end product. For medium companies, the percentage is 65%. And for small companies, the percentage is 74%.

% of Features/Functions	% of responses
Less Than 25%	*4.6%*
25 - 49%	*27.2%*
50 - 74%	*21.8%*
75 - 99%	*39.1%*
100%	*7.3%*

Currently, the 365 companies have a combined 3,682 applications under development. Only 431 or 12% of these projects are on-time and on-budget.

SUCCESS/FAILURE PROFILES

The most important aspect of the research is discovering why projects fail. To do this, The Standish Group surveyed IT executive managers for their opinions about why projects succeed. The three major reasons that a project will succeed are user involvement, executive management support, and a clear statement of requirements. There are other success criteria, but with these three elements in place, the chances of success are much greater. Without them, chance of failure increases dramatically.

Project Success Factors	**% of responses**
1. User Involvement	*15.9%*
2. Executive Management Support	*13.9%*
3. Clear Statement of Requirements	*13.0%*
4. Proper Planning	*9.6%*
5. Realistic Expectations	*8.2%*
6. Smaller Project Milestones	*7.7%*
7. Competent Staff	*7.2%*
8. Ownership	*5.3%*
9. Clear Vision & Objectives	*2.9%*
10. Hard-Working, Focused Staff	*2.4%*
Other	*13.9%*

The survey participants were also asked about the factors that cause projects to be challenged.

Project Challenged Factors	**% of responses**
1. Lack of User Input	*12.8%*
2. Incomplete Requirements & Specifications	*12.3%*
3. Changing Requirements & Specifications	*11.8%*
4. Lack of Executive Support	*7.5%*
5. Technology Incompetence	*7.0%*
6. Lack of Resources	*6.4%*
7. Unrealistic Expectations	*5.9%*
8. Unclear Objectives	*5.3%*
9. Unrealistic Time Frames	*4.3%*
10. New Technology	*3.7%*
Other	*23.0%*

Opinions about why projects are impaired and ultimately canceled ranked incomplete requirements and lack of user involvement at the top of the list.

Project Impaired Factors	**% of responses**
1. Incomplete Requirements	*13.1%*
2. Lack of User Involvement	*12.4%*
3. Lack of Resources	*10.6%*
4. Unrealistic Expectations	*9.9%*
5. Lack of Executive Support	*9.3%*
6. Changing Requirements & Specifications	*8.3%*
7. Lack of Planning	*8.1%*
8. Didn't Need It Any Longer	*7.5%*
9. Lack of IT Management	*6.2%*
10. Technology Illiteracy	*4.3%*
Other	*9.9%*

Another key finding of the survey is that a high percentage of executive managers believe that there are more project failures now than five years ago and ten years ago.

This despite the fact that technology has had time to mature.

	Than 5 Years Ago	Than 10 Years Ago
Significantly More Failures	*27%*	*17%*
Somewhat More Failures	*21%*	*29%*
No Change	*11%*	*23%*
Somewhat Fewer Failures	*19%*	*23%*
Significantly Fewer Failures	*22%*	*8%*

FOCUS GROUPS

To augment the survey results, The Standish Group conducted four focus groups with IT executives of major companies. The attendees were from a cross section of industries, including insurance, state and federal government, retail, banking, securities, manufacturing and service. Two of the focus groups were in Boston. The other two, in San Francisco. Each focus group had an average of ten participants with an overall total of forty-one IT executives. The purpose of these particular focus groups was to solicit opinions on why projects fail. In addition, The Standish Group conducted interviews with various IT managers. Some of their comments are enlightening about the variety of problems besetting project development.

Many of the comments echoed the findings of The Standish Group survey. "We have 500 projects. None are on-time and on-budget. This year, 40% will get canceled," said Edward, Vice President of MIS at a pharmaceutical company.

Other comments went directly to the reasons for failure. Jim, the Director of IT at a major medical equipment manufacturer, said: "Being that it's a mindset, it's very difficult to get all of the management -- it's even on the local level, not even on a worldwide level -- to get all of the management to agree on a set of rules.... That's a challenge in itself because you have to, in some cases, convince them that this is best for the company, not necessarily best for them, but best for the company. And you have to have that buy-in. If you don't have that buy-in, you're going to fail. I don't care how big or how small the project is."

John, Director of MIS at a government agency added: "Probably 90% of application project failure is due to politics!" And Kathy, a programmer at a telecommunication company, offered an even more scathing comment on politics: "Sometimes you have to make a decision you don't like. Even against your own nature. You say well, it's wrong, but you make that decision anyway. It's like taking a hammer to your toe. It hurts."

Bob, the Director of MIS at a hospital, commented on external factors contributing to project failure. "Our biggest problem is competing priorities," he said. "We just had a reorganization today. So now that's going to sap all the resources. And explaining to senior management that, 'Well, it's really taking us the time we said it was going to take. But because you've reorganized the company, I'm going to take another six months on this other project, because I'm doing something else for you.' That's the biggest issue I have." Bill, the Director of MIS at a securities firm, added: "Changes, changes, changes; they're the real killers."

Some of the comments were darkly humorous. "Brain-dead users, just plain brain-dead users," said Peter, an application analyst at a bank. "When the projected started to fail," said Paul, a programmer at a personal products manufacturer, "the management got behind it -- way behind."

The comment most indicative of the chaos in project development came from Sid, a project manager at an insurance company. "The project was two years late and three years in development," he said. "We had thirty people on the project. We delivered an application the user didn't need. They had stopped selling the product over a year before."

CASE STUDIES

For further insight into failure and success, The Standish Group looked carefully at two famous Resolution Type 3 (canceled) projects and two Resolution Type 1 (successful) projects. For purposes of comparison, the project success criteria from the survey of IT executive managers was used to create a "success potential" chart. The success criteria were then weighted based on the input from the surveyed IT managers. The most important criterion, "user involvement," was given 19 "success points". The least important -- "hard-working, focused staff" -- was given three points. Two very important success criteria -- "realistic expectations" and "smaller project milestones" -- were weighted at ten and nine points respectively. Finally, as presented later in this report, each of the case studies was graded.

California DMV

In 1987, the California Department of Motor Vehicles (DMV) embarked on a major project to revitalize their drivers license and registration application process. By 1993, after $45 million dollars had already been spent, the project was canceled.

According to a special report issued by DMV, the primary reason for redeveloping this application was the adoption new technology. They publicly stated: "The specific objective of the 1987 project was to use modern technology to support the DMV mission and sustain its growth by strategically positioning the DMV data processing environment to rapidly respond to change." Also, according to the DMV special report "The phasing was changed several times, but the DMV technical community was never truly confident in its viability...."

The project had no monetary payback, was not supported by executive management, had no user involvement, had poor planning, poor design specifications and unclear objectives. It also did not have the support of the state's information management staff. The DMV project was not rocket science. There are much harder applications than driver licenses and registrations. But because of internal state politics, unclear objectives, and poor planning, the project was doomed from the start.

American Airlines

Early in 1994, American Airlines settled their lawsuit with Budget Rent-A-Car, Marriott Corp. and Hilton Hotels after the $165 million CONFIRM car rental and hotel reservation system project collapsed into chaos. This project failed because there were too many cooks and the soup spoiled. Executive management not only supported the project, they were active project managers. Of course, for a project this size to fail, it must have had many flaws. Other major causes included an incomplete statement of requirements, lack of user involvement, and constant changing of requirements and specifications.

Hyatt Hotels
While Marriott and Hilton Hotels were checking out of their failed reservation system, Hyatt was checking in. Today, you can dial from a cellular airplane telephone at 35,000 feet, check into your Hyatt hotel room, schedule the courtesy bus to pick you up, and have your keys waiting for you at the express desk. This new reservation system was ahead of schedule, under budget, with extra features -- for a mere $15 million of cold cash. They used modern, open systems software with an Informix database and the TUXEDO transaction monitor, on Unix-based hardware. Hyatt had all the right ingredients for success: user involvement, executive management support, a clear statement of requirements, proper planning, and small project milestones.

Banco Itamarati
A year after a strategic redirection, Banco Itamarati, a privately-held Brazilian bank, produced an annual net profit growth of 51% and moved from 47th to 15th place in the Brazilian banking industry. Three fundamental reasons account for Banco Itamarati's success. First, they had a clear vision with documented specific objectives. Second, their top-down level of involvement allowed Banco Itamarati to stay on course. And finally, the bank produced incremental, measurable results throughout the planning/implementation period.

Banco Itamarati's clear business goal is to be one of Brazil's top five privately-held banks by the year 2000. Their objectives include maintaining a close relationship with their customers to improve and maintain an understanding of their needs, offering competitive financial solutions, guaranteeing customer satisfaction, and finally producing balanced results for the Itamarati Group. Banco Itamarati's objectives were incorporated into a strategic plan that clearly identified measurable results and individual ownership.

Their strategic plan made technology a key component of the business strategy. Itamarati used Itautec's GRIP OLTP monitor as a basic tool for integrating software components. According to Henrique Costabile, Director of Organization Development, "We are one of the first banks to implement a client-server architecture that maximizes the potential of this architecture." Executive leadership, a well-communicated plan, and a skilled diverse team provided the foundation for Banco Itamarati to achieve their long-term goal, potentially ahead of schedule.

CASE STUDY CONCLUSIONS
The study of each project included adding up success points on the "success potential" chart.

Success Criteria	Points	DMV	Confirm	HYATT	ITAMARATI
1. User Involvement	19	No (0)	No (0)	Yes (19)	Yes (19)
2. Executive Management Support	16	No (0)	Yes (16)	Yes (16)	Yes (16)
3. Clear Statement of Requirements	15	No (0)	No (0)	Yes (15)	No (0)
4. Proper Planning	11	No (0)	No (0)	Yes (11)	Yes (11)
5. Realistic Expectations	10	Yes (10)	Yes (10)	Yes (10)	Yes (10)
6. Smaller Project Milestones	9	No (0)	No (0)	Yes (9)	Yes (9)
7. Competent Staff	8	No (0)	No (0)	Yes (8)	Yes (8)
8. Ownership	6	No (0)	No (0)	Yes (6)	Yes (6)
9. Clear Vision & Objective	3	No (0)	No (0)	Yes (3)	Yes (3)
10. Hard-Working, Focused Staff	3	No (0)	Yes (3)	Yes (3)	Yes (3)
TOTAL	**100**	**10**	**29**	**100**	**85**

With only 10 success points, the DMV project had virtually no chance of success. With 100 success points, Hyatt's reservation project had all the right ingredients for success. With only 29 success points, the CONFIRM project had little chance of success. With 85, Itamarati, while not as assured as Hyatt, started with a high success probability.

THE BRIDGE TO SUCCESS

Notwithstanding, this study is hardly in-depth enough to provide a real solution to such a daunting problem as the current project failure rates. Application software projects are truly in troubled waters. In order to make order out of the chaos, we need to examine why projects fail. Just like bridges, each major software failure must be investigated, studied, reported and shared. Because it is the product of the ideas of IT managers, the "Success Potential" chart can be a useful tool for either forecasting the potential success of a project or evaluating project failure.

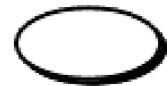
Research at The Standish Group also indicates that smaller time frames, with delivery of software components early and often, will increase the success rate. Shorter time frames result in an iterative process of design, prototype, develop, test, and deploy small elements. This process is known as "growing" software, as opposed to the old concept of "developing" software. Growing software engages the user earlier, each component has an owner or a small set of owners, and expectations are realistically set. In addition, each software component has a clear and precise statement and set of objectives. Software components and small projects tend to be less complex. Making the projects simpler is a worthwhile endeavor because complexity causes only confusion and increased cost.

There is one final aspect to be considered in any degree of project failure. All success is rooted in either luck or failure. If you begin with luck, you learn nothing but arrogance. However, if you begin with failure and learn to evaluate it, you also learn to succeed. Failure begets knowledge. Out of knowledge you gain wisdom, and it is with wisdom that you can become truly successful.

Anexo G. Casos de uso

La captura y documentación de requisitos es una de las tareas más importantes en el desarrollo de aplicaciones informáticas. Puede que sea incluso la más importante de todas, puesto que de una adecuada comprensión de los requisitos del sistema depende en gran parte el éxito o fracaso del proyecto. Quizá nuestro sistema sea el más veloz, el que mejor aprovecha los recursos, el que menos incidencias reportará, el que está diseñado y programado con la mayor elegancia y haciendo uso de las últimas tecnologías. Pero si no hace lo que el usuario realmente necesita, habremos fracasado estrepitosamente.

¿Qué son los Casos de Uso?



Los casos de uso son casos de utilización del sistema, descripciones narrativas de su comportamiento, gracias a las cuales podemos mejorar la comprensión de los requisitos. Estas descripciones cubren tanto el comportamiento normal del sistema, como todas las variantes, de éxito o de fracaso, que pudieran originarse durante un proceso. Un caso de uso capta una funcionalidad visible para el usuario. Logra un objetivo concreto, tangible.
Por tanto, podemos decir que un caso de uso describe la secuencia de eventos y acciones que se producen entre un Actor y un Sistema que interactúan para cumplir un objetivo.

¿Qué ventajas me pueden aportar?

Los casos de uso:

- Documentan los procesos de negocio del sistema.
- Capturan los requisitos del sistema desde la perspectiva del usuario (libres de cuestiones de implementación), y por tanto lo involucran en la revisión y validación de los mismos. Su lenguaje es igualmente comprensible y útil para los miembros el equipo de desarrollo.
- Descubren posibles áreas de colaboración del negocio.
- Permiten separar los procesos del negocio en áreas funcionales.
- Ayudan a identificar posibilidades de reutilización.
- Pueden emplearse para categorizar los requisitos del sistema (Nivel de importancia, Nivel de riesgo, Nivel de Prioridad, Nivel de Dificultad, etc.).
- Pueden ser utilizados para exponer los requisitos a varios niveles: De alto nivel, nivel de diseño detallado, etc.).
- Gracias a ellos podemos identificar y mostrar el impacto que tendrán los cambios de requisitos funcionales sobre la implementación, o el impacto de los cambios de implementación sobre la funcionalidad del sistema.
- Sirven como base para el plan de pruebas del sistema.
- Para la elaboración del manual de usuario pueden aprovecharse gran parte de las descripciones de los casos de uso.
- Fomentan la calidad del sistema, ya que identifican escenarios alternativos y excepciones posibles, en una fase temprana del proceso de desarrollo.

Identificar los Casos de Uso del sistema

Existen dos métodos distintos para identificar los casos de uso:

A) Identificación basada en los actores:
1: Se identifican los actores.

2: Averiguamos qué procesos inician o en cuáles participan:

- ¿Cuáles son sus responsabilidades, de qué tareas se encargan: Crear / Modificar / Eliminar elementos, Introducir / Obtener datos, Mantenimiento / Soporte del sistema?
- ¿Deberán informar al sistema sobre algún evento externo que se produzca? (ej. Llegada de ficheros de datos a su destino, listos para ser procesados)
- ¿Deben ser informados por el sistema sobre algún evento que se produzca (ej.: Error en la ejecución de un proceso desatendido)?
- ¿Necesitan indicar al sistema que efectúe algún proceso concreto en un momento determinado (ej.: Realizar una copia de seguridad de los datos del período)?
- Otros procesos en los que los actores participen como estimuladores del sistema, como receptores de información procedente del sistema, o como colaboradores del mismo en la ejecución de tareas.

B) Identificación basada en los eventos:
1: Se identifican los eventos ante los que el Sistema debe reaccionar.
- Creación/Modificación/Eliminación de elementos.
- Entrada/Solicitud de datos por parte de algún actor.
- Orden de ejecución de algún proceso.
- Notificaciones sobre eventos externos al sistema (p.ej.: El paso del tiempo).
- ¿Es necesario que algún actor sea informado sobre ciertos cambios o acontecimientos que se produzcan en el sistema?
- Cualquier otro evento ante el cual el sistema deba reaccionar.

2: Se relacionan los eventos con actores y con casos de uso.

Al finalizar, deberemos hacernos una última pregunta: ¿Los casos de uso que hemos identificado son capaces de cubrir todos los requisitos funcionales que tenemos anotados?

ERRORES COMUNES EN LA IDENTIFICACIÓN DE CASOS DE USO

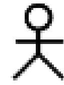
- La Parte por el Todo:
Hay que recordar que un caso de uso está formado por un conjunto de pasos o transacciones necesarios para cumplir un proceso. Sin embargo, un error común en la identificación de casos de uso, es representar los pasos, las operaciones o las transacciones individuales, como casos de uso. Por ejemplo: “Imprimir Factura” sería un caso de uso erróneo, puesto que en realidad se trata de un paso u operación del caso de uso “Comprar Producto/Servicio”. No obstante hay ocasiones en las que un paso o transacción de un caso de uso merece ser representado como un caso de uso aparte (Lo veremos más adelante, en el repaso de “Includes/Extends”).

- “Interacciones con el sistema” y “Objetivos del usuario”:
En ocasiones existen diferencias entre lo que el usuario hace con el sistema, y los verdaderos objetivos del usuario. Por ejemplo, en un procesador de textos “Aplica Negrita” y “Cambia estilo” son interacciones con el sistema. Sin embargo, los verdaderos objetivos del usuario son: “Garantiza el formato del documento” y “Haz que el formato del documento sea igual que el de otro”.

Teniendo en cuenta tanto los objetivos del usuario como las interacciones con el sistema, podremos considerar formas alternativas para el cumplimiento de tales objetivos. Si llegamos muy pronto a la interacción con el sistema, quizá recurriremos a la opción obvia para la solución, pasando por alto otras maneras posiblemente más efectivas de cumplir con los objetivos del usuario.

LOS ACTORES

¿QUÉ ES UN ACTOR DE CASOS DE USO, QUÉ REPRESENTA?



Los actores no forman parte del sistema. Un actor es una entidad externa al sistema que de alguna manera participa en el caso de uso. Generalmente estimula al sistema con eventos de entrada, o bien recibe algo de él. Podemos distinguir varios tipos de actores:

Actor Silencioso:
Es aquel que tiene un interés personal en el comportamiento del caso de uso, incluso si nunca interactúa directamente con el sistema. Ejemplos de este tipo de actores serían: El dueño del sistema, el departamento de procesos de la compañía, etc.
Prestar atención a este tipo de actores mejora la calidad de los casos de uso. Sus intereses se advierten en las validaciones que el sistema realiza, los logs que genera, y las acciones que lleva a cabo. Los casos de uso deben también mostrar cómo el sistema protege estos intereses.

Actor Principal:
Es aquel que invoca al sistema para lograr cierto objetivo. Este actor es frecuentemente, aunque no siempre, quien inicia el caso de uso enviándole un mensaje, pulsando un botón, etc. Hay dos situaciones comunes en las que el que inicia el caso de uso no es el actor principal: Una es cuando un operador telefónico o un dependiente de una tienda inician el caso de uso ante una petición de otra persona, que es la realmente interesada. El otro caso se da cuando el que inicia el caso de uso es "el tiempo".

Los actores principales son importantes al principio de la captura de requisitos y después, justo antes de empezar a desarrollar el sistema. Entre estos dos puntos, no deben preocuparnos demasiado.

Durante la fase inicial de captura de requisitos, obtener una completa lista de actores principales nos permite tener un punto de vista amplio sobre el sistema, detectando y comprendiendo los objetivos que deben cumplirse, basándonos en las necesidades de esos actores principales.

Antes de empezar a desarrollar el sistema, hacer un repaso de esa lista nos permitirá asegurarnos de que dichas necesidades han sido realmente satisfechas, establecer niveles de seguridad para cada caso de uso (administrador, usuario web...), y preparar manuales de usuario para los distintos grupos de usuarios.

Actor de Soporte:
Es aquel que proporciona un servicio al sistema. Puede ser una impresora, otro sistema distinto al nuestro que provee acceso a sus servicios (p.ej.: Web Services), o que invoca a nuestro sistema para lograr un objetivo, etc. Identificar los actores de soporte es muy importante, porque eso nos permitirá detectar las interfaces externas que nuestro sistema utilizará. Un mismo actor puede ejercer de actor principal en un caso de uso, y de actor de soporte en otro.

También es cierto que sobre la conveniencia de identificar como actor a un sistema externo, hay diferentes opiniones entre los gurús del tema. Yo personalmente me quedo con la primera, por los motivos que acabo de exponer, pero tú échales un vistazo y luego decide por ti mismo cuál te parece mejor:

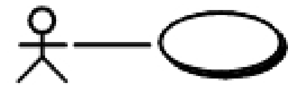
- Todas las interacciones con sistemas externos deben aparecer en el diagrama de casos de uso.
- Sólo se deben mostrar los sistemas externos como actores, cuando ellos sean los que inicien el caso de uso.
- Sólo se deben mostrar los sistemas externos como actores, cuando ellos sean los que necesiten al caso de uso (Martin Fowler, autor de “UML Gota a Gota”, se queda con ésta).
- Algunos piensan que es un enfoque equivocado considerar que un sistema externo es un actor. Paradójicamente, sostienen que un actor es un usuario que desea algo del sistema.

IDENTIFICAR A LOS ACTORES

A parte de lo que ya hemos visto, podemos hacernos las siguientes preguntas como ayuda para identificar a los actores:

¿Quién está interesado en un requisito concreto?
¿En qué dominios de la organización se usará el sistema?
¿Quién será beneficiario de la nueva funcionalidad?
¿Quién proporcionará, usará u obtendrá información?
¿Quién dará soporte y administrará el sistema?
¿Usará el sistema un recurso externo?
¿Interaccionará el nuevo sistema con un sistema antiguo?

LOS ACTORES JUEGAN ROLES



Para explicar lo siguiente, centrémonos por un momento en el caso en que los actores son personas. Un actor puede representar bien el cargo de la persona que usa el sistema, o bien el rol que esa persona está jugando en el momento de usar el sistema. Realmente no es significativo cuál de estos modos empleemos para referirnos al término “actor”. Lo verdaderamente importante es “el objetivo” que cada caso de uso pretende cumplir, ya que nos dice lo que el sistema va a hacer. Muchas veces nos encontraremos con que hay roles cuyas competencias se solapan. Por ejemplo, quizá veamos que el Jefe de Ventas podrá eventualmente actuar como un Comercial, realizando una venta. En estos casos podremos optar por cualquiera de las siguientes soluciones:

- Escribir en la cabecera del caso de uso: El actor principal es el Jefe de Ventas o el Comercial.
- Escribir: El Jefe de Ventas puede jugar también el rol Comercial en este caso de uso. (Si estamos usando diagramas UML, podremos reflejarlo dibujando una flecha de generalización que vaya del Jefe de Ventas al Comercial).
- Crear un nuevo rol con un nombre más genérico, por ejemplo “Vendedor”, y hacer que este sea el actor principal del caso de uso. Escribiremos que el Jefe de Ventas y el Comercial están jugando el rol “Vendedor” en ese caso de uso. (Nuevamente, en UML, podremos reflejarlo dibujando una flecha de generalización que vaya desde el Jefe de Ventas y el Comercial al nuevo rol Vendedor).

Ninguna de estas opciones es mejor que la otra. Simplemente debemos escoger aquella que nos resulte más cómoda de utilizar y creamos que facilita la comprensión a las personas que vayan a leer nuestros casos de uso (los demás miembros del equipo, el usuario, etc.).

La importancia de la frontera o contexto del sistema

Es importante definir la frontera del sistema para distinguir lo que es interno o externo a él, así como las responsabilidades del sistema. El ambiente externo está representado únicamente por actores, por lo que la decisión sobre la frontera que se elija para el sistema tendrá influencias importantes. Por ejemplo, en el sistema de un supermercado, si elegimos como "el sistema" la tienda entera, el Cajero no sería un actor, ya que se encontraría dentro del sistema. Sin embargo, si escogemos como sistema el software y hardware del Terminal Punto de Venta, se entenderían como actores al Cliente y al Cajero.

Escribiendo Casos de Uso

¿Qué son los Diagramas de Casos de Uso?

Los diagramas de casos de uso tienen por objeto permitir conocer rápidamente los actores externos del sistema, y las formas básicas en que lo utilizan.

Un diagrama de casos de uso:

- Explica gráficamente un conjunto de casos de uso, normalmente agrupados por funcionalidad.
- Representa la relación entre los actores y los casos de uso.
- Describe la interacción de los actores con el sistema.

¿Cómo represento las ramificaciones que puede sufrir un Caso de Uso?

Un caso de uso puede contener puntos de decisión. Por ejemplo, en una compra, el cliente puede optar por distintas formas de pago (efectivo, tarjeta, cheque...) Si una de las trayectorias de decisión es muy representativa, y las otras son alternativas poco frecuentes, el caso típico será sobre el que se describa el curso normal, y las otras opciones se describirán como cursos alternativos. Pero para los casos en los que todas las opciones son igualmente importantes y de uso frecuente, podemos seguir la siguiente notación:

1) En la sección principal curso normal se indicarán las ramificaciones.
2) Se creará una subsección por cada ramificación, siguiendo el esquema de descripción acostumbrado.
3) Si alguna subsección tiene a su vez ramificaciones, se describirán como cursos alternativos de esa subsección.

Ejemplo:

Acción de los actores	Respuesta del sistema
1.- ...	2.- ...
3.- El Cliente escoge el tipo de pago: a) Efectivo (ver subsección "Pago en Efectivo") b) Tarjeta (ver subsección "Pago con Tarjeta") c) Cheque (ver subsección "Pago con Cheque")	4.- ...

¿CUÁNDO DEBO USAR LA RELACIÓN INCLUDES Y CUÁNDO LA RELACIÓN EXTENDS?

Includes (Antes llamada "Uses"):
Un caso de uso "incluido" es básicamente un paso del caso de uso base, que decidimos extraer a parte para mejorar la comprensión, por la importancia que tiene el paso por sí mismo, o para factorizar comportamiento común que se repite en varios casos de uso.
El caso de uso base conoce cuándo, dónde y por qué debe ejecutarse el caso de uso incluido. Por tanto, el caso de uso base es quien "inicia", quien "llama" al caso de uso que incluye.
Extends:
Una relación de extensión se utiliza para:

- Modelar las variantes posibles del caso de uso base.
- Modelar la parte de un caso de uso que el usuario puede ver como comportamiento opcional del sistema. De esta forma, se separa el comportamiento opcional del obligatorio.
- Modelar un subflujo separado que se ejecuta sólo bajo ciertas condiciones.
- Modelar varios flujos que se pueden insertar en un punto dado, controlados por la interacción explícita con un actor.

Una relación extends puede reflejar básicamente una extensión (una variante, un curso alternativo) del caso de uso base, que extraemos para modelarlo como caso de uso a parte.

El caso de uso base sigue su curso, pero ante determinadas condiciones, su comportamiento se ve interrumpido con el del caso de uso que lo extiende.

El caso de uso extensión es el que conoce cuándo, dónde y por qué debe incorporar su comportamiento, extendiendo el caso de uso base. El caso de uso base no nombra al caso de uso que le extiende. De hecho no sabe nada de él. El caso de uso extensión nombra al caso de uso base cuando se cumple la condición que lo hace ejecutarse.

Algunos ejemplos del uso de extends:

- Cuando hay varios servicios asíncronos que el usuario puede activar interrumpiendo al caso de uso base. Por ejemplo, en un procesador de texto, el caso de uso base: "Redactar documento", se puede ver interrumpido en cualquier momento si el usuario activa las acciones: "Poner en negrita", "Justificar texto", etc.
- Cuando estamos añadiendo nueva funcionalidad a unos requisitos ya cerrados. Por ejemplo, de un sistema ya desarrollado que ahora queremos ampliar. En estas situaciones podemos crear nuevos casos de uso que extiendan a uno base ya cerrado, evitando así modificarlo directamente. De hecho, se inventó "extends" con el propósito de no tocar requisitos ya cerrados.

Es preferible mantener la extensión dentro del caso de uso base siempre que sea posible. Extraeremos las extensiones como casos de uso aparte cuando:

- La extensión se produce en varios sitios (factorizar comportamiento extendido común)
- La explicación de la extensión puede hacer difícil la comprensión del caso de uso base, por ser larga y/o complicada.
- Queremos ampliar la funcionalidad o comportamiento del caso de uso base, pero sin modificarlo directamente.

Es aconsejable crear casos de uso extensión sólo cuando sea necesario, ya que son más difíciles de comprender para la gente, y también más difíciles de mantener.

NOTA: Con extends los actores tienen algo que ver con los casos de uso que se extienden. Se supone que un actor dado intervendrá tanto en el caso de uso base, como de todas sus extensiones. Sin embargo, con la relación includes, es frecuente que no haya un actor asociado con el caso que se incluye.

Plantillas de Casos de Uso

Las plantillas permiten describir los casos de uso de una manera homogénea, ordenada y estructurada, facilitando un formato de documentación común entre distintos miembros de un equipo de desarrollo, o incluso entre distintos equipos.

Existen numerosas plantillas para documentar casos de uso, pero ninguna de ellas es el modelo perfecto. En realidad, la plantilla perfecta es aquella que reúna las características de forma y contenido que más se adecuen a nuestras necesidades.
Como muestra aquí tenemos una que no es ni demasiado extensa ni demasiado escueta, y que en mi opinión contempla los elementos más importantes para la documentación de casos de uso.

<ID Caso de Uso>: <Nombre>

Características	**Descripción**
Resumen	<Breve descripción de lo que hace el Caso de Uso, el objetivo que logra>
Pre-Condiciones	<Lo que debe cumplirse para ejecutar el Caso de Uso>
Post-Condiciones	<Lo que debe cumplirse una vez finalizado el Caso de Uso con éxito>
Actor Principal	<Nombre del Rol>: <Descripción del Rol del Actor que está interactuando con el Sistema>
Actores Secundarios	<Aquí podemos incluir, por ejemplo, a los actores de soporte>

Curso Normal (Escenario normal)

Paso	**Actor**	**Descripción**
<Paso Nº>	<Nombre del Actor implicado>	<Descripción de lo que ocurre en el paso>

Cursos Alternativos y Extensiones del Curso Normal

Paso	**Variación**	**Descripción**
<Paso Nº>	<Cuál es la condición que desencadena la extensión>	<Descripción de lo que ocurre en la variación o extensión o variante>

Puntos Abiertos

Punto	Descripción
<ID Pto>	<Descripción de las dudas o indefiniciones existentes en el Caso de Uso>

CONSEJOS Y RECORDATORIOS IMPORTANTES

1) No olvidar la perspectiva del usuario
Recordar que, al identificar y describir los casos de uso, es importante mantener una perspectiva cercana al usuario.

2) No perder de vista la frontera del sistema
Es importante definir la frontera del sistema para distinguir lo que es interno o externo a él, así como sus responsabilidades.

3) No confundir un paso o transacción individual como un caso de uso en sí mismo
Salvo cuando por cuestiones de simplificación y/o ayuda a la comprensión se haga necesario.

4) Identificar los verdaderos objetivos del usuario
Hay que distinguir los verdaderos objetivos del usuario de las interacciones obvias del usuario con el sistema. De esta manera podremos siempre descubrir maneras alternativas de resolver un mismo objetivo.

5) Evitar una excesiva descomposición
A menudo se invierte mucho esfuerzo en la estructuración de los casos de uso. Es común tener un caso de uso que contempla bastante lógica, e intentar por todos los medios llegar a un nivel de granularidad mayor, que consideramos más apropiado. Para ello, tomamos ese “super caso de uso” y lo descomponemos en varios sub-casos mediante relaciones include o extends (principalmente includes). A su vez vamos descomponiendo de igual forma cada uno de esos sub-casos, hasta obtener casos de uso “elementales”. Lo que en realidad estaremos haciendo es una “descomposición funcional”, que es precisamente la antítesis de la orientación a objetos. Además, un excesivo nivel de detalle en la estructura de casos de uso no es necesario en las fases tempranas del desarrollo, y sin embargo hacerlo nos costará un tiempo muy útil para otras tareas importantes.

6) Evitar una excesiva abstracción
Un buen análisis es el que consigue reducir un problema complejo en unas cuantas abstracciones sencillas, convirtiéndolo en algo manejable. Ese es nuestro objetivo como analistas. Sin embargo, abstraer excesivamente los casos de uso puede conducirnos a un punto en el que el usuario tenga dificultades para entender lo que queremos comunicarle, y eso no hace más que desvirtuar uno de los objetivos principales de los casos de uso: “La comunicación con el usuario”, además de haber invertido quizá demasiado tiempo en llegar a ese nivel de abstracción.

7) No escribir los casos de uso demasiado escuetos
En la descripción de casos de uso es siempre preferible extenderse a quedarse corto. Hemos de recordar que un caso de uso debe expresar con suficiente claridad las acciones del usuario y las respuestas del sistema a estas acciones. Evitar un texto excesivamente conciso es especialmente importante si las descripciones de los casos de uso van a ser aprovechadas

como base para el manual de usuario. Tampoco es conveniente extendernos demasiado, porque podríamos hacer el documento muy pesado y perder la atención de los lectores.

8) No omitir los posibles cursos alternativos
Aunque el curso principal de un caso de uso es generalmente más fácil de identificar, eso no significa que debamos obviar los cursos alternativos que puedan darse, y más cuando estos reflejen cuestiones importantes. El esfuerzo invertido en los cursos alternativos en una fase temprana como es la construcción de casos de uso, se verá recompensado a la hora de codificar, ya que tendremos identificados los problemas potenciales, y resuelta la lógica de su tratamiento. (No hacerlo provocará que el desarrollador decida por sí mismo la solución a aplicar, casi siempre influido por la facilidad de implementación y no por el modo más conveniente).

9) No complicarse la vida decidiendo si usar "includes" o "extends"
La diferencia entre los dos tipos de relaciones de casos de uso a veces resulta tan sutil que no es fácil decidir cuál de los dos aplicar. El consejo es que no perdamos un mes devanándonos los sesos con esta cuestión. El tipo de recurso que empleemos no debe preocuparnos tanto como para congelar nuestro progreso. Es preferible elegir un único mecanismo con el que nos sintamos confortables, y mantenerlo hasta el final, a tener los dos pero empleados en situaciones muy parecidas, lo que provocará confusión.

10) Mantener separadas las cuestiones sobre la Interfaz de Usuario
Debemos asegurarnos de que lo que escribamos refleje únicamente requisitos funcionales del sistema, no simplemente las acciones del usuario manejando la interfaz de usuario. Para la descripción de esta, podemos emplear otro documento de casos de uso específico, prototipos Html, o cualquier otro medio que se nos ocurra. Pero es importante que la descripción de los requisitos funcionales no vaya vinculada a los requisitos de "usabilidad" o estética de la interfaz de usuario. Son aspectos del sistema bien diferenciados que deben tratarse por separado. No hacerlo puede suponer, entre otras cosas, que el documento se alargue excesivamente y que se debiliten los requisitos, haciendo que un cambio en la interfaz de usuario implique un cambio en la funcionalidad.

BIBLIOGRAFÍA

Booch, Jacobson, Rumbaugh, El Lenguaje Unificado de Modelado. Addison-Wesley, 1999.

Graig Larman, UML y Patrones. Prentice Hall, 1999.

Alistair CockBurn, Writing Effective Use Cases. Editorial Addison-Wesley, 2001

Martin Fowler, Use and Abuse Cases: Artículo de la columna "Methods in Practice" en la revista "Distributed Computing". Disponible en: http://martinfowler.com/articles/abuse.pdf

Top Ten Use Case Mistakes: White Paper de Febrero de 2001, del sitio web "Software Development Online". http://www.sdmagazine.com/documents/s=815/sdm0102c/

ENLACES DE INTERÉS

http://www.pols.co.uk/usecasezone
http://martinfowler.com
http://members.aol.com/acockburn
http://www.vico.org

Anexo H. RAD – Rapid Application Development

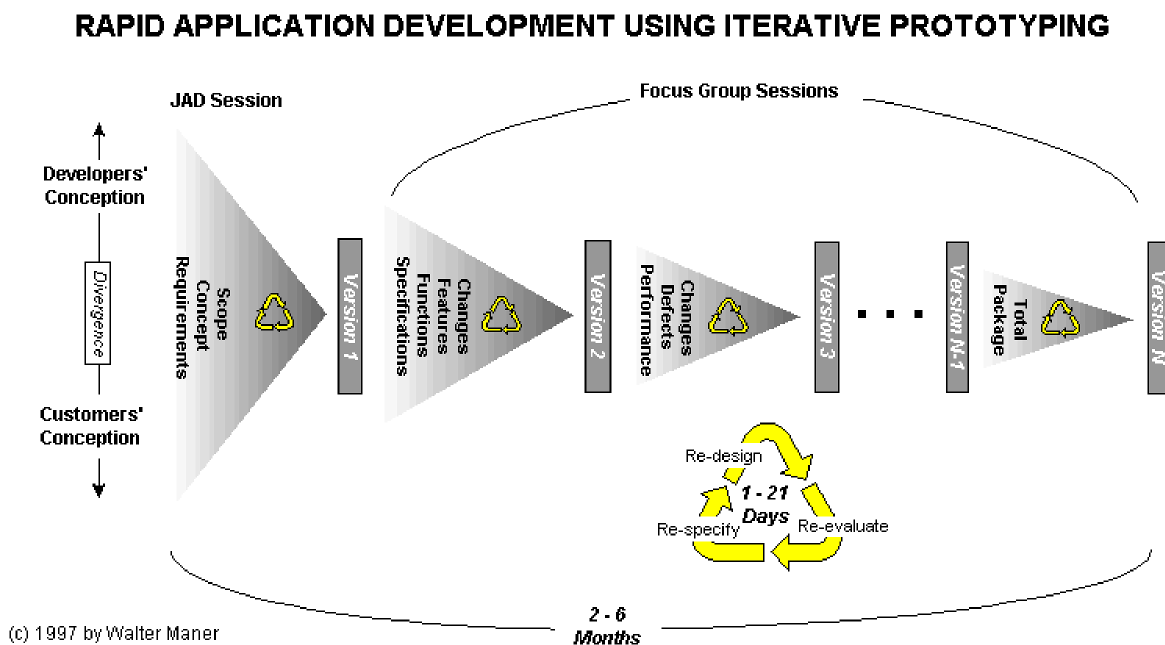
1) DEFINITION

RAD is a software development process that allows usable systems to be built in as little as 60-90 days, often with some compromises.

2) PRINCIPLES BEHIND THE DEFINITION

a) In certain situations, a usable 80% solution can be produced in 20% of the time that would have been required to produce a total solution.
b) In certain situations, the business requirements for a system can be fully satisfied even if some of its operational requirements are not satisfied.
c) In certain situations, the acceptability of a system can be assessed against the agreed minimum useful set of requirements rather than all requirements.

3) CYCLE



4) PROBLEMS ADDRESSED BY RAD

a) With conventional methods, there is a long delay before the customer gets to see any results.
b) With conventional methods, development can take so long that the customer's business has fundamentally changed by the time the system is ready for use.
c) With conventional methods, there is nothing until 100% of the process is finished, then 100% of the software is delivered.

5) WHY USE RAD?

BAD REASONS FOR USING RAD

a) to prevent cost overruns (RAD needs a team already disciplined in cost management)
b) to prevent runaway schedules (RAD needs a team already disciplined in time management)

GOOD REASONS FOR USING RAD

a) to converge early toward a design acceptable to the customer and feasible for the developers
b) to limit a project's exposure to the forces of change
c) to save development time, possibly at the expense of economy or product quality

6) SCHEDULE VERSUS ECONOMY VERSUS PRODUCT QUALITY

A) Tradeoffs determine the pace of development.

1) Efficient Development, balances economy, schedule, and quality
   a) Schedule -- faster than average
   b) Economy -- costs less than average
   c) Product -- better than average quality

2) Sensible RAD tilts away from economy and quality toward fastest schedule
   a) Schedule -- much faster than average
   b) Economy -- costs a little less than average
   c) Product -- a little better than average quality

3) All-out RAD "code like hell"
   a) Schedule -- fastest possible
   b) Economy -- costs more than average
   c) Product -- worse than average quality

B) For RAD, something other than schedule must be negotiable.

1) RAD has a fair chance of success if the customer will negotiate either economy or quality
2) RAD has a better chance for success if the customer will negotiate both economy and quality
3) NOTE: Negotiating quality does NOT mean accepting a higher defect rate. It means accepting a product that is less usable, less fully-featured, or less efficient.

C) So, with RAD, one or more of the following goals may be unachievable.

1) the fewest possible defects (because developers may not have the legal right to modify the source for some plug-in components)
2) the highest possible level of customer satisfaction(because secondary requirements may be sacrificed to stay on schedule)
3) the lowest development costs (because buying reusable components may cost more than building)

7) ABBREVIATED HISTORY OF RAD

Barry Boehm (spiral model) --->
Tom Gilb (evolutionary life cycle) --->
Scott Shultz (RIPP, rapid iterative productive prototyping) --->
James Martin (RAD, circa 1991)

8) CHARACTERISTICS OF RAD

A) RAD USES HYBRID TEAMS

1. Teams should consist of about 6 people, including both developers and full-time users of the system plus anyone else who has a stake in the requirements.

2. Developers chosen for RAD teams should be multi-talented "renaissance" people who are analysts, designers and programmers all rolled into one.

B) RAD USES SPECIALIZED TOOLS THAT SUPPORT ...
"visual" development
creation of fake prototypes (pure simulations)
creation of working prototypes
multiple languages
team scheduling
teamwork and collaboration
use of reusable components
use of standard APIs
version control (because lots of versions will be generated)

C) RAD USES "TIMEBOXING"
Secondary features are dropped as necessary to stay on schedule.

D) RAD USES ITERATIVE, EVOLUTIONARY PROTOTYPING

1. JAD (Joint Application Development) MEETING
   a) High-level end-users and designers meet in a brainstorming session to generate a rough list of initial requirements.
   b) Developers talk and listen
   c) Customers talk and listen
2. ITERATE UNTIL DONE
   a) Developers build / evolve prototype based on current requirements.
   b) Designers review the prototype.
   c) Customers try out the prototype, evolve their requirements.
   d) FOCUS GROUP meeting: Customers and developers meet to review product together, refine requirements, generate change requests: Developers listen, customers talk.
   e) Requirements and change requests are "timeboxed":
      - Changes that cannot be accommodated within existing timeboxes are eliminated.
      - If necessary to stay "in the box," secondary requirements are dropped.
3. NOTES
   a) Iterations require between 1 day and 3 weeks.
   b) At some stage, exploratory prototypes may evolve into operational prototypes.
   c) Focus Group Sessions
      last about 2 hours
      are led by an experienced facilitator, who keeps the group "on focus"
      by having clear goals regarding the kind of information that needs to be elicited
      by preparing an issue-oriented agenda in advance of the meeting
      by ensuring that adequate discussion is directed toward each issue
      by ensuring everyone has an adequate opportunity to participate

are followed by a report from the facilitator

9) IMPORTANT RAD CONSTRAINTS

a) "Fitness for a business purpose" must be the criterion for acceptance of deliverables.
b) All constituencies that can impact application requirements must have representation on the development team throughout the process.
c) Development teams must be empowered to make some decisions traditionally left to management.
d) End-to-end timescale must be 6 months or less.
e) Iteration must be used in such a way that the development process converges toward an acceptable business solution.
f) Prototyping must incorporate evolving requirements quickly, in real time, and gain consensus early.
g) There must be a "buy before build" bias.
h) Customers, developers and management must accept informal deliverables.
   1. Paper prototypes rather than full-scale systems
   2. Notes from user workshops rather than formal requirements documents
   3. Notes from designers' meetings rather than formal design documents
   4. PRINCIPLE: Create the minimum documentation necessary to facilitate future development and maintenance.

10) WHEN RAD WORKS AND WHEN IT DOESN'T

RAD TENDS TO WORK WHEN

a) The application will be run standalone.
b) Major use can be made of preexisting class libraries (APIs).
c) Performance is not critical.
d) Product distribution will be narrow (in-house or vertical market).
e) Project scope (macro-schedule) is constrained.
f) Reliability is not critical.
g) System can be split into several independent modules.
h) The product is aimed at a highly specialized IS (information systems) market.
i) The project has strong micro-schedule constraints (timeboxes).
j) The required technology is more than a year old.

RAD TENDS TO FAIL WHEN

a) Application must interoperate with existing programs.
b) Few plug-in components are available.
c) Optimal performance is required.
d) Product development can't take advantage of high-end IS tools (e.g., 4GLs).
e) Product distribution will be wide (horizontal or mass market).
f) RAD becomes QADAD (Quick And Dirty Application Development).
g) RAD methods are used to build operating systems (reliability target too high for RAD), computer games (performance target too high for RAD).
h) Technical risks are high due to use of "bleeding" edge technology.
i) The product is mission- or life-critical.
j) The system cannot be modularized (defeats parallelism).

11. EVALUATION OF RAD

ADVANTAGES OF RAD

a) Buying may save money compared to building
b) Deliverables sometimes easier to port (because they make greater use of high-level abstractions, scripts, intermediate code)
c) Development conducted at a higher level of abstraction (because RAD tools operate at that level)
d) Early visibility (because of prototyping)
e) Greater flexibility (because developers can redesign almost at will)
f) Greatly reduced manual coding (because of wizards, code generators, code reuse)
g) Increased user involvement (because they are represented on the team at all times)
h) Possibly fewer defects (because CASE tools may generate much of the code)
i) Possibly reduced cost (because time is money, also because of reuse)
j) Shorter development cycles (because development tilts toward schedule and away from economy and quality)
k) Standardized look and feel (because APIs and other reusable components give a consistent appearance)

DISADVANTAGES

a) Buying may not save money compared to building
b) Cost of integrated toolset and hardware to run it
c) Harder to gauge progress (because there are no classic milestones)
d) Less efficient (because code isn't hand crafted)
e) Loss of scientific precision (because no formal methods are used)
f) May accidentally empower a return to the uncontrolled practices of the early days of software development
g) More defects (because of the "code-like-hell" syndrome)
h) Prototype may not scale up, a B-I-G problem
i) Reduced features (because of timeboxing, software reuse)

j) Reliance on third-party components may sacrifice needed functionality, add unneeded functionality and create legal problems
k) Requirements may not converge (because the interests of customers and developers may diverge from one iteration to the next)
l) Standardized look and feel (undistinguished, lackluster appearance)
m) Successful efforts difficult to repeat (no two projects evolve the same way)
n) Unwanted features (through reuse of existing components)

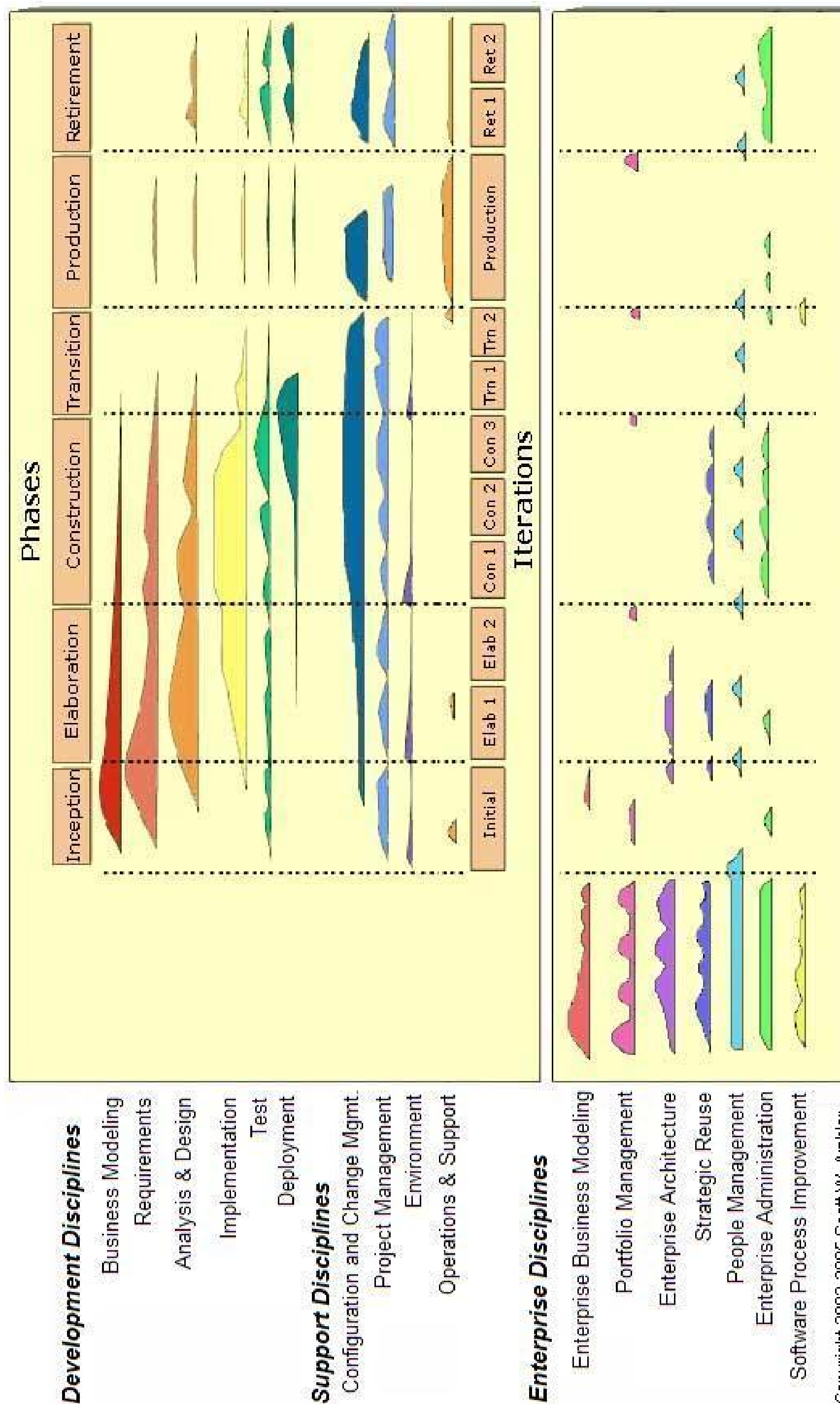
12. SUMMARY

"In order to ensure high responsiveness, projects are designed with fixed timescales, sacrificing functionality if necessary. This allows the development team to focus on the pieces of functionality that have the highest business value, and deliver that functionality rapidly. Change is often the reason for delays in application development. In long linear development processes, changes in functionality requirements or project scope, particularly after a lot of time has been invested in planning, design, development and testing, cause many months to be lost and significant expense to be incurred for redesigning and redevelopment. RAD combats scope and requirements creep by limiting the project's exposure to change -- shortening the development cycle and limiting the cost of change by incorporating it up-front before large investments are made in development and testing." – Sun Microsystems

Anexo I. EUP – Enterprise Unified Process

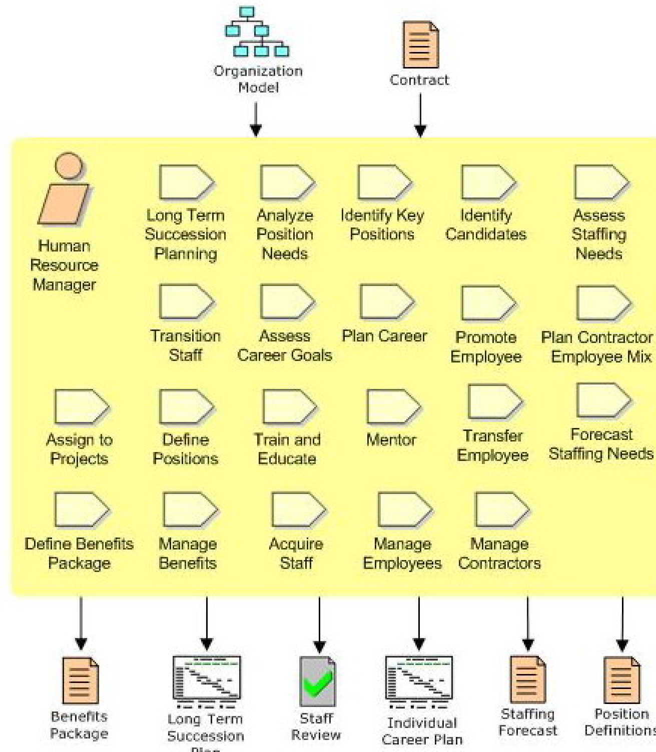
El Enterprise Unified Process (EUP) es una extensión, no un sustituto, del *Rational Unified Process* (RUP). Una organización primero debería adoptar RUP y después mirar todo el ciclo de vida del producto ayudándose con EUP. EUP añade dos nuevas fases a RUP: Producción y Retiro (*Retirement*).

EUP también añade 8 disciplinas: *Operations & Support, Enterprise Administration, Enterprise Architecture, Enterprise Business Modeling, People Management, Portfolio Management, Software Process Improvement (SPI,* y *Strategic Reuse.*



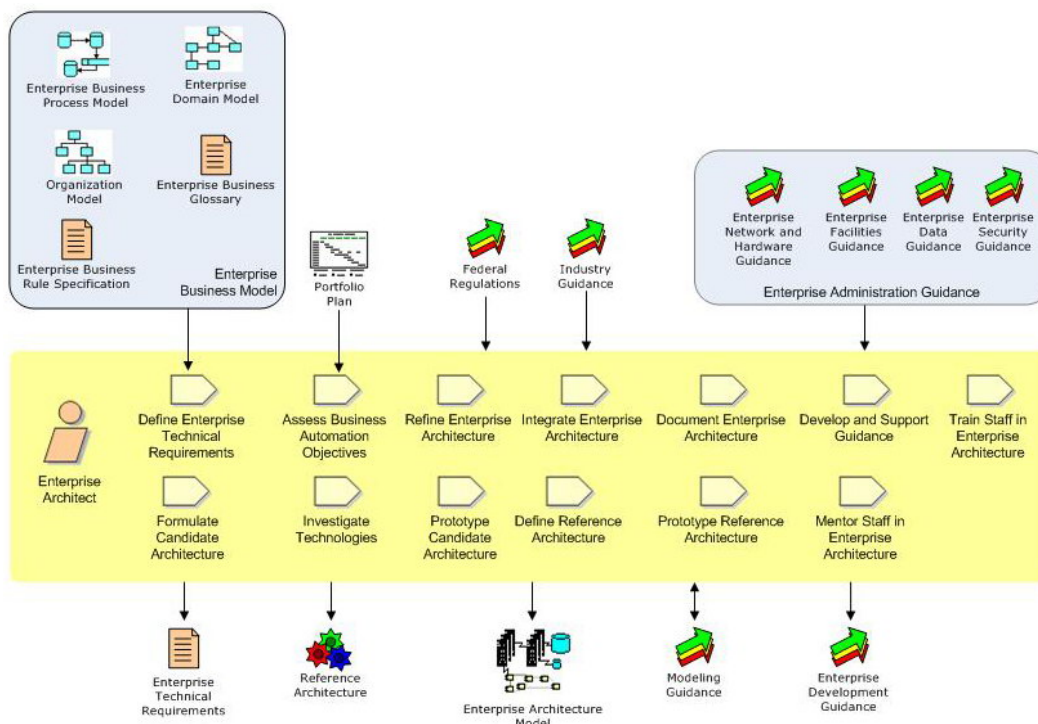
DISCIPLINA DE GESTIÓN DE PERSONAL

El objetivo es gestionar y mejorar la efectividad de los trabajadores de la organización. Esta disciplina describe los procesos de organización, supervisión, educación, motivación, etc.



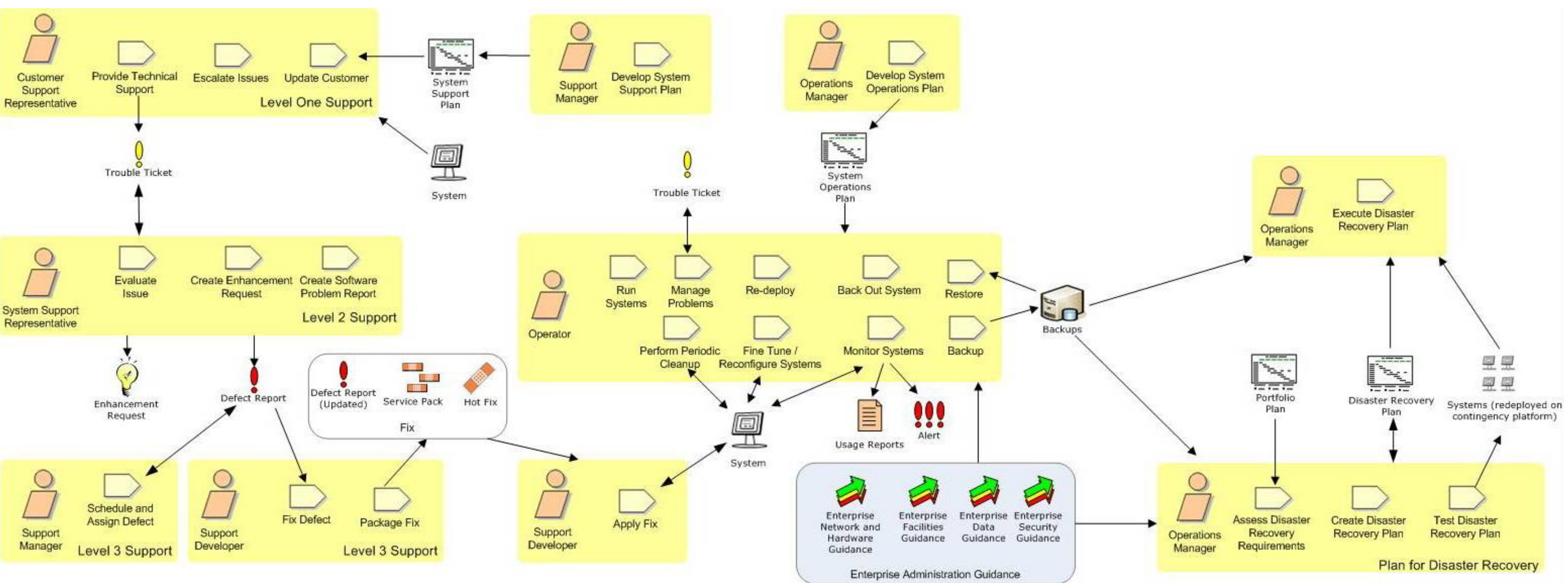
DISCIPLINA ARQUITECTURA DE EMPRESA

Engloba todos los asuntos de la empresa. Sus artefactos incluyen modelos que la definen, prototipos y modelos funcionales que demuestran cómo funciona y *frameworks* que faciliten su uso. Los arquitectos de la empresa, deberían ser miembros activos del equipo de desarrolladores de software.



DISCIPLINA DE SOPORTE Y OPERACIONES

El objetivo de esta disciplina es operar y dar soporte al software en el entorno de producción. El primer propósito es asegurar que el software funciona correctamente, que la red esté operativa, supervisada y que se puedan hacer y restaurar copias de seguridad de los datos necesarios. Se elaboran planes para casos de emergencia. El soporte está fuertemente ligado con el *feedback* de los usuarios: debe responder a sus preguntas, analizar los problemas que encuentran, recoger las peticiones para nuevas funcionalidades, y hacer y aplicar las correcciones al software.

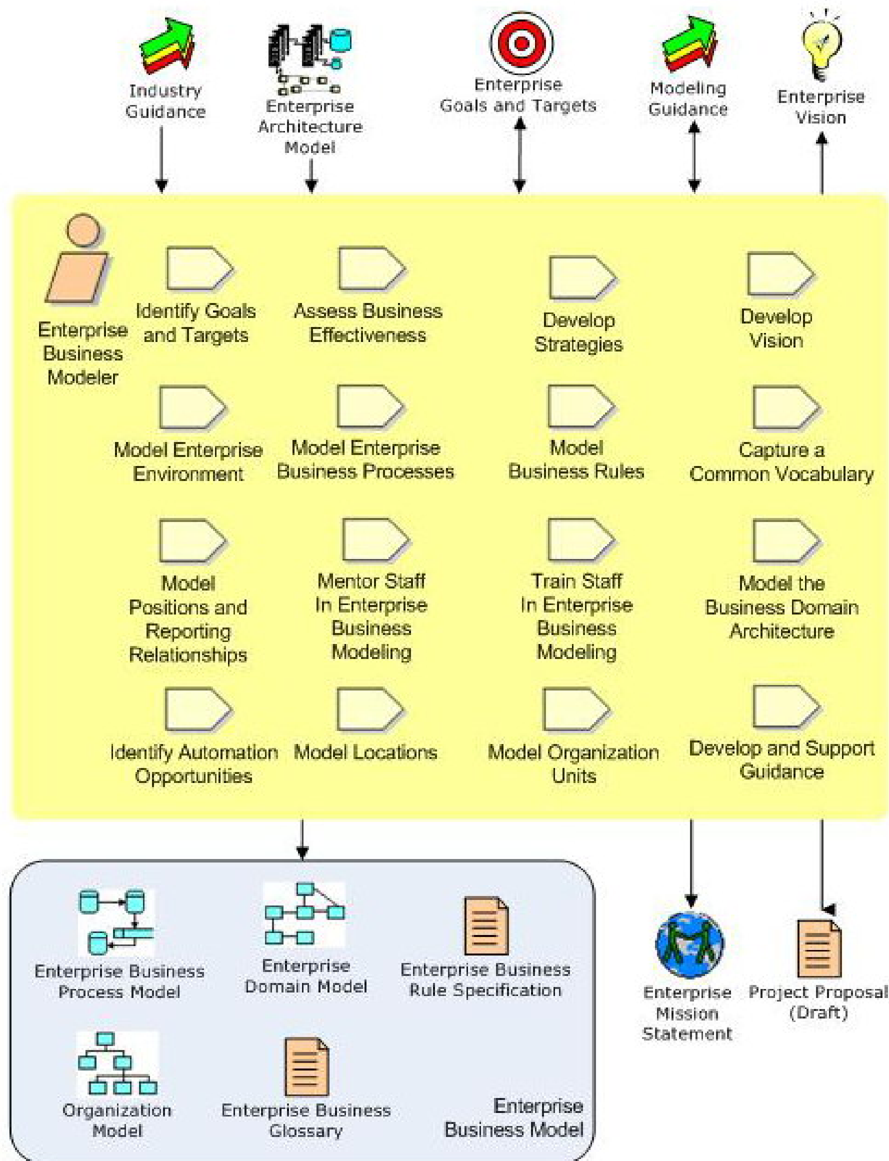


DISCIPLINA DE GESTIÓN DE CARTERA (PORTFOLIO)

En esta disciplina se pretende ayudar a mejorar la eficiencia global y la efectividad de los proyectos de la organización seleccionando y gestionando el desarrollo del software. Para conseguirlo, se busca la máxima adaptación del proyecto con los objetivos de la empresa. La cartera de software es la colección de proyectos, propuestos y en progreso, así como también los sistemas implantados en la propia organización.

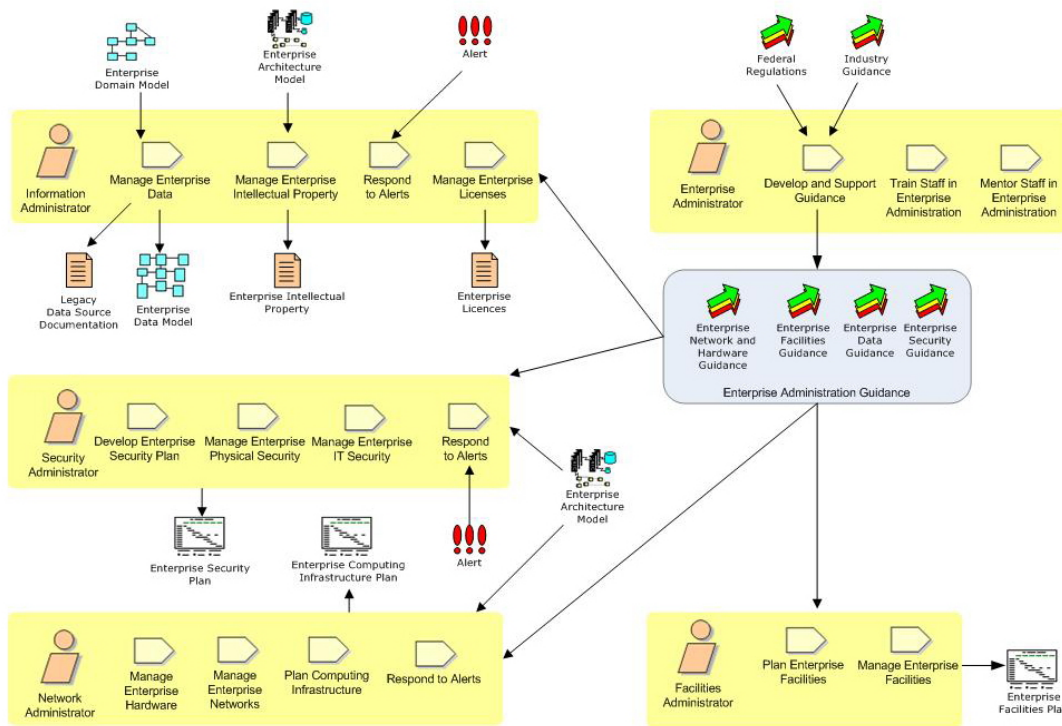
Disciplina de Modelo de negocio de la empresa

El propósito principal es explorar la estructura del negocio y los procesos de la empresa, proporcionando un entendimiento común del negocio, actividades, clientes, proveedores...



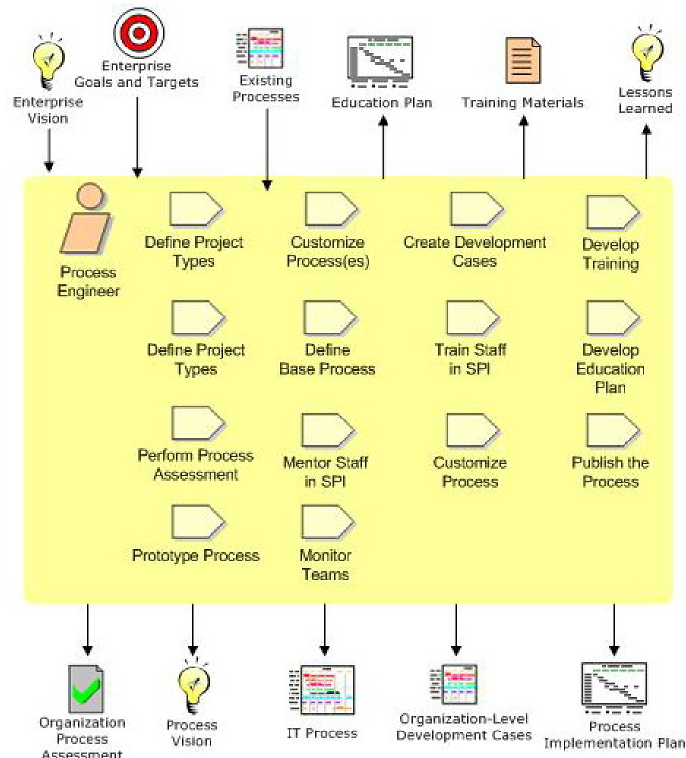
DISCIPLINA DE ADMINISTRACIÓN DE EMPRESA

Define cómo una organización crea, mantiene, gestiona y despliega las características destacadas de forma segura. El fin no es sólo añadir burocracia, sino hacer más eficiente estas actividades.



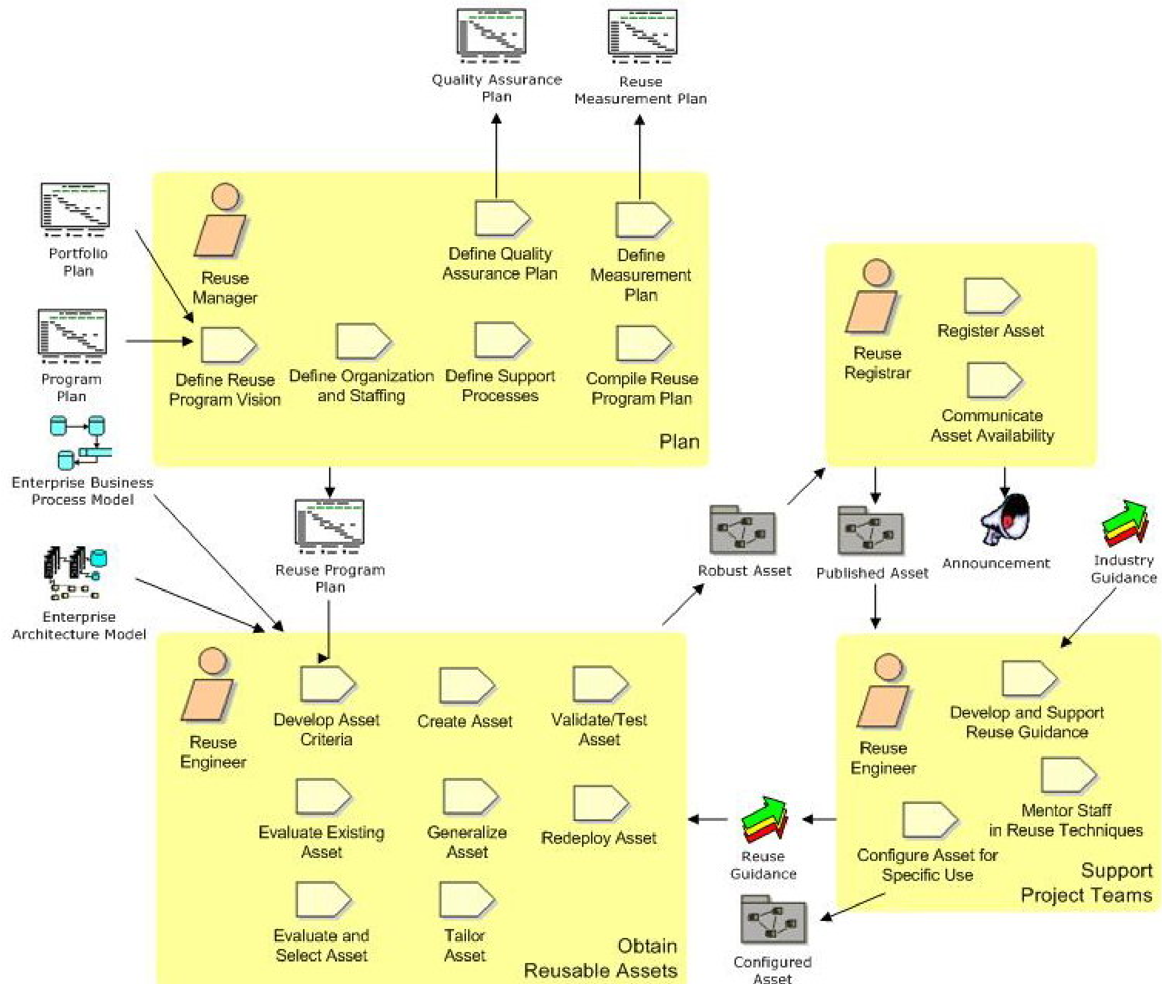
DISCIPLINA DE MEJORA DEL PROCESO DE SOFTWARE

Esta disciplina trata la necesidad de gestionar, mejorar y ayudar a los diferentes procesos a lo largo de toda la organización.



DISCIPLINA DE ESTRATEGIA DE REUTILIZACIÓN

El propósito es averiguar la mejor estrategia para poder reutilizar entre diferentes proyectos.



Anexo J. Scrum + XP, ejemplo de aplicación

Adaptive Engineering of Large Software Projects with Distributed/Outsourced Teams

Jeff Sutherland
PatientKeeper, Inc.
jeff.sutherland@computer.org

Anton Viktorov
Starsoft Development Labs
anton.viktorov@starsoftlabs.com

Jack Blount
SirsiDynix
jack@dynix.com

1. INTRODUCTION

Scrum is an Agile software development process designed to add energy, focus, clarity, and transparency to project teams developing complex systems. It leverages artificial life research [Langton 1992] by allowing teams to operate close to the edge of chaos to foster rapid system evolution. It capitalizes on robot subsumption architectures [Brooks 1991] by enforcing a simple set of rules that allows rapid self-organization of software teams to produce systems with evolving architectures. A properly implemented Scrum will increase speed of development, align individual and organization objectives, create a culture driven by performance, support shareholder value creation, achieve stable and consistent communication of performance at all levels, and enhance individual development and quality of life.

Scrum for software development teams began at Easel Corporation in 1993, where we built the first object-oriented design and analysis (OOAD) tool that incorporated round-trip engineering. In a Smalltalk development environment, code was autogenerated from a graphic design tool and changes to the code from the Smalltalk integrated development environment (IDE) were immediately reflected back into design.

We needed a development process that supported small teams where visualization of design could result immediately in working code. This led to an extensive review of the literature and the real experience from leaders of hundreds of software development projects. Key factors that influenced the introduction of Scrum at Easel Corporation were fundamental problems inherent in software development [DeGrace and Stahl 1990].

- Requirements are not fully understood before the project begins,
- Users know what they want only after they see an initial version of the
- software,
- Requirements change often during the software construction process,
- And new tools and technologies make implementation strategies unpredictable

“All-at-Once” models of software development uniquely fit object-oriented implementation of software and help resolve these challenges. They assume that creation of software involves simultaneously working on requirements, analysis, design, coding, and testing, then delivering the entire system all at once.

1.1. “ALL-AT-ONCE” DEVELOPMENT MODELS

The simplest “All-at-Once” model is a single super-programmer creating and delivering an application from beginning to end. This is the fastest way to deliver a product that has good internal architectural consistency and is the “hacker” model of implementation. For example, in a predecessor to the first Scrum, one individual spent two years writing every line of code

for the Matisse object database [Matisse Software 2003] used to drive $10B nuclear reprocessing plants worldwide. At less than 50,000 lines of code, the nuclear engineers said it was the fastest and most reliable database ever benchmarked for nuclear plants. IBM has shown that a variant of this approach called the Surgical Team is the most productive software development process [Brooks 1995]. The Surgical Team approach has a fatal flaw in that there are at most one or two individuals even in a large company that can execute this model. For example, it took three years for an outstanding team of developers to understand the conceptual elegance of the Matisse object server well enough to maintain it. The single-programmer model does not scale well to large projects.

The next level of "All-at-Once" development is handcuffing two programmers together, as in pair programming in the eXtreme Programming paradigm [Beck 1999]. Here, two developers working at the same terminal deliver a component of the system together. This has been shown to deliver better code (usability, maintainability, flexibility, extendibility) faster than two developers working individually [Wood and Kleb 2003]. The challenge is to achieve a similar productivity effect with more than two people.

Scrum, a scalable, team-based "All-at-Once" model, was motivated by the Japanese approach to team-based new product development combined with simple rules to enhance team self-organization as used in the Brooks subsumption architecture [Brooks 1991]. At Easel we were already using an iterative and incremental approach to building software [Larman 2004]. It was implemented in slices in which an entire piece of fully integrated functionality worked at the end of an iteration. What intrigued us was Takeuchi and Nonaka's description of the team-building process for setting up and managing a Scrum [Takeuchi and Nonaka 1986]. The idea of building a self empowered team in which a daily global view of the product caused the team to self organize seemed like the right idea. The approach to managing the team, which had been so successful at Honda, Canon, and Fujitsu, also resonated with the systems thinking approach promoted by Professor Senge at MIT [Senge 1990].

1.2. HYPERPRODUCTIVITY IN SCRUM

The hyperproductive state achieved in 1993-1994 during the first Scrum was the result of three primary factors. The first was the Scrum process itself, characterized by 15 minute daily meetings where each person answers three questions - what did you accomplish yesterday, what will you do today, and what impediments are getting in your way? This is now part of the definitive Scrum organizational pattern [Beedle, Devos et al. 1999]. Second, the team implemented all XP engineering processes [Beck 1999] including pair programming, continuous builds, and aggressive refactoring. And third, the team systematically stimulated rapid evolution of the software system. Development tasks, originally planned to take days, could often be accomplished in hours using someone else's code as a starting point.

One of the interesting complexity phenomena of the first Scrum was an observed "punctuated equilibrium" effect [Gould 2002]. This occurs in biological evolution when a species is stable for long periods of time and then undergoes a sudden jump in capability. Dennis Hillis simulated this effect on an early super-computer, the Connection Machine.

"The artificial organisms in Hillis's particular world evolved not by steady progress of hill climbing but by the sudden leaps of punctuated equilibrium… with artificial organisms Hillis had the power to examine and analyze the genotype as easily as the realized phenotypes… While the population seemed to be resting during the periods of equilibrium… the underlying genetic makeup was actively evolving. The sudden increase in fitness was no more an instant

occurrence than the appearance of a newborn indicates something springing out of nothing; the population seemed to be gestating its next jump. Specifically, the gene pool of the population contained a set of epistatic genes that could not be expressed unless all were present; otherwise the alleles for these genes would be recessive." [Levy 1993]

A fully integrated component design environment leads to unexpected, rapid evolution of a software system with emergent, adaptive properties resembling the process of punctuated equilibrium. Sudden leaps in functionality resulted in earlier than expected delivery of software in the first Scrum.

This aspect of self-organization is now understood as a type of Set-Based Concurrent Engineering (SBCE) practiced at Toyota [Sobek, Ward et al. 1999]. Developers consider sets of possible solutions and gradually narrow the set of possibilities to converge on a final solution. Decisions on how and where to implement a feature in a set of components was delayed until the last possible moment. The most evolved component is selected "just in time" to absorb new functionality, resulting in minimal coding and a more elegant architecture. Thus emergent architecture, a core principle in all Agile processes, is not random evolution. Properly implemented, it is an SBCE technique viewed as a best business practice in some of the world's leading corporations.

2. THE SIRSIDYNIX DISTRIBUTED SCRUM

The hyperproductive state achieved by many Scrum teams has increased productivity by an order of magnitude. The question for this paper is whether a large, distributed, outsourced team can achieve the same effect.

Many U.S., European, or Japanese companies outsource software development to Eastern Europe, Russia, or the Far East. Typically, remote teams operate independently and communication problems limit productivity. While there is a large amount of research literature on project management, distributed development, and outsourcing strategies as isolated domains, there are few detailed studies of best project management practices on large systems that are both distributed and outsourced.

Best current Scrum practice is for local Scrum teams at all sites to synchronize once a day via a Scrum of Scrums meeting. Here we describe something rarely seen on large, distributed teams. At SirsiDynix, all Scrum teams consist of developers distributed across different sites. Any team member from any site can work on any team task. While some Agile companies operate in this geographically transparent manner on a small scale, SirsiDynix has been successful in using fully integrated Scrum teams with over 50 developers in the U.S., Canada, and Russia. They have created a new implementation of platform and system architecture for a complex Integrated Library System (ILS). An ILS system can best be compared to a vertical market ERP system with a public portal interface used by more than 200 million people. New best practices for distributed Scrum seen on this project consist of (1) daily Scrum meetings of all developers from multiple sites, (2) daily meetings of Product Owner team (3) hourly automated builds from one central repository, (4) no distinction between developers at different sites on the same team, (5) and seamless integration of XP practices like pair programming with Scrum. While similar practices have been implemented on small distributed Scrum teams [Sutherland 2001] this is the first documented project that demonstrates Scrum hyperproductivity for large distributed/outsourced teams building complex enterprise systems.

3. DISTRIBUTED TEAM MODELS

Here we consider three distributed Scrum models commonly observed in practice.

- Isolated Scrums - Teams are isolated across geographies. In many cases off-shore teams are not cross-functional. In some cases, offshore teams are non-Scrum teams.
- Distributed Scrum of Scrums - Scrum teams are isolated across geographies and integrated by a Scrum of Scrums that means regularly across geographies.
- Totally Integrated Scrums - Scrum teams are cross-functional with members distributed across geographies. In the SirsiDynix case, the Scrum of Scrums was localized as all ScrumMasters were in Utah.

Most outsourced development efforts use a degenerative form of the Isolated Scrums model where outsourced teams are not cross-functional and not Agile. Requirements may be created in the U.S. and developed in Dubai, or development may occur in Germany and quality assurance in India. The authors have experienced cross cultural communication problems compounded by disparities in work types in many companies around the world where they were directly responsible for development projects. In the worst case, outsourced teams are not using Scrum and their productivity is typical of inhouse waterfall projects further delayed by lag time induced by cross continent communications.

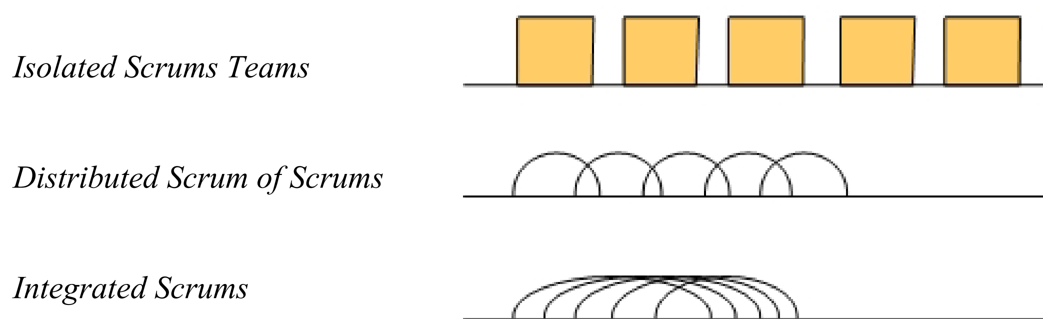


Figure 1. Strategies for distributed Scrum teams [Takeuchi and Nonaka 2004].

The latest thinking in the Project Management Institute Guide to the Project Management Body of Knowledge (PMBOK) models is a degenerative case of isolated non-Scrum teams [Nidiffer and Dolan 2005]. This is a spiral waterfall methodology which layers the Unified Modeling Language (UML) and the Rational Unified Process (RUP) onto teams which are not cross-functional [Zanoni and Audy 2003]. It partitions work across teams, creates teams with silos of expertise, and incorporates a phased approach laden with artifacts that violate the principles of lean development [Poppendieck 2005].

Best practice recommended by the Scrum Alliance is a Distributed Scrum of Scrums model. This model partitions work across cross-functional, isolated Scrum teams while eliminating most dependencies between teams. Scrum teams are linked by a Scrum-of-Scrums where ScrumMasters (team leaders/project managers) meet regularly across locations. This encourages communication, cooperation, and cross fertilization.

An Integrated Scrums model has all teams fully distributed and each team has members at multiple locations. While this appears to create communication and coordination burdens, the daily Scrum meetings help to break down cultural barriers and disparities in work styles. On large enterprise implementations, it can organize the project into a single whole with a rapidly evolving global code base. The virtual nature of this approach provides location transparency

and creates performance characteristics similar to a small co-located team. The hyperproductive Web team at IDX Systems during 1996-2000 achieved ten times the performance of the industry average for teams of large systems [Sutherland 2001]. The SirsiDynix model outlined in this paper is a good example of best practices for Integrated Scrums. This may be the most productive distributed team ever documented, delivering a large Java enterprise system with more than one million lines of code.

4. SirsiDynix Case Study

4.1. SirsiDynix Background

SirsiDynix provides global technology solutions for libraries to assist people in discovering and using knowledge, resources and other valuable content for their educations, jobs and entertainment. In concert with key industry partners, SirsiDynix supports this strategic role for libraries by offering a comprehensive integrated suite of technology solutions for improving the internal productivity of libraries and enhancing their capabilities for meeting the needs of people and communities. SirsiDynix has approximately 4,000 library and consortia clients, serving more than 200 million people through more than 20,000 library outlets in the Americas, Europe, Africa, the Middle East and Asia-Pacific.

Jack Blount, President and CEO of Dynix and now CTO of the merged SirsiDynix company, negotiated an outsource agreement with StarSoft who staffed the project with more than 20 qualified engineers in less than 60 days. Significant development milestones were completed in just a few weeks and all joint development projects were efficiently tracked and continue to be on schedule.

4.2. StarSoft Background

StarSoft Development Labs, Inc. is a fast-growing software outsourcing service provider in Russia and Eastern Europe. Headquartered in Cambridge, Massachusetts, USA, StarSoft operates development centers in St. Petersburg, Russia and Dnepropetrovsk, Ukraine, employing over 450 professionals. StarSoft has experience handling development efforts varying in size and duration from just several engineers working for a few months to large-scale projects involving dozens of developers and spanning over several years. A CMM Level 3 company, StarSoft successfully uses Agile development methodologies for the benefits of its clients.

5. Hidden Costs of Outsourcing

The hidden costs of outsourcing can be significant beginning with startup costs. Barthelemy [Barthelemy 2001] surveyed 50 companies and found that 14% of outsourcing operations were failures. In the remainder, costs of transitioning to a new vendor often canceled out most of the company's savings from lower labor costs in other countries. The average time from evaluating outsourcing to beginning of vendor performance was 18 months for projects smaller than the SirsiDynix contract. As a result, the MIT Sloan Management Review counsels readers not to outsource critical IT functions and to spend more time planning. The German Institute for Economic Research analyzed 43,000 German manufacturing firms from 1992-2000 and found that outsourcing services led to poor corporate performance, while outsourcing production helped [Gorzig and Stephan 2002]. While this is a manufacturing study rather than software development, it suggests that outsourcing core development may provide gains not seen otherwise.

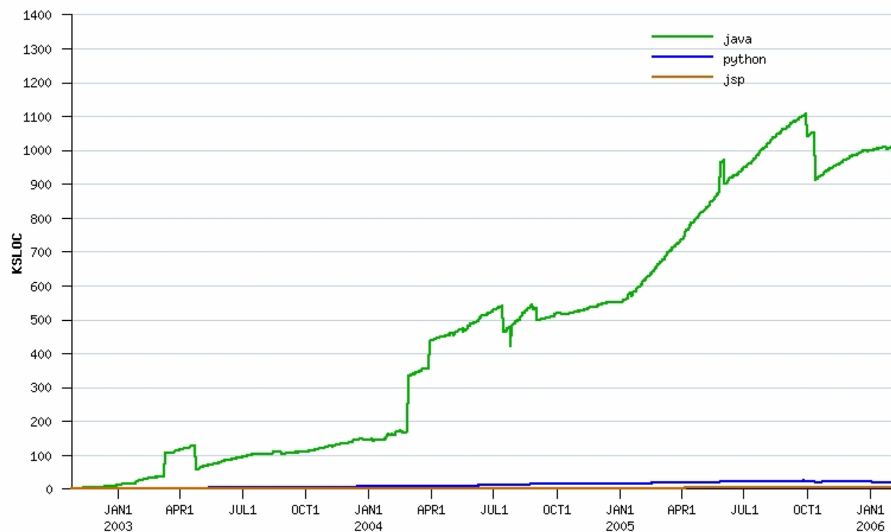


Figure 2. SirsiDynix lines of new Java code in thousands from 2000-2006.

Large software projects are very high risk. The 2003 Standish Chaos Report show success rates of only 34%. 51% of projects are over budget or lacking critical functionality. 15% are total failures [StandishGroup 2003].

SirsiDynix sidestepped many of the hidden costs, directly outsourced primary production and used Integrated Scrums to control the risk. The goals of both increasing output per team member and increasing overall output by increasing team size were achieved. Production velocity more than doubled when they increased the size of the 30 person North American development team and added 26 people from StarSoft on 1 December 2005.

6. Intent of the Integrated Scrums Model

An Agile company building a large product and facing time-to-market pressure needs to quickly double or triple productivity within a constrained budget. The local talent pool is not sufficient to expand team size and salary costs are much higher than outsourced teams. On the other hand, outsourcing is only a solution if Agile practices are enhanced by capabilities of the outsourced teams. The primary driver is enhanced technical capability resulting in dramatically improved throughput of new application functionality. Cost savings are a secondary driver.

7. Context

Software complexity and demands for increased functionality are exponentially increasing in all industries. When the lead author of this paper flew F-4 aircraft in combat in 1967, 8% of pilot functions were supported by software. In 1982, the F16 software support was 45%, and by 2000, the F22 was augmented 80% of pilot capabilities with software [Nidiffer and Dolan 2005]. Demands for ease of use, scalability, reliability, and maintainability increase with complexity.

SirsiDynix was confronted with the requirement to completely re-implement a legacy library system with over 12,500 installed sites across the globe. The large number of developers required over many years in the midst of a changing business environment threatened to obsolete many feature requirements in the middle of the project. To complicate matters

further, the library software industry was in a consolidating phase. Dynix started the project in 2002 and merged with Sirsi in 2005 to form SirsiDynix.

Fortunately, Dynix started the project with a scalable Agile process that could adapt to changing requirements throughout the project. Time to market demanded more than doubling of output. That could only happen by augmenting resources with Agile teams. StarSoft was selected because of their history of successful XP implementations and their experience with systems level software.

The combination of high risk, large scale, changing market requirements, merger and acquisition business factors, and the SirsiDynix experience with Scrum combined with StarSoft success with XP led them to choose an Integrated Scrums implementation. Jack Blount's past experience with Agile development projects at US Data Authority, TeleComputing and JD Edwards where he had used Isolated Scrums and Distributed Scrum of Scrums models was a key factor in his decision to structure the project as Integrated Scrums.

8. FORCES

8.1. COMPLEXITY DRIVERS

The Systems and Software Consortium (SSCI) of large defense contractors has outlined drivers, constraints, and enablers that force organizations to invest in real-time project management information systems. Scalable Scrum implementations with minimal tooling are one of the best real-time information generators in the software industry.

SSCI complexity drivers are described as [Nidiffer and Dolan 2005]:

- Increasing problem complexity shifting focus from requirements to objective capabilities that must be met by larger teams and strategic partnerships.
- Increasing solution complexity which shifts attention from platform architectures to enterprise architectures and fully integrated systems.
- Increasing technical complexity from integrating stand alone systems to integrating across layers and stacks of communications and network architectures.
- Increasing compliance complexity shifting from proprietary to open standards.
- Increasing team complexity shifting from a single implementer to strategic teaming and mergers and acquisitions.

SirsiDynix faced all of these issues. Legacy products were difficult to sell to new customers. They needed a new product with complete functionality for the library enterprise based on today's technologies that was highly scalable, easily expandable, and used the latest computer and library standards.

The Horizon 8.0 architecture supports a wide variety of users from publication acquisition to cataloging, searching, reserving, circulating, or integrating information from local and external resources. The decision was made to use Java with J2EE, a modular design, database independency, maximum use of free platforms and tools, and wide support of MARC21, UNIMARC, Z39.50 and other ILS standards.

The project uses a three-tier architecture and uses Hibernate as a database abstraction layer. Oracle 10g, MS SQL, and IBM DB2 support is provided. The JBoss 4 Application server is used with a Java GUI Client with WebStart bootstrap. It is a crossplatform product supporting

MS Windows 2000, XP, 2003, Red Hat Linux, and Sun Solaris. Built-in multi-language support has on-the-fly resource editing for ease of localization. Other key technologies are JAAS, LDAP, SSL, Velocity, Xdoclet, JAXB, JUnit, and Jython.

8.2. Top Issues in Distributed Development

The SSCI has carefully researched top issues in distributed development [Nidiffer and Dolan 2005], all of which had to be handled by SirsiDynix and StarSoft.

- Strategic: Difficult leveraging available resources, best practices are often deemed proprietary, are time consuming and difficult to maintain.
- Project and process management: Difficulty synchronizing work between distributed sites.
- Communication: Lack of effective communication mechanisms.
- Cultural: Conflicting behaviors, processes, and technologies.
- Technical: Incompatible data formats, schemas, and standards.
- Security: Ensuring electronic transmission confidentiality and privacy.

The unique way in which SirsiDynix and StarSoft implemented an Integrated Scrums model carefully addressed all of these issues.

9. Solution: Integrated Scrums

There are three roles in a Scrum: the Product Owner, the ScrumMaster, and the Team. SirsiDynix used these roles. Scrum itself solves the strategic distribution problem of building a high velocity, real-time reporting organization with an open source process that is easy to implement and low-overhead to maintain [Sutherland 2005].

For large programs, a chief ScrumMaster to run a Scrum of Scrums and a chief Product Owner to centrally manage a single consolidated and prioritized product backlog is essential. SirsiDynix colocated the Scrum of Scrums and the Product Owner team in Utah.

9.1. Team Formation

The second major challenge is process management, particularly synchronizing work between sites. This was achieved by splitting teams across sites and fine tuning daily Scrum meetings

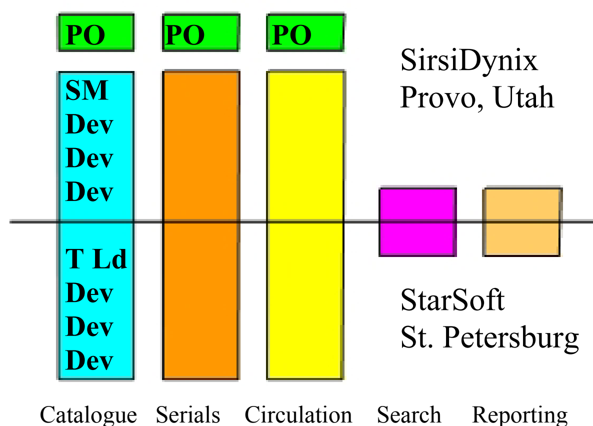


Figure 3. Scrum teams split across sites. PO=Product Owner, SM=ScrumMaster, TLd=Technical Lead.

Teams at SirsiDynix were split across the functional areas needed for a integrated library system. Half of a Scrum team is typically in Provo, Utah, and the other half in St. Petersburg. There are typically 3-5 people on the Utah part of the team and 4 or more on the St. Petersburg portion of the team. The Search and

Reporting Teams are smaller. There are smaller numbers of team members in Seattle, Denver, St. Louis, and Waterloo, Canada.

9.2. SCRUM MEETINGS

Teams meet across geographies at 7:45am Utah time which is 17:45 St. Petersburg time. The team has found it necessary to answer the three Scrum questions in writing and distribute the answers by email before the Scrum meeting. This shortens the time needed for teleconference on the joint meeting and helps overcome any language barriers. Each individual reports on what they did since the last meeting, what they intend to do next, and what impediments are blocking their progress.

Email exchange on the three questions before the daily Scrum teleconference was used throughout the project to enable phone meetings to proceed more smoothly and efficiently. These daily team calls helped the people in Russia and the U.S. learn to understand each other. Most outsourced development projects do not hold formal daily calls and the communication bridge is never formed.

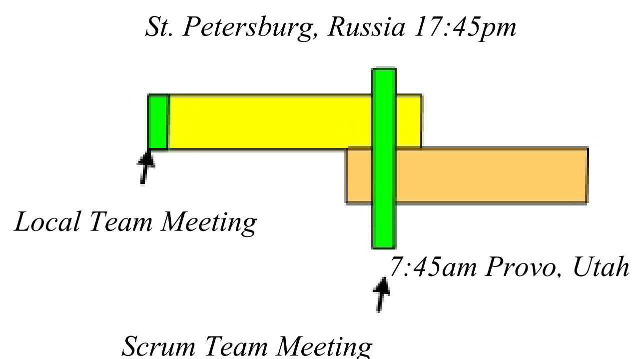


Figure 5. Scrum Team meetings

Local sub-teams have an additional standup meeting at the beginning of the day in St. Petersburg. Everyone is using the same process and technologies and daily meetings coordinate activities within the teams.

ScrumMasters are all in Provo, Utah or Waterloo, Canada, and meet in a Scrum of Scrums every Monday morning. Here work is coordinated across teams. Architects are directly allocated to production Scrum teams and all located in Utah. An Architecture group also meets on Monday after the Scrum of Scrums meeting and controls the direction of the project architecture through the Scrum meetings. A Product Owner resident in Utah is assigned to each Scrum team. A chief Product Owner meets regularly with all Product Owners to assure coordination of requirements.

SirsiDynix achieved strong central control of teams distributed across geographies by centrally locating ScrumMasters, Product Owners, and Architects. This enabled them to get consistent performance across all distributed teams.

9.3. SPRINTS

Sprints are two weeks on the SirsiDynix project. There is a Sprint planning meeting that is the same as an XP release planning meeting in which requirements from User Stories are broken down into development tasks. Most tasks require a lot of questions from the Product Owners and tasks occassionally take more time than initial estimates.

The lag time for Utah Product Owner response to questions on User Stories forces multitasking in St. Petersburg and this is not the ideal situation. Sometimes new tasks are discovered after querying Product Owners during the Sprint with additional feature details.

Code is feature complete and demoed at the end of each Sprint. If it meets the Product Owner's functional requirement, it is considered done. It is not deliverable code and SirsiDynix wants to strengthen its definition of "done" to include testing. Failure to do this allows work in progress to cross Sprint boundaries, introducing wait times and greater risk into the project.

9.4. PRODUCT SPECIFICATIONS

Requirements are in the form of User Stories used in many Scrum and XP implementations. Some of them are lengthy and detailed, others are not. A lot of questions result after receiving the document in St. Petersburg which are resolved by in daily Scrum meetings, by instant messaging, or by email.

Story for Simple Renewals Use Case - Patron brings item to staff to be renewed. Patron John Smith checked out "The Da Vinci Code" the last time he was in the library. Today he is back in the library to pick up something else and brings "The Da Vinci Code" with him. He hands it to the staff user and asks for it to be renewed. The staff user simply scans the item barcode at checkout, and the system treats it as a renewal since the item is already checked out to John. This changes the loan period (extends the due date) for the length of the renewal loan. Item and patron circulation history are updated with a new row showing the renewal date and new due date. Counts display for the number of renewals used and remaining. The item is returned to Patron John Smith.

Assumptions:

- Item being renewed is currently checked out to the active patron
- No requests or reservations outstanding
- Item was not overdue
- Item does not have a problem status (lost, cr, etc)
- No renew maximums have been reached
- No block/circ maximums have been reached
- Patron's subscriptions are active and not within renewal period
- No renewal charges apply
- No recalls apply
- Renewal is from Check Out (not Check In)
- Staff User has renewal privileges

Verification (How to verify completion):

- Launch Check Out
- Retrieve a patron who has an item already checked out but not yet overdue
- Enter barcode for checked out item into barcode entry area (as if it is being checked out), and press <cr>.
- System calculates new due date according to circ rules and agency parameters.
- The renewal count is incremented (Staff renewal with item)
- If user views "Circulation Item Details", the appropriate Renewals information should be updated (renewals used/remaining)

- Cursor focus returns to barcode entry area, ready to receive next scan (if previous barcode is still displayed, it should be automatically replaced by whatever is entered next)
- A check of the item and patron circulation statistics screens show a new row for the renewal with the renewal date/time and the new due date.

For this project, St. Petersburg staff liked a detailed description because the system is a comprehensive and complex system designed for specialized librarians. As a result, there is a lot of knowledge that needs to be embedded in the product specification.

The ways libraries work in St. Petersburg are very different than English libraries. Russian libraries operate largely via manual operations. While processes look similar to English libraries on the surface, the underlying details are quite different. Therefore, user stories do not have sufficient detail for Russian programmers.

9.5. TESTING

Developers write unit tests. The Test team and Product Owners do manual testing. An Automation Test team in Utah creates scripts for an automated testing tool. Stress testing is as needed.

The test first approach is encouraged although not mandated. Tests are written simultaneously with code most of the time. GUIs are not unit tested. Manual testing is not currently complete for the Sprint Demo leading to a lot of open work in progress.

Component	Test Cases	Tested
Acquisitions	529	384
Binding	802	646
Cataloging	3,101	1,115
Circulation	3,570	1,089
Common	0	0
ERM	0	0
Pac Searching	1,056	167
Serials	2,735	1,714
Sub total	11,793	5,115

Figure 4. Test Cases Created vs. Tested

During the Sprint, the Product Owner tests features that are in the Sprint backlog. Testers receive a stable Sprint build only after the Sprint demo. The reason for this is a low tester/developer ratio.

There are 30 team members in North America and 26 team members in St. Petersburg on this project. The St. Petersburg team has one project leader, 3 technical team leaders, 18 developers, 1 test lead, and 3 testers. This low tester/developer ratio and makes it impossible to have a fully tested package of code at the end of the Sprints. Fixing this problem could accelerate production in the future.

9.6. CONFIGURATION MANAGEMENT

SirsiDynix was using CVS as source code repository when the decision was made to engage an outsourcing firm. At that time, SirsiDynix made a decision that CVS could not be used

effectively because of lack of support for distributed development, largely seen in long code synchronization times. Other tools were evaluated and Perforce was chosen as the best solution.

StarSoft had seen positive results on many projects using Perforce. It is fast, reliable and offers local proxy servers for distributed teams. Although not a cheap solution, it has been very effective for the SirsiDynix project.

Automated builds run every hour with email generated back to developers. It takes 12 minutes to do a build, 30 minutes if the database changes. StarSoft would like to see faster builds and true concurrent engineering. Right now builds are only stable every two weeks at Sprint boundaries.

9.7. Pair Programming, Refactoring, and other XP practices

StarSoft is an XP company and tries to introduce XP practices into all their projects. Pair programming is done on more complicated pieces of functionality. Refactoring was planned for future Sprints and not done in every iteration as in XP. Some radically refactoring has occurred as the project approaches completion without loss of functionality. Continuous integration is implemented as hourly builds. On this project, these three engineering practices were used with Scrum as the primary methodology.

9.8. Measuring Progress

The project uses the Jira <http://www.atlassian.com> issue tracking and project management software. This gives everyone on the project a real-time view into the state of Sprints. It also provides comprehensive management reporting tools. The Figure below shows the Sprint burn-down chart and a snapshot of Earned Business Value on the project along with a synopsis of bug status.

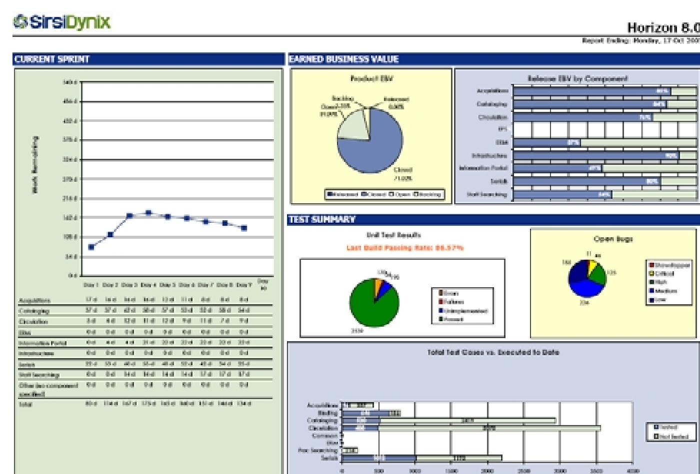


Figure 6. SirsiDynix Horizon 8.0 Project Dashboard

Data from Jira can be downloaded into Excel to create any requested data analysis. High velocity complex projects need an automated tool to track status across teams and geographies. The best tools support bug tracking and status of development tasks in one system to avoid extra work on data entry by developers. Such tools should track tasks completed by developers and work remaining. They provide more detailed and useful data than time sheets, which should be avoided. Time sheets are extra overhead that do not provide useful information on the state of the project, and are de-motivating to developers.

Other companies like PatientKeeper [Sutherland 2005] have found tools that incorporate both development tasks and defects that can be packaged into a Sprint Backlog are highly useful for complex development projects. Thousands of tasks and dozens of Sprints can be easily maintained and reviewed simultaneously with the right tool.

10. Integrated Scrums Model Resulting Context

Collaboration of SirsiDynix and StarSoft turned the Horizon 8.0 project into one of the most productive Scrum projects ever documented. For example, data is provided in the table below on a project that was done initially with a waterfall team and then reimplemented with a Scrum team [Cohn 2004]. The waterfall team took 9 months with 60 people and generated 54000 lines of code. It was re-implemented by a Scrum team of 4.5 people in 12 months. The resulting 50,803 lines of code had more functionality and higher quality.

	SCRUM	Waterfall	SirsiDynix
Person Months	54	540	827
Lines of Java	50,803	54,000	671,688
Function Points	959	900	12,673
FP per dev/month	17.8	2.0	15.3
FP per dev/month (industry average)	12.5	12.5	3

Figure 7. Function Points/Developer Month for collocated vs. distributed projects.

Capers Jones of Software Productivity Research has published extensive tables on average number of function points per lines of code for all major languages [Jones 1996]. Since the average lines of code per function point for Java is 53, we can estimate the number of function points in the Scrum application. The waterfall implementation is known to have fewer function points.

Distributed team working on Horizon 8.0 generated 671,688 lines of code in 14.5 months with 56 people. During this period they radically refactored the code on two occasions and reduced the code based by 275,000. They have not been penalized for radical refactoring as that is rarely done in large waterfall projects in the database from which Capers derived his numbers.

Jones has also shown from his database of tens of thousands of projects that industry average productivity is 12.5 function points per developer/month for a project of 900 function points and that this drops to 3 for a project with 13000 function points [Jones 2000].

The SirsiDynix project is almost as productive as the small Scrum project with a collocated team of 4.5 people. For a globally dispersed team, it is one of the most productive projects ever documented at a run rate of five times industry average.

11. Conclusions

It is extremely easy to integrate Scrum with XP practices even on large distributed teams. This can improve productivity, reduce project risk, and enhance software quality.

What is new in this paper is that single teams with members distributed across sites can enhance code ownership and improve autonomy essential to team self-organization. One Scrum meeting a day was necessary which included all team members across geographies. Written communication prior to joint meetings was needed to improve communication and reduce misunderstandings due to cultural and distance barriers. Project leaders in Provo, Utah, and St. Petersburg had a remarkable common view of the project because of the transparency and frequency of communications.

Automated communication of Product and Sprint backlogs throughout the organization combined with upward reporting of Scrum status to management can tightly align a global organization.

The issues of Product Backlog being “ready” for implementation in a Sprint and working software being “done” at the end of a Sprint are key areas where even the best teams need improvement.

REFERENCES

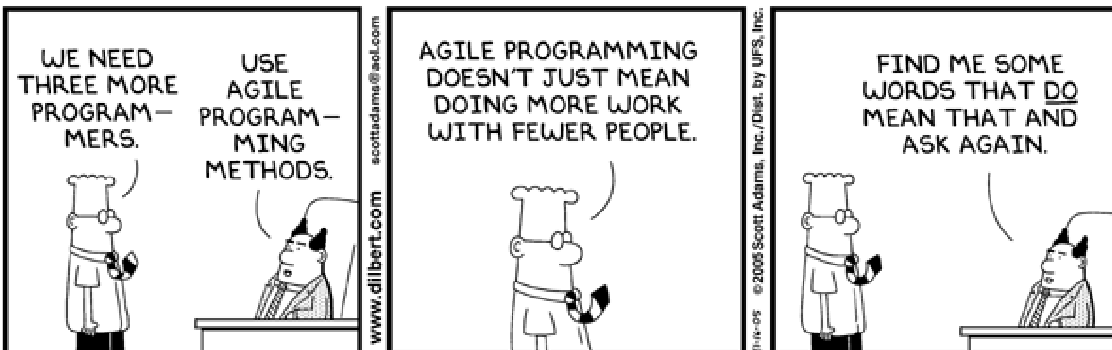
- Barthelemy, J., 2001, "The Hidden Costs of Outsourcing," MITSloan Management Review 42(3): 60-69.
- Beck, K., 1999, Extreme Programming Explained: Embrace Change, Addison-Wesley (Boston).
- Beedle, M., M. Devos, et al., 1999, Scrum: A Pattern Language for Hyperproductive Software Development, in Pattern Languages of Program Design, N. Harrison, Addison-Wesley (Boston). 4: 637-651.
- Brooks, F. P., 1995, The Mythical Man Month: Essays on Software Engineering, Addison-Wesley.
- Brooks, R. A., 1991, "Intelligence without representation," Artificial Intelligence 47: 139-159. Cohn, M., 2004, User Stories Applied: For Agile Software Development, Addison-Wesley.
- DeGrace, P. and L. H. Stahl, 1990, Wicked problems, righteous solutions: a catalogue of modern software engineering paradigms, Yourdon Press (Englewood Cliffs, N.J.).
- Gorzig, B. and A. Stephan, 2002, Outsourcing and Firm-level Performance.
- Gould, S. J., 2002, The structure of evolutionary theory, Belknap Press of Harvard University Press (Cambridge, Mass.).
- Jones, C., 1996, Programming Languages Table, Release 8.2, (Burlington, MA).
- Jones, C., 2000, Software assessments, benchmarks, and best practices / Capers Jones, Addison-Wesley (Boston, Mass.).
- Langton, C. G., 1992, Life at the Edge of Chaos, Artificial Life II, SFI Studies in the Sciences of Complexity, Held Feb 1990 in Sante Fe, NM, Addison-Wesley.
- Larman, C., 2004, Agile & Iterative Development: A Manager's Guide, Addison-Wesley (Boston).
- Levy, S., 1993, Artificial Life: A Report from the Frontier Where Computers Meet Biology, Vintage, Reprint edition (New York).
- Matisse Software, 2003, The Emergence of the Object-SQL Database, (Mountain View, CA).
- Nidiffer, K. E. and D. Dolan, 2005, "Evolving Distributed Project Management," IEEE Software 22(5): 63-72.
- Poppendieck, M., 2005, A History of Lean: From Manufacturing to Software Development, JAOO Conference, Aarhus, Denmark, EOS.

- Senge, P. M., 1990, The Fifth Discipline: the Art and Practice of the Learning Organization, Currency (New York).
- Sobek, D. K. I., A. C. Ward, et al., 1999, "Toyota's Principles of Set-Based Concurrent Engineering," Sloan Management Review 40(2): 67-83.
- StandishGroup, (2003), "2003 Chaos Chronicles." www.standishgroup.com/press/article.php?id=2.
- Sutherland, J., 2001, "Agile Can Scale: Inventing and Reinventing Scrum in Five Companies," Cutter IT Journal 14(12): 5-11.
- Sutherland, J., 2005, Future of Scrum: Parallel Pipelining of Sprints in Complex Projects with Details on Scrum Type C Tools and Techniques, (Brighton, MA).
- Sutherland, J., 2005, Scrum Evolution: Type A, B, and C Sprints, Agile 2005 Conference, Denver, CO.
- Takeuchi, H. and I. Nonaka, 1986, "The New Product Development Game," Harvard Business Review (January-February).
- Takeuchi, H. and I. Nonaka, 2004, Hitotsubashi on Knowledge Management, John Wiley & Sons (Asia) (Singapore).
- Wood, W. A. and W. L. Kleb, 2003, "Exploring XP for Scientific Research," IEEE Software 20(3): 30-36.
- Zanoni, R. and J. L. N. Audy, 2003, Projected Management Model for Physically Distributed Software Development Environment, HICSS'03, Hawaii, IEEE.

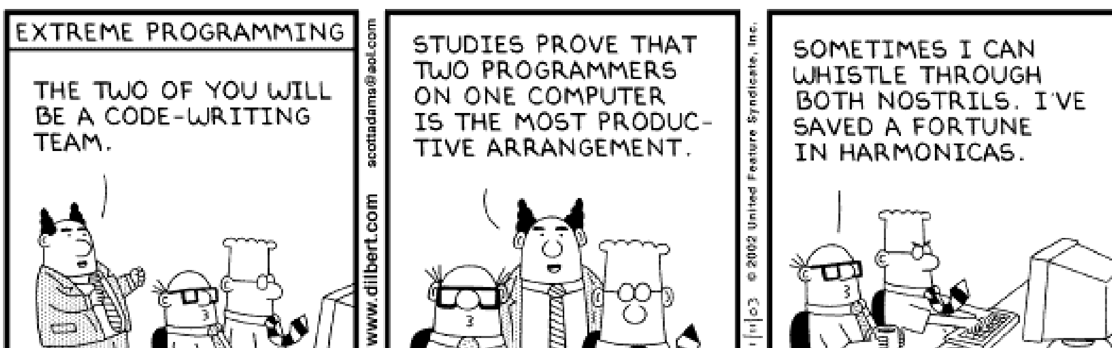
Anexo K. Dilbert y los métodos ágiles

Copyright © 2003 United Feature Syndicate, Inc.

Copyright © 2003 United Feature Syndicate, Inc.

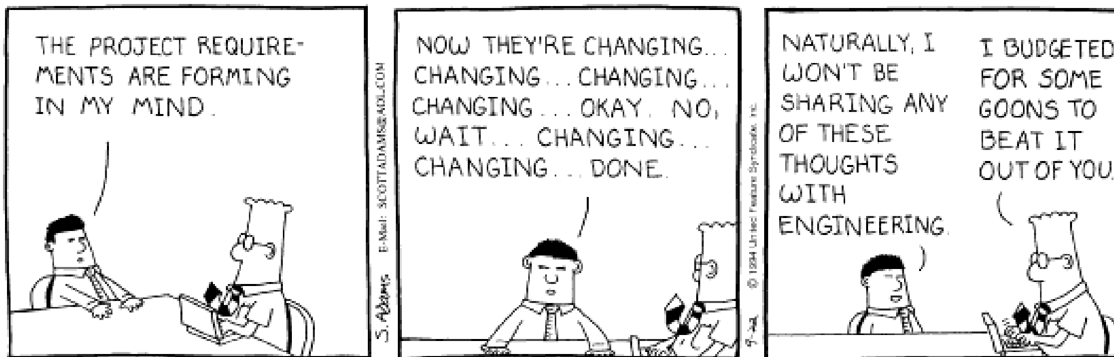


© Scott Adams, Inc./Dist. by UFS, Inc.

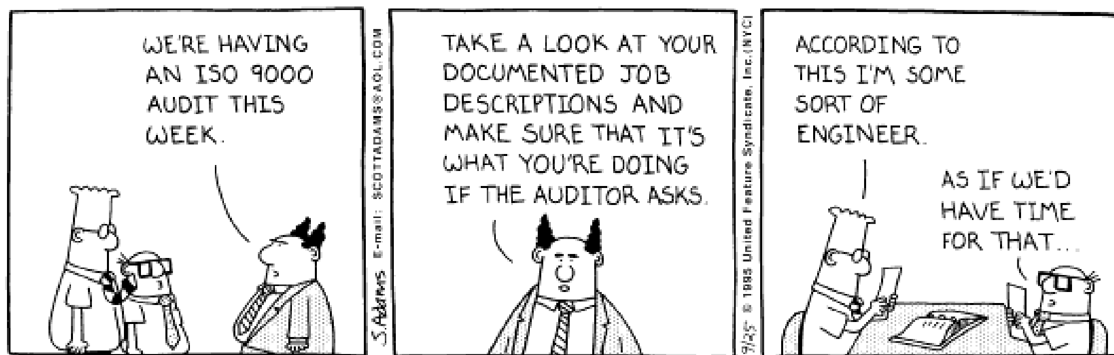


Copyright © 2003 United Feature Syndicate, Inc.

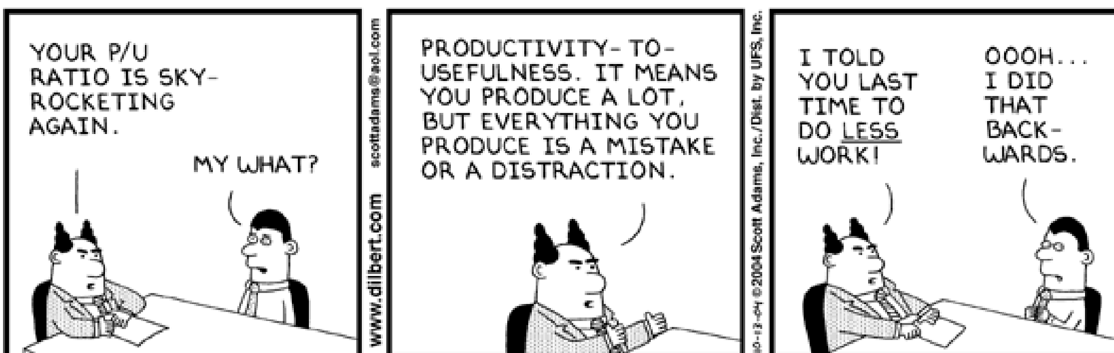
DILBERT © United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited.



DILBERT © United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited.



DILBERT © United Feature Syndicate, Inc.
Redistribution in whole or in part prohibited.



© UFS, Inc.

Anexo L. Epigramas sobre la programación

Alan J. Perlis
Yale University

This text has been published in SIGPLAN Notices Vol. 17, No. 9, September 1982, pages 7 - 13. I'm offering it here online until ACM stops me.

The phenomena surrounding computers are diverse and yield a surprisingly rich base for launching metaphors at individual and group activities. Conversely, classical human endeavors provide an inexhaustible source of metaphor for those of us who are in labor within computation. Such relationships between society and device are not new, but the incredible growth of the computer's influence (both real and implied) lends this symbiotic dependency a vitality like a gangly youth growing out of his clothes within an endless puberty.

The epigrams that follow attempt to capture some of the dimensions of this traffic in imagery that sharpens, focuses, clarifies, enlarges and beclouds our view of this most remarkable of all mans' artifacts, the computer.

1) One man's constant is another man's variable.
2) Functions delay binding: data structures induce binding. Moral: Structure data late in the programming process.
3) Syntactic sugar causes cancer of the semi-colons.
4) Every program is a part of some other program and rarely fits.
5) If a program manipulates a large amount of data, it does so in a small number of ways.
6) Symmetry is a complexity reducing concept (co-routines include sub-routines); seek it everywhere.
7) It is easier to write an incorrect program than understand a correct one.
8) A programming language is low level when its programs require attention to the irrelevant.
9) It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

10) Get into a rut early: Do the same processes the same way. Accumulate idioms. Standardize. The only difference (!) between Shakespeare and you was the size of his idiom list - not the size of his vocabulary.
11) If you have a procedure with 10 parameters, you probably missed some.
12) Recursion is the root of computation since it trades description for time.
13) If two people write exactly the same program, each should be put in micro-code and then they certainly won't be the same.
14) In the long run every program becomes rococo - then rubble.
15) Everything should be built top-down, except the first time.
16) Every program has (at least) two purposes: the one for which it was written and another for which it wasn't.
17) If a listener nods his head when you're explaining your program, wake him up.
18) A program without a loop and a structured variable isn't worth writing.
19) A language that doesn't affect the way you think about programming, is not worth knowing.

20) Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.
21) Optimization hinders evolution.
22) A good system can't have a weak command language.
23) To understand a program you must become both the machine and the program.
24) Perhaps if we wrote programs from childhood on, as adults we'd be able to read them.

25) One can only display complex information in the mind. Like seeing, movement or flow or alteration of view is more important than the static picture, no matter how lovely.
26) There will always be things we wish to say in our programs that in all known languages can only be said poorly.
27) Once you understand how to write a program get someone else to write it.
28) Around computers it is difficult to find the correct unit of time to measure progress. Some cathedrals took a century to complete. Can you imagine the grandeur and scope of a program that would take as long?
29) For systems, the analogue of a face-lift is to add to the control graph an edge that creates a cycle, not just an additional node.

30) In programming, everything we do is a special case of something more general - and often we know it too quickly.
31) Simplicity does not precede complexity, but follows it.
32) Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis.
33) The 11th commandment was "Thou Shalt Compute" or "Thou Shalt Not Compute" - I forget which.
34) The string is a stark data structure and everywhere it is passed there is much duplication of process. It is a perfect vehicle for hiding information.
35) Everyone can be taught to sculpt: Michelangelo would have had to be taught how not to. So it is with the great programmers.
36) The use of a program to prove the 4-color theorem will not change mathematics - it merely demonstrates that the theorem, a challenge for a century, is probably not important to mathematics.
37) The most important computer is the one that rages in our skulls and ever seeks that satisfactory external emulator. The standardization of real computers would be a disaster - and so it probably won't happen.
38) Structured Programming supports the law of the excluded muddle.
39) Re graphics: A picture is worth 10K words - but only those to describe the picture. Hardly any sets of 10K words can be adequately described with pictures.

40) There are two ways to write error-free programs; only the third one works.
41) Some programming languages manage to absorbe change, but withstand progress.
42) You can measure a programmer's perspective by noting his attitude on the continuing vitality of FORTRAN.
43) In software systems it is often the early bird that makes the worm.
44) Sometimes I think the only universal in the computing field is the fetch-execute-cycle.
45) The goal of computation is the emulation of our synthetic abilities, not the understanding of our analytic ones.
46) Like punning, programming is a play on words.
47) As Will Rogers would have said, "There is no such thing as a free variable."
48) The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.
49) Giving up on assembly language was the apple in our Garden of Eden: Languages whose use squanders machine cycles are sinful. The LISP machine now permits LISP programmers to abandon bra and fig-leaf.

50) When we understand knowledge-based systems, it will be as before - except our finger-tips will have been singed.
51) Bringing computers into the home won't change either one, but may revitalize the corner saloon.
52) Systems have sub-systems and sub-systems have sub-systems and so on ad finitum - which is why we're always starting over.
53) So many good ideas are never heard from again once they embark in a voyage on the semantic gulf.

54) Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.
55) A LISP programmer knows the value of everything, but the cost of nothing.
56) Software is under a constant tension. Being symbolic it is arbitrarily perfectible; but also it is arbitrarily changeable.
57) It is easier to change the specification to fit the program than vice versa.
58) Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
59) In English every word can be verbed. Would that it were so in our programming languages.
60) Dana Scott is the Church of the Lattice-Way Saints.

61) In programming, as in everything else, to be in error is to be reborn.
62) In computing, invariants are ephemeral.
63) When we write programs that "learn", it turns out we do and they don't.
64) Often it is means that justify ends: Goals advance technique and technique survives even when goal structures crumble.
65) Make no mistake about it: Computers process numbers - not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity.
66) Making something variable is easy. Controlling duration of constancy is the trick.
67) Think of all the psychic energy expended in seeking a fundamental distinction between "algorithm" and "program".
68) If we believe in data structures, we must believe in independent (hence simultaneous) processing. For why else would we collect items within a structure? Why do we tolerate languages that give us the one without the other?
69) In a 5 year period we get one superb programming language. Only we can't control when the 5 year period will begin.

70) Over the centuries the Indians developed sign language for communicating phenomena of interest. Programmers from different tribes (FORTRAN, LISP, ALGOL, SNOBOL, etc.) could use one that doesn't require them to carry a blackboard on their ponies.
71) Documentation is like term insurance: It satisfies because almost no one who subscribes to it depends on its benefits.
72) An adequate bootstrap is a contradiction in terms.
73) It is not a language's weaknesses but its strengths that control the gradient of its change: Alas, a language never escapes its embryonic sac.
74) It is possible that software is not like anything else, that it is meant to be discarded: that the whole point is to always see it as soap bubble?
75) Because of its vitality, the computing field is always in desperate need of new cliches: Banality soothes our nerves.
76) It is the user who should parametrize procedures, not their creators.
77) The cybernetic exchange between man, computer and algorithm is like a game of musical chairs: The frantic search for balance always leaves one of the three standing ill at ease.
78) If your computer speaks English it was probably made in Japan.
79) A year spent in artificial intelligence is enough to make one believe in God.

80) Prolonged contact with the computer turns mathematicians into clerks and vice versa.
81) In computing, turning the obvious into the useful is a living definition of the word "frustration".
82) We are on the verge: Today our program proved Fermat's next-to-last theorem!
83) What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak.
84) Motto for a research laboratory: What we work on today, others will first think of tomorrow.
85) Though the Chinese should adore APL, it's FORTRAN they put their money on.
86) We kid ourselves if we think that the ratio of procedure to data in an active data-base system can be made arbitrarily small or even kept small.
87) We have the mini and the micro computer. In what semantic niche would the pico computer fall?

88) It is not the computer's fault that Maxwell's equations are not adequate to design the electric motor.
89) One does not learn computing by using a hand calculator, but one can forget arithmetic.

90) Computation has made the tree flower.
91) The computer reminds one of Lon Chaney - it is the machine of a thousand faces.
92) The computer is the ultimate polluter. Its feces are indistinguishable from the food it produces.
93) When someone says "I want a programming language in which I need only say what I wish done," give him a lollipop.
94) Interfaces keep things tidy, but don't accelerate growth: Functions do.
95) Don't have good ideas if you aren't willing to be responsible for them.
96) Computers don't introduce order anywhere as much as they expose opportunities.
97) When a professor insists computer science is X but not Y, have compassion for his graduate students.
98) In computing, the mean time to failure keeps getting shorter.
99) In man-machine symbiosis, it is man who must adjust: The machines can't.

100) We will never run out of things to program as long as there is a single program around.
101) Dealing with failure is easy: Work hard to improve. Success is also easy to handle: You've solved the wrong problem. Work hard to improve.
102) One can't proceed from the informal to the formal by formal means.
103) Purely applicative languages are poorly applicable.
104) The proof of a system's value is its existence.
105) You can't communicate complexity, only an awareness of it.
106) It's difficult to extract sense from strings, but they're the only communication coin we can count on.
107) The debate rages on: Is PL/I Bachtrian or Dromedary?
108) Whenever two programmers meet to criticize their programs, both are silent.
109) Think of it! With VLSI we can pack 100 ENIACs in 1 sq.cm.

110) Editing is a rewording activity.
111) Why did the Roman Empire collapse? What is the Latin for office automation?
112) Computer Science is embarrassed by the computer.
113) The only constructive theory connecting neuroscience and psychology will arise from the study of software.
114) Within a computer natural language is unnatural.
115) Most people find the concept of programming obvious, but the doing impossible.
116) You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.
117) It goes against the grain of modern education to teach children to program. What fun is there in making plans, acquiring discipline in organizing thoughts, devoting attention to detail and learning to be self-critical?
118) If you can imagine a society in which the computer-robot is the only menial, you can imagine anything.
119) Programming is an unnatural act.
120) Adapting old programs to fit new machines usually means adapting new machines to behave like old ones.

Anexo M. Resumen en transparencias

1. INTRODUCCIÓN

    1.1 Precedentes: métodos clásicos: cascada, V, espiral
    1.2 Necesidad de nuevos métodos, diferencias
    1.3 Ágil: El Manifiesto Ágil y los 12 principios ágiles

2. PRINCIPALES MÉTODOS ÁGILES

    2.1 Extreme Programming – XP
    2.2 Scrum
    2.3 Crystal
    2.4 Feature Driven Development – FDD
    2.5 Rational/Enterprise/Agile Unified Process – RUP, EUP, AUP
    2.6 Dynamic Systems Development Method – DSDM
    2.7 Adaptive Software Development – ASD
    2.8 Open Source Software Development – OSS
    2.9 Lean Software Development – LSD
    2.10 Agile Modelling – AM
    2.11 Evolutionary Project Management – Evo
    2.12 Internet–Speed Development – ISD
    2.13 Microsoft Solutions Framework – MSF
    2.14 Pragmatic Programming – PP

3. SOFTWARE

    3.1 Herramientas para automatizar
    3.2 Programas

4. CONCLUSIONES

    4.1 Resumen
    4.2 Estadísticas
    4.3 Relaciones: complementariedad, similitudes
    4.4 Crítica a las prácticas y métodos ágiles

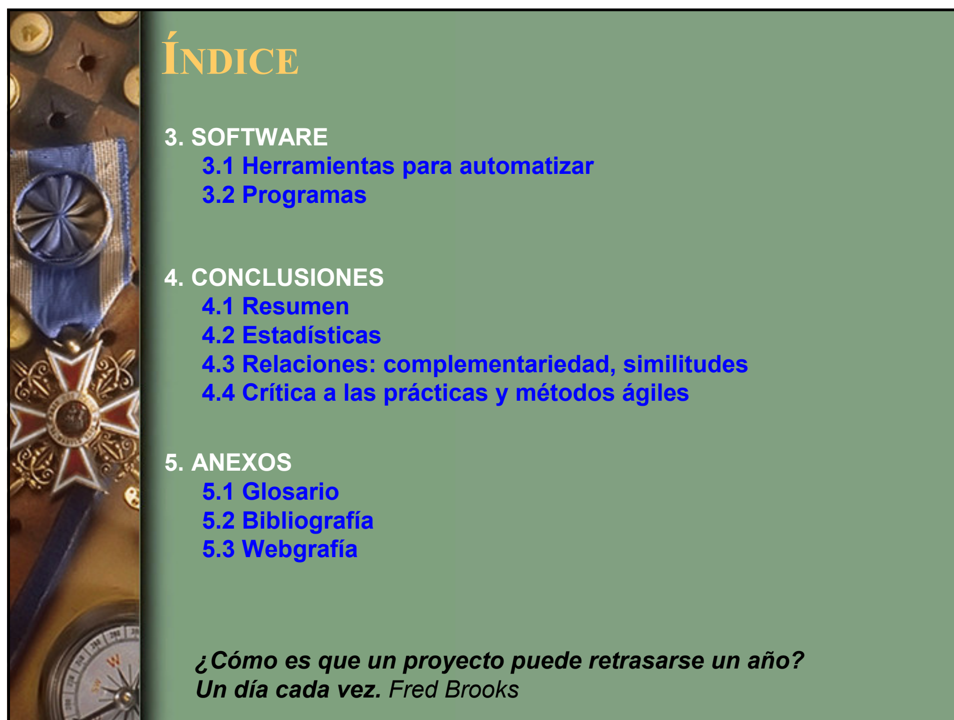
5. ANEXOS

    5.1 Glosario
    5.2 Bibliografía
    5.3 Webgrafía

ÍNDICE

1. INTRODUCCIÓN
- 1.1 Precedentes: métodos clásicos: cascada, V, espiral
- 1.2 Necesidad de nuevos métodos, diferencias
- 1.3 Ágil: El Manifiesto Ágil y los 12 principios ágiles

2. PRINCIPALES MÉTODOS ÁGILES
- 2.1 Extreme Programming – XP
- 2.2 Scrum
- 2.3 Crystal
- 2.4 Feature Driven Development – FDD
- 2.5 Rational/Enterprise/Agile Unified Process – RUP, EUP, AUP
- 2.6 Dynamic Systems Development Method – DSDM
- 2.7 Adaptive Software Development – ASD
- 2.8 Open Source Software Development – OSS
- 2.9 Lean Software Development – LSD
- 2.10 Agile Modelling – AM
- 2.11 Evolutionary Project Management – Evo
- 2.12 Internet–Speed Development – ISD
- 2.13 Microsoft Solutions Framework – MSF
- 2.14 Pragmatic Programming – PP



1. Introducción

Introducción – Índice

1.1 Precedentes: métodos clásicos: cascada, V, espiral

- Análisis previo y completo de requisitos.
- Diseño de todo el proyecto antes de empezar a desarrollarlo.
- Todo cambio no previsto es un problema y se corrige para volver al plan inicial, pero no se cambia el plan.
- El cliente recibe el código muy tarde y no se le consulta de forma continua.
- Enfatizan la documentación.

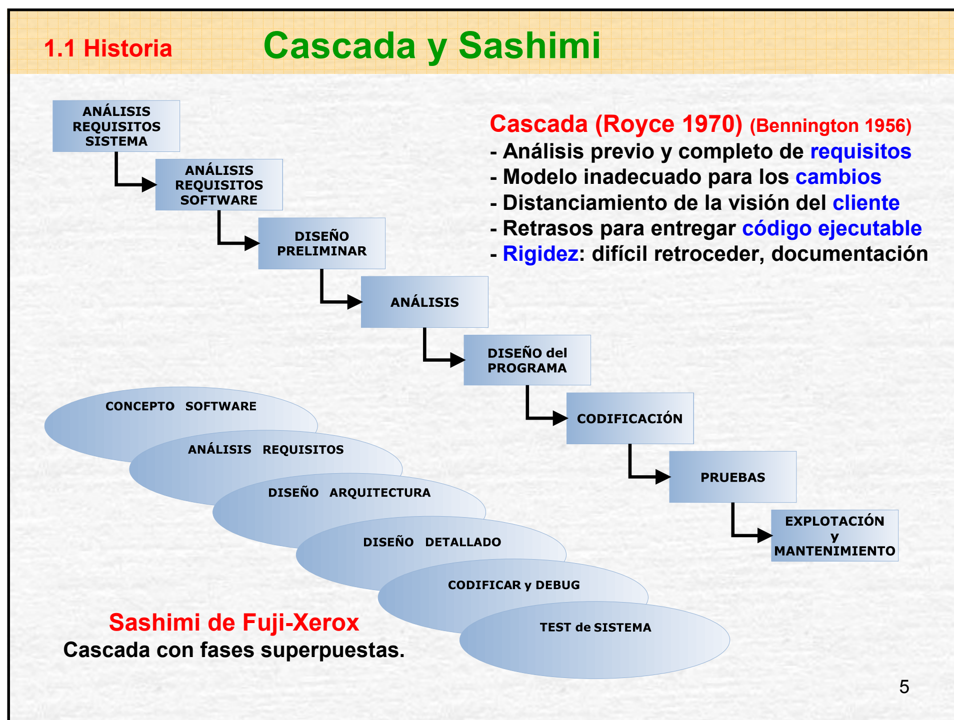
1.2 Necesidad de nuevos métodos, diferencias

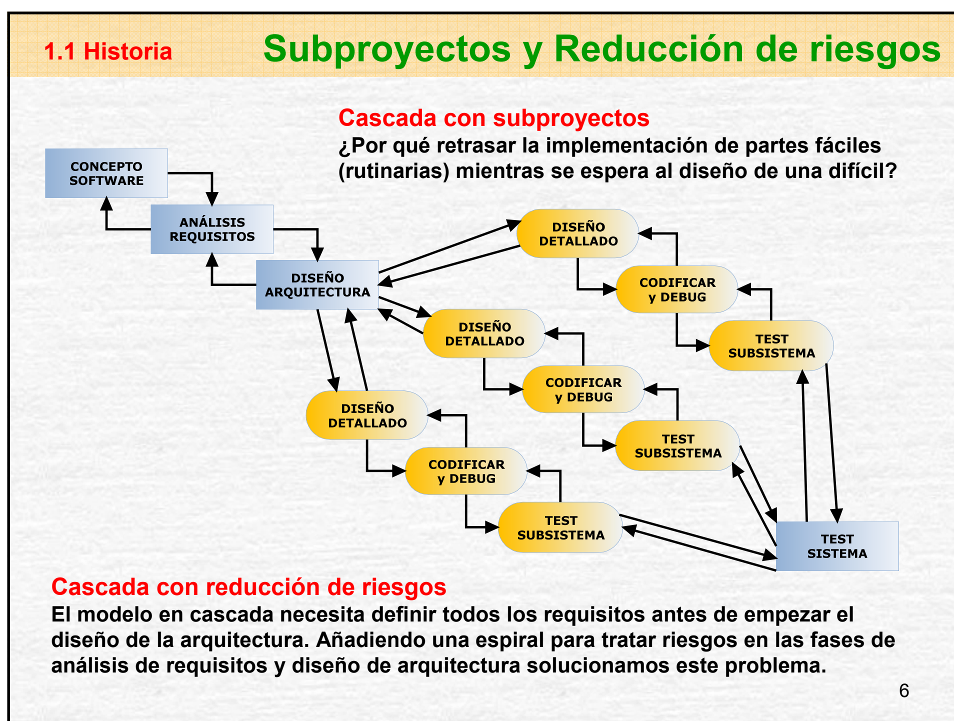
- Gracias al *feedback*, pretenden evitar que el cliente, tras mucho tiempo y documentación, diga *"Es lo que pedí, pero no es lo que necesitaba"*.
- Muchos proyectos sobrepasaron tiempo y coste acordados.

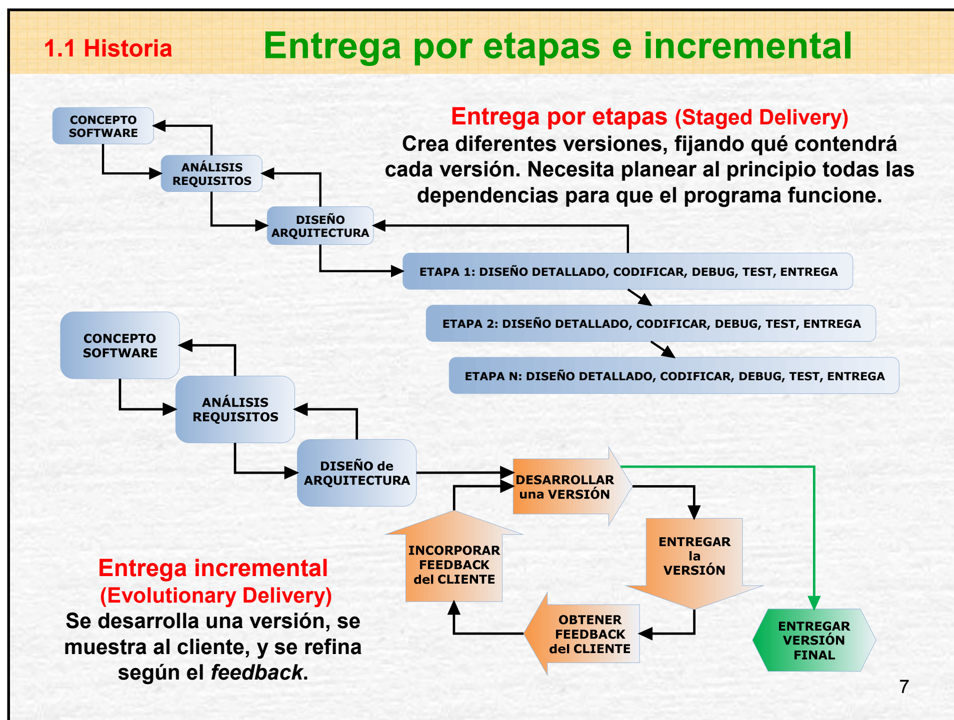
1.3 Ágil: El Manifiesto Ágil y los 12 principios ágiles

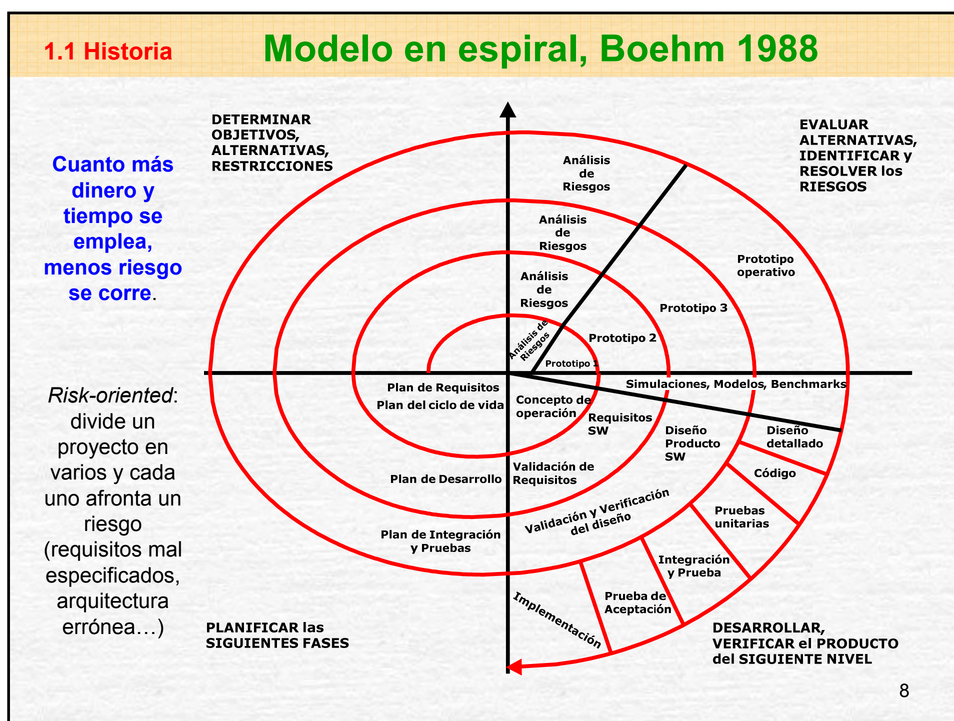
- Bases comunes a todos los métodos ágiles.

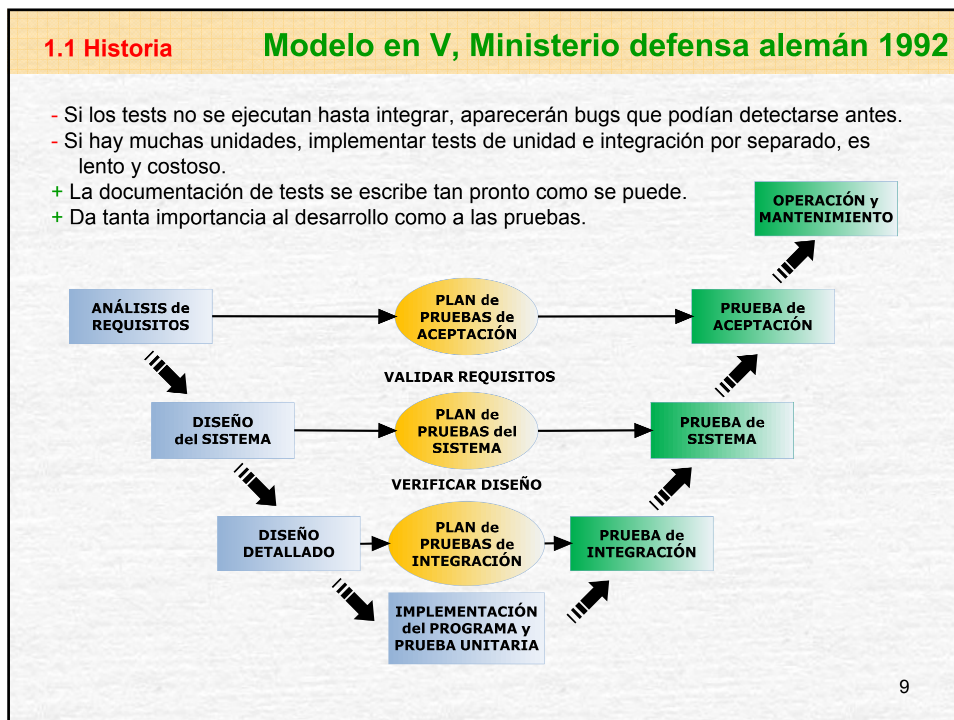
4

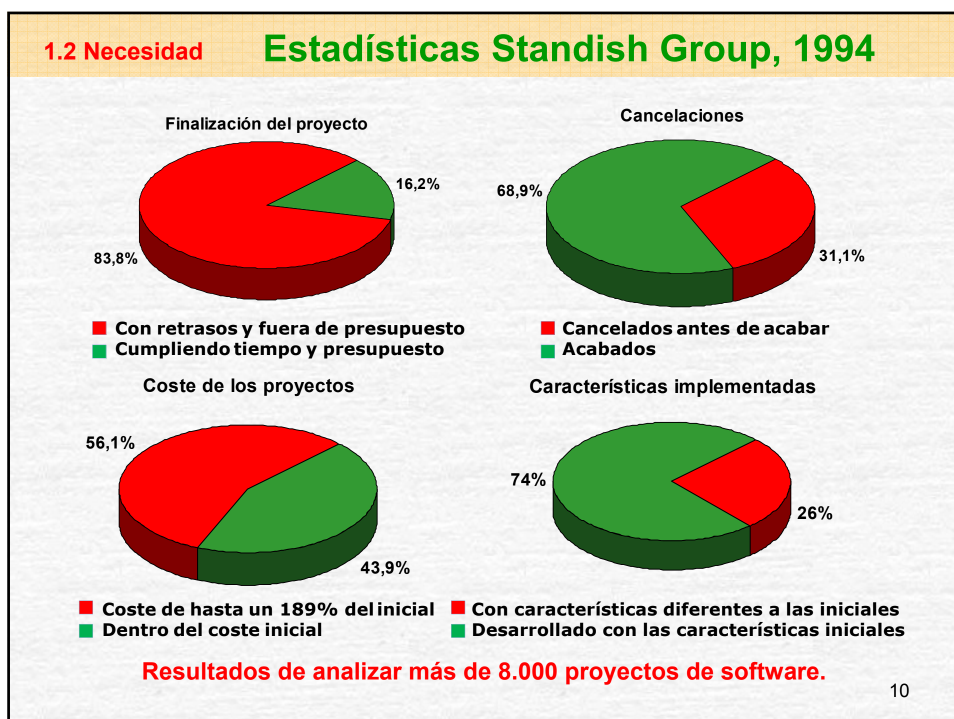

1.1 Historia
Cascada y Sashimi
ANÁLISIS REQUISITOS SISTEMA
ANÁLISIS REQUISITOS SOFTWARE
DISEÑO PRELIMINAR
ANÁLISIS
DISEÑO del PROGRAMA
CODIFICACIÓN
PRUEBAS
EXPLOTACIÓN y MANTENIMIENTO
Cascada (Royce 1970) (Bennington 1956)
- Análisis previo y completo de requisitos
- Modelo inadecuado para los cambios
- Distanciamiento de la visión del cliente
- Retrasos para entregar código ejecutable
- Rigidez: difícil retroceder, documentación
CONCEPTO SOFTWARE
ANÁLISIS REQUISITOS
DISEÑO ARQUITECTURA
DISEÑO DETALLADO
CODIFICAR y DEBUG
TEST de SISTEMA
Sashimi de Fuji-Xerox
Cascada con fases superpuestas.
5


1.1 Historia
Subproyectos y Reducción de riesgos
Cascada con subproyectos
¿Por qué retrasar la implementación de partes fáciles (rutinarias) mientras se espera al diseño de una difícil?
CONCEPTO SOFTWARE
ANÁLISIS REQUISITOS
DISEÑO ARQUITECTURA
DISEÑO DETALLADO
CODIFICAR y DEBUG
TEST SUBSISTEMA
DISEÑO DETALLADO
CODIFICAR y DEBUG
TEST SUBSISTEMA
DISEÑO DETALLADO
CODIFICAR y DEBUG
TEST SUBSISTEMA
TEST SISTEMA
Cascada con reducción de riesgos
El modelo en cascada necesita definir todos los requisitos antes de empezar el diseño de la arquitectura. Añadiendo una espiral para tratar riesgos en las fases de análisis de requisitos y diseño de arquitectura solucionamos este problema.
6


1.1 Historia
Entrega por etapas e incremental
Entrega por etapas (Staged Delivery)
Crea diferentes versiones, fijando qué contendrá cada versión. Necesita planear al principio todas las dependencias para que el programa funcione.
CONCEPTO SOFTWARE
ANÁLISIS REQUISITOS
DISEÑO ARQUITECTURA
ETAPA 1: DISEÑO DETALLADO, CODIFICAR, DEBUG, TEST, ENTREGA
ETAPA 2: DISEÑO DETALLADO, CODIFICAR, DEBUG, TEST, ENTREGA
ETAPA N: DISEÑO DETALLADO, CODIFICAR, DEBUG, TEST, ENTREGA
CONCEPTO SOFTWARE
ANÁLISIS REQUISITOS
DISEÑO de ARQUITECTURA
DESARROLLAR una VERSIÓN
ENTREGAR la VERSIÓN
OBTENER FEEDBACK del CLIENTE
INCORPORAR FEEDBACK del CLIENTE
ENTREGAR VERSIÓN FINAL
Entrega incremental
(Evolutionary Delivery)
Se desarrolla una versión, se muestra al cliente, y se refina según el feedback.
7


1.1 Historia
Modelo en espiral, Boehm 1988
Cuanto más dinero y tiempo se emplea, menos riesgo se corre.
Risk-oriented: divide un proyecto en varios y cada uno afronta un riesgo (requisitos mal especificados, arquitectura errónea…)
DETERMINAR OBJETIVOS, ALTERNATIVAS, RESTRICCIONES
EVALUAR ALTERNATIVAS, IDENTIFICAR y RESOLVER los RIESGOS
PLANIFICAR las SIGUIENTES FASES
DESARROLLAR, VERIFICAR el PRODUCTO del SIGUIENTE NIVEL
Análisis de Riesgos
Análisis de Riesgos
Análisis de Riesgos
Análisis de Riesgos
Prototipo operativo
Prototipo 3
Prototipo 2
Prototipo 1
Simulaciones, Modelos, Benchmarks
Plan de Requisitos
Plan del ciclo de vida
Concepto de operación
Requisitos SW
Diseño Producto SW
Diseño detallado
Código
Plan de Desarrollo
Validación de Requisitos
Validación y Verificación del diseño
Pruebas unitarias
Plan de Integración y Pruebas
Integración y Prueba
Prueba de Aceptación
Implementación
8


1.1 Historia
Modelo en V, Ministerio defensa alemán 1992
- Si los tests no se ejecutan hasta integrar, aparecerán bugs que podían detectarse antes.
- Si hay muchas unidades, implementar tests de unidad e integración por separado, es lento y costoso.
+ La documentación de tests se escribe tan pronto como se puede.
+ Da tanta importancia al desarrollo como a las pruebas.
OPERACIÓN y MANTENIMIENTO
ANÁLISIS de REQUISITOS
PLAN de PRUEBAS de ACEPTACIÓN
PRUEBA de ACEPTACIÓN
VALIDAR REQUISITOS
DISEÑO del SISTEMA
PLAN de PRUEBAS del SISTEMA
PRUEBA de SISTEMA
VERIFICAR DISEÑO
DISEÑO DETALLADO
PLAN de PRUEBAS de INTEGRACIÓN
PRUEBA de INTEGRACIÓN
IMPLEMENTACIÓN del PROGRAMA y PRUEBA UNITARIA
9


1.2 Necesidad
Estadísticas Standish Group, 1994
Finalización del proyecto
16,2%
83,8%
Con retrasos y fuera de presupuesto
Cumpliendo tiempo y presupuesto
Cancelaciones
68,9%
31,1%
Cancelados antes de acabar
Acabados
Coste de los proyectos
56,1%
43,9%
Coste de hasta un 189% del inicial
Dentro del coste inicial
Características implementadas
74%
26%
Con características diferentes a las iniciales
Desarrollado con las características iniciales
Resultados de analizar más de 8.000 proyectos de software.
10

1.3 Necesidad

Tradicionales vs. ágiles

MÉTODOS TRADICIONALES	MÉTODOS ÁGILES
Proceso **secuencial**, replicable, controlado	Procesos simultáneos, únicos, **aleatorios**
Orientado a **proceso** (goal-driven)	Guiado por **sucesos** (accident-driven)
Mayor o menor **resistencia a los cambios**	Aceptan e incluso **fomentan el cambio**
Se trata con el **cliente sólo en reuniones**	El **cliente** forma parte del **equipo**
Equipos grandes, >15-20 miembros	**Equipos pequeños** o medianos
Procesos con **muchas normas**	Procesos con **pocas reglas**
Mucha documentación	**Poca documentación**
Mucho análisis, diseño y arquitectura	**Poco análisis y diseño y arquitectura**
Procesos y herramientas	Individuos e interacciones
Documentación comprensible	Software que funciona
Negociación de contratos	Colaboración con el cliente
Seguir un plan	Responder al cambio

11

1.4 Ágiles

El Manifiesto Ágil

Estamos descubriendo mejores maneras de desarrollar software, tanto por nuestra propia experiencia como ayudando a terceros. A través de esta experiencia hemos aprendido a valorar:

Individuos e interacciones sobre procesos y herramientas.
Software que funciona sobre documentación exhaustiva.
Colaboración con el cliente sobre negociación de contratos.
Responder ante el cambio sobre seguimiento de un plan.

Es decir, aunque los elementos a la derecha tienen valor, nosotros valoramos por encima de ellos los que están a la izquierda.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham

Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt

Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin

Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

12

1.4 Ágiles

Los 12 principios ágiles

1) Nuestra mayor prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor.

2) Aceptamos requisitos cambiantes, incluso en etapas avanzadas. Los procesos ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.

3) Se entrega software frecuentemente, con una periodicidad desde un par de semanas a un par de meses, con preferencia por los periodos más cortos posibles.

4) Los responsables de negocio y los desarrolladores deben trabajar juntos diariamente a lo largo del proyecto.

5) Construimos proyectos con profesionales motivados. Les damos el entorno y soporte que necesitan, y confiando en ellos para que realicen el trabajo.

6) El método más eficiente y efectivo de comunicar la información a un equipo de desarrollo y entre los miembros del mismo es la conversación cara a cara.

13

1.4 Ágiles

Los 12 principios ágiles

7) La principal medida de progreso es software que funciona.

8) Los procesos ágiles promueven el desarrollo sostenible. Patrocinadores, desarrolladores y usuarios deben ser capaces de mantener un ritmo constante de forma indefinida.

9) La atención continua a la excelencia técnica y los buenos diseños mejoran la agilidad.

10) La Simplicidad, arte de maximizar la cantidad de trabajo no realizado, es esencial.

11) Las mejores arquitecturas, requisitos y diseños surgen de equipos que se autoorganizan.

12) A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo, entonces mejora y ajusta su comportamiento de acuerdo a sus conclusiones.

La diferencia entre un atracador de bancos y un teórico del método CMM es que con el atracador se puede negociar. Ken Orr, DSDM

14

2\. MÉTODOS

MÉTODOS ÁGILES – Índice

2. Principales métodos ágiles

2.1 Extreme Programming – XP
2.2 Scrum
2.3 Crystal
2.4 Feature Driven Development – FDD
2.5 Rational/Enterprise/Agile Unified Process, – RUP, EUP, AUP
2.6 Dynamic Systems Development Method – DSDM
2.7 Adaptive Software Development – ASD
2.8 Open Source Software Development – OSS
2.9 Lean Software Development – LSD
2.10 Agile Modelling – AM
2.11 Evolutionary Project Management – Evo
2.12 Internet-Speed Development – ISD
2.13 Microsoft Solutions Framework – MSF
2.14 Pragmatic Programming – PP

Haga un plan para tirarlo, lo hará de todos modos. Fred Brooks
Plan the work, and work the plan. Tom DeMarco

15

2.1 XP

Extreme Programming - XP

Kent Beck, 1999; segunda versión en 2004

Beck y Cunningham programaban en Smalltalk, adecuado para XP. Primera prueba para el proyecto C3 de Chrysler.

"*eXtreme*" viene de llevar al "extremo" principios de sentido común:
"si diseñar es bueno, diseñemos todo el tiempo"
"si las pruebas son buenas, probemos todo el tiempo"

16

2.1 XP

Principios (1999)

1. **Juego de planteamiento:** Los programadores estiman el esfuerzo necesario para implementar las *stories* del cliente en tarjetas **CRC** Clase-Responsabilidad-Colaborador y éste decide qué contendrán las versiones.
2. **Releases pequeñas o Entregas frecuentes:** en cada iteración sólo se implementan unas cuantas historias. Se creará una versión pequeña pero operativa, al menos cada 2 o 3 meses.
3. **Metáfora:** El sistema entre cliente y programadores se define con metáforas (como arquitectura simplificada, imagen mental compacta del sistema).
4. **Diseño simple:** diseñar la solución más simple posible. Menos código, menos errores. Para eliminar la duplicación de código se utiliza la refactorización. XP no usa un análisis completo mediante diagramas UML. **El Código es el diseño**.
5. **Pruebas continuas o Inmediatez:** Desarrollo **test-driven**, según pruebas (se escriben ANTES que el código, **test-first**, y son automáticas). La prueba unitaria (programador) verifica una clase o un conjunto. La prueba funcional (Cliente) verifica todo el sistema, o gran parte. Al incorporar código, se ejecutan todas las pruebas existentes, no sólo las asociadas al nuevo código. Es el **Test de regresión**.
6. **Refactoring:** Reestructurar código internamente para ser más claro y eficiente. *Si funciona bien, arréglalo de todos modos*.

17

2.1 XP

Principios (1999)

7. **Programación en Parejas** (*pair programming):* dos personas escriben el código en una computadora, turnándose su uso. Quien no escribe, piensa la estrategia y revisa el código del otro en tiempo real.
8. **Propiedad Colectiva:** Cualquiera puede cambiar cualquier parte del código cuando quiera, si antes escribe su prueba. Un programador puede modificar cualquier clase, no sólo las escritas por él.
9. **Integración continua:** Si un trozo de código está listo, se integra en el sistema. Se integra varias veces cada día.
10. **Semanas de 40 horas:** Horas extras controladas. Descanso necesario.
11. **Cliente On-site:** Representante del cliente con el equipo para tener feedback rápido y colaborar en las pruebas.
12. **Estándares de codificación:** Comunicación a través del código, los comentarios no están muy bien vistos: si el código es tan oscuro que necesita comentario, se debe reescribir.

- IMPLÍCITOS

13. **Lugar de trabajo abierto:** el equipo trabaja junto, sin barreras.
14. **Contratos de alcance opcional:** Los contratos que fijan plazos de entrega, coste, requisitos y calidad (alcance global) implican riesgos.

"De todo lo que valga la pena hablar, vale la pena dejar constancia por escrito."

18

2.1 XP

Papeles

Programador: Escribe las pruebas unitarias y el código simple y optimizado. Estima el tiempo para implementar las historias. Comunicación con todo el equipo.

Cliente "en casa": Escribe las historias y las pruebas funcionales, decidiendo cuándo dar por válida cada funcionalidad. Fija la prioridad de implementación de los requisitos. Siempre debe acompañar al equipo.

Probador (tester): Ayuda al cliente a escribir las pruebas funcionales que ejecutan regularmente. Informa de los resultados.

Supervisor (Tracker): Sigue cada iteración (con *feedback*) y ajusta el calendario previsto.

Entrenador o Instructor: Conocedor de XP, guiará a los miembros del equipo y es el responsable del proceso en conjunto. Ayuda, no da órdenes.

Consultor (asesor): miembro externo con el conocimiento técnico específico necesario para guiar al equipo en problemas concretos.

Gestor o Director (Manager, Big Boss): Toma las decisiones. Determina la situación actual y distingue cualquier dificultad o deficiencia.

Algunos papeles pueden combinarse en una sola persona. En un equipo XP no existen los papeles de Diseñador o de Arquitecto, ni Jefe Supremo que decida qué diseños son buenos y cuáles no. Las decisiones de diseño las toma el equipo.

19

2.1 XP

Proceso

EXPLORACIÓN
PLANTEAMIENTO
ITERACIONES PARA ENTREGAR
PRODUCCIÓN
MANTENIMIENTO
MUERTE
Actualizaciones regulares
HISTORIAS
HISTORIAS PARA LA PRÓXIMA ITERACIÓN
PRIORIDADES
ESTIMACIONES de ESFUERZO
REVISIONES CONTINUAS
PROGRAMACIÓN POR PAREJAS
ANÁLISIS
DISEÑO
PLAN PARA PRUEBAS
PRUEBAS
FEEDBACK
INTEGRACIÓN CONTINUA
TEST
BASE de DATOS COLECTIVA
VERSIÓN PEQUEÑA
VERSIONES Actualizadas
VERSIÓN FINAL

" Si usted quiere probar XP, por Dios, no intente tragárselo de una vez. Escoja el peor problema en su proceso actual y pruebe a resolverlo como lo haría XP " – Kent Beck

20

2.1 XP

Nuevos valores y principios (2004)

Valores:

Comunicación + Simplicidad + Feedback + Valor + Respeto

Principios:

1) **Humanidad**: las personas necesitan mantener el empleo, sentirse valoradas, pertenecer al grupo, ampliar sus habilidades y perspectivas, y ser entendidas.
2) **Economía**: diseño incremental, para dar valor comercial, según *feedback*.
3) **Beneficio mutuo**: toda actividad debe beneficiar a toda persona y organización.
4) **Auto-similitud**: se debe poder reutilizar código para soluciones similares.
5) **Mejora**: la perfección no existe, pero debe buscarse (*feedback, tests…)*
6) **Diversidad**: Equipos con diferentes conocimientos, habilidades y caracteres.
7) **Reflexión**: un equipo analiza el porqué del éxito o fracaso.
8) **Flujo**: continuidad y firmeza, pequeños incrementos en la dirección correcta.
9) **Oportunidad**: un problema es una oportunidad para aprender y mejorar.
10) **Redundancia**: resolver problemas críticos y difíciles de varias maneras. Así, si una solución falla, las otras evitarán un desastre.
11) **Fracaso**: si no puede hacerlo bien, equivóquese para aprender.
12) **Calidad**: al máximo, pero provocar retrasos intentando buscar la perfección es peor que probar y equivocarse.
13) **Pequeños pasos**: proceder iterativamente avanzando lentamente.
14) **Responsabilidad aceptada**: la responsabilidad sólo puede aceptarse.

21

2.1 XP

Nuevas y antiguas prácticas

NUEVAS PRÁCTICAS
- Historias
- Ciclo semanal
- Ciclo trimestral
- Slack
- Despliegue incremental
- Cliente involucrado
- Contrato alcance negociado
- Pay-Per-Use
- Compartir espacio
- Equipo completo
- Área de trabajo informativa
- Trabajo enérgico

ANTIGUAS PRÁCTICAS
- Juego de planteamiento
- Entregas frecuentes
- Cliente on-site
- Metáforas
- Paz sostenible

NUEVAS PRÁCTICAS
- Programación por parejas
- Continuidad del equipo
- Equipos que se reducen
- Diseño incremental
- Programación test-first
- Análisis Raíz-causa
- Build de 10 minutos
- Integración continua
- Código y pruebas
- Código compartido
- Base de código única
- Despliegue diario

ANTIGUAS PRÁCTICAS
- Programación por parejas
- Pruebas constantes
- Refactoring
- Diseño simple
- Integración continua
- Propiedad código colectiva
- Estándares de codificación

En la primera edición, XP tenía 4 valores, 15 principios básicos, y 12 prácticas.
En el nuevo XP hay 5 valores, 14 principios, 13 prácticas primarias y 11 secundarias que reflejan las originales en parte. Metáforas y los estándares de codificación desaparecen.

22

2.2 SCRUM **SCRUM**

Jeff Sutherland y Ken Schwaber, 1995 (origen 1986)

Empresa DuPont: dos procesos: definidos y empíricos (requerían constante supervisión y adaptación a los cambios, originaron el Daily Scrum meeting, para evitar largas y poco productivas reuniones semanales).

Scrum debe verse más como un método de gestión no sólo de software, sino aplicable a otras disciplinas como la consultoría.

Origen en Honda, Canon, Fuji-Xerox, Brother o Toyota, que conseguían multiplicar por 4 la eficiencia y por 12 la calidad respecto a sus competidores

Hay experiencias que integran XP y *Scrum*. *Scrum* proporciona el *Project management framework*, que se sostiene por las prácticas de XP. Uno de los beneficios de la unión es permitir escalar XP a proyectos mayores.

23

2.2 SCRUM **Papeles**

Scrum Master: Asegura que se usan las prácticas, valores y reglas de *Scrum* y que progresa como estaba previsto. Actúa con el equipo y el cliente. Soluciona problemas.

Propietario del producto: Responsable del proyecto, lo gestiona, controla, y hace visible la *Product backlog list*. Elegido por el *Scrum Master*, cliente y dirección. Toma las últimas decisiones respecto la *Product backlog list*, estima el esfuerzo de los puntos del *Backlog* y los concreta en funcionalidades a desarrollar.

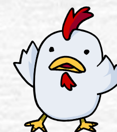
Equipo de Scrum: Se auto-organiza para lograr lo propuesto en cada *sprint*. Estima esfuerzos, crea y revisa la *Product Backlog list* e identifica problemas.

Cliente: Participa en los puntos del *Backlog* para diseñar o mejorar el sistema.

Gestión (dirección): Toma la última decisión y se encarga de los documentos, normas y convenciones. Identifica objetivos y requisitos. Ayuda a seleccionar el *product owner*, valorar los progresos y reducir el *Backlog* con el *Scrum Master*.

¿Qué nombre ponemos al restaurante?

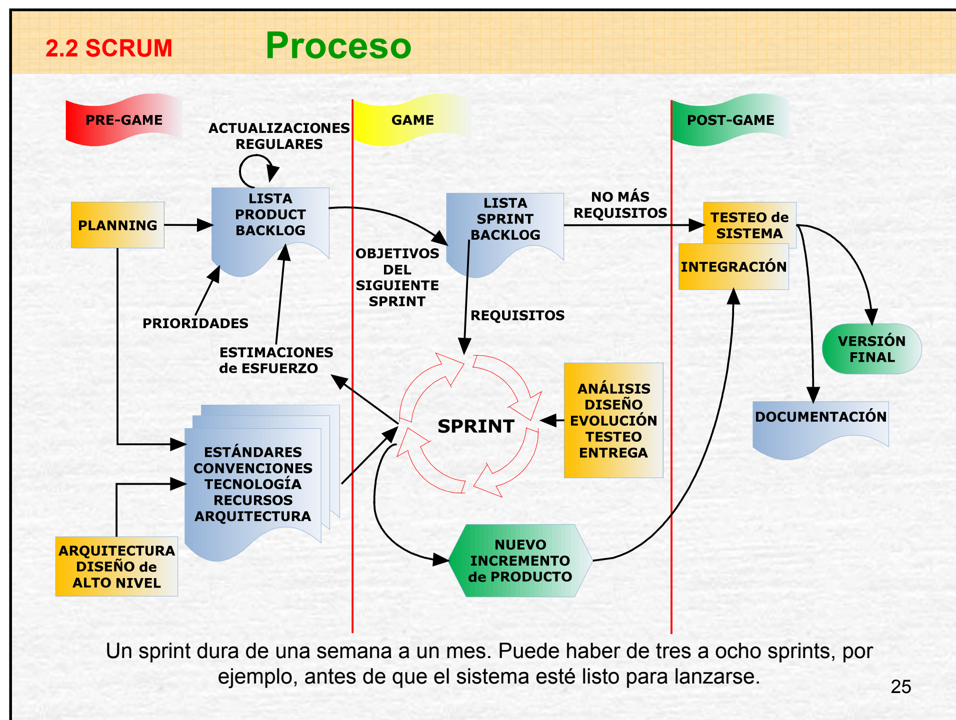
"Jamón y Huevos"



No, gracias. Yo estaré comprometido, pero tú sólo involucrada.

"Si te pueden echar por dejar que el proyecto fracase, eres un cerdo; si puedes mantener tu puesto de trabajo incluso si el proyecto fracasa, eres una gallina"

24



2.2 SCRUM Fases

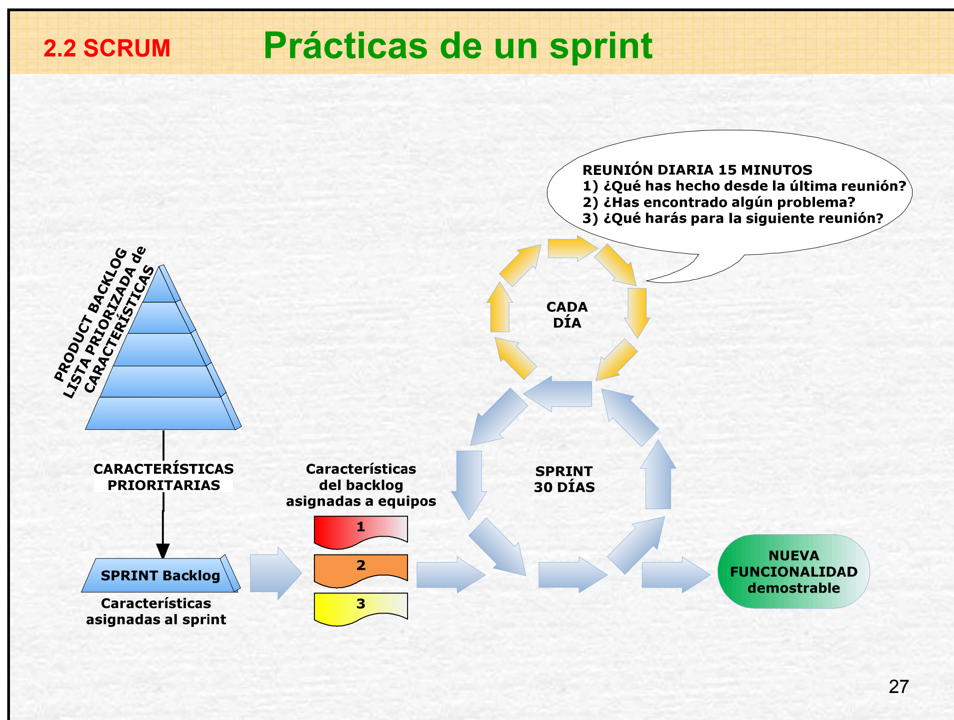
Pre-game, subfase Planning: Define el sistema a desarrollar y asegurar la financiación. Se crea una lista (actualizada en cada iteración), *Product Backlog list* con los requisitos conocidos priorizados y se estima su esfuerzo. El *planning* incluye la definición del equipo del proyecto, herramientas, análisis de riesgos, necesidades de formación y la aprobación de la gestión de comprobaciones.

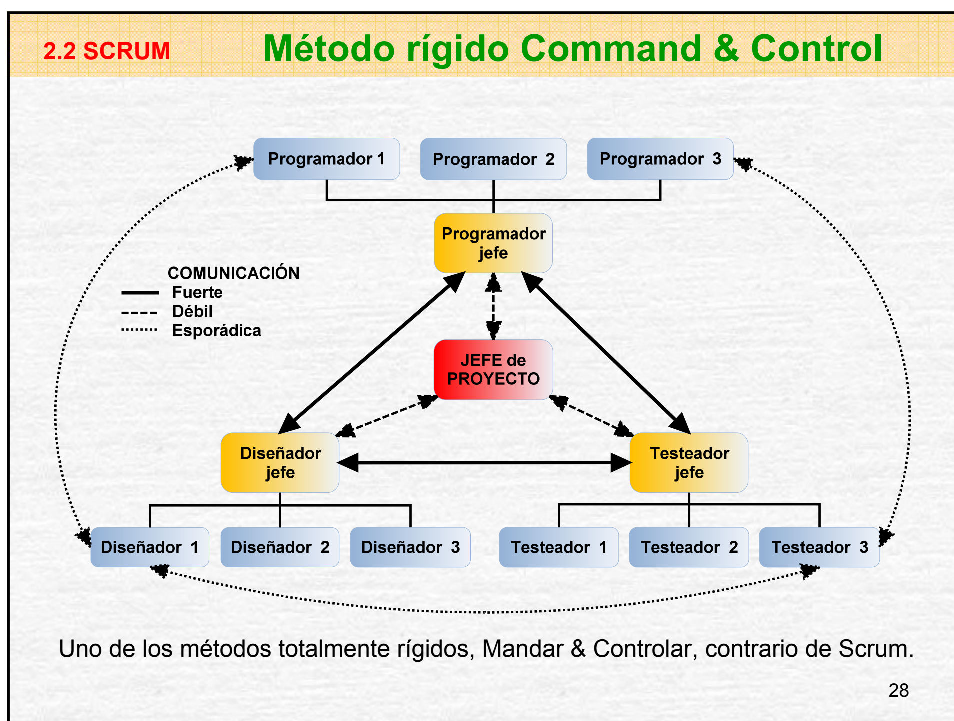
Pre-game, Subfase Diseño de alto nivel / Arquitectura: Se crea el diseño a alto nivel basándose en los requisitos actuales del *Backlog*. Se preparan contenidos de las versiones a lanzar.

Desarrollo (game): Las diferentes variables (calendario, calidad, requisitos, recursos, tecnologías, herramientas, métodos de desarrollo), que pueden cambiar durante el proceso, se controlan constantemente en los *sprints*. Los *sprints* son ciclos iterativos *(1 semana a 1 mes)* donde se desarrollan o mejoran funciones para tener nuevos incrementos. Cada *sprint* incluye las fases de requisitos, análisis, diseño, desarrollo y entrega.

Post-game: Todos los requisitos completados. No se añade ni mejora ninguna función. Sistema listo para lanzarse: se integra, se pone a prueba y se documenta.

26


2.2 SCRUM
Prácticas de un sprint
REUNIÓN DIARIA 15 MINUTOS
1) ¿Qué has hecho desde la última reunión?
2) ¿Has encontrado algún problema?
3) ¿Qué harás para la siguiente reunión?
PRODUCT BACKLOG
LISTA PRIORIZADA de
CARACTERÍSTICAS
CADA
DÍA
CARACTERÍSTICAS
PRIORITARIAS
Características
del backlog
asignadas a equipos
SPRINT
30 DÍAS
1
2
3
SPRINT Backlog
Características
asignadas al sprint
NUEVA
FUNCIONALIDAD
demostrable
27


2.2 SCRUM
Método rígido Command & Control
Programador 1
Programador 2
Programador 3
Programador
jefe
COMUNICACIÓN
Fuerte
Débil
Esporádica
JEFE de
PROYECTO
Diseñador
jefe
Testeador
jefe
Diseñador 1
Diseñador 2
Diseñador 3
Testeador 1
Testeador 2
Testeador 3
Uno de los métodos totalmente rígidos, Mandar & Controlar, contrario de Scrum.
28

2.3 CRYSTAL

Familia de métodos Crystal

Alistair Cockburn, 2002

Ingeniería de software es una aberración; es un juego cooperativo de invención y cooperación

CRITICIDAD del SISTEMA

L6	L20	L40	L80	L200	L500
E6	E20	E40	E80	E200	E500
D6	D20	D40	D80	D200	D500
C6	C20	C40	C80	C200	C500
TRANSPARENTE	AMARILLO	NARANJA	ROJO	GRANATE	AZUL

TAMAÑO del PROYECTO

29

2.3 CRYSTAL

Principios

Entrega frecuente. Entregar software a los clientes con frecuencia, puede ser diaria, semanal, mensualmente...

Comunicación osmótica. Todos trabajan en la misma sala.

Mejora reflexiva. Tomarse un tiempo (desde horas hasta alguna semana) para pensar con profundidad qué se está haciendo, cotejar notas, reflexionar y discutir.

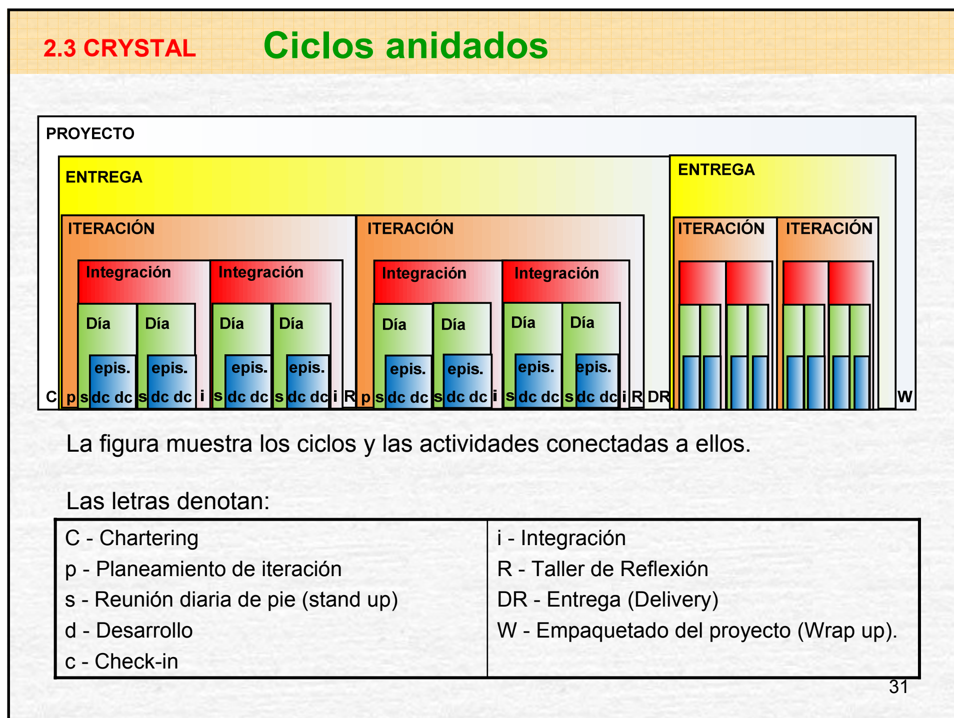
Seguridad personal. Poder decir la verdad. Decir al manager que la agenda no es realista, o a un colega que su código necesita mejorarse.

Foco. Saber qué se está haciendo y tener tranquilidad y tiempo para hacerlo. Contribuirá el *sponsor* o Patrocinador Ejecutivo.

Fácil acceso a usuarios expertos. Contacto directo con expertos semanalmente o con llamadas telefónicas.

Ambiente técnico con prueba automatizada, gestión de configuración e integración frecuente.

30



2.3 CRYSTAL

Prácticas

Exploración de 360°. Verificar requisitos, modelo de dominio, tecnología y plan de proyecto. Es preliminar al desarrollo (como inicio de RUP). En *Crystal Clear* debe durar pocos días (<14). Debe resultar una lista de los casos de uso esenciales.

Victoria temprana. Mejor pequeños triunfos iniciales que aspirar a una gran victoria tardía. La primera victoria: el Esqueleto Ambulante. En vez de *"lo peor primero"* de XP, (puede bajar la moral) se prefiere "lo más fácil primero, lo más difícil segundo".

Esqueleto ambulante. Versión simple, pero completa. No suele ser robusto; sólo camina, y carece de la funcionalidad de la aplicación real, que se agregará por pasos. Es diferente de una *spike* (para saber si se está en la dirección correcta).

Rearquitectura incremental. No es conveniente interrumpir el desarrollo para corregir la arquitectura. La arquitectura debe evolucionar, manteniendo el sistema en ejecución.

Radiadores de información. Es una lámina comprensible, actualizada y visible. Puede ser una página web, pero mejor en una pared. Muestra la iteración actual, pruebas pasadas / pendientes, el número de casos de uso o historias entregadas…

32

2.3 CRYSTAL **Papeles y artefactos**

Patrocinador. Se encarga de la Declaración de Misión con Prioridades de Compromiso (Trade-off). Consigue los recursos y define la totalidad del proyecto.

Usuario experto. Junto con el experto comercial, crea la Lista de Actores-Objetivos y el Archivo de Casos de Uso y Requisitos. Debe familiarizarse con el uso del sistema, sugerir atajos de teclado, modos de operación, información a visualizar simultáneamente, navegación

Diseñador principal. Produce la Descripción Arquitectónica. El Diseñador Principal tiene funciones de coordinador, arquitecto, mentor y programador más experto. Se supone que debe ser al menos un profesional de nivel 3.

Diseñador-Programador. unto con el Diseñador Principal se encarga de: Borradores, Modelo Común de Dominio, Notas y Diagramas de Diseño, Código Fuente, Código de Migración, Pruebas, Empaquetado del sistema.

Experto comercial. Junto con el Usuario Experto crea la Lista de Actores-Objetivos y el Archivo de Casos de Uso y Requisitos. Conoce las reglas y políticas del negocio.

Coordinador. Con la ayuda del equipo, crea el Mapa de Proyecto, el Plan de Entrega, el Estado del Proyecto, la Lista de Riesgos, el Plan y Estado de Iteración y la Agenda de Visualización.

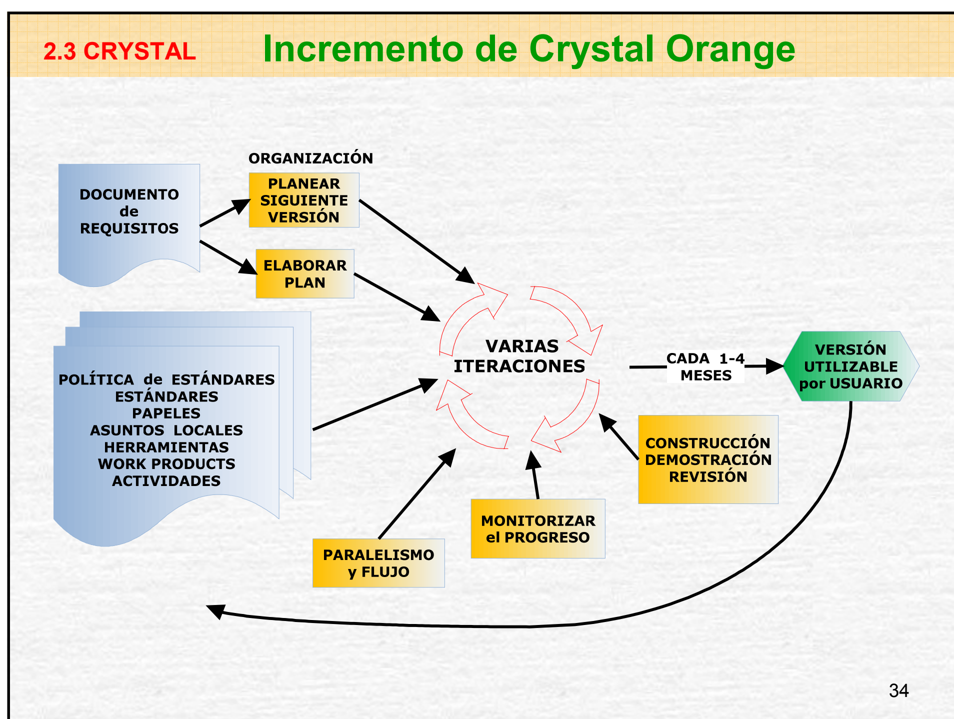
Verificador. Confecciona el informe de bugs.

Escritor técnico. Elabora el manual del usuario.

Crystal Orange introduce nuevos papeles: Diseñador de interfaz usuario, Diseñador de base de datos, Ayudante técnico, Analista/diseñador comercial, Usuario experto, Arquitecto, Mentor de diseño, Reutilización (Reuse point), Escritor, Tester.

33

2.3 CRYSTAL **Incremento de Crystal Orange**



34

2.4 FDD **Feature Driven Development - FDD**

Palmer y Felsing, 2002

- Se centra sólo en diseño y desarrollo.
- Adecuado para procesos críticos (enfatiza calidad).
- No exige la presencia del cliente.
- Bastante jerarquizado: programador jefe dirige a propietarios de clases que dirigen a equipos de características.
- Consta de 5 procesos secuenciales (las dos últimas fases iterativas) para diseñar y construir el sistema:

Elaborar modelo general → Construir lista de características → Planear por característica → Diseñar por característica → Construir por característica

35

2.4 FDD **Papeles**

Papeles clave

- *Manager del proyecto*
- *Arquitecto jefe*
- *Jefe de desarrollo*
- *Programador jefe*
- *Propietarios de las clases*
- *Experto del dominio*

Papeles de apoyo

- *Gestor de versiones*
- *Experto del lenguaje*
- *Ingeniero de las build*
- *Toolsmith*
- *Administrador de Sistemas*

Papeles adicionales

- *Probadores*
- *Distribuidores o Desplegadores (deployers)*
- *Escritores técnicos*

Un miembro del equipo puede tener varios papeles, y un papel puede ser compartido por varias personas.

36

2.4 FDD

Prácticas

Modelado de los objetos del dominio: Exploración del dominio del problema. Los resultados se engloban en el framework donde se agregan las funcionalidades.

Desarrollar por característica: Desarrollar y seguir el progreso con una lista de funciones importantes para el cliente formada por funcionalidades.

Propiedad individual de la clase (código): Cada clase tiene un único responsable de su fiabilidad, rendimiento, e integridad conceptual.

Equipos de características: Equipos pequeños y formados dinámicamente.

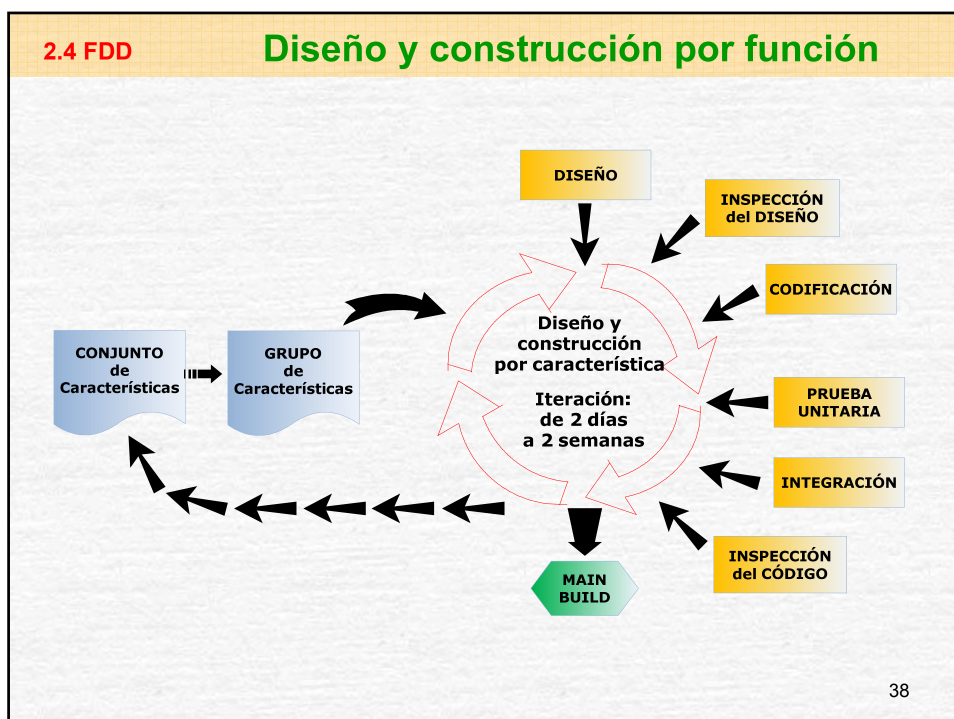
Inspección: Uso de los mejores mecanismos para detectar fallos.

Builds regulares: Siempre hay un sistema que funciona para mostrar. Son las bases donde se añadirán nuevas funcionalidades.

Gestión de configuración: Permite la identificación y control de las últimas versiones de cada fichero de código fuente completado.

Informes de progreso: Se informa a todos los niveles organizativos necesarios del progreso basándose en partes completadas.

37



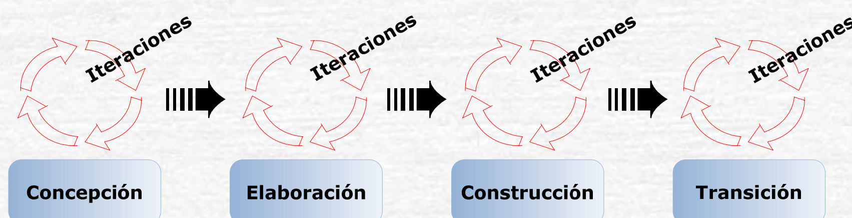
2.5 RUP, EUP, AUP **Rational Unified Process – RUP & AUP**

Philippe Kruchten, 2000

Empezó en la Rational Corporation para complementar UML

Historia: 1988: Objectory 1.0, 1998: RUP 5.0, 2004: EUP y 2005: AUP

Un proyecto RUP tiene cuatro fases que se dividen en iteraciones, cada una con el propósito de crear un fragmento de software operativo. Cada iteración dura de 2 semanas o menos a 6 meses.



Concepción Elaboración Construcción Transición

39

2.5 RUP, EUP, AUP **Fases de RUP**

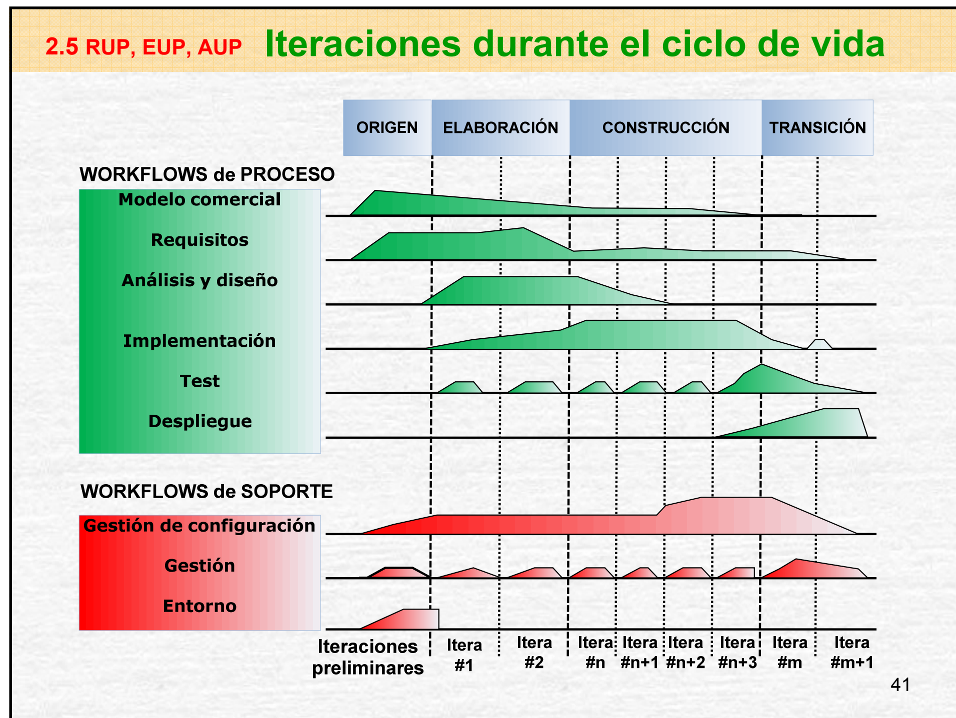
Fase Concepción (inception): Se definen los objetivos del ciclo de vida para tener en cuenta las necesidades de cada participante. Se establecen las posibilidades, criterio de aceptación y casos de uso. Se diseñan posibles arquitecturas, y se estima el calendario y costo para el proyecto entero.

Fase de Elaboración: Establece la base de la arquitectura y se define el plan. Los requisitos y planes son suficientemente estables. RUP pone énfasis en la automatización de herramientas. Se identifican la mayoría de los casos de uso y se crea un prototipo ejecutable.

Fase de Construcción: Se desarrollan los componentes y características restantes y se integran en el producto para pasar las pruebas. RUP considera esta fase un proceso industrial enfatizando gestionar recursos y controlar costos, horarios, y calidad. Se crean rápido una o más versiones, pero con calidad.

Fase de transición: El software está maduro (según número e importancia de los cambios pedidos) para mostrarse a los usuarios. Probar las versiones beta, preparar a los usuarios y gestores de los sistemas, y pasar el producto a los departamentos de marketing, distribución y ventas. Ahora se escribe la documentación.

40



2.5 RUP, EUP, AUP **Prácticas de RUP**

Desarrollo iterativo: En pequeños incrementos, breves iteraciones para identificar riesgos y problemas rápido.

Gestionar requisitos: Identificar los requisitos cambiantes y adaptar el software. Los requisitos pueden priorizarse, filtrarse y seguirse. Requisitos bien definidos facilitan la comunicación.

Arquitecturas basadas en componentes: Da flexibilidad: se aíslan partes probables de cambiar y se manejan mejor. Los componentes reutilizables ahorran tiempo.

Software de visualización de modelos: Facilitan entender sistemas complejos. Un método como UML captura la arquitectura del sistema y diseño objetivamente.

Verificar la calidad: En cada iteración, para encontrar fallos pronto reduciendo el costo de arreglarlo.

Consideración de los cambios: Cualquier cambio en los requisitos debe tratarse. La madurez del software se ve en la frecuencia e importancia de los cambios hechos.

42

2.5 RUP, EUP, AUP **Enterprise Unified Process – EUP**

Respecto a RUP, EUP es una extensión que añade 2 fases y disciplinas:

Fase de PRODUCCIÓN
El sistema ha sido desplegado, hasta que sea reemplazado por una nueva versión o producto, o sea retirado.

Fase de RETIRO
El sistema se elimina totalmente de la producción. Las actividades son convertir y archivar datos, y asegurarse que no habrá efectos colaterales.

Disciplina SOPORTE Y OPERACIONES
En Construcción (o Elaboración) se diseñará un plan de S&O, documentos y manuales de formación. En Transición, se mejorarán.

Disciplinas DE EMPRESA, core enterprise disciplines
Son 7: *Enterprise Administration, Enterprise Architecture, Enterprise Business Modeling, People Management, Portfolio Management, Software Process Improvement (SPI),* y *Strategic Reuse.* Convierten el *Unified Process* en un ciclo de vida de un sistema de IT completo no sólo un proceso de desarrollo. El trabajo más significativo ocurre antes de la fase Concepción.

43

2.5 RUP, EUP, AUP **Agile Unified Process – AUP**

Agile UP 1.1, es una simplificación del RUP, ahora de IBM.

Su ciclo de vida tiene parte de comportamiento en cascada a grosso modo, pero es iterativo en sus pasos y proporciona versiones incrementales

PRINCIPIOS DE AGILE UP

1) Los empleados saben lo que hacen, conocen a alto nivel el proyecto.

2) Simplicidad y documentación concisa.

3) Agilidad, seguimiento de los principios ágiles.

4) Centrarse en actividades que proporcionen valor al cliente.

5) Independencia de herramientas software usadas.

6) Adaptar el producto a las necesidades.

DELIVERABLES mínimos:

Sistema, código fuente, conjunto de tests de regresión, scripts de instalación, documentación, notas de la versión, modelos de requisitos y de diseño.

44

2.5 RUP, EUP, AUP **Agile Unified Process**

Fase	Objetivos	Milestone
Concepción	Identificar el alcance inicial del proyecto, arquitectura principal, y obtener los fundamentos del proyecto inicial y la aceptación de los participantes.	Objetivos del Ciclo de Vida (LCO)
Elaboración	Comprobar la arquitectura del sistema.	Arquitectura de Ciclo de Vida (LCA)
Construcción	Construir software que funcione a base de incrementos regulares, y según las prioridades de los participantes.	Capacidad Operacional Inicial (IOC)
Transición	Validar y desplegar el sistema en el entorno de producción.	Versión de Producto (PR)

Papeles principales:
Administrador de base de datos ágil, Modelador ágil, Managers de configuración, de tests y de proyecto, Desplegador, Desarrollador, Ingeniero de proceso, Revisor, Participante, Escritor técnico, Tester, Especialista de herramientas.

45

2.5 RUP, EUP, AUP **Agile Unified Process**

Disciplina	Objetivos
Modelo	Entender el negocio de la organización, el dominio del problema que ataca el proyecto, e identificar una solución viable.
Implementación	Transformar el modelo(s) en código ejecutable y realizar un testeo básico, en particular, tests unitarios.
Test	Realizar una evaluación objetiva para asegurar la calidad. Esto incluye encontrar defectos, validar que el sistema funciona como se preveía, y verificar que se reúnen los requisitos.
Despliegue	Planear la entrega del sistema y ejecutar el plan para que los usuarios finales dispongan del sistema.
Gestión de configuración	Gestionar el acceso a los work products del proyecto. Esto incluye controlar las versiones de los work products y sus cambios.
Gestión de proyecto	Dirigir las actividades del proyecto: gestión de riesgos, asignar tareas, hacer seguimientos, coordinar y asegurar que se cumple el calendario y el presupuesto.
Entorno	Respaldar el resto del esfuerzo asegurando que el equipo dispone de las guías, estándares y herramientas soft y hard cuando las necesite.

46

2.6 DSDM

Dynamic Systems Development Method

DSDM Consortium, 1994

DSDM: en lugar de fijar la cantidad de funcionalidad en un producto, y después ajustar tiempo y recursos para alcanzar esa funcionalidad, **se prefiere ajustar la cantidad de funcionalidad conforme a un tiempo y recursos fijados**.

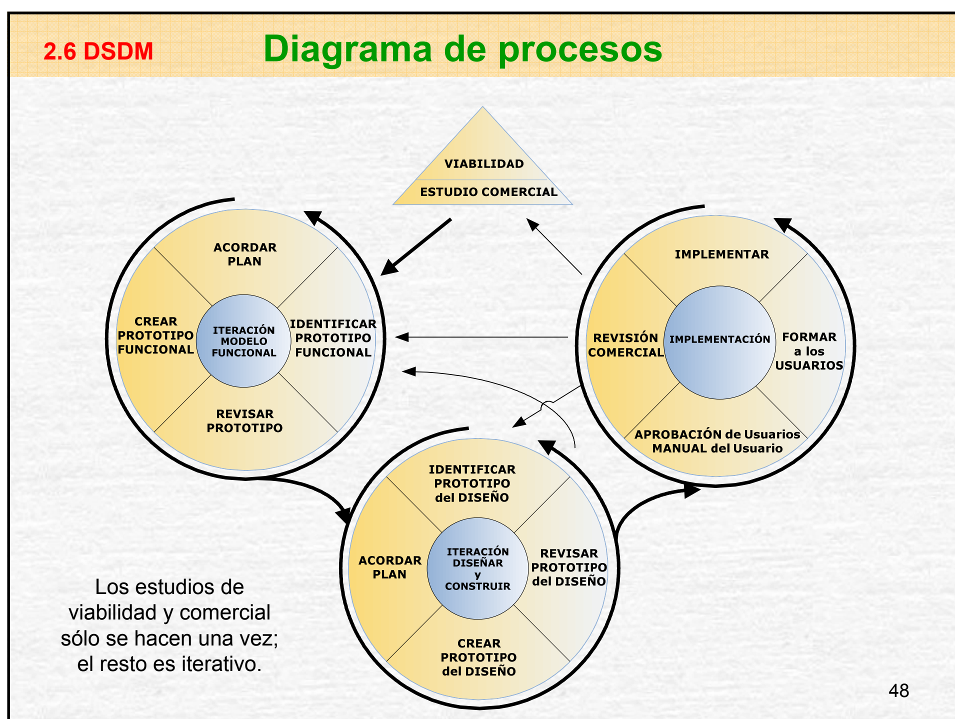
DSDM es un framework para el desarrollo de RAD sin ánimo de lucro y no propietario, mantenido por el DSDM Consortium. Además de ser un método, proporciona un framework de controles para RAD, complementado con consejos sobre cómo usarlos eficazmente.

Las iteraciones son time-boxes que duran un tiempo predefinido (de unos días a unas semanas), y la iteración debe completarse en ese tiempo. El tiempo permitido se decide de antemano y garantiza los resultados buscados.

47

2.6 DSDM

Diagrama de procesos



Los estudios de viabilidad y comercial sólo se hacen una vez; el resto es iterativo.

48

2.6 DSDM **Prácticas**

Usuario activo: Se requiere algún usuario entendido: feedback rápido y preciso.

Equipos autorizados para tomar decisiones: Evitar procesos que se tarde mucho en decidir; los ciclos de desarrollo deben ser rápidos.

Entrega frecuente de productos: facilita corregir decisiones erróneas.

Adecuar las entregas al propósito comercial: Construir el producto adecuado antes de construirlo bien: "**Build the right product before you build it right**".

Desarrollo iterativo e incremental: Los requisitos cambian. Con el desarrollo iterativo, pueden encontrarse y corregirse los errores pronto.

Los cambios durante el desarrollo son reversibles: Con iteraciones cortas y asegurando poder volver al paso anterior, puede corregirse el camino.

Establecer una baseline a alto nivel para los requisitos: Congelar requisitos centrales sólo a alto nivel; los requisitos de los detalles pueden cambiar. Al avanzar, se congelan más requisitos.

Pruebas continuas: Probar es incremental para programadores y usuarios. Tests de regresión (probar lo que antes funcionaba después de actualizar el código).

Cooperación y colaboración entre participantes: Organización comprometida con DSDM. Elegir qué partes se entregan y cuáles no requiere acuerdo común.

49

2.6 DSDM **Papeles**

Patrocinador Ejecutivo. Autoridad y responsabilidad financiera, tiene la última palabra.

Visionario. Usuario con la percepción más exacta de los objetivos del proyecto. Asegura que se cumplan los requisitos esenciales.

Usuario embajador. Conoce la comunidad de usuarios a la que se dirige el proyecto. Asegura un diseño adecuado. Revisa la documentación, crea la documentación para el usuario y supervisa los tests de usuarios y su formación.

Usuario Asesor (Advisor). Representa otros puntos de vista importantes. Aprueba los diseños y los prototipos.

Jefe de Proyecto. Coordina e informa, elabora el calendario, supervisa el progreso, controla la gestión de riesgos, trata problemas inesperados, etc.

Coordinador técnico. Define la arquitectura del sistema y es responsable de la calidad técnica, control técnico, configuración del sistema y control de versiones.

Líder del Equipo. Se encarga que todo el equipo vaya en la misma dirección. Gestiona el control de cambios y la documentación del proyecto. Informa del progreso.

Desarrolladores. Modelan e interpretan los requisitos, convirtiéndolos en prototipos y código entregable. No se diferencia analista, diseñador y programador.

Testeador. Realiza los tests (no de usuario ni tests de unidad) según la estrategia de test del plan de desarrollo. El *tester* es parte del equipo de desarrollo.

Escriba. El *scribe* registra los requisitos, acuerdos y decisiones alcanzadas en las reuniones, talleres y sesiones acerca de los prototipos. Distribuye la documentación.

50

2.7 ASD

Adaptive Software Development – ASD

James Highsmith (*Cutter Consortium*), 2000

ASD requiere complementarse con otros métodos. Profundiza en la filosofía del movimiento ágil.

"Si un componente tiene una fiabilidad del 99.9%, cuando juntamos 200, la fiabilidad se reduce a un 82%, y si ponemos 1.000 bajaría hasta un 37%"

Las revisiones de calidad son bastante escasas (sólo al final de cada ciclo).

La presencia del cliente en ASD se apoya con sesiones JAD, desarrollo de aplicaciones compartidas (*Joint Application Development*).

COLABORAR
ESPECULAR
APRENDER

Una **sesión JAD** es básicamente un taller donde diseñadores y representantes del cliente discuten las características del producto.

51

2.7 ASD

Fases del ciclo de vida

Prácticas	- desarrollo iterativo - planning basado en características (basadas en componentes) - revisiones en grupo enfocadas al cliente
Participantes en una sesión JAD	- un ayudante para planear y moderar la sesión, - un secretario para tomar actas, - el jefe del proyecto, - representantes del cliente, y - representantes de los programadores.

BUCLE de APRENDIZAJE

INICIO de PROYECTO → PLAN de CICLO ADAPTATIVO → INGENIERÍA CONCURRENTE de COMPONENTES → REVISIÓN de CALIDAD → CONTROL CALIDAD FINAL y ENTREGA

Especular | Colaborar | Aprender

52

2.8 OSS Open Source Software Development

Cómo se gestiona un proceso OSS:

- Sistemas con un gran número de voluntarios
- El trabajo no se asigna; cada uno escoge la tarea que le interese
- No existe ningún diseño explícito a nivel de sistema
- No hay plan del proyecto, calendarios o lista de entregas
- El sistema crece con incrementos pequeños
- Los programas se testean frecuentemente
- Software en continua mejora, "nunca" se dan por acabados

Éxitos: Linux, Servidor web Apache, lenguaje Perl, Sendmail…

DESCUBRIR PROBLEMA → ENCONTRAR VOLUNTARIOS → IDENTIFICAR SOLUCIÓN → ESCRIBIR CÓDIGO y TESTEAR → REVISAR CAMBIOS CÓDIGO → ENTREGAR CÓDIGO y DOCUMENTACIÓN → GESTIONAR VERSIONES

Aunque el código OSS no es un producto comercial (usa la General Public License), empresas como Redhat empaquetan partes y las comercializan.

53

2.8 OSS Papeles y motivaciones

PAPELES:

1) **Líderes del proyecto**: tienen la responsabilidad global del proyecto y normalmente han escrito el código inicial.

2) **Programadores voluntarios**: crean y presentan código. Son:

a) Miembros Senior o programadores con más autoridad global.
b) Programadores Periféricos que crean y presentan cambios del código.
c) Contribuyentes Ocasionales.
d) Ayudantes y diseñadores acreditados.

3) **Otros individuos:** prueban, identifican *bugs* e informan de fallos.

4) **Posters** participan en *newsgroups* y discusiones; no programan.

MOTIVACIONES y PAUTAS:

a) **Tecnológica**: necesidad de código robusto, ciclos de desarrollo rápidos, normas estrictas de calidad, fiabilidad y estabilidad, plataformas y estándares abiertos.

b) **Barata**: la necesidad colectiva de costo y riesgo compartido.

c) **Socio-política**: satisfacer el reto de programadores particulares, la reputación del los colegas, el deseo de trabajo útil, comunidad orientada al idealismo.

La cultura OSS es como una fusión de colectivismo e individualismo: dar a la comunidad es lo que hace un héroe al individuo ante los ojos de los demás.

54

2.9 LSD

Lean Software Development - LSD

LD por Bob Charette, 1990
LSD por Tom y Mary Poppendieck, 2003

(lean = sin grasa). Darrel Norton también lo desarrolló.

ELIMINAR DESPERDICIO
DESCUBRIR DESPERDICIO

Una solución al 80% hoy, en vez de una al 100% mañana. Charette

CICLO de PROYECTO (el juego cooperativo): Atraer, Plasmarlo por escrito, Entregar, Cosechar

CICLO de ENTREGA (1 semana - 3 meses): Calibrar y entregar el plan, Iterar, Entregar a usuarios reales, Completar (celebrar y reflexionar)

ITERACIÓN (1 semana - 3 meses): Estimar y planear, Desarrollar, Completar (celebrar y reflexionar)

CICLO de ITERACIÓN (30 minutos - 3 días): Desarrollar, Integrar, Ejecutar test de sistema

EPISODIOS de DESARROLLO (minutos - horas): Diseñar, Codificar y testear, Comprobar

Entre equipos
En el equipo
Semana
Reunión de pie
Día

CICLO de MANTENIMIENTO: Informar del bug, Integrar, Aceptar solución al bug

55

2.9 LSD

7 Principios y 22 herramientas (D. Norton)

Principio 1:
Eliminar desperdicios
H1: Ver los desperdicios. Son 7: Inventario, demasiados procesos, demasiadas funciones, transporte (cambiar de tareas o proyecto), Espera, Movimiento y Defectos.
H2: Observar el mapa de flujo de valor.

Principio 2:
Amplificar el conocimiento
H3: Feedback
H4: Iteraciones
H5: Sincronización
H6: Desarrollo set-based.

Principio 3: Decidir tan tarde como sea posible
H7: “options thinking”
H8: El último momento responsable
H9: Tomar decisiones

SINCRONIZACIÓN CON EL EQUIPO
LISTA PRIORIZADA de CARACTERÍSTICAS
SINCRONIZAR CON EL CLIENTE
INCREMENTO de FUNCIONALIDAD
Reuniones semanales
Sincronización entre equipos
Reuniones diarias

56

2.9 LSD — 7 Principios y 22 herramientas (D. Norton)

Principio 4: Entregar tan rápido como sea posible
H10: Sistemas de arrastre (pull systems)
H11: Teoría de colas
H12: Costo del Retraso

Principio 5: Otorgar poder al equipo
H13: Autodeterminación
H14: Motivación
H15: Liderazgo
H16: Aptitud, competencia

Principio 6: Integridad incorporada.
H17: Integridad percibida
H18: Integridad conceptual
H19: Refactoring
H20: Testeos

Principio 7: Ver la totalidad
H21: Medidas
H22: Contratos

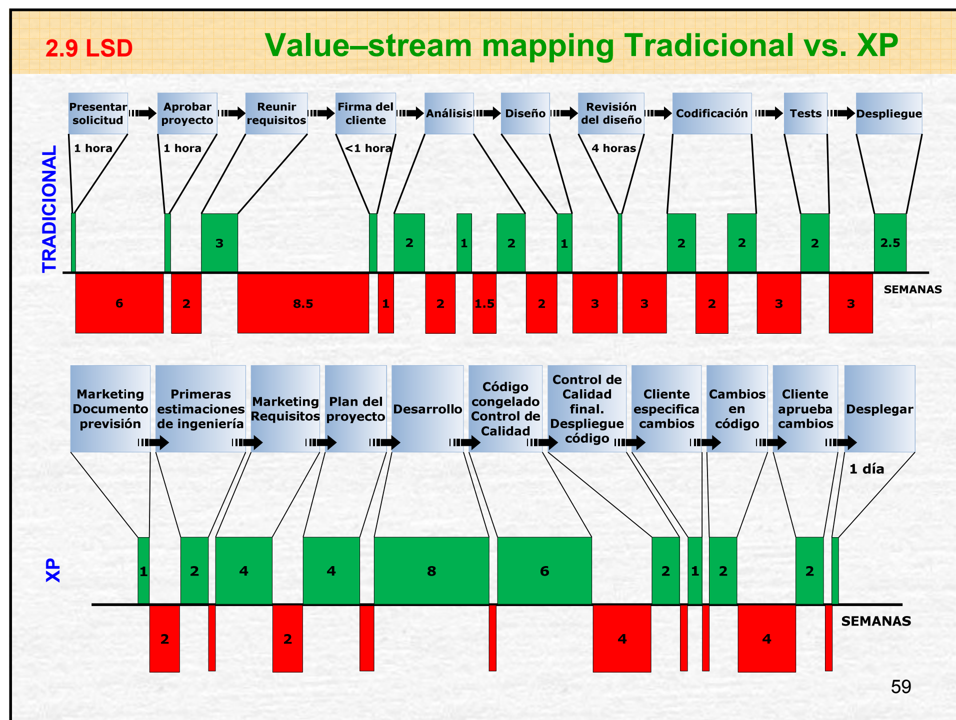
57

2.9 LSD — Value–stream mapping ágil

El mapa de flujo de valor ayuda a enfocar la atención en los productos y en el cliente en lugar de en las organizaciones, recursos, tecnologías, procesos, etc. Se mira el proceso desde el punto de vista del cliente, señalando cuándo se está aportando valor al cliente y cuándo no (esperas).

El cliente recibe lo que necesita hoy, no lo que necesitaba ayer. Norton

58



2.10 AM

Agile Modelling – AM

Scott Ambler (2002)
Agile Model-Driven Development

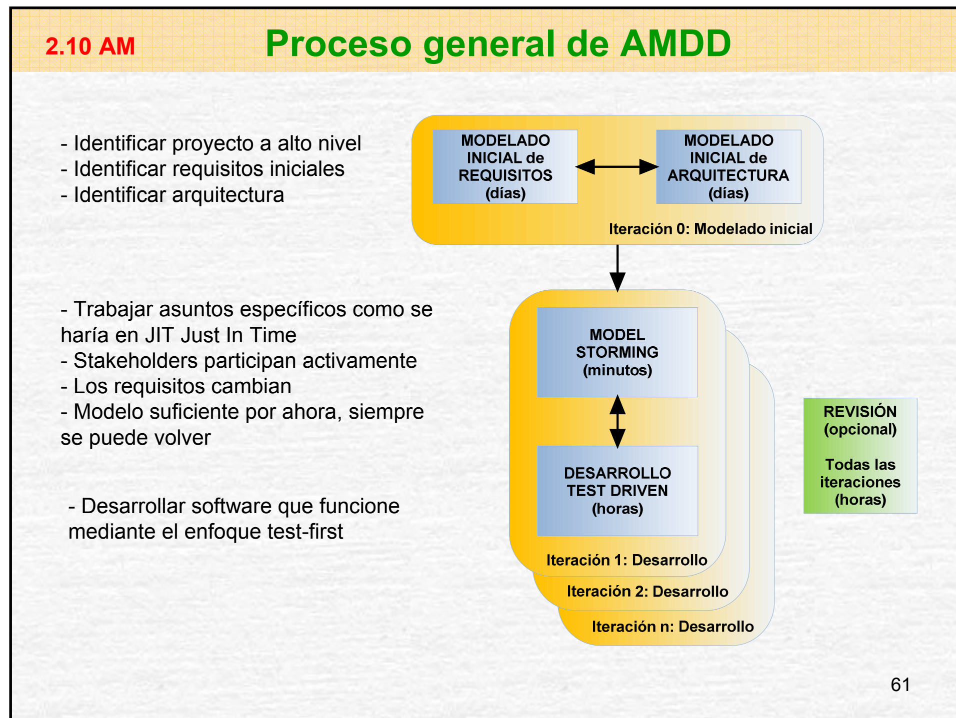
AM es una estrategia de modelado (de clases, de datos, de procesos) pensada para contrarrestar la idea de que los métodos ágiles no modelan y no documentan. Orientar el modelado de una manera efectiva y ágil.

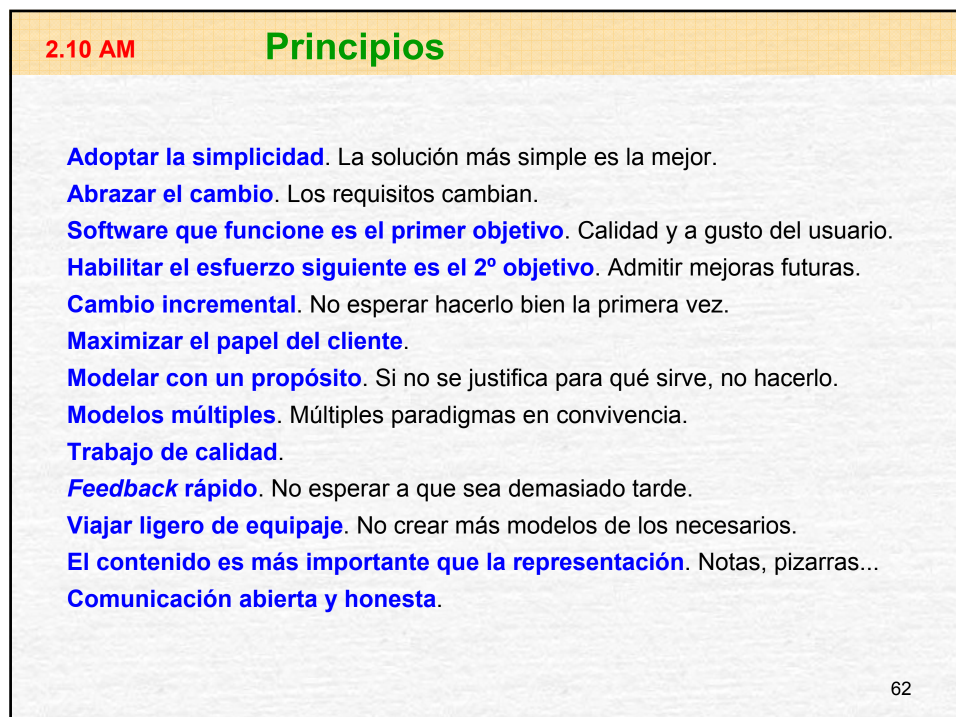
Se recomienda su uso con XP, MSF, RUP o EUP.

Sus ideas más destacables son:

- Al principio del proyecto, se hace el modelo inicial, para conocer los requisitos fundamentales e identificar una posible arquitectura.
- Durante los ciclos de desarrollo, se modela unos minutos, luego se implementa durante horas o días hasta necesitar hacer más modelos.
- Las prácticas de AM de “Modelar con otros” y “Propiedad colectiva” proporcionan las revisiones necesarias.

60


2.10 AM
Proceso general de AMDD
- Identificar proyecto a alto nivel
- Identificar requisitos iniciales
- Identificar arquitectura
MODELADO INICIAL de REQUISITOS (días)
MODELADO INICIAL de ARQUITECTURA (días)
Iteración 0: Modelado inicial
- Trabajar asuntos específicos como se haría en JIT Just In Time
- Stakeholders participan activamente
- Los requisitos cambian
- Modelo suficiente por ahora, siempre se puede volver
MODEL STORMING (minutos)
DESARROLLO TEST DRIVEN (horas)
REVISIÓN (opcional)
Todas las iteraciones (horas)
- Desarrollar software que funcione mediante el enfoque test-first
Iteración 1: Desarrollo
Iteración 2: Desarrollo
Iteración n: Desarrollo
61


2.10 AM
Principios
Adoptar la simplicidad. La solución más simple es la mejor.
Abrazar el cambio. Los requisitos cambian.
Software que funcione es el primer objetivo. Calidad y a gusto del usuario.
Habilitar el esfuerzo siguiente es el 2º objetivo. Admitir mejoras futuras.
Cambio incremental. No esperar hacerlo bien la primera vez.
Maximizar el papel del cliente.
Modelar con un propósito. Si no se justifica para qué sirve, no hacerlo.
Modelos múltiples. Múltiples paradigmas en convivencia.
Trabajo de calidad.
Feedback rápido. No esperar a que sea demasiado tarde.
Viajar ligero de equipaje. No crear más modelos de los necesarios.
El contenido es más importante que la representación. Notas, pizarras...
Comunicación abierta y honesta.
62

2.10 AM Prácticas

AM propone unas prácticas principales:

- Colaboración activa de los participantes.
- Aplicación de los artefactos correctos.
- Propiedad colectiva de todos los elementos.
- Crear diversos modelos en paralelo.
- Crear contenido simple.
- Iterar a otro artefacto cuando nos quedemos bloqueados.
- Modelo en incrementos pequeños.
- Modelar con otros miembros del equipo.
- Demostrarlo con código.
- Utilizar las herramientas más simples (CASE, o mejor pizarras, tarjetas, post-it).
- Exhibir públicamente los modelos para que los vea todo el equipo.
- Diseñar modelos de manera simple.
- Guardar la información en un único sitio y sin repetirla.

Y también unas prácticas complementarias:

- Aplicación de estándares de modelado.
- Aplicación adecuada de patrones de modelado.
- Descartar los modelos temporales.
- Formalizar modelos de contrato.
- Actualizar sólo cuando lo que se estaba haciendo era perjudicial.

63

2.11 Evo Evolutionary Project Management – Evo

Tom Gilb, 1988 (desde 1976)

En proyectos evolutivos, las Metas se desarrollan tratando de comprender de quiénes vienen (Participantes), qué es lo que son (medios y fines) y cómo expresarlas (cuantificables, significantes y verificables). Poca documentación.

Una herramienta de Evo es la "***Herramienta-?***"; consiste en preguntar "por qué" a cada meta o requisito aparente, haciéndolo iterativamente sobre las respuestas que se van dando, para determinar la raíz del requisito real.

Evo utiliza el lenguaje **Planguage** para cuantificar. Un ejemplo de la especificación de una meta de satisfacción del cliente:

```
CUSTOMER.SATISFACTION
SCALE: promedio de satisfacción del cliente 1-5
PAST [2003] 2.5
GOAL [2004] 3.5
```

64

2.11 Evo

Planear – Hacer – Estudiar – Actuar

Influencias de Evo: el método PSDA de Schewart y los valores de Deming, padres del control estadístico de calidad que dieron lugar a 6Sigma (Total Quality Management), entre otros.

PLANEAR
Elaborar un plan para mejorar la calidad en un proceso.

HACER
Ejecutar el plan, primero a pequeña escala.

ACTUAR
Hacer el plan definitivo o estudiar ajustes.

ESTUDIAR
Evaluar el feedback para confirmar o ajustar el plan.

Evo es una metodología probada desde hace mucho tiempo en *NASA*, *HP*, *Douglas Aircraft*, la Marina británica, etc. El estándar MIL-STD-498 del Departamento de Defensa y su correspondiente estándar civil IEEE doc 12207 homologan el uso del modelo de entrega evolutiva. DSDM y RUP han logrado un reconocimiento comparable.

65

2.11 Evo

Principios

1) Se entregarán temprano y con frecuencia resultados con valor para los participantes.
2) El siguiente paso de entrega de Evo será el que proporcione el mayor valor para el participante en ese momento.
3) Los pasos de Evo entregan requisitos especificados de forma evolutiva.
4) No sabemos los requisitos por anticipado, pero podemos descubrirlos más rápidamente intentando proporcionar valor real a participantes reales.
5) Evo es ingeniería de sistemas holística (todos los aspectos necesarios del sistema deben ser completos y correctos) y con entrega a un ambiente de participantes reales.
6) Los proyectos de Evo requieren una arquitectura abierta (requisitos cambian).
7) El equipo, concentrará su energía hacia el éxito del paso actual, sin gastar energías en pasos futuros hasta que hayan dominado los actuales.
8) Se aprende de la experiencia: qué funciona y aporta valor. Evo ataca los problemas al principio.
9) Una Entrega temprana, a tiempo, porque se ha priorizado desde el inicio.
10) Evo debería permitirnos poner a prueba nuevos procesos de trabajo y deshacernos de los que funcionan mal tan pronto como los identifiquemos.

66

2.11 Evo Pilares

1) **Metas, Valores y Costos** – Cuánto y cuántos recursos. Las Metas y Valores del equipo se llaman también, objetivos, metas estratégicas, requisitos, propósitos, fines, ambiciones, cualidades e intenciones.

2) **Soluciones** – Banco de ideas sobre la forma de alcanzar las Metas y Valores dentro del rango de los Costos.

3) **Estimación de Impacto** – Confrontar Soluciones con Metas y Costos para averiguar si se tienen ideas adecuadas para lograr Metas y Costos.

4) **Plan Evolutivo** – Inicialmente una idea general de la secuencia a desarrollar y evolucionar hacia las Metas. Los detalles evolucionan junto con el resto del plan a medida que se desarrolla el producto/servicio.

5) **Funciones** – Describen qué hace el sistema. Son totalmente secundarias.

67

2.11 Evo Tabla de estimación de impacto

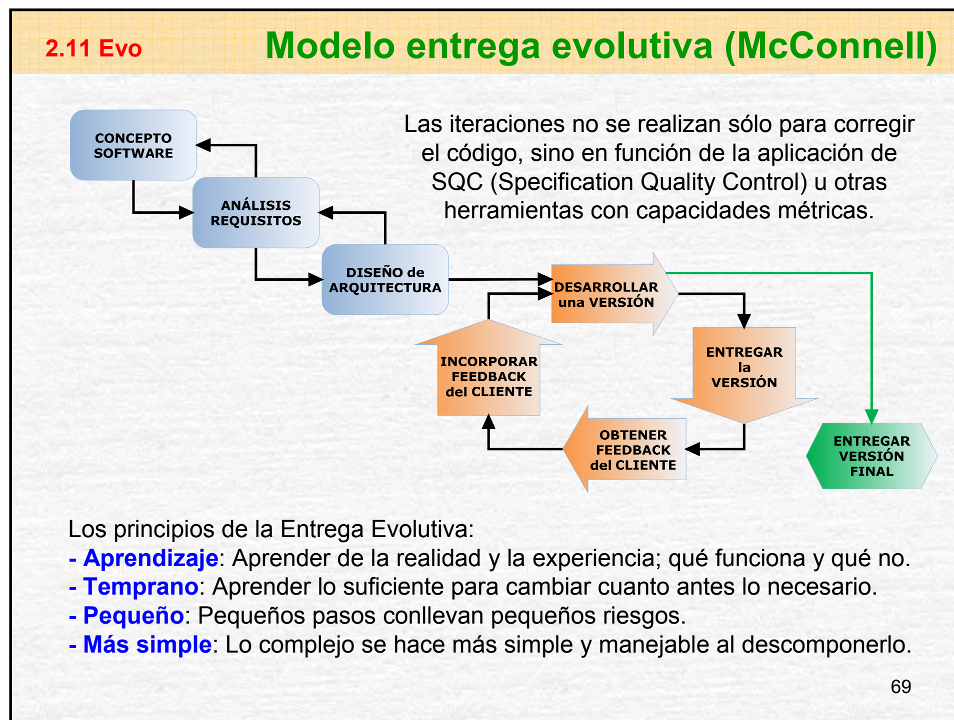
Ejemplo: se estima que la solución A proporcionará:
- un 10% del total (desde el paso anterior), en nuestra escala de “usabilidad”
- efectos negativos (-10%) respecto al objetivo de seguridad
- un 50% de mejora, hacia el objetivo de rendimiento

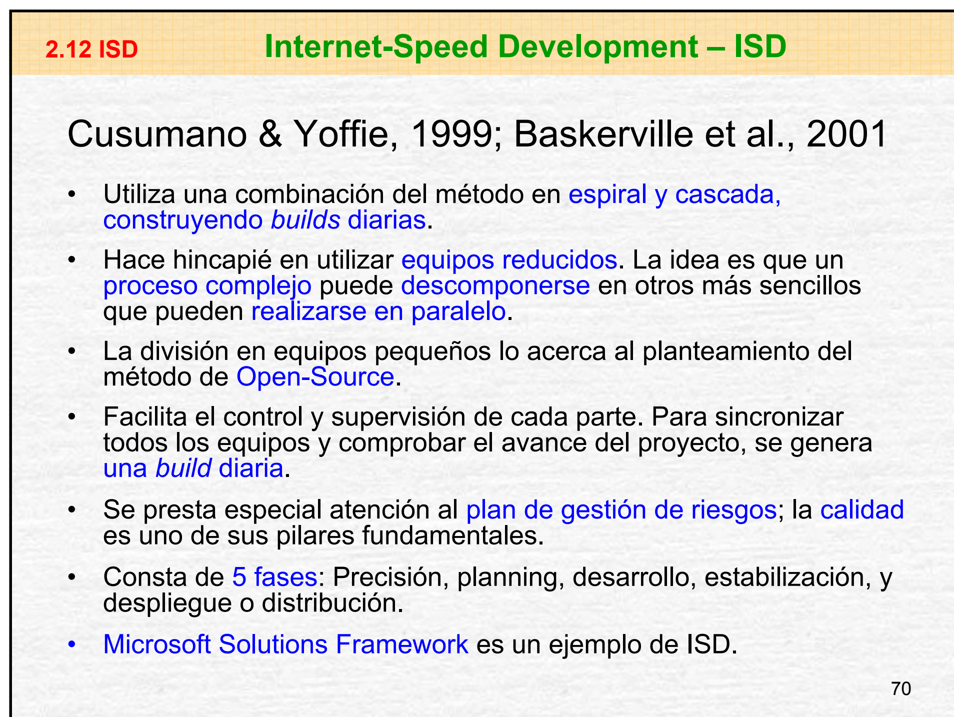
En total, la calidad se vería aumentada un 10%-10%+50%=50%.

Respecto a recursos de desarrollo, consumirá el 20%. Dividiendo la calidad total entre los recursos consumidos, la solución C obtiene la mejor relación.

	SOLUCIÓN A	SOLUCIÓN B	SOLUCIÓN C
Usabilidad	**10 %**	**10 %**	**0 %**
Seguridad	**- 10 %**	**20%**	**60 %**
Rendimiento	**50 %**	**50 %**	**- 10 %**
Calidad total del producto	**50 %**	**80 %**	**50 %**
Consumo de recursos de desarrollo	**20 %**	**10 %**	**5 %**
Calidad producto / Recursos desarrollo	**2,5**	**8**	**10**

68


2.11 Evo
Modelo entrega evolutiva (McConnell)
CONCEPTO SOFTWARE
ANÁLISIS REQUISITOS
DISEÑO de ARQUITECTURA
DESARROLLAR una VERSIÓN
ENTREGAR la VERSIÓN
OBTENER FEEDBACK del CLIENTE
INCORPORAR FEEDBACK del CLIENTE
ENTREGAR VERSIÓN FINAL
Las iteraciones no se realizan sólo para corregir el código, sino en función de la aplicación de SQC (Specification Quality Control) u otras herramientas con capacidades métricas.
Los principios de la Entrega Evolutiva:
- Aprendizaje: Aprender de la realidad y la experiencia; qué funciona y qué no.
- Temprano: Aprender lo suficiente para cambiar cuanto antes lo necesario.
- Pequeño: Pequeños pasos conllevan pequeños riesgos.
- Más simple: Lo complejo se hace más simple y manejable al descomponerlo.
69


2.12 ISD
Internet-Speed Development – ISD
Cusumano & Yoffie, 1999; Baskerville et al., 2001
• Utiliza una combinación del método en espiral y cascada, construyendo builds diarias.
• Hace hincapié en utilizar equipos reducidos. La idea es que un proceso complejo puede descomponerse en otros más sencillos que pueden realizarse en paralelo.
• La división en equipos pequeños lo acerca al planteamiento del método de Open-Source.
• Facilita el control y supervisión de cada parte. Para sincronizar todos los equipos y comprobar el avance del proyecto, se genera una build diaria.
• Se presta especial atención al plan de gestión de riesgos; la calidad es uno de sus pilares fundamentales.
• Consta de 5 fases: Precisión, planning, desarrollo, estabilización, y despliegue o distribución.
• Microsoft Solutions Framework es un ejemplo de ISD.
70

2.12 ISD
Fases y actividades
PREVISIÓN
ANALIZAR REQUISITOS
DEFINIR METAS y RESTRICCIONES
FORMAR EQUIPO
DEFINIR METAS y RESTRICCIONES
Crear EVALUACIÓN de RIESGOS
PLANNING
DEFINIR REQUISITOS
ESPECIFICAR REQUISITOS de las FUNCIONES
DEFINIR ESPECIFICACIONES FUNCIONALES
CREAR el PLANNING
Estimación de coste
Estimación de Riesgos
Crear planes de trabajo
Crear horarios
CREAR DISEÑO
Crear modelo Use-case
Diseño conceptual
Diseño lógico
Diseño físico
Arquitectura
DESARROLLO
DESARROLLAR FUNCIONES
CREAR BUILD DIARIA
DESARROLLAR INFRAESTRUCTURA
Más funciones a desarrollar
¿ Todas las funciones hechas ?
FINALIZAR ALCANCE
71

2.12 ISD
Fases y actividades
ESTABILIZACIÓN
RESOLVER BUGS
TESTEAR
Hay bugs
¿ HAY BUGS ?
No hay bugs
DESPLEGAR VERSIÓN PILOTO
Hay bugs
¿ HAY BUGS ?
No hay bugs
REVISAR
DESPLIEGUE
DESPLEGAR los COMPONENTES CORE
DESPLEGAR la SOLUCIÓN en casa del CLIENTE
ESTABILIZAR el DESPLIEGUE
TRANSFERIR el proyecto a SOPORTE y OPERACIONES
APROBACIÓN FINAL del CLIENTE
REVISAR PROYECTO
72

2.13 MSF

Microsoft Solutions Framework - MSF

MSF es un conjunto de **principios, modelos, disciplinas, conceptos, pautas** y prácticas probadas elaborados por Microsoft desde 1994. MSF **es un framework** flexible y escalable, adaptable a las necesidades de cada proyecto **y no un método**.

MSF tiene una perspectiva de entrega de soluciones; MOF la de servicio de gestión. MSF pone énfasis en los proyectos y MOF en hacer funcionar (running) el entorno de producción.

Muchos métodos son complementos adecuados para MSF: AM, RUP, XP, DSDM…

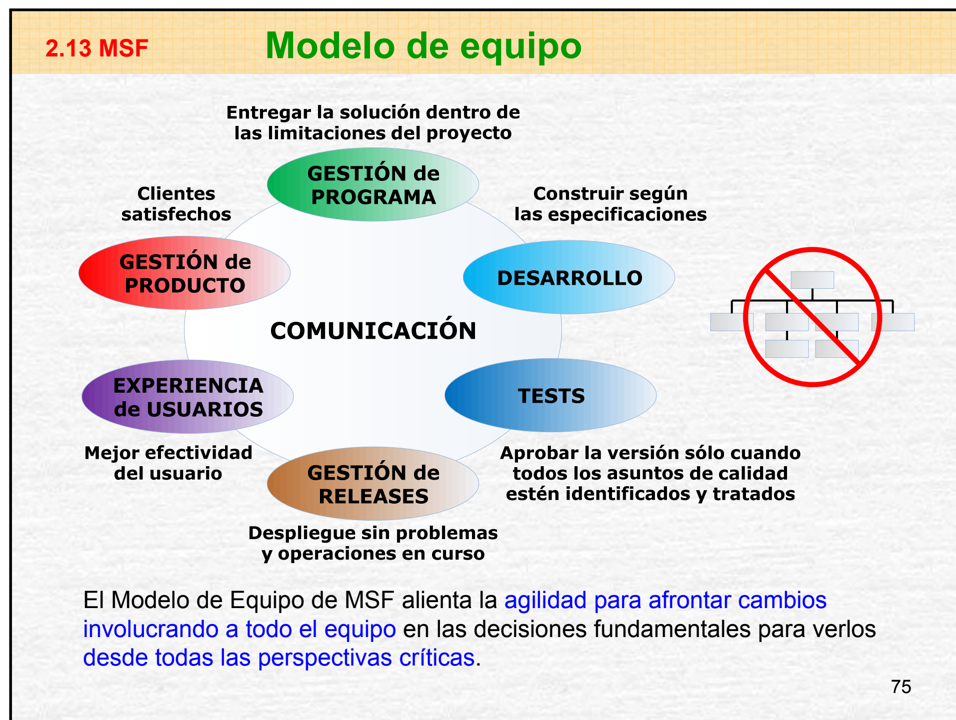
73

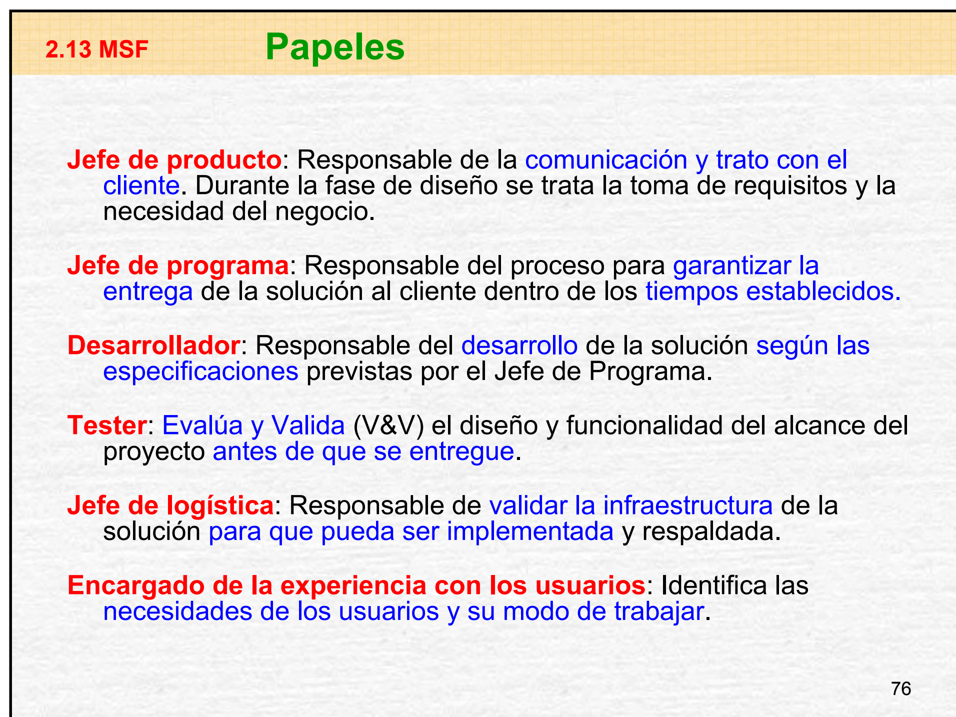
2.13 MSF

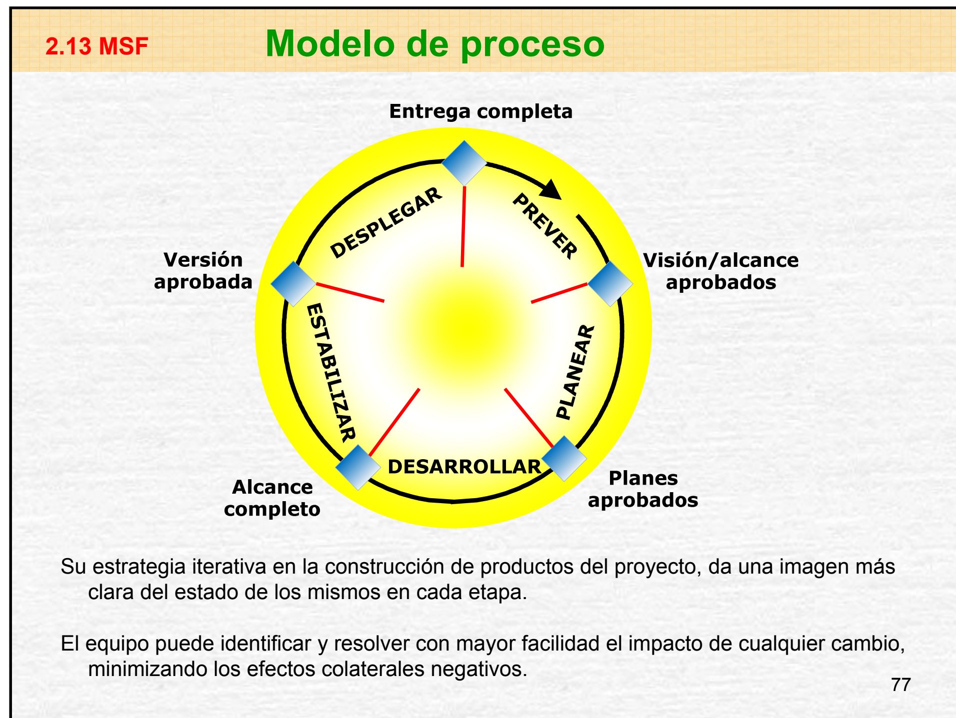
Principios y prácticas

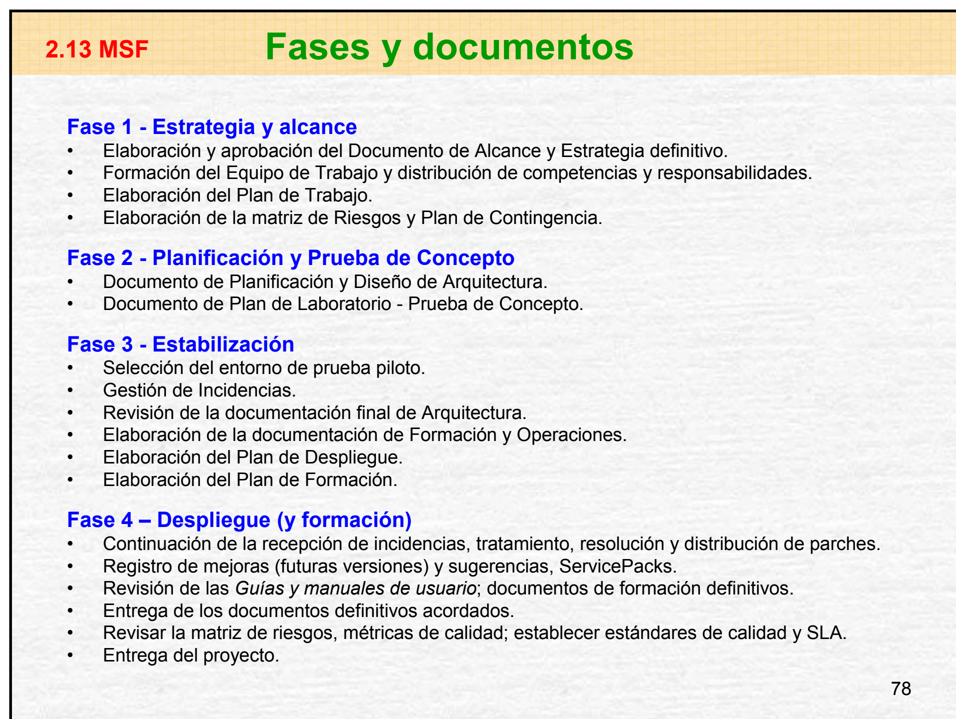
MSF ha diseñado su Modelo de Equipo y Modelo de Proceso para anticiparse al cambio y controlarlo.

74


2.13 MSF
Modelo de equipo
Entregar la solución dentro de las limitaciones del proyecto
GESTIÓN de PROGRAMA
Clientes satisfechos
Construir según las especificaciones
GESTIÓN de PRODUCTO
DESARROLLO
COMUNICACIÓN
EXPERIENCIA de USUARIOS
TESTS
Mejor efectividad del usuario
GESTIÓN de RELEASES
Aprobar la versión sólo cuando todos los asuntos de calidad estén identificados y tratados
Despliegue sin problemas y operaciones en curso
El Modelo de Equipo de MSF alienta la agilidad para afrontar cambios involucrando a todo el equipo en las decisiones fundamentales para verlos desde todas las perspectivas críticas.
75


2.13 MSF
Papeles
Jefe de producto: Responsable de la comunicación y trato con el cliente. Durante la fase de diseño se trata la toma de requisitos y la necesidad del negocio.
Jefe de programa: Responsable del proceso para garantizar la entrega de la solución al cliente dentro de los tiempos establecidos.
Desarrollador: Responsable del desarrollo de la solución según las especificaciones previstas por el Jefe de Programa.
Tester: Evalúa y Valida (V&V) el diseño y funcionalidad del alcance del proyecto antes de que se entregue.
Jefe de logística: Responsable de validar la infraestructura de la solución para que pueda ser implementada y respaldada.
Encargado de la experiencia con los usuarios: Identifica las necesidades de los usuarios y su modo de trabajar.
76


2.13 MSF
Modelo de proceso
Entrega completa
DESPLEGAR
PREVER
Versión aprobada
Visión/alcance aprobados
ESTABILIZAR
PLANEAR
DESARROLLAR
Alcance completo
Planes aprobados
Su estrategia iterativa en la construcción de productos del proyecto, da una imagen más clara del estado de los mismos en cada etapa.
El equipo puede identificar y resolver con mayor facilidad el impacto de cualquier cambio, minimizando los efectos colaterales negativos.
77


2.13 MSF
Fases y documentos
Fase 1 - Estrategia y alcance
• Elaboración y aprobación del Documento de Alcance y Estrategia definitivo.
• Formación del Equipo de Trabajo y distribución de competencias y responsabilidades.
• Elaboración del Plan de Trabajo.
• Elaboración de la matriz de Riesgos y Plan de Contingencia.
Fase 2 - Planificación y Prueba de Concepto
• Documento de Planificación y Diseño de Arquitectura.
• Documento de Plan de Laboratorio - Prueba de Concepto.
Fase 3 - Estabilización
• Selección del entorno de prueba piloto.
• Gestión de Incidencias.
• Revisión de la documentación final de Arquitectura.
• Elaboración de la documentación de Formación y Operaciones.
• Elaboración del Plan de Despliegue.
• Elaboración del Plan de Formación.
Fase 4 – Despliegue (y formación)
• Continuación de la recepción de incidencias, tratamiento, resolución y distribución de parches.
• Registro de mejoras (futuras versiones) y sugerencias, ServicePacks.
• Revisión de las Guías y manuales de usuario; documentos de formación definitivos.
• Entrega de los documentos definitivos acordados.
• Revisar la matriz de riesgos, métricas de calidad; establecer estándares de calidad y SLA.
• Entrega del proyecto.
78

2.13 MSF

Control de cambios

MSF también se adapta a un desarrollo muy documentado y a escala de misión crítica que requiera niveles más altos de estructura (CMMI, PMBOK, ISO9000), pero sus disciplinas no admiten modelos no iterativos o no incrementales.

1 IDENTIFICAR
2 ESPECIFICAR
3 CAMBIAR
4 EVALUAR
BASE de CONOCIMIENTOS

Los métodos ágiles son antiguos en Microsoft, por ejemplo, uno de los textos clásicos de RAD, *Rapid Development* (McConnell 1996). Algunas prácticas de RAD, hoy no serían ágiles como fijar metas de antemano, contratar a terceros para realizar parte del código (*outsourcing*), trabajar en oficinas privadas u ofrecerse para horas extras.

79

2.13 MSF

Control de riesgos

1. IDENTIFICAR
Describir Riesgo
2. ANALIZAR
Documento de estimación de riesgos TOP 10
3. PLAN
4. SEGUIMIENTO
5. CONTROL
Eliminar Riesgo

La Disciplina de Gestión de Riesgo que se aplica a todos los miembros del Modelo de Equipo de MSF emplea el principio fundacional que literalmente se expresa como permanecer ágil y esperar el cambio.

80

2.14 PP

Pragmatic Programming

Hunt & Thomas, 2002

La programación pragmática no define un proceso, fases, workproducts, papeles… sino que son 70 consejos de programación ilustrados con ejemplos para ser un buen programador. Hace hincapié en automatizar las pruebas e incluso parte de la documentación.

Principios de la filosofía :

1) Responsabilícese de lo que hace; piense en soluciones y no en excusas.

2) No diseñe o programe mal, arregle las inconsistencias en cuanto pueda.

3) Tome un papel activo para introducir cambios donde los crea necesarios.

4) Haga software que satisfaga a su cliente, pero sepa cuándo parar

5) Constantemente amplíe su conocimiento.

6) Mejore sus habilidades de comunicación.

81

2.14 PP

Reglas del Pragmatic Programmer

1) Preocúpese de su destreza. Es la forma de hacer las cosas bien.
2) Piense en su trabajo. Critique y aprecie su trabajo.
3) Proporcione opciones, no dé excusas. No diga que no puede hacerse; explique lo que puede hacerse.
4) No viva con ventanas que tienen errores. Arregle los malos diseños y decisiones poco acertadas.
5) Sea un catalizador para el cambio. Usted no puede forzar el cambio en las personas. En cambio, muéstreles cómo podría ser el futuro y podría ayudarles a participar creándolo.
6) Recuerde el conjunto. No se centre en detalles cuando hay problemas graves.
7) Haga de la calidad un requisito.
8) Invierta regularmente en su carpeta de conocimiento. Aprender debe ser un hábito.
9) Analice críticamente lo que lee y oye.
10) Son las dos cosas: lo que dice y cómo lo dice. Una gran idea hay que comunicarla bien.
11) DRY – *Don't repear Yourself*. No se repita mientras escribe código.
12) Hágalo fácil de re-usar. Si es fácil re-usar, las personas lo harán.
13) Elimine efectos entre cosas no relacionadas. Diseñe componentes autónomos, independientes, con propósito único y bien definido.
14) No hay últimas decisiones. Piense en el cambio.
15) Use balas trazadoras (*tracer bullets*) para encontrar el problema.
16) Use prototipos para aprender. Su valor no queda en el código, sino en usted.
17) Programe cerca del dominio del problema. Planee en el lenguaje del usuario.

82

2.14 PP

Reglas del Pragmatic Programmer

18) Prevea para evitar las sorpresas. Descubrirá los problemas potenciales.
19) Itere el horario y calendario con el código. Use la experiencia para precisar el tiempo que necesitará.
20) Guarde el conocimiento en el texto sencillo. El texto sencillo (plain) no quedará obsoleto y simplifica la depuración y pruebas.
21) Use el poder de las consolas de sistema (*shell*). Son más rápidas que los GUI.
22) Use un único editor bien. El editor debe ser una extensión de su mano
23) Utilice siempre el control de código fuente. Es una máquina de tiempo para su trabajo, puede volver atrás.
24) Arregle el problema, no culpe.
25) No se ponga nervioso cuando haga el *debug*.
26) “select" no está roto. Es raro encontrar un *bug* en el sistema operativo o en el compilador, o incluso en productos de otras empresas o bibliotecas. Lo más probable es que el error esté en su aplicación.
27) No suponga, demuestre. Demuestre sus hipótesis en situaciones reales y condiciones límite.
28) Aprenda un lenguaje de manipulación de texto. Parte del cada día que trabaja con texto. Puede automatizarse.
29) Escriba código que escribe código. Los generadores de código aumentan su productividad evitan la duplicación.
30) No puede escribir software perfecto. Proteja código y usuarios de errores inevitables.
31) Diseñe con los contratos. Úselos para documentar y verificar que el código no hace ni más ni menos de lo que debe hacer.

83

2.14 PP

Reglas del Pragmatic Programmer

32) Provoque un error grave (crash) para terminar el programa pronto. Un programa muerto hace menos daño que uno dañado.
33) Use las aserciones para prevenir l imposible. Las aserciones validan sus suposiciones. Úselas para proteger su código.
34) Use las excepciones para problemas excepcionales. Las excepciones pueden padecer de falta de legibilidad y problemas de mantenimiento, como código spaghetti.
35) Termine lo que empieza. La rutina u objeto que asignan un recurso debe ser responsables de liberarlo.
36) Minimice el acoplamiento entre módulos. Ley de Demeter, LoD (“Sólo habla con tus amigos inmediatos”).
37) Configure, no integre. Implemente las opciones de tecnología como opciones de configuración.
38) Ponga las abstracciones en el código, los detalles en metadatos. Programe para el caso general, y ponga los casos específicos fuera del código base compilado.
39) Analice el diagrame de flujo para mejorar la concurrencia.
40) Diseñe usando servicios. Objetos independientes, concurrentes detrás de interfaces consistentes y bien definidos.
41) Siempre diseñe para usar concurrencia. Serán interfaces claras, menos suposiciones.
42) Use pizarras para coordinar volúmenes de trabajo (workflow).
43) No programe por casualidad o coincidencia. Sólo confíe en cosas fiables.

84

2.14 PP

Reglas del Pragmatic Programmer

44) Estime el orden de sus algoritmos. Estime cuánto tiempo es probable que le lleve antes de programar.
45) Compruebe sus estimaciones. Cronometre su código en el entorno objetivo.
46) Separe las vistas de los modelos. Gane flexibilidad diseñando su aplicación en términos de modelos y vistas.
47) Refactorice pronto, refactorice a menudo. Arregle la raíz del problema.
48) Diseñe para testear. Piense en testear antes de escribir ninguna línea de código.
49) Pruebe su software, o lo harán los usuarios. Que no encuentren *bugs* por usted.
50) No use código mágico que no entienda.
51) No recoja requisitos; excave para encontrarlos. Suelen ocultarse bajo suposiciones.
52) Trabaje con un usuario para pensar como un usuario. Verá cómo se usará el sistema.
53) Las abstracciones viven más mucho tiempo que los detalles. Invierta en la abstracción, no la aplicación.
54) Use un glosario del proyecto con el vocabulario específico.
55) No piense "*outside the box*" (sin ideas preconcebidas); "*find the box*" (encuentre el punto de vista). Ante un problema, identifique restricciones reales. ¿tiene que hacerse así? ¿es necesario hacerlo?
56) Empiece cuando esté preparado. Su vida es experiencia. No ignore las dudas insignificantes.
57) Algunas cosas es más fácil hacerlas que describirlas.
58) No sea un esclavo de los métodos formales. No adopte ninguna técnica a ciegas.

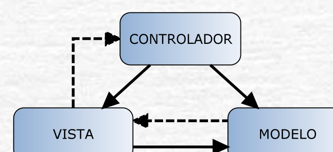
85

2.14 PP

Reglas del Pragmatic Programmer

59) Las herramientas costosas no producen mejores diseños.
60) Organice los equipos según su funcionalidad. No separe diseñadores de programadores, etc.
61) No use los procedimientos manuales cuando pueda automatizarlos.
62) Testee pronto, a menudo, y de forma automática.
63) El código no está hecho hasta que pasa todos los tests.
64) Use "saboteadores" para testear su test. Introducir *bugs* a propóstito.
65) Testee los estados, no sólo el código.
66) Encuentre *bugs* una vez. A partir de entonces, pruebas automáticas.
67) El inglés es simplemente un idioma de la programación. Escriba documentos como escribiría el código: DRY (*don't repeat yourself*), use metadatos, MVC…
68) Construya la documentación en el código. La documentación separada no será correcta ni actualizada.
69) Sobrepase las expectativas de los usuarios. Entregue un poco más.
70) Firme su trabajo. Como un artesano, debería estar orgulloso de él.

MVC: Modelo Vista Controlador. Las líneas sólidas indican una asociación directa, y las punteadas, indirecta.



86

3. Software

Software – Índice

3.1 Herramientas para automatizar. **Deben ser:**

1) **fáciles de escribir:** el cliente debería ser capaces de escribirlas.
2) **automatizables**, de forma desatendida.
3) **cubrir todos componentes de una aplicación**, en distintos niveles de abstracción (pruebas unitarias, pruebas de integración, pruebas funcionales, etc.)

El *framework* xUnit cumple estos requisitos (*JUnit* para JAVA; Gamma y Beck).

Veremos algunas herramientas habituales para:

- pruebas automáticas
- integración continua
- refactorización
- trabajo colaborativo
- estándares de codificación

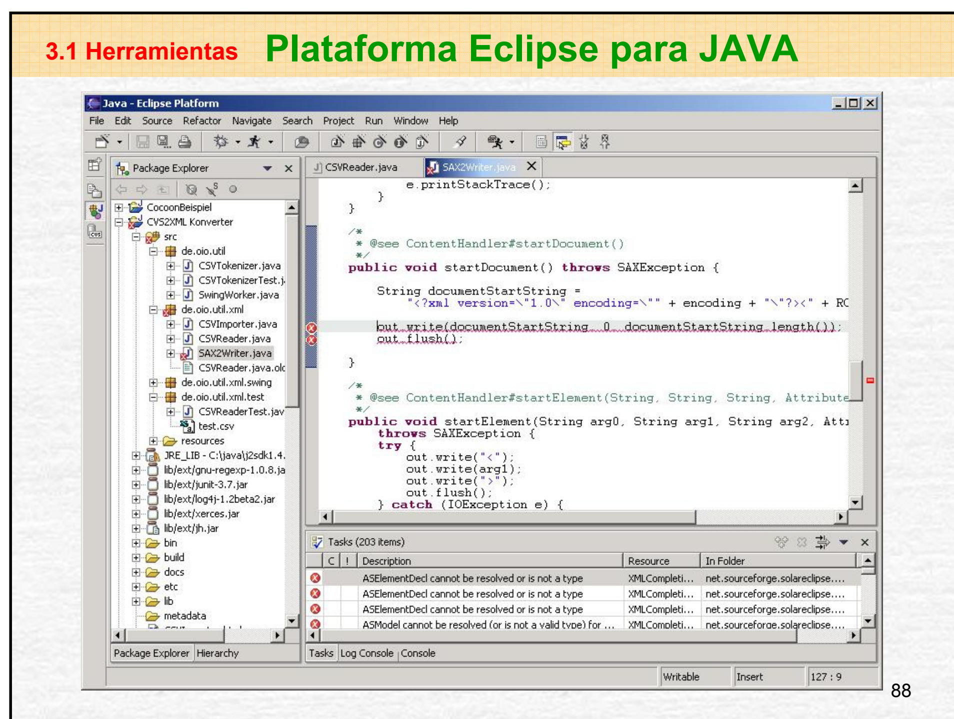
3.2 Software

- XPlanner
- Rally
- TargetProcess
- Evo Task Administrator
- V1Agile Team y Agile Enterprise
- Microsoft Visual Studio
- ExtremePlanner
- Atlassian
- xProcess

87

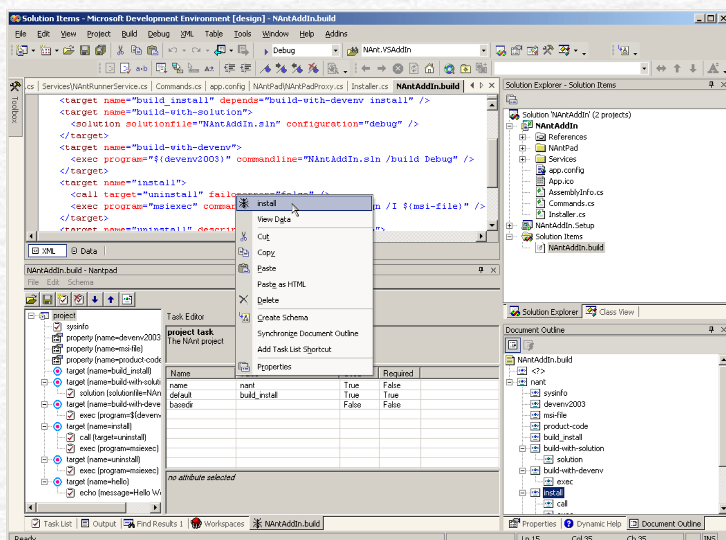
3.1 Herramientas

Plataforma Eclipse para JAVA



88

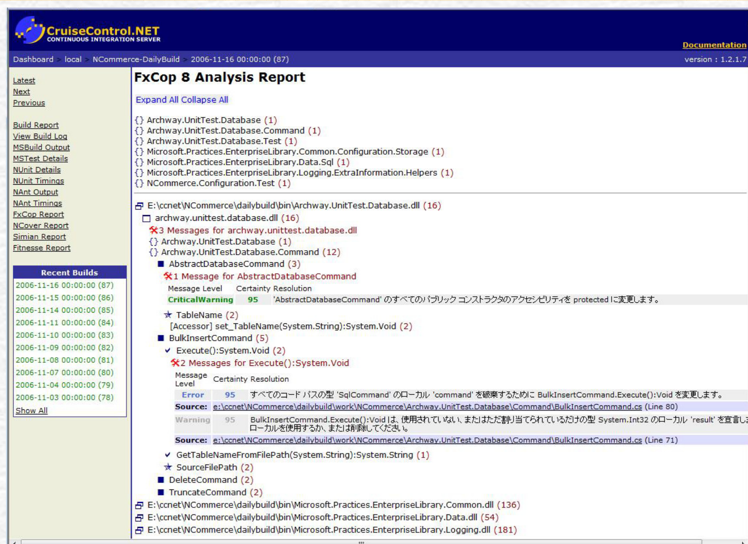
3.1 Herramientas **Integración continua con Nant (.NET)**



Ant o Nant son *frameworks* para automatizar todas las fases de integración continua (incluyendo accesos al repositorio, transferencias por red o ejecución de pruebas Junit).

89

3.1 Herramientas **Integración automatizada con CruiseControl**



Al detectar alguna modificación (se sube nuevo código al repositorio), se despierta y ejecuta toda la cadena de integración y notifica el resultado por e-mail u otros medios.

90

3.1 Herramientas **Control de versiones con CVS**

CVS Repository Exploring - Eclipse Platform

File Edit Navigate Search Project Run Window Help

Java Structure Compare

Compilation Unit
SBCreator
getSessionBean(String, String)

Java Source Compare

SBCreator.java 1.5 | SBCreator.java 1.4

```
return beans.get(beanClassName);

//Trace.log(1, "*** trying to get context ...");
Context ctx = new InitialContext();

/*
Trace.log(1, "*** got initial context --> " + ctx);
Trace.log(1, "*** try looking up JNDIUtil");
*/
Object obj = ctx.lookup("CatalogJNDIUtil");
//Trace.log(1, "*** starting narrowing from " + obj.getCla
CatalogJNDIUtilHome jndiUtilHome = (CatalogJNDIUtilHome) P
//Trace.log(5, "*** got object " + jndiUtilHome);
CatalogJNDIUtil jndiUtil = jndiUtilHome.create();

/*
Trace.log(5, "*** created obj " + jndiUtil);
Trace.log(5, "*** returning with [" + XIntrospector.invoke
*/
return XIntrospector.invoke("get" + jndiName + "Home", jr

tch(IntrospectionException e) {
Trace.log(1, "SBCreator.getSessionBean() ERROR: " + e);
```

```
return beans.get(beanClassName);

try {
Trace.log(1, "*** trying to get context ...");
Context ctx = new InitialContext();

Trace.log(1, "*** got initial context --> " +
Trace.log(1, "*** try looking up JNDIUtil");
Object obj = ctx.lookup("CatalogJNDIUtil");
Trace.log(1, "*** starting narrowing from " +
CatalogJNDIUtilHome jndiUtilHome = (CatalogJND
Trace.log(5, "*** got object " + jndiUtilHome)
CatalogJNDIUtil jndiUtil = jndiUtilHome.create

Trace.log(5, "*** created obj " + jndiUtil);
Trace.log(5, "*** returning with [" + XIntrosp
return XIntrospector.invoke("get" + jndiName

} catch(IntrospectionException e) {
Trace.log(1, "SBCreator.getSessionBean() ERROR
} catch(NamingException e) {
Trace.log(1, "SBCreator.getSessionBean() could
Trace.log(1, "SBCreator.getSessionBean(): " +
e.printStackTrace();
```

CVS Resource History - SBCreator.java

Revision	Tags	Date	Author	Comment
1.5		11.06.01 16:02	tjaeger	Trace-Outputs rausgenommen
1.4	bkk-0_9	04.06.01 18:53	sschluff	Neues Session-Bean zur Verwaltung von P...
1.3		02.06.01 15:02	kkoehler	Methode getArticleService eingefügt
1.2		29.05.01 18:02	sschluff	Standard-Header eingefuegt.

91

3.1 Herramientas **Control de bugs con Bugzilla**

http://bugzilla.mozilla.org/show_bug.cgi?id=52094

G35 Forum | CNN.com | The New York Times ... | Netflix - Rent DVDs ... | millennium | we... | Post to MT Blog | List & Edit Entries | m...

mozilla.org

Bugzilla Version 2.17.1

Bugzilla Bug 52094 hyatt should give ben $50 Last modified: 2003-07-14 12:14

Query page Enter new bug

Bug#: 52094 alias:
Product: Browser
Component: Tracking
Status: VERIFIED
Resolution: WONTFIX
Assigned To: hyatt@mozilla.org (David Hyatt)
QA Contact: jrgm@netscape.com
URL: http://www.zachlipton.com/ben
Summary: hyatt should give ben $50
Status Whiteboard:
Keywords: helpwanted, meta, modern, nsonly, pp, testcase

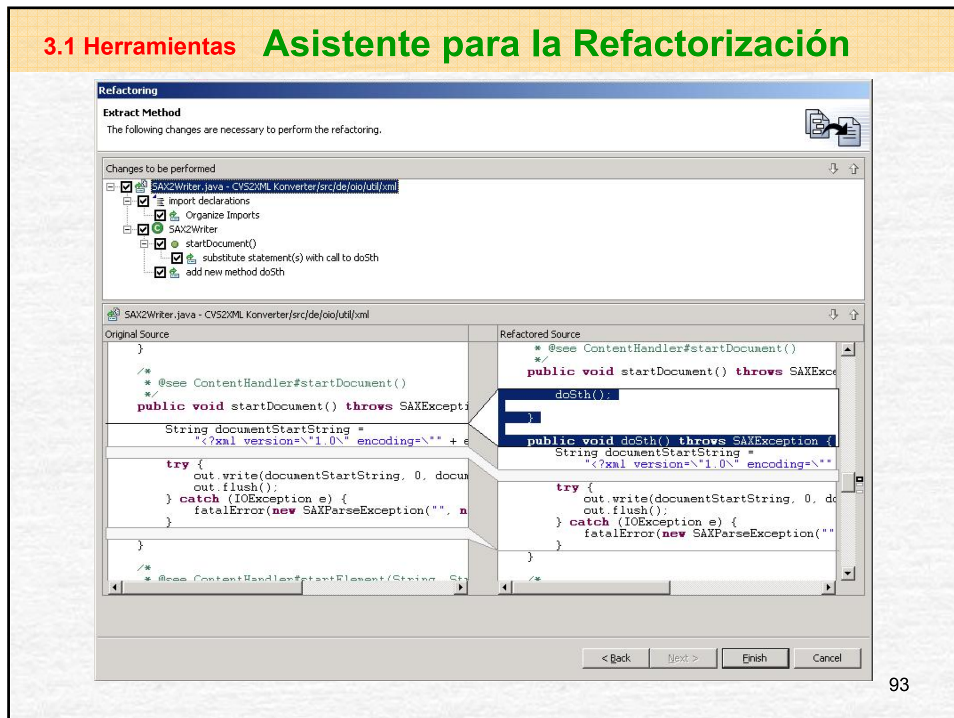
Hardware: PC
OS: Windows 2000
Version: Trunk
Priority: P1
Severity: blocker
Target Milestone: Future

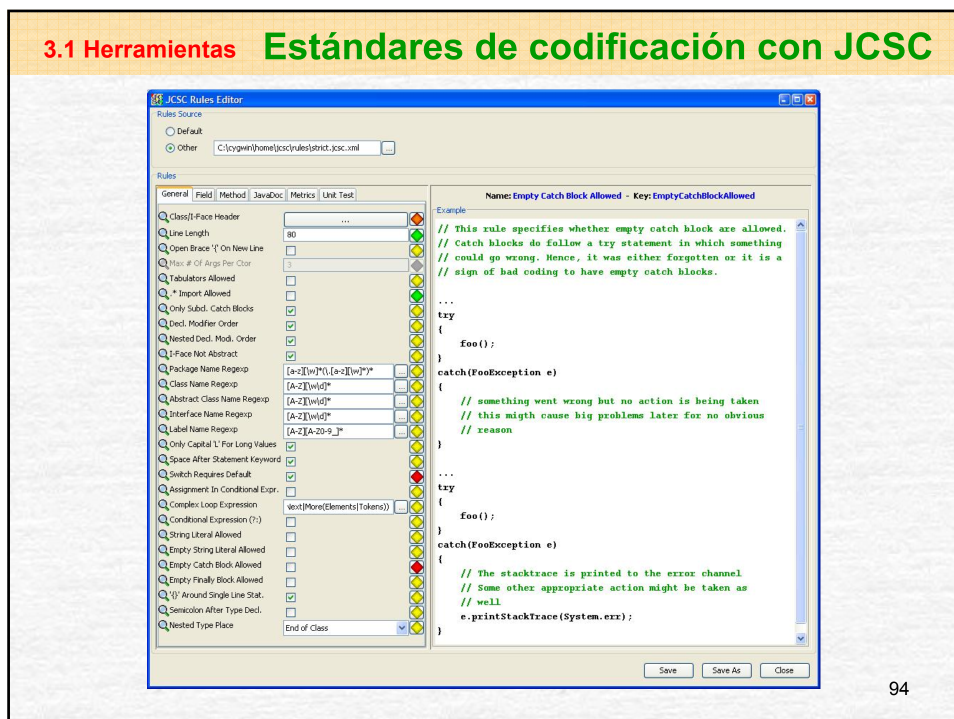
Reporter: ben@netscape.com (Ben Goodger)
Add CC:
CC: adam@gimp.org, andersma@luther.edu, bhart@cvip.net, blaker@netscape.com, brian@mozdev.org
Remove selected CCs

Flags: (Help!) Requestee:
blocking1.4.x
blocking1.5a

Attachment	Type	Created	Flags	Actions

92


3.1 Herramientas
Asistente para la Refactorización
Refactoring
Extract Method
The following changes are necessary to perform the refactoring.
Changes to be performed
SAX2Writer.java - CVS2XML Konverter/src/de/oio/util/xml
import declarations
Organize Imports
SAX2Writer
startDocument()
substitute statement(s) with call to doSth
add new method doSth
Original Source
Refactored Source
< Back
Next >
Finish
Cancel
93

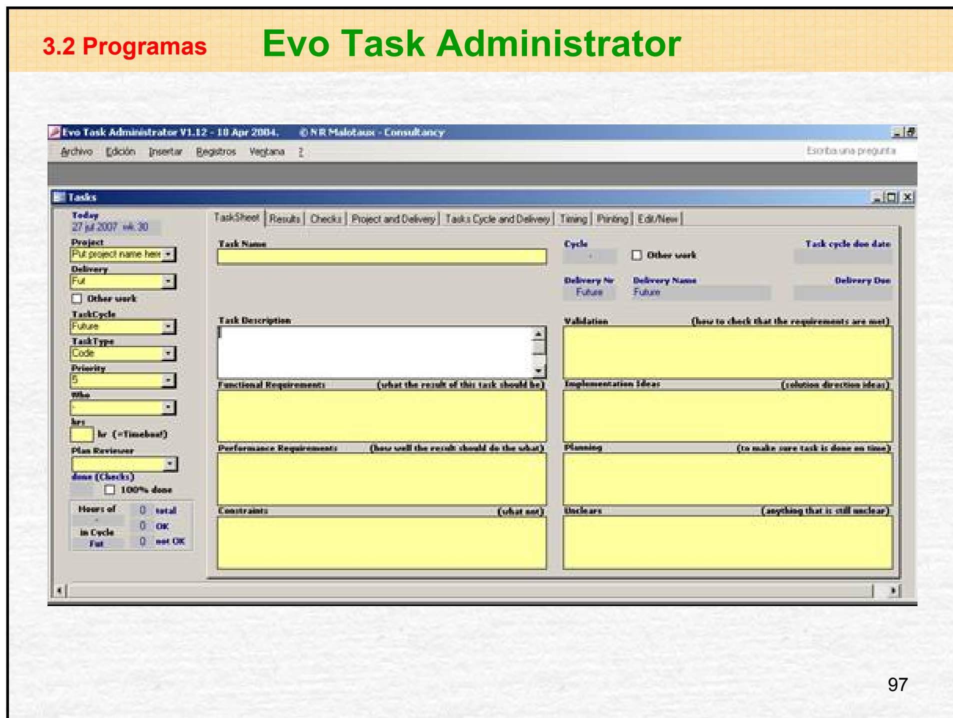

3.1 Herramientas
Estándares de codificación con JCSC
JCSC Rules Editor
Rules Source
Default
Other
Rules
Name: Empty Catch Block Allowed - Key: EmptyCatchBlockAllowed
Example
// This rule specifies whether empty catch block are allowed.
// Catch blocks do follow a try statement in which something
// could go wrong. Hence, it was either forgotten or it is a
// sign of bad coding to have empty catch blocks.
Save
Save As
Close
94

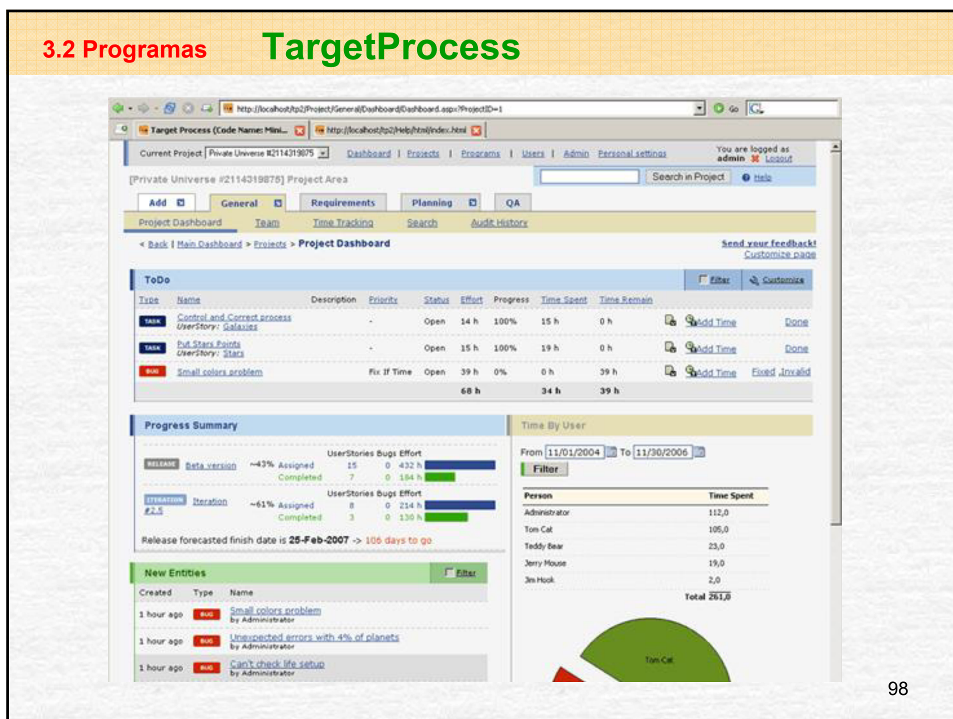
3.2 Programas **XPlanner**

95

3.2 Programas **Rally**

96


3.2 Programas
Evo Task Administrator
97


3.2 Programas
TargetProcess
98

3.2 Programas

VersionOne Agile Enterprise

99

3.2 Programas

ExtremePlanner

extremeplanner

Current Project: Demo Project Select All Projects User: Administrator

Summary | Releases | Iterations | Stories | Tasks | Test Cases [Projects | Users | Admin]

Project Activity: Demo Project

[Status | Activity | Metrics | Export]

Activity From: 2007-02-11 Through: 2007-02-12 View

Include: Story changes Task changes Files added Comments added

Additions

Type	ID	Description	Status	Created	Created By
Story	S9	Accept gift certificates	Proposed	2/12/07 12:30 PM	Administrator
Task	T8	Allow admins to create gift certificate codes Story: S9	Active	2/12/07 12:33 PM	Administrator
Task	T9	Update checkout system to use credit from gift certificates Story: S9	Active	2/12/07 12:33 PM	Administrator
Comment		Checked with marketing, and they would like to use tracking codes on the certificates Story: S9		2/12/07 12:34 PM	Administrator

Updates

Type	ID	Description	Status	Updated	Updated By
Task	T2	Populate product database from Excel Story: S2	Active	2/12/07 12:29 PM	Administrator

100

3.2 Programas

Atlassian

Propone 7 módulos de software utilizados por más de 7.000 usuarios:

JIRA: aplicación para el seguimiento de *bugs*, *issues*, características, tareas, versiones, y la gestión de proyectos.

CONFLUENCE: una *wiki* de empresa que facilita colaborar y compartir.

BAMBOO: servidor continuo para la integración. Aporta medidas en tiempo real, proporciona feedback, métricas, patrones... y se integra con otras herramientas de desarrollo.

CLOVER: entorno para garantizar la calidad de los propiostests unitarios (los tests unitarios miran la calidad del código). Permite encontrar partes de código no testeadas, "código muerto" (que ya no se usa y puede eliminarse de forma segura), integrarse con otras plataformas de test, *plug-ins*.

CROWD: herramienta web que simplifica la gestión centralizada de identidades (pertenencia a grupos) y el aprovisionamiento de aplicaciones.

CRUCIBLE: facilita revisar los cambios al código, añadir comentarios.

FISHEYE: almacén o repositorio. Facilita trabajar con código que cambia constantemente, supervisar, buscar archivos, compartir, etc.

101

3.2 Programas

Microsoft Visual Studio

102

4. Resumen

Conclusiones - Índice

Costo del cambio según Boehm (base 67 proyectos, 1987) y según Beck en XP.

Boehm afirmó que encontrar y arreglar un problema después entregarse al cliente cuesta 100 veces más que hacerlo en el diseño inicial; para sistemas pequeños, el factor de acerca a 5.

COSTE DEL CAMBIO

PROGRAMACIÓN TRADICIONAL

EXTREME PROGRAMMING

Requisitos Análisis Diseño Implementación Testeo Producción

TIEMPO

CASCADA

ANÁLISIS

DISEÑO

IMPLEMENTACIÓN

TEST

ITERATIVO

XP

XP no sólo es iterativo sino que hace todas las fases simultáneamente (nótese filas o columnas de las 4 fases).

103

4.1 Resumen

Cronología y relaciones entre métodos ágiles

1990

2000

Método prototipos (e.g. Lantz, 1986)

Nuevo juego de desarrollo de producto (Takeuchi & Nonaka, 1986)

Fiction of universal methods (Malouin & Landry, 1983)

Modelo en ESPIRAL Boehm, 1988

Enfoque ORIENTADO a OBJETOS

EVOlutionary life-Cycle (Gilb, 1988)

Amethodological IS development (Baskerville, 1992; Truex et al., 2001

Rapid Application Development (RAD) (e.g. Martin, 1991)

Methodology Engineering (Kumar & Welke, 1992)

SCRUM (Schwaber, 1995; Schwaber & Beedle, 2001)

RADical Software Development (Bayer & Highsmith, 1994)

DYNAMIC SYSTEMS DEVELOPMENT METHOD (DSDM) 1995

Internet Technologies, Distributed software development

Enfoque Sync-and-stabilize (Microsoft) (Cusumano & Selby, 1995, 1997)

Open Source Software (OSS) development

Unified Modeling Language (UML)

CRYSTAL FAMILY METHODOLOGIES (Cockburn, 1998, 2001)

ADAPTIVE SOFTWARE DEVELOPMENT (ASD) (Highsmith, 2000)

Internet-Speed Development (Cusumano & Yoffie 1999; Baskerville et al. 2001)

Internet Speed development (Truex et al., 1999)

RATIONAL UNIFIED PROCESS (RUP) (Kruchten, 2000)

EXTREME PROGRAMMING (XP) (Beck, 1999)

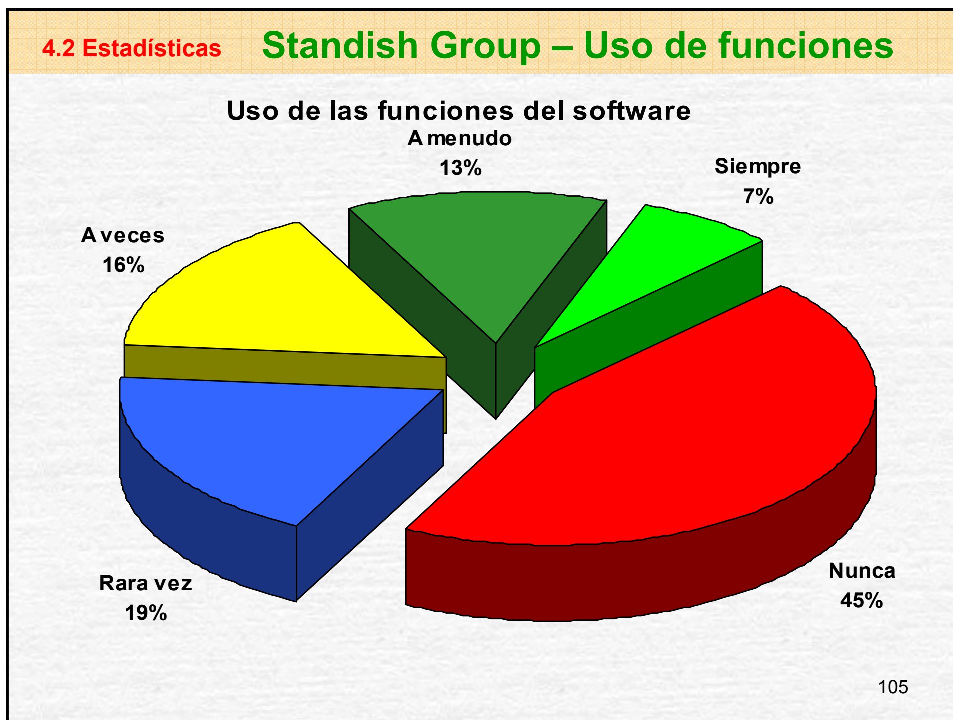
FEATURE DRIVEN DEVELOPMENT (FDD) (Palmer & Felsing, 2002)

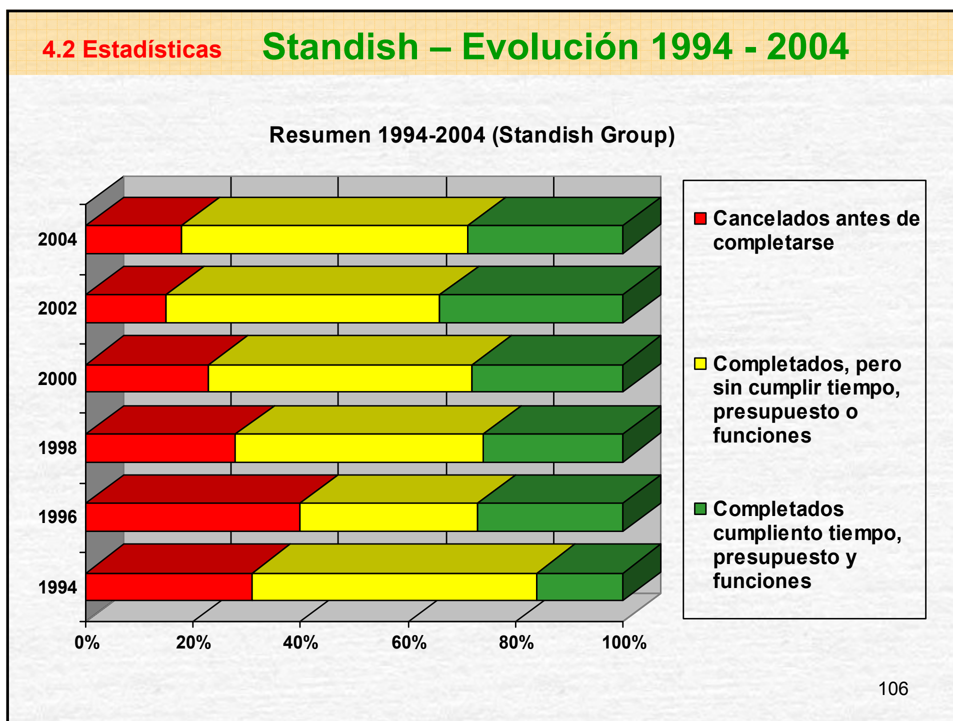
AGILE MANIFESTO Beck et al., 2001

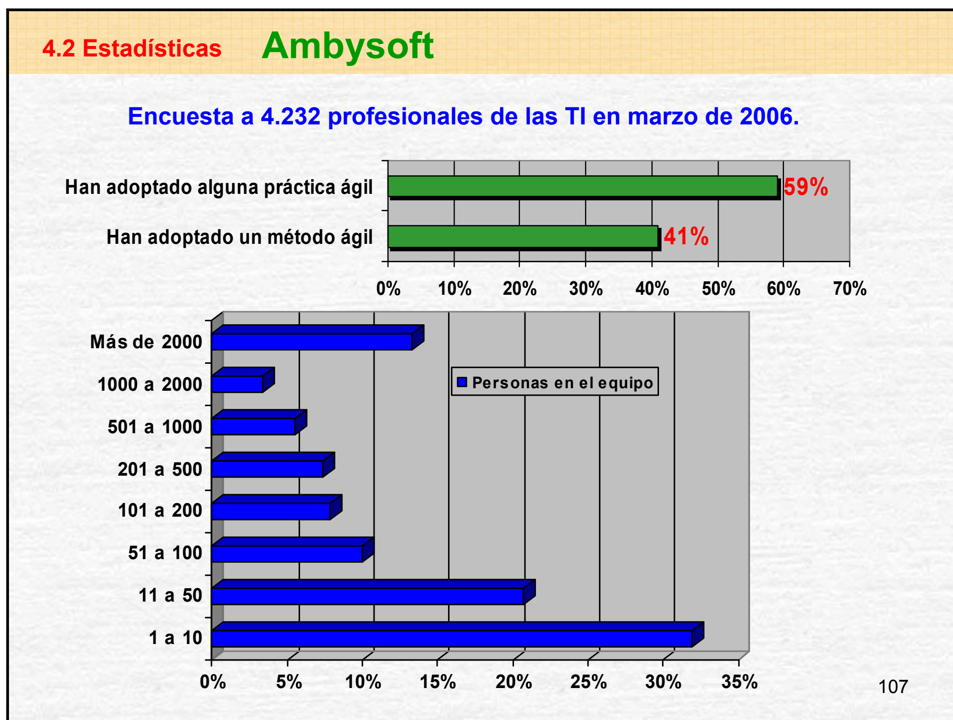
PRAGMATIC PROGRAMMING (Hunt & Thomas, 2000)

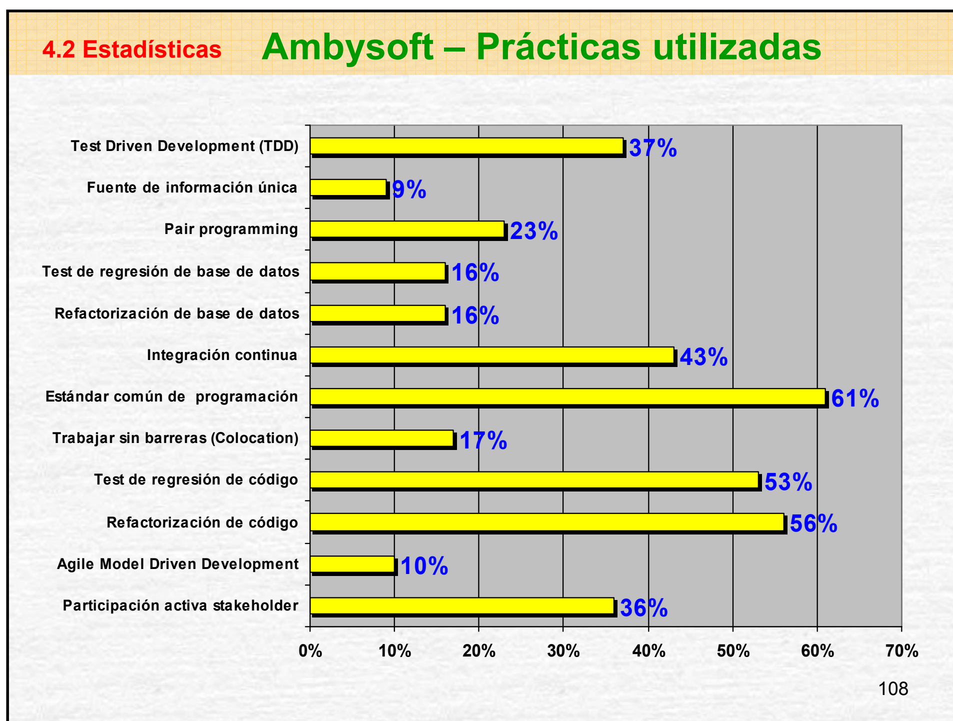
AGILE MODELING - (AM) (Ambler, 2002)

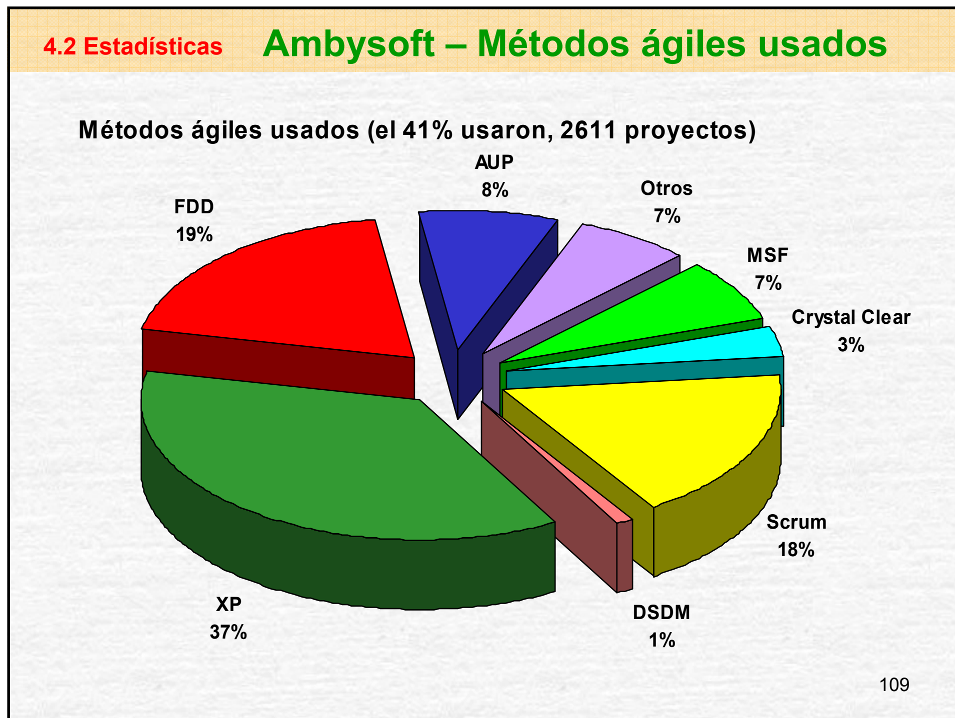
104

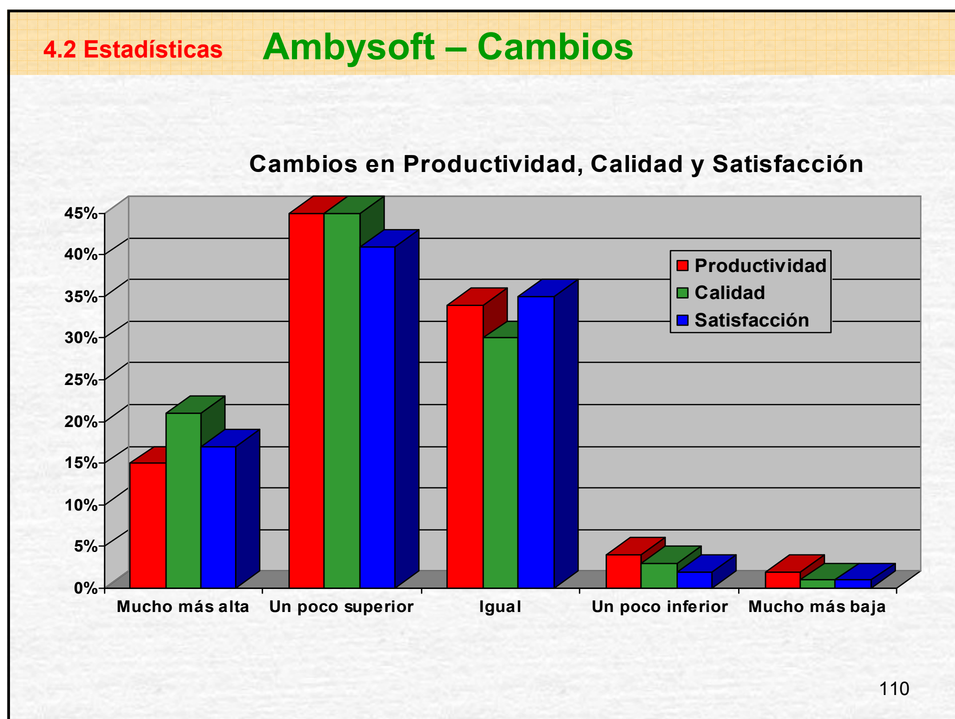

4.2 Estadísticas
Standish Group – Uso de funciones
Uso de las funciones del software
A menudo
13%
Siempre
7%
A veces
16%
Rara vez
19%
Nunca
45%
105

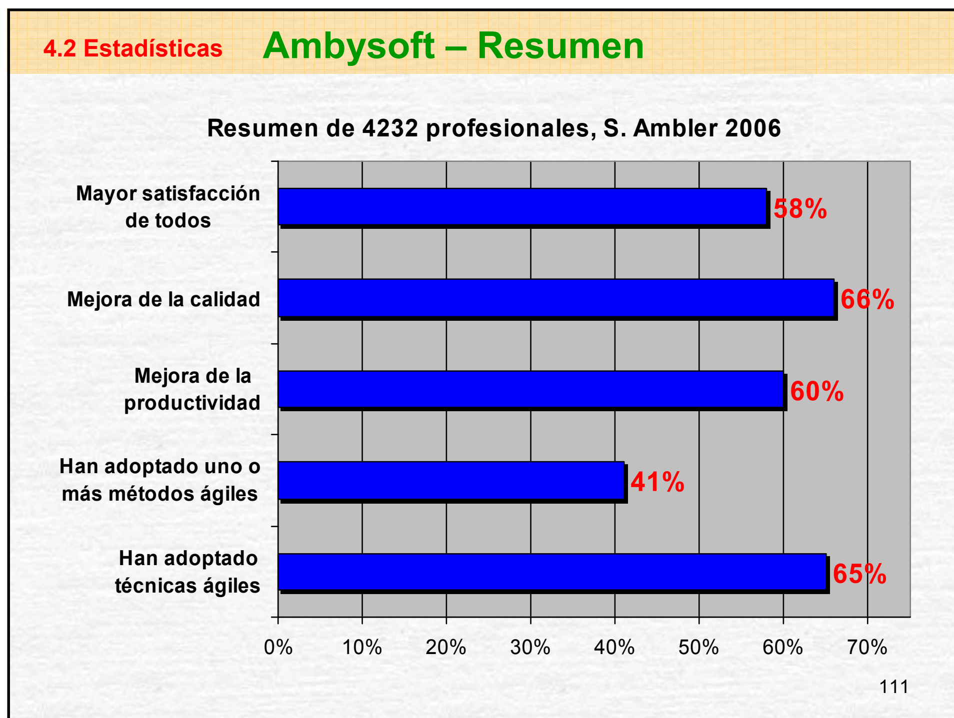

4.2 Estadísticas
Standish – Evolución 1994 - 2004
Resumen 1994-2004 (Standish Group)
2004
2002
2000
1998
1996
1994
0%
20%
40%
60%
80%
100%
Cancelados antes de completarse
Completados, pero sin cumplir tiempo, presupuesto o funciones
Completados cumpliento tiempo, presupuesto y funciones
106

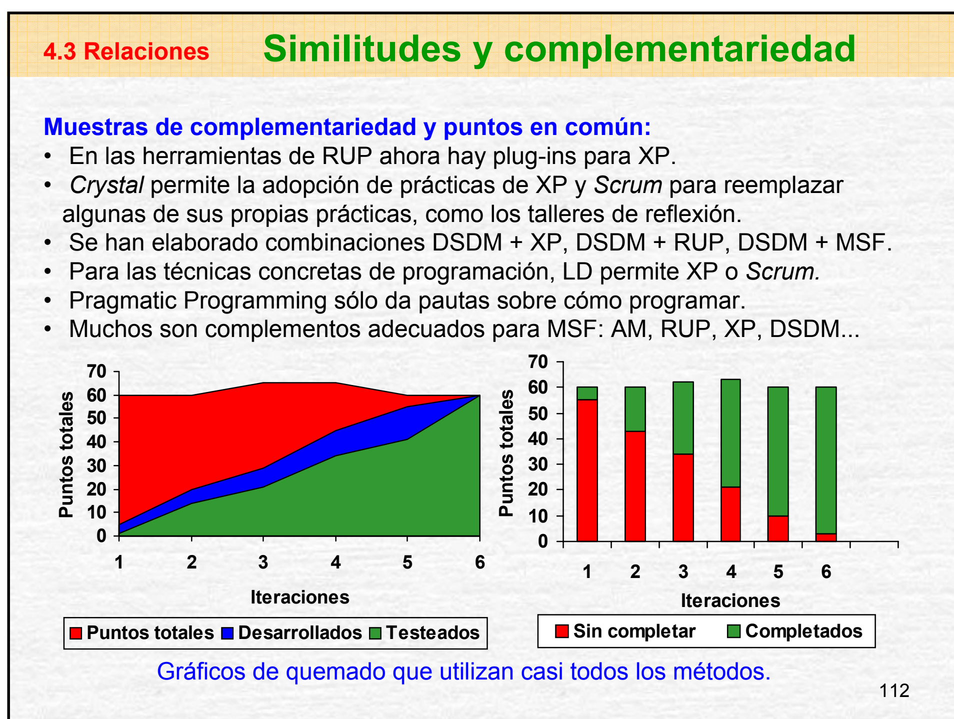

4.2 Estadísticas
Ambysoft
Encuesta a 4.232 profesionales de las TI en marzo de 2006.
Han adoptado alguna práctica ágil
Han adoptado un método ágil
59%
41%
0% 10% 20% 30% 40% 50% 60% 70%
Más de 2000
1000 a 2000
501 a 1000
201 a 500
101 a 200
51 a 100
11 a 50
1 a 10
Personas en el equipo
0% 5% 10% 15% 20% 25% 30% 35%
107

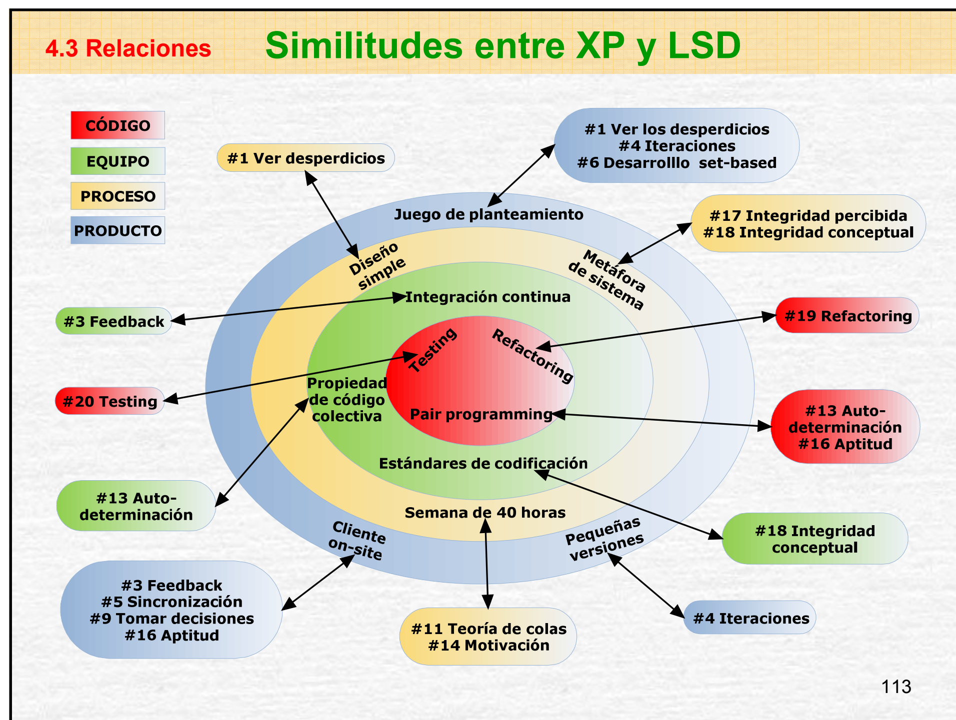

4.2 Estadísticas
Ambysoft – Prácticas utilizadas
Test Driven Development (TDD) 37%
Fuente de información única 9%
Pair programming 23%
Test de regresión de base de datos 16%
Refactorización de base de datos 16%
Integración continua 43%
Estándar común de programación 61%
Trabajar sin barreras (Colocation) 17%
Test de regresión de código 53%
Refactorización de código 56%
Agile Model Driven Development 10%
Participación activa stakeholder 36%
0% 10% 20% 30% 40% 50% 60% 70%
108


4.2 Estadísticas
Ambysoft – Métodos ágiles usados
Métodos ágiles usados (el 41% usaron, 2611 proyectos)
AUP 8%
Otros 7%
FDD 19%
MSF 7%
Crystal Clear 3%
Scrum 18%
XP 37%
DSDM 1%
109


4.2 Estadísticas
Ambysoft – Cambios
Cambios en Productividad, Calidad y Satisfacción
45%
40%
35%
30%
25%
20%
15%
10%
5%
0%
Productividad
Calidad
Satisfacción
Mucho más alta
Un poco superior
Igual
Un poco inferior
Mucho más baja
110


4.2 Estadísticas
Ambysoft – Resumen
Resumen de 4232 profesionales, S. Ambler 2006
Mayor satisfacción de todos
58%
Mejora de la calidad
66%
Mejora de la productividad
60%
Han adoptado uno o más métodos ágiles
41%
Han adoptado técnicas ágiles
65%
0%
10%
20%
30%
40%
50%
60%
70%
111


4.3 Relaciones
Similitudes y complementariedad
Muestras de complementariedad y puntos en común:
• En las herramientas de RUP ahora hay plug-ins para XP.
• Crystal permite la adopción de prácticas de XP y Scrum para reemplazar algunas de sus propias prácticas, como los talleres de reflexión.
• Se han elaborado combinaciones DSDM + XP, DSDM + RUP, DSDM + MSF.
• Para las técnicas concretas de programación, LD permite XP o Scrum.
• Pragmatic Programming sólo da pautas sobre cómo programar.
• Muchos son complementos adecuados para MSF: AM, RUP, XP, DSDM...
Puntos totales
Iteraciones
Puntos totales
Desarrollados
Testeados
Sin completar
Completados
Gráficos de quemado que utilizan casi todos los métodos.
112



4.4 Crítica

Crítica a las prácticas ágiles

Programación en parejas: se paga el doble y el rendimiento no es el doble. A la gente no le gusta que la corrijan.

Código: En lenguajes como C++ (no Smalltalk), cuanto más óptimo es, más difícil es de leer.

Requisitos: Si no se escribe, cada uno la recuerda a su manera

Documentación: leer código fuente es lento (sobre todo si está optimizado), UML o casos de uso facilitan entender el programa.

Pruebas: Dijkstra afirmaba que “la prueba de programas puede ser una manera muy efectiva para mostrar la presencia de bugs, pero es irremediablemente inadecuada para mostrar su ausencia”.

Refactorización: se debe refactorizar porque no se ha diseñado al principio.

Cliente en casa: es caro tener a un buen profesional de la empresa a disposición total; lo necesitan en su empresa.

Contrato de alcance parcial: es difícil aceptar “lo que salga de la iteración”.

Workproducts: aunque no se citen XP usa muchos: historias, tarjetas CRC,

114

5. Anexos

Glosario

80/20: el 80% del valor proviene del 20% de las características o funcionalidades que no se hacen funcionar nunca.

Caórdica: de *Chaordic*, una combinación de caos y orden, palabra inventada por Dee Hock, fundador y anterior CEO de Visa International.

Caso de uso (use case): Es una pieza de funcionalidad bien delimitada y reutilizable que da valor a N Actores que interactúan con el sistema.

CRC, tarjetas. Las tarjetas *Clase-Responsabilidad-Colaborador* son simples tarjetas de papel, de 4x6 (o 3x5) pulgadas, y es una técnica que reemplaza a los diagramas en la representación de modelos.

Excelencia técnica: "Hacer lo correcto, correctamente".

Experiencia, nivel de: En los métodos ágiles se definen tres niveles de experiencia. Un programador de Nivel 1 es capaz de "seguir los procedimientos"; uno de Nivel 2 es capaz de "apartarse de los procedimientos específicos y encontrar otros distintos" y uno de Nivel 3 es capaz de "manejar con fluidez, mezclar e inventar procedimientos".

Framework: estructura de apoyo para organizar y puede desarrollarse software utilizándola. Puede incluir programas de apoyo, librerías de código, un lenguaje de script, u otro software para ayudar a desarrollar y unir los diferentes componentes de un proyecto de software. Un ejemplo, el .NET de Microsoft.

MoSCoW: Método para establecer prioridades. Significa:

M - MUST, Debe tener esto.

S - SHOULD Debería tener esto si fuera posible.

C - COULD Podría tener esto si no afecta a ninguna función.

W - WON'T No tendrá esto ahora, pero quizá sí en un futuro.

115

5. Anexos

Glosario

Prueba (test) unitaria o de unidad: (a nivel de método) verifica una clase, o un pequeño conjunto de clases y son responsabilidad del programador.

Prueba (test) funcional o de aceptación: verifica todo el sistema, o una gran parte, y son propuestas por el cliente.

QA (Quality Assessment o Quality Assurance): El Control de calidad está formado por la evaluación, y mediciones de los procesos de diseño, desarrollo, producción, instalación, servicio y documentación.

Requisitos no funcionales: Cuestiones como usabilidad, seguridad y rendimiento.

SLA: Service Level Agreement. Es un acuerdo de nivel de servicio por el que una compañía se compromete a prestar un servicio a otra bajo determinadas condiciones y con un nivel de calidad y prestaciones mínimas

Smalltalk: Lenguaje de programación considerado el primero en utilizar el paradigma orientado a objetos. En Smalltalk todo es un objeto, incluso el propio entorno Smalltalk. Se caracteriza por su orientación a objetos pura, tipado dinámico, herencia simple, interactúa entre objetos mediante envío de mensajes y posee un recolector de basura. Es multiplataforma y puede compilar en tiempo de ejecución o interpretado. Smalltalk tuvo gran influencia en la creación de Java o Ruby.

Smoke Test: test simple para asegurar que el software funciona hasta un punto mínimo. Proviene de prácticas de hardware, donde se dejaba el circuito a ver si se quemaba).

Spike: Una púa es un trozo desechable de código, usado para comprender cómo podría resolverse un problema de programación, para saber si se está en la dirección correcta; es diferente del esqueleto ambulante de *Crystal*. Se lo llama así porque "va de punta a punta, pero es muy fino".

116

5. Anexos

Glosario

Stakeholder: cada uno de los participantes en un proyecto como el usuario final, el contratista, el comprador, etc.

Test-first: Cuando se programa por parejas, antes de escribir código, se escriben los tests automatizados para verificarlo.

Testear: Verificar y Validar. Los métodos ágiles usan este término para referirse a cualquier actividad V&V, no sólo a las pruebas manuales o automáticas.

Test de regresión: Test realizado después de realizar un cambio (arreglo o mejora) para asegurarse de que el sistema no ha sufrido un retroceso (la funcionalidad que antes funcionaba debe seguir funcionando).

UML: Unified Modelling Language. Notación estándar que permite modelar visualmente todos los procesos implicados en el análisis, diseño y desarrollo orientado a objetos de un sistema.

V&V: Validar y Verificar. Ver Testear.

Validación: Comprobar (testeando o repasando) para asegurarse que el sistema tal y como está construido o especificado, cumple las necesidades.

Verificación: Comprobar (testeando o repasando) para asegurarse de que el sistema se ha construido o especificado de forma precisa y con integridad.

Wiki: significa rápido en hawaiano. Una wiki es una página web que facilita que todo el mundo contribuya a ampliarla y a crear enlaces entre sus páginas. En general, Wikis son un tipo de software que ayuda a comunicarse en línea, donde editar es tan fácil como leer, siendo la herramienta perfecta para colaborar en grupo.

Work product: Cualquier documento que requiere utilizar un método. Se suele traducir al castellano como artefacto.

117

5. Anexos

Bibliografía básica

- Ambler, S. Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. John Wiley & Sons, 2002.
- Beck, K. Extreme programming explained: Embrace change. Addison-Wesley, 1999.
- Beck, K & Andres, C. Extreme programming explained: Embrace change. Second edition. Addison-Wesley, 2004.
- Cockburn, A. Agile Software Development. Addison-Wesley, 2002.
- Gilb, T. Principles of Software Engineering Management. Addison-Wesley, 1988.
- Highsmith, J. Adaptive Software Development: A Collaborative Approach to Managing Complex Systems. Dorset House Publishing, 2000.
- Highsmith, J. Agile software development ecosystems. Pearson Education, 2002.
- Hunt, A. & Thomas, D. The Pragmatic Programmer. Addison Wesley, 2000.
- Kruchten, P. The Rational Unified Process: an Introduction. Addison-Wesley, 2000.
- Palmer, S. & Felsing, J. A Practical Guide to Feature-Driven Development. Prentice-Hall, 2002.
- Poppendieck, Mary & Tom. Lean Software Development: An Agile Toolkit for Software Development Managers. Addison-Wesley, 2003.
- Schwaber, K. & Beedle, M. Agile Software Development With Scrum. Prentice-Hall, 2002.
- Stapleton, J. Dynamic Systems Development Method – The method in practice. Addison Wesley, 1997.
- Turner, M. Microsoft Solutions Framework Essentials: Building successful technology solutions. Microsoft Press, 2006.

118

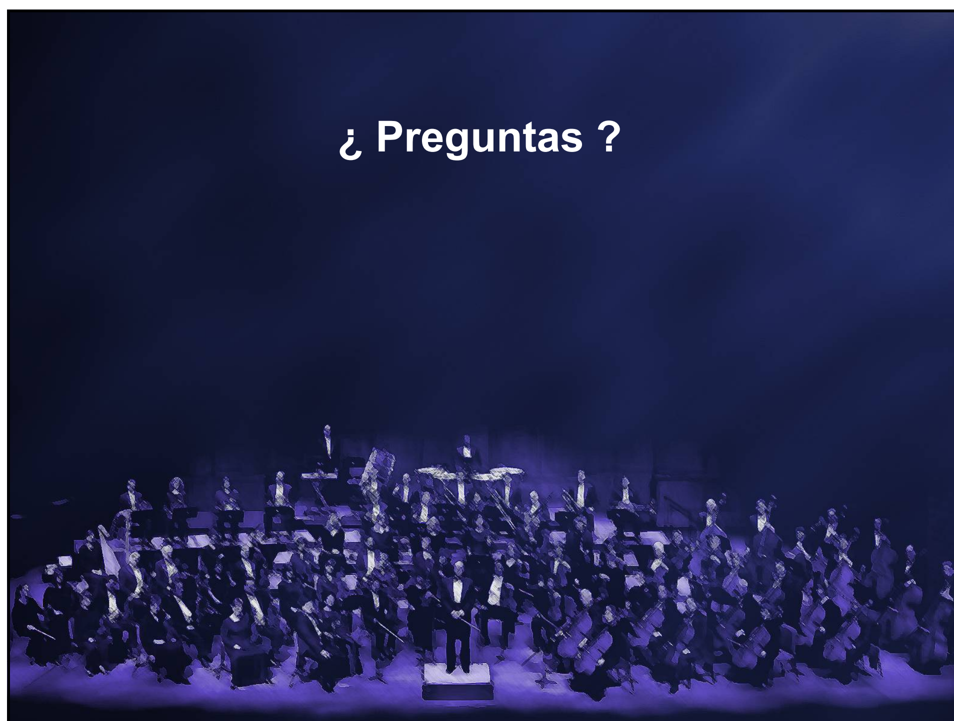
5. Anexos

Bibliografía web

Scrum	www.controlchaos.com http://jeffsutherland.com www.scrumalliance.org
XP	www.extremeprogramming.com www.extremeprogramming.org www.xprogramming.com www.programacionextrema.org www.martinfowler.com www.xpday.info www.xp2007.org
Crystal	http://alistair.cockburn.us http://agile.csc.ncsu.edu/crystal.html
FDD	www.nebulon.com/fdd www.featuredrivendevelopment.com www.togethercommunity.com www.petercoad.com
ASD	www.adaptivesd.com
RUP EUP AUP	www.rational.com www.enterpriseunifiedprocess.info www.ambysoft.com/unifiedprocess
AM	www.agilemodelling.com www.ambysoft.com www.agiledata.org

MSF	www.microsoft.com/msf
Evo	www.gilb.com
LSD	www.poppendieck.com
OSS	www.opensource.org
DSDM	www.dsdm.org
PP	www.pragmaticprogrammer.com
Agile & Agile Project Manifesto	www.agilemanifesto.org www.pmdoi.org
Agile Alliance	www.agilealliance.org
Cutter Consortium	www.cutter.com
Agile 2007 conference	http://agile2007.com
Miscelánea	www.agile-spain.com www.agileadvice.com www.agileJournal.com www.agiledeveloper.com www.agilemanagement.net www.codinghorror.com http://ebe.cpsc.ucalgary.ca/ebe www.stevemcconnell.com www.vtt.fi

119



CAPÍTULO

6

CONTENIDO

6.1. Métodos ágiles en Internet

6.2. Índice de ilustraciones

6.3. Índice de tablas

6.4 Bibliografía

BIBLIOGRAFÍA E ÍNDICES

"Adaptar viejos programas para que funcionen en ordenadores nuevos normalmente significa adaptar los nuevos ordenadores para comportarse como los viejos."

Alan J. Perlis

6. Bibliografía e índices

6.1. Métodos ágiles en Internet

Filosofía ágil y herramientas	
www.agilemanifesto.org www.agile-spain.com www.vtt.fi http://agile2007.com www.agilealliance.org www.pmdoi.org www.apln.org www.agileadvice.com www.agileJournal.com www.agilemanagement.net/Articles/Papers www.cutter.com http://www.agiledeveloper.com/	Manifesto for Agile Software Develop. Software ágil VTT publications Agile 2007 Conference Agile Alliance Agile Project Manifesto Agile Project Leadership Network Cutter Consortium Artículos por Venkat Subramaniam
www.codinghorror.com www.testing.com www.refactoring.com www.construx.com/survivalguide/home.htm www.complianceautomation.com www.developertesting.com www.booch.com/architecture/index.jsp	Problemas de programación Testeos Refactorizar Steve McConnell's Survival Guide site Characteristics of Good Requirements Testeos y calidad de software Grady Booch, Handbook of Software Architecture

Métodos	
www.extremeprogramming.com www.extremeprogramming.org www.xprogramming.com www.programacionextrema.org www.martinfowler.com www.xpday.info www.xp2007.org www.xbreed.net	XP Ron Jeffries Martin Fowler XBreed (Scrum + XP)
www.controlchaos.com http://jeffsutherland.com www.scrumalliance.org	Creador de Scrum Jeff Sutherland Scrum Alliance
http://alistair.cockburn.us http://agile.csc.ncsu.edu/crystal.html	Alistair Cockburn, CRYSTAL Crystal-Adaptive Software Development
www.nebulon.com/fdd www.featuredrivendevelopment.com www.togethercommunity.com www.petercoad.com	FDD
www.opensource.org	Open Source Initiative – OSS
www.dsdm.org	Dynamic Systems Dev. Method – DSDM
www.adaptivesd.com	Adaptive Software Development – ASD

www.enterpriseunifiedprocess.info www.rational.com www.ambysoft.com/unifiedprocess/agileUP.html www.rational.com/uml	Enterprise Unified Process (EUP) Rational Unified Process (RUP) AgileUP UML
www.pragmaticprogrammer.com	Pragmatic Programming
www.poppendieck.com	Lean Software Development - LSD
www.gilb.com	Evo
www.agilemodelling.com www.ambysoft.com www.agiledata.org	Agile Modelling – AM, AMDD
www.sei.cmu.edu/cmm	CMM
www.stevemcconnell.com htttp://csweb.cs.bgsu.edu/maner/domains/RAD.htm www.blueink.biz/RapidApplicationDevelopment.aspx http://www.gantthead.com/process/processMain.cfm?ID=2-19516-2	Steve McConnell RAD Rapid Application Development RAD

GLOSARIOS DE INGENIERÍA DE SOFTWARE:

Freedman, A. 9th Computer Glossary: The Complete Illustrated Dictionary (con CD-ROM). New York, NY: American Management Association, 2000.

IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology.

Nader, J. Illustrated Dictionary of Computing (con CD-ROM). Prentice Hall, 1998.

GLOSARIOS DE GESTIÓN DE PROYECTOS

www.pmforum.org/library/glossary/index.htm. "Wideman Comparative Glossary of Project Management Terms"

GLOSARIOS DE INGENIERÍA DE CALIDAD

www.1stnclass.com/quality_glossary.htm
www.asq.org/glossary. "American Society of Quality Glossary of Terms"

6.2. Índice de ilustraciones

Ilustración 1. Método en cascada, contra el que luchan todos los métodos ágiles. 14
Ilustración 2. Resumen estadísticas 1994-2004 de Standish Group. 15
Ilustración 3. Esquema del modelo iterativo. 17
Ilustración 4. Entrega incremental de Steve McConnel (Microsoft, 1996). 17
Ilustración 5. Comparación del orden de las fases entre métodos de cascada, iterativo y XP Programming. 17
Ilustración 6. Opinión sobre el cambio en productividad, calidad y satisfacción al aplicar métodos o algunas de las prácticas ágiles. 19
Ilustración 7. Distribución de los métodos ágiles utilizados en la encuesta de Ambysoft. 19
Ilustración 8. Prácticas ágiles adoptadas (Ambysoft 2006). 20
Ilustración 9. Mejoras al aplicar técnicas ágiles, según encuesta de Ambysoft (2006). 20
Ilustración 10. Visión cronológica general de los métodos ágiles más destacables y sus influencias. 22
Ilustración 11. Fases y pasos en un proyecto XP. 23
Ilustración 12. Fases y workproducts del método de gestión Scrum, punto de referencia para muchos otros. 23
Ilustración 13. Prácticas de un sprint en Scrum. 24
Ilustración 14. Modelo original en cascada de Royce (1970). 30
Ilustración 15. Modelo en espiral de Barry Boehm (1988). 32
Ilustración 16. Modelo iterativo de desarrollo. 32
Ilustración 17. Modelo en V. 34
Ilustración 18. Una mejora del modelo en cascada, modelo Sashimi, cascada con fases superpuestas. 34
Ilustración 19. Otra mejora del modelo en cascada, cascada con subproyectos. 35
Ilustración 20. Evolución del método en cascada, entrega por etapas. 36
Ilustración 21. Entrega incremental. 36
Ilustración 22. Consecuencias preocupantes de los métodos no ágiles. 37
Ilustración 23. Eje temporal de la evolución de los métodos de desarrollo. 41
Ilustración 24. Kent Beck. 48
Ilustración 25. Ward Cunningham. 48
Ilustración 26. Pasos del proceso de Extreme Programming. 51
Ilustración 27. Valores y prácticas de Extreme Programming. 55
Ilustración 28. Coste de los cambios según Boehm y Kent Beck. 59
Ilustración 29. Relación entre las viejas y nuevas prácticas de XP. 63
Ilustración 30. Caricatura de una reunión de Scrum. 70
Ilustración 31. Ken Schwaber. 70
Ilustración 32. Proceso de Scrum. 71
Ilustración 33. Fábula agilista sobre el cerdo y la gallina usados en Scrum. 73
Ilustración 34. Ficha de Product Backlog utilizada en Scrum. 74
Ilustración 35. Prácticas de un sprint en Scrum. 75
Ilustración 36. Ficha de Sprint Backlog utilizada en Scrum. 75
Ilustración 37. Alistair Cockburn. 81
Ilustración 38. Dimensiones de las metodologías Crystal. 82
Ilustración 39. Un incremento de Crystal Orange. 85
Ilustración 40. Ciclos anidados de Crystal Clear para mostrar las actividades diarias. 86
Ilustración 41. Dos ejemplos de gráficos de quemado para mostrar el progreso del proyecto. 91
Ilustración 42. Procesos de FDD. Las dos últimas fases son iterativas. 94
Ilustración 43. Procesos Diseño y Construcción por característica de FDD. 96
Ilustración 44. Fases de RUP. 101
Ilustración 45. Iteraciones de RUP durante el ciclo de vida. 102
Ilustración 46. Diagrama de procesos de DSDM. 113
Ilustración 47. Jim Highsmith. 118
Ilustración 48. El ciclo de ASD. 118
Ilustración 49. Fases del ciclo de vida de ASD. 119
Ilustración 50. Fases visibles de un proyecto OSS. 122
Ilustración 51. Bob Charette. 126
Ilustración 52. La base de todos los principios de LSD. 128
Ilustración 53. Mapa de flujo de valor (Value stream mapping) de un método ágil. 129
Ilustración 54. Mapa de flujo de valor de un método tradicional. 129
Ilustración 55. Mapa de flujo de valor de Kent Beck y su XP. 129
Ilustración 56. Método en cascada y método iterativo. 130

Ilustración 57. Sincronización total en LSD. 131
Ilustración 58. Ciclo completo de desarrollo de un proyecto aplicando el método Lean. 131
Ilustración 59. Enfoques Breadth-first (global) y Depth-first (detalles). 132
Ilustración 60. Método rígido Mandar & Controlar (Command &Control), contrario a Scrum. 134
Ilustración 61. Tiempo de ciclo de los procesos. 144
Ilustración 62. Funcionamiento general de Agile Model Driven Development. 147
Ilustración 63. Diagrama de artefactos de Agile Modeling / EUP. 151
Ilustración 64. Método PSDA: Planear-Hacer-Estudiar-Actuar. 152
Ilustración 65. Elementos de Evo. 153
Ilustración 66. Entorno Planear-Hacer-Estudiar-Actuar en el entorno de Evo. 156
Ilustración 67. Modelo de entrega evolutiva, basado en Steve McConnell. 158
Ilustración 68. Esquema con las cinco fases y actividades de Internet-Speed. 160
Ilustración 69. Complementariedad entre MOF y MSF. 161
Ilustración 70. Modelo de proceso de MSF. 162
Ilustración 71. Principios de Microsoft Solutions Framework. 166
Ilustración 72. Modelo de equipo de MSF. 167
Ilustración 73. Control de cambios de MSF. 167
Ilustración 74. Control de riesgos de MSF. 168
Ilustración 75. MVC, diagrama que muestra la relación entre el modelo, la vista y el controlador. 173
Ilustración 76. Ejemplo de plataforma abierta de desarrollo, Eclipse, desde donde ejecutar JUnit. 176
Ilustración 77. Acceso a JUnit desde Java Studio Enterprise. 177
Ilustración 78. El framework Nant (.NET) automatiza las fases de un proceso de integración continua. 178
Ilustración 79. Muestra de informe generado por CruiseControl en la versión para .NET. 179
Ilustración 80. Información de monitorización que proporciona CruiseControl. 179
Ilustración 81.Control de versiones con CVS en Eclipse. 180
Ilustración 82. Bugzilla para la gestión de bugs a través de Internet. 181
Ilustración 83. Asistente de refactorización automática. 182
Ilustración 84. Opciones de refactorización manual desde NetBeans IDE 5. 182
Ilustración 85. JCSC para homogeneizar estilos y estándares de codificación 183
Ilustración 86. Resumen de integraciones y entregas en XPlanner. 185
Ilustración 87. Perfil de un participante de XPlanner. 186
Ilustración 88. Iteración de XPlanner. 187
Ilustración 89. Historia, mostrada en la característica tarjeta. 188
Ilustración 90. Tarea en XPlanner. 189
Ilustración 91. Métricas de la iteración de XPlanner. 190
Ilustración 92. Estadísticas y gráficos de quemado. 191
Ilustración 93. Campos a rellenar de cada tarea. 192
Ilustración 94. Propiedades del proyecto y de las entregas. 192
Ilustración 95. Ciclo de tareas y entregas. 193
Ilustración 96. Calendario y estimaciones de duración de ciclo. 193
Ilustración 97. Relación entre los programas de Rally, CRM y las herramientas de desarrollo 194
Ilustración 98. Detalle de una característica. 194
Ilustración 99. Product backlog. 195
Ilustración 100. Editor de casos de uso para especificar requisitos. 195
Ilustración 101. Estado de una iteración de un test case. 196
Ilustración 102. Resumen de test cases. 196
Ilustración 103. Editor de test cases. 197
Ilustración 104. Resumen del estado de los defectos. 197
Ilustración 105. Resumen del estado de los defectos de toda la iteración. 198
Ilustración 106. Añadir un nuevo defecto o bug. 198
Ilustración 107. Vista personalizada de defectos. 199
Ilustración 108. Configurar permisos para crear, editar y borrar, según el grupo de pertenencia. 199
Ilustración 109. Seguimiento de varios equipos con Program roadmap. 200
Ilustración 110. Estado global del programa. 200
Ilustración 111. Plan de la versión. 201
Ilustración 112. Estado de la versión. 201
Ilustración 113. Métricas de la iteración. 202
Ilustración 114. Plan de la iteración. 202
Ilustración 115. Estado de las tareas de una iteración. 203
Ilustración 116. Gráficos de quemado de la iteración y acumulativo. 203

Ilustración 117. Ejemplo de notificaciones recibidas. 204
Ilustración 118. Vista de las tareas asignadas a una persona. 204
Ilustración 119. Recoger el feedback. 205
Ilustración 120. Valoración de los requisitos. 205
Ilustración 121. Detalle de requisito. 206
Ilustración 122. Página principal del "dashboard" (cuadro de mandos) de la versión. 206
Ilustración 123. Detalle de la versión. 207
Ilustración 124. Vista general del plan de una iteración. 208
Ilustración 125. Seguimiento de la iteración. 209
Ilustración 126. Resumen de la iteración. 209
Ilustración 127. Agile Enterprise dispone de más de 50 gráficos predefinidos. 210
Ilustración 128. Horas de las tareas asignadas a cada persona. 210
Ilustración 129. La página de inicio de TargetProcess es personalizable. 211
Ilustración 130. Descomposición de un proyecto en características, historias y tareas. 212
Ilustración 131. Control de releases: fechas, horas, progreso, estado de las iteraciones, etc. 212
Ilustración 132. Para mover cualquier elemento, sólo hay que arrastrar y soltar. 213
Ilustración 133. Basta con arrastrar y soltar para mover historias o bugs a las iteraciones. 214
Ilustración 134. Es posible personalizar las partes de cada proceso. 214
Ilustración 135. Time Tracking facilita un seguimiento preciso y completo de la iteración. 215
Ilustración 136. Descomposición de historias en tareas, mostrando si están retrasadas. 215
Ilustración 137. Gráfico de quemado: esfuerzo total, estimado y completado. 216
Ilustración 138. Asignación de usuarios. 216
Ilustración 139. Gestión de bugs integrada. 217
Ilustración 140. Test Runner simplifica las pruebas manuales. 218
Ilustración 141. TargetProcess da toda la información para asegurar la calidad. 218
Ilustración 142. La Bug Submission Tool permite adjuntar capturas de pantalla para describir bugs. 219
Ilustración 143. Es posible añadir información muy rápido introduciendo los datos en modo texto. 219
Ilustración 144. Se pueden ver las diferencias entre versiones. 220
Ilustración 145. TargetProcess facilita información sobre todo el equipo y la carga que llevan. 220
Ilustración 146. Carga de trabajo para cada desarrollador. 221
Ilustración 147. Página de inicio de ExtremePlanner. 222
Ilustración 148. Informe de actividades. 222
Ilustración 149. Gestión de versión, historias y tareas. 223
Ilustración 150. Datos para especificar una historia. 223
Ilustración 151. Seguimiento de las tareas de cada historia para formar una versión. 224
Ilustración 152. Gráfico de quemado. 224
Ilustración 153. Informe de las historias con sus respectivas tareas, avances, estados, autores... 225
Ilustración 154. Plan de una versión y posibilidad de exportar informes a Excel, Word y XML. 225
Ilustración 155. Un control de acceso permite seguir un proyecto y mantener partes reservadas. 226
Ilustración 156. Integración para actualizar información de tareas desde Eclipse o MS Visual Studio. 226
Ilustración 157.Bamboo facilita la integración continua. 227
Ilustración 158. Crucible para la revisión de código. 228
Ilustración 159. FishEye o repositorio de Atlassian. 228
Ilustración 160. Vista del entorno que ve un participante con sus tareas asignadas, calendarios, etc. 229
Ilustración 161. Diagrama de proceso. 230
Ilustración 162. Vista del calendario, gráfico de quemado, tareas... de un proyecto en xProcess. 231
Ilustración 163. En el Team Explorer de Visual Studio Team System, podemos ver en una misma ventana los requisitos, las tareas y los bugs. 232
Ilustración 164. Entorno de Microsoft Visual Studio. 234
Ilustración 165. Ojalá pudiera parar y pensar un momento. 236
Ilustración 166. Cambio en la concepción: Cascada-Iterativo-XP. 236
Ilustración 167. Árbol genealógico de los métodos ágiles. Extraído del artículo New Directions on Agile Methods: A Comparative Analysis (Abrahamsson et al.). 237
Ilustración 168. Gastos derivados del cambio en XP y en los métodos tradicionales según Beck. 250
Ilustración 169. Comparación-integración entre XP y LSD. 252
Ilustración 170. Estadística sobre el uso de las funciones del software. 253
Ilustración 171. Mejoras de 1994 a 2004 al aplicar conceptos ágiles. 254
Ilustración 172. Sobrecosto económico y de tiempo respecto a las estimaciones iniciales. 254
Ilustración 173. Tamaño de los grupos encuestados. 255
Ilustración 174. Porcentajes de proyectos que utilizaron prácticas o métodos ágiles. 255

Ilustración 175. Prácticas ágiles concretas utilizadas. 256
Ilustración 176. Cambios en la productividad, calidad y satisfacción al aplicar prácticas ágiles 257
Ilustración 177. Resumen de beneficios al aplicar conceptos ágiles. 257
Ilustración 178. Métodos utilizados en base a un estudio donde el 41% de 2611 proyectos que afirmaron usar métodos ágiles. 258

6.3. Índice de tablas

Tabla 1. Diferencias de enfoques: metodológico, el requerido en la práctica y el proporcionado por los métodos ágiles. 16
Tabla 2. Resumen de diferencias entre los métodos tradicionales y los ágiles. 16
Tabla 3. Esquema de los métodos que se detallarán, con sus autores y año de publicación. 21
Tabla 4: Historia de los principales modelos precursores de los métodos ágiles. 29
Tabla 5. Diferencias entre métodos ágiles y tradicionales. 38
Tabla 6: Boehm compara los métodos ágiles y los orientados a procesos (plan-driven) y añade el ejemplo del Open Source Software. 42
Tabla 7. Un ejemplo sencillo de organización de un proyecto XP. 52
Tabla 8. Lista de actividades en secuencia vertical cronológica. 86
Tabla 9. Ejemplo de un plan de características, con la típica codificación de colores. 99
Tabla 10. Las prácticas de RUP. 104
Tabla 11. Fases y milestones de AUP. 106
Tabla 12. Disciplinas de AgileUP. 106
Tabla 13. Descripción de los papeles y disciplinas donde actúan. 108
Tabla 14. Deliverables mínimos para Agile UP. 109
Tabla 15. Otros artefactos secundarios de AUP. 111
Tabla 16. Workproducts facilitados por la empresa. 112
Tabla 17. Principios de DSDM. 116
Tabla 18. Preguntas antes de adoptar DSDM. 117
Tabla 19. Características de ciclos de desarrollo adaptativos. 120
Tabla 20. Posibles artefactos para el modelado de análisis. 150
Tabla 21. Ejemplo conceptual de tabla de impacto. 155
Tabla 22. Tabla de impacto para elegir una de tres soluciones según tres parámetros. 155
Tabla 23. Funciones de las diferentes partes de Visual Studio. 233
Tabla 24. Visual Team System: ahorros en la fase de desarrollo. 234
Tabla 25. Visual Team System: ahorros en la fase de prueba. 234
Tabla 26. La teoría del enfoque metodológico, la práctica, y el enfoque ágil. 238
Tabla 27. Resumen comparativo de los métodos ágiles más importantes. 241
Tabla 28. Relación entre las prácticas de la programación extrema, las del modelo bazar y las herramientas libres que dan soporte a las prácticas. 247

6.4. Bibliografía

Ambler, S. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. John Wiley & Sons, 2002.

Beck, K. *Extreme programming explained: Embrace change*. Addison-Wesley, 1999.

Beck, K & Andres, C. *Extreme programming explained: Embrace change. Second Edition*. Addison-Wesley, 2004.

Bergquist, M. & Ljungberg, J. *The power of gifts: organizing social relationships in open source communities*. Information Systems Journal 11(4): 305–320, 2001.

Boehm, B. *Software Engineering Economics*. Prentice Hall, 1981.

Boehm, B. *Get ready for the Agile Methods, with care*. Computer 35(1): 64–69, 2002.

Brooks, F. *The Mythical ManMonth: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, 1995.

Coad, P., LeFebvre, E. & De Luca, J. *Java Modeling In Color With UML: Enterprise Components and Process*. Prentice Hall, 2000.

Cockburn, A. *Agile Software Development*. Addison-Wesley, 2002.

Cockburn A. *Crystal Clear: A Human-Powered Methodology for Small Teams.* The Agile Software Development Series. Addison-Wesley, 2004.

Crowston, K. & Scozzi, B. *Open source software projects as virtual organisations: competency rallying for software development*. IEE Proceedings – Software 149(1): 3–17, 2002.

Favaro, J. M*anaging Requirements for Business Value*. IEEE Software, 19: 15–17, 2002.

Feller, J. & Fitzgerald, B. *A Framework Analysis of the Open Source Software Development Paradigm*. 21st Annual International Conference on Information Systems, Australia, 2000.

Gilb, T. *Principles of Software Engineering Management*. Addison-Wesley, 1988.

Highsmith, J. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing, 2000.

Highsmith, J. & Cockburn, A. *Agile Software Development: The Business of Innovation*. Computer 34(9): 120–122, 2001.

Highsmith, J. *Agile software development ecosystems*. Pearson Education, 2002.

Humphrey, W. *A discipline for software engineering*. Addison-Wesley Longman, 1995.

Hunt, A. & Thomas, D. *The Pragmatic Programmer*. Addison-Wesley, 2000.

Hunt, A. & Subramaniam, V. *Practices of an agile developer. Working in the real world*. The pragmatic bookshelf, 2006.

Hunt, J. *Agile Software Construction*. Springer, 2006.

Jeffries, R., Anderson, A. & Hendrickson, C. *Extreme Programming Installed*. Addison-Wesley, 2001.

Koch, A. *Agile Software Development, Evaluating the Methods For Your Organization*. Artech House Publishers, 2005.

Kruchten, P. *The Rational Unified Process: an introduction*. Addison-Wesley, 2000.

Lerner, J. & Tirole, J. The Simple Economics of Open Source. 2001. http://www.people.hbs.edu/jlerner/publications.html

Maurer, F. & Martel, S. *On the Productivity of Agile Software Practices: An Industrial Case Study*. http://ebe.cpsc.ucalgary.ca/ebe/Wiki.jsp?page=Publications, 2002.

Mockus, A., Fielding, R. & Herbsleb, J. *A Case Study of Open Source Software Development: The Apache Server*. 22nd International Conference on Software Engineering, ICSE 2000, Limerick, Ireland, 263–272, 2000.

O'Reilly, T. *Lessons from Open Source Software Development*. Communications of the ACM Vol. 42(No. 4): 32–37, 1999.

Palmer, S. & Felsing, J. *A Practical Guide to Feature-Driven Development*. Prentice-Hall, 2002.

Poppendieck, Mary & Tom. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, 2003.

Rising, L. & Janoff, N. *The Scrum software development process for small teams*. IEEE Software 17(4): 26–32, 2000.

Schwaber, K. *Scrum Development Process. OOPSLA'95 Workshop on Business Object Design and Implementation*. Springer-Verlag, 1995.

Schwaber, K. & Beedle, M. *Agile Software Development with Scrum*. Prentice-Hall, 2002.

Sharma, S., Sugumaran, V. & Rajagopalan, B. (2002). A framework for creating hybrid-open source software communities. Information Systems Journal 12(1): 7–25, 2002.

Stapleton, J. *Dynamic Systems Development Method – The method in practice*. Addison-Wesley, 1997.

Turner, M. *Microsoft Solutions Framework Essentials: Building Successful Technology Solutions*. Microsoft Press, 2006.