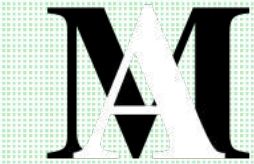
Metodologías Ágiles en el Desarrollo de Software

Alicante, 12 de Noviembre de 2003



Organización:

Grupo ISSI

(Ingeniería del Software y Sistemas de Información)

Colaboradores:

UNIVERSIDAD POLITECNICA DE VALENCIA

Taller realizado en el marco de las VIII Jornadas de Ingeniería del Software y Bases de Datos, JISBD 2003.
Alicante, del 12 al 14 de Noviembre de 2003

Actas

Metodologías Ágiles en el Desarrollo de Software

Alicante – España
12 de Noviembre de 2003

Editado Por:
Patricio Letelier Torres
Emilio A. Sánchez López

Grupo ISSI
(Ingeniería del Software y Sistemas de Información)

Comité organizador

Patricio Letelier Torres

Mª Carmen Penadés

José Hilario Canós

Emilio A. Sánchez López

Comité Técnico

Esperanza Marcos, *Universidad Rey Juan Carlos*
Rafael Corchuelo, *Universidad de Sevilla*
Mikel Emaldi, *European Software Institute (ESI)*
César Pérez-Chirinos, *Banco de España*
José Hilario Canós, *Universidad Politécnica de Valencia*

Presentación

En las dos últimas décadas las notaciones de modelado y posteriormente las herramientas han sido las "balas de plata" para el deseado éxito en el desarrollo de software. El proceso de desarrollo asumido en este contexto llevaba asociada una marcada tendencia hacia el control del proceso mediante una rigurosa definición de actividades, artefactos y roles. Este esquema "tradicional" para abordar el desarrollo de software ha demostrado ser efectivo en proyectos de gran envergadura donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el contexto es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. En la práctica, para muchos equipos de desarrollo, ante las dificultades para utilizar metodologías tradicionales, se llegó a la resignación de prescindir del "buen hacer" de la ingeniería del software con el objetivo de ajustarse a estas restricciones. Ante esta situación, las metodologías ágiles aparecen como una posible respuesta para llenar este vacío metodológico. Por estar especialmente orientadas para proyectos pequeños, las metodologías ágiles constituyen una solución a medida, con una elevada simplificación que a pesar de ello no renuncia a las prácticas esenciales para asegurar la calidad del producto.

El tema es de rabiosa actualidad. La curiosidad que siente la mayor parte de ingenieros de software, profesores, e incluso alumnos, sobre los métodos ágiles hace prever una fuerte proyección industrial de las metodologías ágiles. Por un lado, para muchos equipos de desarrollo el uso de metodologías tradicionales les resulta muy lejano a su forma de trabajo actual considerando las dificultades de su introducción e inversión asociada en formación y herramientas. Por otro, las características de los proyectos para los cuales las metodologías ágiles han sido especialmente pensadas se ajustan a un amplio rango de proyectos industriales de desarrollo de software; aquellos en los cuales los equipos de desarrollo son pequeños, con plazos reducidos, requisitos volátiles, nuevas tecnologías, etc. Esto último abriría interesantes canales de cooperación entre la industria y la universidad.

El objetivo de este taller ha sido constituir un primer punto de encuentro en España para académicos y gentes de industria interesados en metodologías ágiles.

Esperamos que este taller cumpla con dichas espectativas y genere nuevas iniciativas y vínculos de colaboración.

Los editores.

Índice

Métodologías Ágiles en el Desarrollo de Software

José H. Canós, Patricio Letelier y Mª Carmen Penadés

DSIC -Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia
{ jhcanos | letelier | mpenades }@dsic.upv.es

RESUMEN

El desarrollo de software no es una tarea fácil. Prueba de ello es que existen numerosas propuestas metodológicas que inciden en distintas dimensiones del proceso de desarrollo. Por una parte tenemos aquellas propuestas más tradicionales que se centran especialmente en el control del proceso, estableciendo rigurosamente las actividades involucradas, los artefactos que se deben producir, y las herramientas y notaciones que se usarán. Estas propuestas han demostrado ser efectivas y necesarias en un gran número de proyectos, pero también han presentado problemas en otros muchos. Una posible mejora es incluir en los procesos de desarrollo más actividades, más artefactos y más restricciones, basándose en los puntos débiles detectados. Sin embargo, el resultado final sería un proceso de desarrollo más complejo que puede incluso limitar la propia habilidad del equipo para llevar a cabo el proyecto. Otra aproximación es centrarse en otras dimensiones, como por ejemplo el factor humano o el producto software. Esta es la filosofía de las metodologías ágiles, las cuales dan mayor valor al individuo, a la colaboración con el cliente y al desarrollo incremental del software con iteraciones muy cortas. Este enfoque está mostrando su efectividad en proyectos con requisitos muy cambiantes y cuando se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Las metodologías ágiles están revolucionando la manera de producir software, y a la vez generando un amplio debate entre sus seguidores y quienes por escepticismo o convencimiento no las ven como alternativa para las metodologías tradicionales. En este trabajo se presenta resumidamente el contexto en el que surgen las metodologías ágiles, sus valores, principios y comparación con las metodologías tradicionales. Además se describen brevemente las principales propuestas, especialmente Programación Extrema (eXtreme Programming, XP) la metodología ágil más popular en la actualidad.

PALABRAS CLAVE. Procesos de Software, Metodologías Ágiles, Programación Extrema (XP)

1. INTRODUCCIÓN

En las dos últimas décadas las notaciones de modelado y posteriormente las herramientas pretendieron ser las "balas de plata" para el éxito en el desarrollo de software, sin embargo, las expectativas no fueron satisfechas. Esto se debe en gran parte a que otro importante elemento, la metodología de desarrollo, había sido postergado. De nada sirven buenas notaciones y herramientas si no se proveen directivas para su aplicación. Así, esta década ha comenzado con un creciente interés en metodologías de desarrollo. Hasta hace poco el proceso de desarrollo llevaba asociada un marcado énfasis en el control del proceso mediante una rigurosa definición de roles, actividades y artefactos, incluyendo modelado y documentación detallada. Este esquema "tradicional" para abordar el desarrollo de software ha demostrado ser efectivo y necesario en proyectos de gran tamaño (respecto a tiempo y recursos), donde por lo general se exige un alto grado de ceremonia en el proceso. Sin embargo, este enfoque no resulta ser el más adecuado para muchos de los proyectos actuales donde el entorno del sistema es muy cambiante, y en donde se exige reducir drásticamente los tiempos de desarrollo pero manteniendo una alta calidad. Ante las dificultades para utilizar metodologías tradicionales con estas restricciones de tiempo y flexibilidad, muchos equipos de desarrollo se resignan a prescindir del “buen hacer” de la ingeniería del software, asumiendo el riesgo que ello conlleva. En este escenario, las metodologías ágiles emergen como una posible respuesta para llenar ese vacío metodológico. Por estar especialmente orientadas para proyectos pequeños, las metodologías ágiles constituyen una solución a medida para ese entorno, aportando una elevada simplificación que a pesar de ello no renuncia a las prácticas esenciales para asegurar la calidad del producto.

Las metodologías ágiles son sin duda uno de los temas recientes en ingeniería de software que están acaparando gran interés. Prueba de ello es que se están haciendo un espacio destacado en

la mayoría de conferencias y workshops celebrados en los últimos años. Es tal su impacto que actualmente existen 4 conferencias internacionales de alto nivel y específicas sobre el tema[1]. Además ya es un área con cabida en prestigiosas revistas internacionales. En la comunidad de la ingeniería del software, se está viviendo con intensidad un debate abierto entre los partidarios de las metodologías tradicionales (referidas peyorativamente como "metodologías pesadas") y aquellos que apoyan las ideas emanadas del "Manifiesto Ágil"[2]. La curiosidad que siente la mayor parte de ingenieros de software, profesores, e incluso alumnos, sobre las metodologías ágiles hace prever una fuerte proyección industrial. Por un lado, para muchos equipos de desarrollo el uso de metodologías tradicionales les resulta muy lejano a su forma de trabajo actual considerando las dificultades de su introducción e inversión asociada en formación y herramientas. Por otro, las características de los proyectos para los cuales las metodologías ágiles han sido especialmente pensadas se ajustan a un amplio rango de proyectos industriales de desarrollo de software; aquellos en los cuales los equipos de desarrollo son pequeños, con plazos reducidos, requisitos volátiles, y/o basados en nuevas tecnologías.

El artículo está organizado como sigue. En la sección 2 se introducen las principales características de las metodologías ágiles, recogidas en el Manifiesto y se hace una comparación con las tradicionales. La sección 3 se centra en *eXtreme Programming* (XP), presentando sus características particulares, el proceso que se sigue y las prácticas que propone. En la sección 4 se citan otros métodos ágiles, enumerándose sus principales características. Finalmente aparecen las conclusiones.

2. METODOLOGÍAS ÁGILES

En febrero de 2001, tras una reunión celebrada en Utah-EEUU, nace el término “ágil” aplicado al desarrollo de software. En esta reunión participan un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores o impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó *The Agile Alliance*[3], una organización, sin ánimo de lucro, dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones para que adopten dichos conceptos. El punto de partida es fue el Manifiesto Ágil, un documento que resume la filosofía “ágil”.

2.1. El Manifiesto Ágil.

Según el Manifiesto se valora:

- **Al individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas.** La gente es el principal factor de éxito de un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente. Es mejor crear el equipo y que éste configure su propio entorno de desarrollo en base a sus necesidades.
- **Desarrollar software que funciona más que conseguir una buena documentación**. La regla a seguir es “no producir documentos a menos que sean necesarios de forma inmediata para tomar un decisión importante”. Estos documentos deben ser cortos y centrarse en lo fundamental.

[1] XP Agile Universe: `www.agileuniverse.com`. Conference on eXtreme Programming and Agile Processes in Software Engineering: `www.xp2004.org`. Agile Development Conference (EEUU): `www.agiledevelopmentconference.com`. Agile Development Conference (Australia): `www.softed.com/adc2003/`

[2] `agilemanifesto.org`

[3] `www.agilealliance.com`

- **La colaboración con el cliente más que la negociación de un contrato**. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- **Responder a los cambios más que seguir estrictamente un plan**. La habilidad de responder a los cambios que puedan surgir a los largo del proyecto (cambios en los requisitos, en la tecnología, en el equipo, etc.) determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

Los valores anteriores inspiran los doce principios del manifiesto. Son características que diferencian un proceso ágil de uno tradicional. Los dos primeros principios son generales y resumen gran parte del espíritu ágil. El resto tienen que ver con el proceso a seguir y con el equipo de desarrollo, en cuanto metas a seguir y organización del mismo. Los principios son:

I. *La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.*
II. *Dar la bienvenida a los cambios. Se capturan los cambios para que el cliente tenga una ventaja competitiva.*
III. *Entregar frecuentemente software que funcione desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.*
IV. *La gente del negocio y los desarrolladores deben trabajar juntos a lo largo del proyecto.*
V. *Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos para conseguir finalizar el trabajo.*
VI. *El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.*
VII. *El software que funciona es la medida principal de progreso.*
VIII. *Los procesos ágiles promueven un desarrollo sostenible. Los promotores, desarrolladores y usuarios deberían ser capaces de mantener una paz constante.*
IX. *La atención continua a la calidad técnica y al buen diseño mejora la agilidad.*
X. *La simplicidad es esencial.*
XI. *Las mejores arquitecturas, requisitos y diseños surgen de los equipos organizados por sí mismos.*
XII. *En intervalos regulares, el equipo reflexiona respecto a cómo llegar a ser más efectivo, y según esto ajusta su comportamiento.*

2.2. Comparación

La Tabla 1 recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales ("no ágiles"). Estas diferencias que afectan no sólo al proceso en sí, sino también al contexto del equipo así como a su organización.

3. PROGRAMACIÓN EXTREMA (*EXTREME PROGRAMMING*, XP)

XP[4] [2] es una metodología ágil centrada en potenciar las relaciones interpersonales como clave para el éxito en desarrollo de software, promoviendo el trabajo en equipo, preocupándose por el aprendizaje de los desarrolladores, y propiciando un buen clima de trabajo. XP se basa en realimentación continua entre el cliente y el equipo de desarrollo, comunicación fluida entre todos los participantes, simplicidad en las soluciones implementadas y coraje para enfrentar los cambios. XP se define como especialmente adecuada para proyectos con requisitos imprecisos y muy cambiantes, y donde existe un alto riesgo técnico.

[4] www.extremeprogramming.org,www.xprogramming.com, c2.com/cgi/wiki?ExtremeProgramming

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe contrato tradicional o al menos es bastante flexible	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones
Grupos pequeños (<10 integrantes) y trabajando en el mismo sitio	Grupos grandes y posiblemente distribuidos
Pocos artefactos	Más artefactos
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

Tabla 1. Diferencias entre metodologías ágiles y no ágiles

Los principios y prácticas son de sentido común pero llevadas al extremo, de ahí proviene su nombre. Kent Beck, el padre de XP, describe la filosofía de XP en [2] sin cubrir los detalles técnicos y de implantación de las prácticas. Posteriormente, otras publicaciones de experiencias se han encargado de dicha tarea. A continuación presentaremos las características esenciales de XP organizadas en los tres apartados siguientes: historias de usuario, roles, proceso y prácticas.

3.1. Las Historias de Usuario

Son la técnica utilizada para especificar los requisitos del software. Se trata de tarjetas de papel en las cuales el cliente describe brevemente las características que el sistema debe poseer, sean requisitos funcionales o no funcionales. El tratamiento de las historias de usuario es muy dinámico y flexible. Cada historia de usuario es lo suficientemente comprensible y delimitada para que los programadores puedan implementarla en unas semanas [12].

Beck en su libro [2] presenta un ejemplo de ficha (*customer story and task card*) en la cual pueden reconocerse los siguientes contenidos: fecha, tipo de actividad (nueva, corrección, mejora), prueba funcional, número de historia, prioridad técnica y del cliente, referencia a otra historia previa, riesgo, estimación técnica, descripción, notas y una lista de seguimiento con la fecha, estado cosas por terminar y comentarios. A efectos de planificación, las historias pueden ser de una a tres semanas de tiempo de programación (para no superar el tamaño de una iteración). Las historias de usuario son descompuestas en tareas de programación (task card) y asignadas a los programadores para ser implementadas durante una iteración.

3.2. Roles XP

Los roles de acuerdo con la propuesta original de Beck son:

- **Programador.** El programador escribe las pruebas unitarias y produce el código del sistema.

- **Cliente.** Escribe las historias de usuario y las pruebas funcionales para validar su implementación. Además, asigna la prioridad a las historias de usuario y decide cuáles se implementan en cada iteración centrándose en aportar mayor valor al negocio.

- **Encargado de pruebas (*Tester*)**. Ayuda al cliente a escribir las pruebas funcionales. Ejecuta las pruebas regularmente, difunde los resultados en el equipo y es responsable de las herramientas de soporte para pruebas.

- Encargado de seguimiento (*Tracker*). Proporciona realimentación al equipo. Verifica el grado de acierto entre las estimaciones realizadas y el tiempo real dedicado, para mejorar futuras estimaciones. Realiza el seguimiento del progreso de cada iteración.

- Entrenador (*Coach*). Es responsable del proceso global. Debe proveer guías al equipo de forma que se apliquen las prácticas XP y se siga el proceso correctamente.

- Consultor. Es un miembro externo del equipo con un conocimiento específico en algún tema necesario para el proyecto, en el que puedan surgir problemas.

- Gestor (*Big boss*). Es el vínculo entre clientes y programadores, ayuda a que el equipo trabaje efectivamente creando las condiciones adecuadas. Su labor esencial es de coordinación.

3.3. Proceso XP

El ciclo de desarrollo consiste (a grandes rasgos) en los siguientes pasos [12]:

1. El cliente define el valor de negocio a implementar.
2. El programador estima el esfuerzo necesario para su implementación.
3. El cliente selecciona qué construir, de acuerdo con sus prioridades y las restricciones de tiempo.
4. El programador construye ese valor de negocio.
5. Vuelve al paso 1.

En todas las iteraciones de este ciclo tanto el cliente como el programador aprenden. No se debe presionar al programador a realizar más trabajo que el estimado, ya que se perderá calidad en el software o no se cumplirán los plazos. De la misma forma el cliente tiene la obligación de manejar el ámbito de entrega del producto, para asegurarse que el sistema tenga el mayor valor de negocio posible con cada iteración.

El ciclo de vida ideal de XP consiste de seis fases [2]: Exploración, Planificación de la Entrega (*Release*), Iteraciones, Producción, Mantenimiento y Muerte del Proyecto.

3.4. Prácticas XP

La principal suposición que se realiza en XP es la posibilidad de disminuir la mítica curva exponencial del costo del cambio a lo largo del proyecto, lo suficiente para que el diseño evolutivo funcione. Esto se consigue gracias a las tecnologías disponibles para ayudar en el desarrollo de software y a la aplicación disciplinada de las siguientes prácticas.

- **El juego de la planificación**. Hay una comunicación frecuente el cliente y los programadores. El equipo técnico realiza una estimación del esfuerzo requerido para la implementación de las historias de usuario y los clientes deciden sobre el ámbito y tiempo de las entregas y de cada iteración.
- **Entregas pequeñas**. Producir rápidamente versiones del sistema que sean operativas, aunque no cuenten con toda la funcionalidad del sistema. Esta versión ya constituye un resultado de valor para el negocio. Una entrega no debería tardar más 3 meses.
- **Metáfora**. El sistema es definido mediante una metáfora o un conjunto de metáforas compartidas por el cliente y el equipo de desarrollo. Una metáfora es una historia compartida que describe cómo debería funcionar el sistema (conjunto de nombres que actúen como vocabulario para hablar sobre el dominio del problema, ayudando a la nomenclatura de clases y métodos del sistema).
- **Diseño simple**. Se debe diseñar la solución más simple que pueda funcionar y ser implementada en un momento determinado del proyecto.
- **Pruebas**. La producción de código está dirigida por las pruebas unitarias. Éstas son son establecidas por el cliente antes de escribirse el código y son ejecutadas constantemente ante cada modificación del sistema.

- **Refactorización (*Refactoring*)**. Es una actividad constante de reestructuración del código con el objetivo de remover duplicación de código, mejorar su legibilidad, simplificarlo y hacerlo más flexible para facilitar los posteriores cambios. Se mejora la estructura interna del código sin alterar su comportamiento externo [8].
- **Programación en parejas**. Toda la producción de código debe realizarse con trabajo en parejas de programadores. Esto conlleva ventajas implícitas (menor tasa de errores, mejor diseño, mayor satisfacción de los programadores, …).
- **Propiedad colectiva del código**. Cualquier programador puede cambiar cualquier parte del código en cualquier momento.
- **Integración continua**. Cada pieza de código es integrada en el sistema una vez que esté lista. Así, el sistema puede llegar a ser integrado y construido varias veces en un mismo día.
- **40 horas por semana**. Se debe trabajar un máximo de 40 horas por semana. No se trabajan horas extras en dos semanas seguidas. Si esto ocurre, probablemente está ocurriendo un problema que debe corregirse. El trabajo extra desmotiva al equipo.
- **Cliente in-situ**. El cliente tiene que estar presente y disponible todo el tiempo para el equipo. Éste es uno de los principales factores de éxito del proyecto XP. El cliente conduce constantemente el trabajo hacia lo que aportará mayor valor de negocio y los programadores pueden resolver de manera inmediata cualquier duda asociada. La comunicación oral es más efectiva que la escrita.
- **Estándares de programación**. XP enfatiza que la comunicación de los programadores es a través del código, con lo cual es indispensable que se sigan ciertos estándares de programación para mantener el código legible.

El mayor beneficio de las prácticas se consigue con su aplicación conjunta y equilibrada puesto que se apoyan unas en otras. Esto se ilustra en la Figura 1 (obtenida de [2]), donde una línea entre dos prácticas significa que las dos prácticas se refuerzan entre sí. La mayoría de las prácticas propuestas por XP no son novedosas sino que en alguna forma ya habían sido propuestas en ingeniería del software e incluso demostrado su valor en la práctica (ver [1] para un análisis histórico de ideas y prácticas que sirven como antecedentes a las utilizadas por las metodologías ágiles). El mérito de XP es integrarlas de una forma efectiva y complementarlas con otras ideas desde la perspectiva del negocio, los valores humanos y el trabajo en equipo.

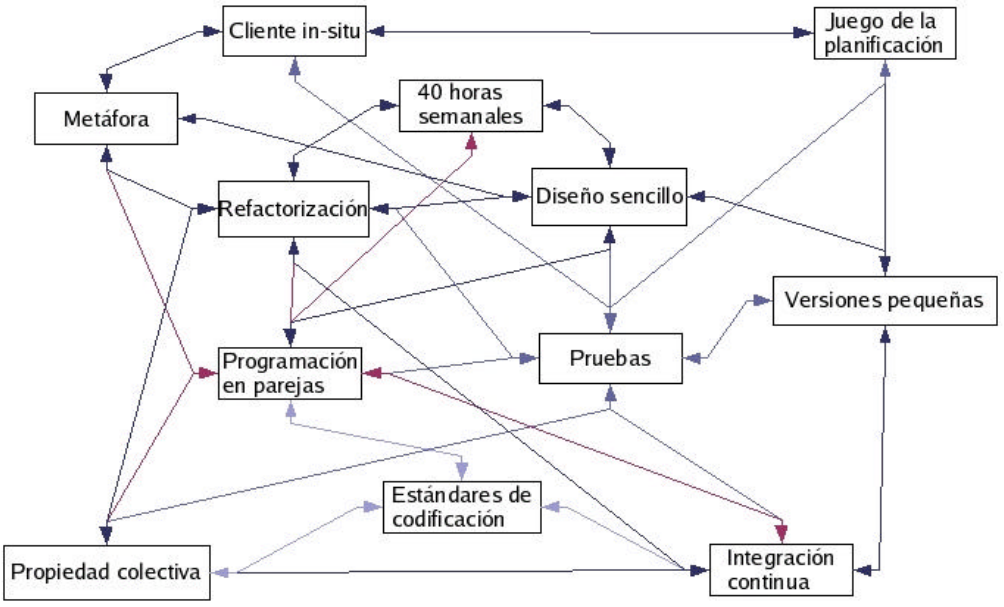


Figura 1. Las prácticas se refuerzan entre sí

3. OTRAS METODOLOGÍAS ÁGILES

Aunque los creadores e impulsores de las metodologías ágiles más populares han suscrito el manifiesto ágil y coinciden con los principios enunciados anteriormente, cada metodología tiene características propias y hace hincapié en algunos aspectos más específicos. A continuación se resumen otras metodologías ágiles. La mayoría de ellas ya estaban siendo utilizadas con éxito en proyectos reales pero les faltaba una mayor difusión y reconocimiento.

- **SCRUM**[5] [16]. Desarrollada por Ken Schwaber, Jeff Sutherland y Mike Beedle. Define un marco para la gestión de proyectos, que se ha utilizado con éxito durante los últimos 10 años. Está especialmente indicada para proyectos con un rápido cambio de requisitos. Sus principales características se pueden resumir en dos. El desarrollo de software se realiza mediante iteraciones, denominadas *sprints*, con una duración de 30 días. El resultado de cada *sprint* es un incremento ejecutable que se muestra al cliente. La segunda característica importante son las reuniones a lo largo proyecto, entre ellas destaca la reunión diaria de 15 minutos del equipo de desarrollo para coordinación e integración.

- **Crystal Methodologies**[6] [5]. Se trata de un conjunto de metodologías para el desarrollo de software caracterizadas por estar centradas en las personas que componen el equipo y la reducción al máximo del número de artefactos producidos. Han sido desarrolladas por Alistair Cockburn. El desarrollo de software se considera un juego cooperativo de invención y comunicación, limitado por los recursos a utilizar. El equipo de desarrollo es un factor clave, por lo que se deben invertir esfuerzos en mejorar sus habilidades y destrezas, así como tener políticas de trabajo en equipo definidas. Estas políticas dependerán del tamaño del equipo, estableciéndose una clasificación por colores, por ejemplo Crystal Clear (3 a 8 miembros) y Crystal Orange (25 a 50 miembros).

- **Dynamic Systems Development Method**[7] (**DSDM**) [17]. Define el marco para desarrollar un proceso de producción de software. Nace en 1994 con el objetivo de crear una metodología RAD unificada. Sus principales características son: es un proceso iterativo e incremental y el equipo de desarrollo y el usuario trabajan juntos. Propone cinco fases: estudio viabilidad, estudio del negocio, modelado funcional, diseño y construcción, y finalmente implementación. Las tres últimas son iterativas, además de existir realimentación a todas las fases.

- **Adaptive Software Development**[8] (**ASD**) [9]. Su impulsor es Jim Highsmith. Sus principales características son: iterativo, orientado a los componentes software más que a las tareas y tolerante a los cambios. El ciclo de vida que propone tiene tres fases esenciales: especulación, colaboración y aprendizaje. En la primera de ellas se inicia el proyecto y se planifican las características del software; en la segunda desarrollan las características y finalmente en la tercera se revisa su calidad, y se entrega al cliente. La revisión de los componentes sirve para aprender de los errores y volver a iniciar el ciclo de desarrollo.

- **Feature-Driven Development**[9] (**FDD**) [3]. Define un proceso iterativo que consta de 5 pasos. Las iteraciones son cortas (hasta 2 semanas). Se centra en las fases de diseño e implementación del sistema partiendo de una lista de características que debe reunir el software. Sus impulsores son Jeff De Luca y Peter Coad.

- **Lean Development**[10] (**LD**) [15]. Definida por Bob Charette's a partir de su experiencia en proyectos con la industria japonesa del automóvil en los años 80 y utilizada en numerosos proyectos de telecomunicaciones en Europa. En LD, los cambios se consideran riesgos, pero si se manejan adecuadamente se pueden convertir en oportunidades que mejoren la

[5] www.controlchaos.com
[6] www.crystalmethodologies.org
[7] www.dsdm.org
[8] www.adaptivesd.com
[9] www.featuredrivendevelopment.com
[10] www.poppendieck.com

productividad del cliente. Su principal característica es introducir un mecanismo para implementar dichos cambios.

4. CONCLUSIONES

No existe una metodología universal para hacer frente con éxito a cualquier proyecto de desarrollo de software. Toda metodología debe ser adaptada al contexto del proyecto (recursos técnicos y humanos, tiempo de desarrollo, tipo de sistema, etc. Históricamente, las metodologías tradicionales han intentado abordar la mayor cantidad de situaciones de contexto del proyecto, exigiendo un esfuerzo considerable para ser adaptadas, sobre todo en proyectos pequeños y con requisitos muy cambiantes. Las metodologías ágiles ofrecen una solución casi a medida para una gran cantidad de proyectos que tienen estas características. Una de las cualidades más destacables en una metodología ágil es su sencillez, tanto en su aprendizaje como en su aplicación, reduciéndose así los costos de implantación en un equipo de desarrollo. Esto ha llevado hacia un interés creciente en las metodologías ágiles. Sin embargo, hay que tener presente una serie de inconvenientes y restricciones para su aplicación, tales como: están dirigidas a equipos pequeños o medianos (Beck sugiere que el tamaño de los equipos se limite de 3 a 20 como máximo, otros dicen no más de 10 participantes), el entorno físico debe ser un ambiente que permita la comunicación y colaboración entre todos los miembros del equipo durante todo el tiempo, cualquier resistencia del cliente o del equipo de desarrollo hacia las prácticas y principios puede llevar al proceso al fracaso (el clima de trabajo, la colaboración y la relación contractual son claves), el uso de tecnologías que no tengan un ciclo rápido de realimentación o que no soporten fácilmente el cambio, etc.

Falta aún un cuerpo de conocimiento consensuado respecto de los aspectos teóricos y prácticos de la utilización de metodologías ágiles, así como una mayor consolidación de los resultados de aplicación. La actividad de investigación está orientada hacia líneas tales como: métricas y evaluación del proceso, herramientas específicas para apoyar prácticas ágiles, aspectos humanos y de trabajo en equipo. Entre estos esfuerzos destacan proyectos como NAME[11] (*Network for Agile Methodologies Experience*) en el cual hemos participado como nodo en España.

Aunque en la actualidad ya existen libros asociados a cada una de las metodologías ágiles existentes y también abundante información en internet, es XP la metodología que resalta por contar con la mayor cantidad de información disponible y es con diferencia la más popular.

BIBLIOGRAFIA

[1] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J. "Agile software development methods Review and analysis". VTT Publications. 2002.

[2] Beck, K.. "Extreme Programming Explained. Embrace Change", Pearson Education, 1999. Traducido al español como: "Una explicación de la programación extrema. Aceptar el cambio", Addison Wesley, 2000.

[3] Coad P., Lefebvre E., De Luca J. "Java Modeling In Color With UML: Enterprise Components and Process". Prentice Hall. 1999.

[4] Cockbun, A. "Agile Software Development". Addison-Wesley. 2001.

[5] Fowler, M., Beck, K., Brant, J. "Refactoring: Improving the Design of Existing Code". Addison-Wesley. 1999

[6] Highsmith J., Orr K. "Adaptive Software Development: A Collaborative Approach to Managing Complex Systems". Dorset House. 2000.

[7] Jeffries, R., Anderson, A., Hendrickson, C. "Extreme Programming Installed". Addison-Wesley. 2001

[8] Poppendieck M., Poppendieck T. "Lean Software Development: An Agile Toolkit for Software Development Managers". Addison Wesley. 2003.

[9] Schwaber K., Beedle M., Martin R.C. "Agile Software Development with SCRUM". Prentice Hall. 2001.

[10] Stapleton J. "Dsdm Dynamic Systems Development Method: The Method in Practice". Addison-Wesley. 1997.

[11] name.case.unibz.it/

Hacia un proceso metodológico dirigido por modelos para el desarrollo ágil de sistemas de información Web

Paloma Cáceres, Esperanza Marcos
Grupo de Investigación Kybele
Universidad Rey Juan Carlos
Madrid
{p.caceres, e.marcos}@escet.urjc.es

Resumen. En la actualidad existen multitud de plataformas de explotación para los sistemas de información Web, así como diferentes tecnologías de implementación para el desarrollo de dichos sistemas. Con el fin de garantizar una especificación independiente de la plataforma de explotación y de la tecnología de implementación, OMG propone realizar una arquitectura basada en modelos (MDA). En base a esta arquitectura, se está definiendo un marco metodológico basado en modelos para el desarrollo de sistemas de información Web, denominado MIDAS. En este trabajo presentamos el proceso metodológico dirigido por modelos, de dicho marco de trabajo, para desarrollar de forma ágil sistemas de información Web.

Palabras clave: Procesos ágiles, metodologías ágiles, metodologías de desarrollo.

1. Introducción

Hoy en día, el desarrollo de los sistemas de información Web (SIW) ha despertado un gran interés tanto a nivel empresarial como a nivel investigador y docente.

La necesidad de metodologías específicas para el desarrollo de SIW es también conocida (Fraternali, 2000). Las metodologías tradicionales no incorporan métodos específicos para contemplar determinadas características de estos sistemas (Lowe and Hall, 1999) y son demasiado pesadas y burocráticas puesto que exigen la generación de un gran volumen de documentación, impidiendo así un desarrollo ágil y rápido, Fowler (2001). Así han aparecido las metodologías y procesos denominados *ágiles*, que garantizan un proceso de desarrollo suficiente pero no excesivo.

Por otra parte, la creciente aparición de diferentes tecnologías y plataformas asociadas al desarrollo de sistemas de información, hace demasiado específico el modelado de un sistema. En este sentido, OMG ha propuesto la *Arquitectura Dirigida por Modelos* (MDA), (Miller y Mukerji, 2001) que garantiza la especificación completa de un sistema en base a modelos, independientes y específicos de una tecnología y plataforma.

Estos motivos nos han llevado a proponer un proceso software dirigido por modelos para el desarrollo ágil de SIW dentro del marco metodológico de MIDAS[1], presentado en Marcos et al. (2002).

El resto del artículo se organiza de la siguiente forma: la sección 2 hace una breve introducción a MDA; la sección 3 presenta el proceso metodológico de MIDAS y por último, las conclusiones y trabajos futuros se presentan en la sección 4.

[1] Este trabajo se ha llevado a cabo dentro de los proyectos: EDAD (07T/0056/2003 1) financiado por la Comunidad Autónoma de Madrid y DAWIS (TIC 2002-04050-C02-01) financiado por la Subdirección General de Proyectos de Investigación del MCyT (Ministerio de Ciencias y Tecnología) y por la Universidad Rey Juan Carlos (GC-2003-12).

2. Arquitectura dirigida por modelos

MDA (Miller and Mukerji, 2001) es un marco de trabajo para el desarrollo del software, definido por OMG (Object Management Group), que define una arquitectura orientada a modelos y unas guías de transformación entre los mismos para recoger las especificaciones de un sistema, donde los productos que se generan son modelos formales, que incluso pueden llegar a ser comprendidos por ordenadores.

MDA propone la realización de: Modelos independientes de computación (**CIM**) que son modelos del más alto nivel de abstracción que identifican el contexto del sistema. Modelos Independientes de la Plataforma (**PIM**) que proporcionan la especificación formal de la estructura y función del sistema, sin tener en cuenta aspectos técnicos e independientes de cualquier tecnología de implementación; Modelos Específicos de la Plataforma (**PSM**) que proporcionan modelos en términos de constructores de implementación que están disponibles en una tecnología específica.

Las reglas de transformación se aplican para transformar: **PIM en PIM** que van generalmente asociadas a los pasos que se suceden entre el modelado de la especificación, análisis y diseño; **PIM en PSM** que se realizan cuando el PIM está suficientemente refinado y se transforma en un modelo dependiente de la insfraestructura final de ejecución; un PIM se transforma en uno o varios PSM; **PSM en PSM** que son necesarios para la realización y despliegue de componentes; **PSM en PIM** que permiten la abstración de modelos a partir de implementaciones específicas de una plataforma y dependientes de una tecnología concreta.

3. Proceso metodológico de MIDAS

En esta sección se presenta el proceso metodológico de MIDAS dirigido por modelos para el desarrollo ágil de sistemas de información Web (SIW). En primer lugar y dado que uno de nuestros objetivos era la agilidad, se estudió la viabilidad de aplicar un proceso *ágil* para el desarrollo de SIW. Fue presentado en Cáceres y Marcos (2001) y se concluyó que era viable la aplicación de dichos procesos a los desarrollos Web. En segundo lugar, y debido a las ventajas que también aporta MDA, se decidió ver la posibilidad de integración entre las prácticas ágiles y las dirigidas por modelos. Surge entonces una propuesta de integración que se presenta en este trabajo, junto con el proceso metodológico propuesto en MIDAS.

➤ **Propuesta dirigida por modelos**

Al comienzo de nuestro trabajo, se apostó por el modelado conceptual como uno de los requisitos fundamentales a incluir en nuestro proceso metodológico puesto que permitía la representación de los requisitos y establecía el vínculo entre el espacio del problema -*lo que tratamos de resolver*- y el espacio de la solución -*cómo lo vamos a resolver*- (Sánchez y Pastor, 2001). Se definió un conjunto de modelos para el desarrollo de SIW, con dos objetivos concretos: El primero, que el modelado conceptual fuese independiente de cualquier tecnología, pero dirigido a entorno Web puesto que éste es nuestro ámbito de estudio. El segundo fue la definición de guías de transformación entre modelos. Se comenzó el estudio de MDA y se presentó una propuesta concreta de modelado basada en MDA para plataforma Web y tecnología XML y objeto-relacional en Cáceres et al. (2003).

➤ **Agilidad en el desarrollo dirigido por modelos**

Pero dado que seguíamos apostando por un proceso ágil, había que analizar la viabilidad de un proceso *ágil* y *dirigido por modelos*. En este sentido nos encontramos que, según puede apreciarse en la figura 1, los procesos ágiles y los procesos dirigidos por modelos no contemplan igualmente el espacio del problema (*qué es lo que hay que resolver*) y el espacio de la solución (*cómo resolverlo*), Weneger (2002). Nuestro área de interés, parte sombreada de la figura 1, se centró entonces en el acercamiento entre ambas propuestas, en base al estudio de cinco aspectos diferenciadores entre ambos procesos, indicados en la tabla 1 (Weneger, 2002).

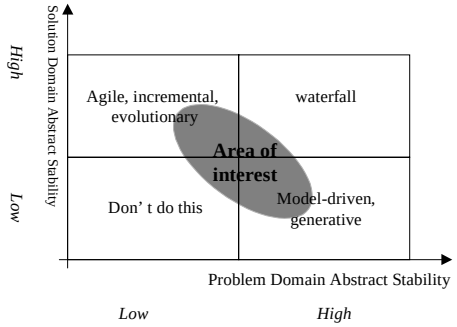


Fig. 1.- Relación entre el espacio del problema y de la solución

Aspectos	Agil	Dirigido por modelos
Personas	Alta prioridad; se facilita relación cliente-desarrollador	No prioritario; el modelo del espacio del problema es la base de la discusión entre cliente-desarrollador
Proceso	Prioridad media; incremental y evolutivo	Tiende al proceso en cascada, poco incremental
Tecnología	Baja prioridad; sólo cobra importancia al final.	Es relevante; se usa para la generación del software (usando un PSM)
Modelos	Artefacto secundario; se producen cuando es absolutamente necesario	Artefacto prioritario; fuente de la implementación
Software	Artefacto prioritario; es la única medida de progreso	Artefacto secundario; depende del espacio de la solución

Tabla 1. Aspectos diferenciadores entre los procesos ágiles y los procesos dirigidos por modelos

Finalmente propusimos lo siguiente:

Personas. Al igual que en los procesos ágiles, nosotros sí apostamos por darle una alta prioridad a la relación con el cliente. Aunque en los procesos dirigidos por modelos, el cliente entra a formar parte del proyecto a partir del modelo del espacio del problema, para nosotros es fundamental esa relación desde el principio: para establecer cuál es el espacio del problema, para definirlo y para utilizarlo posteriormente como base de la discusión entre el cliente-analista, entre el cliente-diseñador/arquitecto, entre el cliente-desarrollador. El cliente es parte fundamental durante todo el proceso, puesto que es la forma de garantizar que el producto final cumpla sus expectativas.

Proceso. Es una parte muy relevante dentro de nuestra propuesta. El modelo de proceso propuesto en MIDAS tiene mucho que ver con el de los procesos ágiles puesto que es un modelo iterativo, incremental, adaptativo y con prototipado, lejano por tanto del modelo en cascada indicado en los procesos dirigidos por modelos. El proceso iterativo e incremental aporta grandes ventajas puesto que permite la obtención de versiones del producto software antes de la entrega final del mismo, Jacobson et al. (2000). Ambler

(2003) confirma además, la posibilidad de realizar una implementación iterativa e incremental, a través del desarrollo ágil dirigido por modelos.

Tecnología. Nuestra propuesta en este aspecto se basa en tecnología XML y objeto-relacional (Cáceres et al., 2003). A diferencia de los procesos ágiles, este punto lo consideramos relevante, puesto que los PSM indicados por MDA, son específicos de una tecnología concreta; luego es obligatorio identificarla.

Modelos. Este elemento tiene una alta prioridad en nuestra propuesta. Los modelos son los artefactos que generamos y nuestra única documentación junto con el código. Aquí disentimos respecto a los procesos ágiles. Nosotros consideramos que el conocimiento acerca del sistema y del negocio no puede mantenerse exclusivamente en la mente de los desarrolladores y del cliente. Según Atkinson y Kühne (2003), es conveniente tanto para satisfacción del cliente como para el buen desarrollo del proyecto que se deje constancia de ello por escrito con una notación concisa; en nuestro caso se deja constancia a través de *modelos* en notación UML y UML extendido.

Software. El objetivo de cada iteración propuesta en MIDAS es la obtención de un prototipo o versión del producto software a través de ciclos cortos de desarrollo, con la finalidad de garantizar el progreso del *software*. Gracias a que el proceso es además incremental, el producto final se obtiene a través de versiones lo que permite entregas al cliente de forma rápida, previas a la versión final. Por lo tanto, sí que dotamos de una alta prioridad también a este aspecto, tanto a nivel de prototipo, como de versión no definitiva.

Hay que destacar en este subapartado que, según afirma Mellor et al. (2003), el hecho de tener identificado un conjunto de modelos específicos así como sus reglas de transformación (Cáceres et al., 2003), es también una forma ágil de desarrollo, puesto que cada modelo es autónomo, completo y se enlaza con otros modelos.

➤ Proceso metodológico de MIDAS

En MIDAS se ha combinado la arquitectura de MDA con una arquitectura de capas donde cada capa representa una vista del sistema (Kulkarni y Reddy, 2003). De esta forma tenemos una representación bidimensional de nuestro proceso metodológico (ver figura 2).

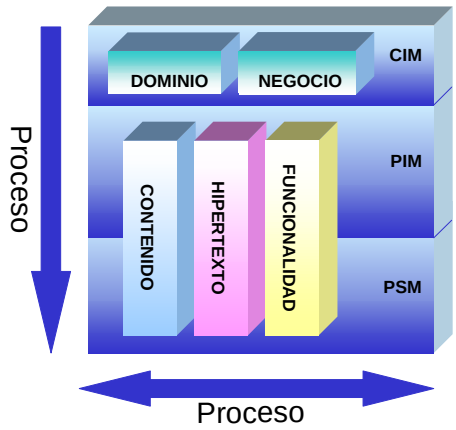


Fig. 2.- Dimensiones del proceso metodológico de MIDAS

La dimensión correspondiente al eje *x*, representa las diferentes vistas del SIW que en nuestra propuesta son la vista del hipertexto, la vista del contenido y la vista de la funcionalidad. La dimensión correspondiente al eje *y*, representa la parte independiente

de computación primero (CIM) y después, la parte independiente (PIM) y dependiente de tecnología (PSM).
La intersección de ambas dimensiones ha dado lugar a un conjunto de modelos que representan, a nivel de CIM el modelado conceptual del contexto o entorno del sistema, a través del modelado del dominio y del negocio; a nivel de PIM el modelado del hipertexto, del contenido y de la funcionalidad del sistema, desde el modelado conceptual hasta la etapa de diseño (excluido el diseño lógico); y a nivel de PSM representan también el modelado del hipertexto, del contenido y de la funcionalidad del sistema, desde la etapa de diseño lógico hasta la implementación.
El proceso de desarrollo de MIDAS, mostrado en la figura 3, comienza siempre en la definición de los CIM, y posteriormente evolucionará hacia los PIM y hasta los PSM. Sin embargo desde el punto de vista del eje *x*, y una vez completados los CIM, el proceso puede comenzar en diferentes puntos a nivel PIM. Puede comenzar en la vista de contenido o bien en la vista de funcionalidad con la definición del modelo conceptual de datos o de casos de uso, respectivamente. A continuación, las guías de transformación permiten pasar a definir, bien la vista de hipertexto a nivel de PIM, o bien continuar dentro de las mismas vistas para definir, ya a nivel de PSM, el modelado lógico. Esta decisión vendrá dada por las necesidades del cliente o bien por una priorización de requisitos realizada la etapa de especificación de requisitos.

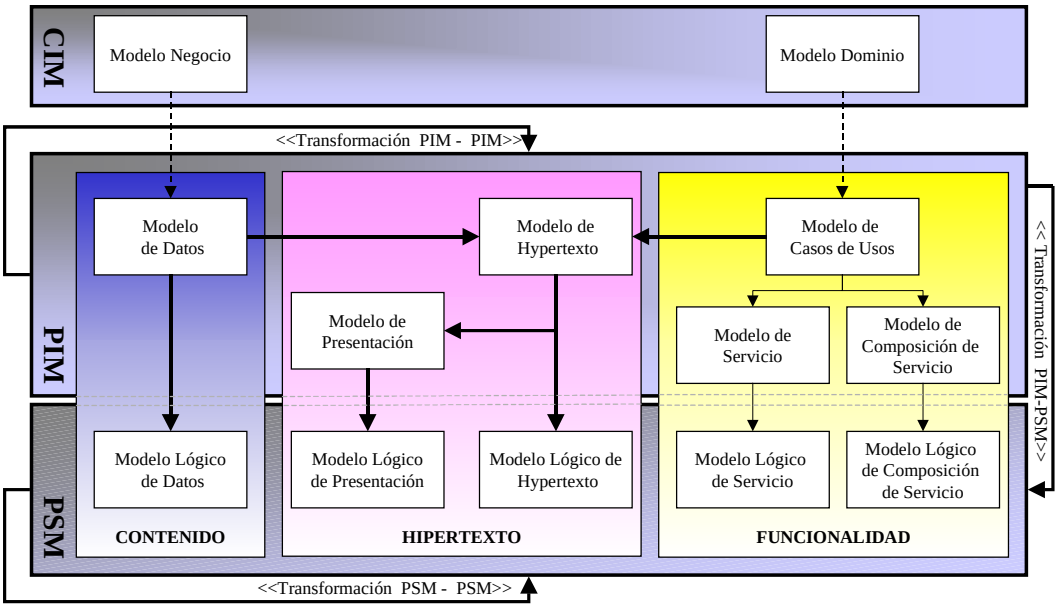


Fig. 3.- Proceso de desarrollo de MIDAS

4. Conclusiones y Trabajos Futuros

En este artículo se ha descrito el proceso metodológico de MIDAS, dirigido por modelos para el desarrollo ágil de sistemas de información Web. Este proceso metodológico combina además, la arquitectura de MDA con una arquitectura de capas, típica de las aplicaciones de negocio. Todo ello ha dado lugar a un conjunto específico de modelos y guías de transformación entre los mismos con la finalidad de realizar un desarrollo ágil de sistemas de información Web y que además se ha concretado para la tecnología XML y objeto-relacional.

Actualmente se están refinando las guías de transformación de los modelos relacionados con la lógica de negocio. Como trabajos futuros se planea automatizar la implementación de los modelos y las transformaciones entre ellos, por medio de una herramienta CASE.

Referencias

Atkinson, C., Kühne, T. Model-Driven Development: A Metamodeling Foundation. IEEE Software, Vol. 4, pp. 36-41, septiembre-octubre de 2003.

Ambler, S. (2003). Agile Model Driven Development is Good Enough. IEEE Software, Vol. 4, pp. 71-73, septiembre-octubre de 2003.

Booch, G. (2001). Developing the Future. Software Solutions. *Communications of ACM*, vol. 44 (3), pp. 119-121, March 2001.

Cáceres, P.; Marcos, E. (2001) Procesos ágiles para el desarrollo de aplicaciones Web. Taller de Web Engineering de las Jornadas de Ingeniería del Software y Bases de Datos de 2001. (JISBD2001). En http://www.dlsi.ua.es/webe01/articulos/s112.pdf

Cáceres, P.; Marcos, E.; Vela, B. A MDA-Based Approach for Web Information System Development. Workshop in Software Model Engineering (WiSME@UML'2003) in conjunction with UML Conference. Octubre, 2003. San Francisco, USA. Aceptado.

Fraternali, P. (2000). Tools and Approaches for Developing Data-Intensive Web Applications: a Survey. Retrieved July 2000 from the World Wide Web: http://toriisoft.com

Jacobson, I., Booch, G., Rumbaugh, J. (2000). El proceso unificado de desarrollo del software. Addison Wesley.

Lowe, D.; Hall, W. (1999). Hypermedia & the Web. An Engineering Approach. J. Wiley and Sons.

Fowler, M. (2001). The New Methodology. Retrieved May 2001 from http://www.martinfowler.com/articles/newMethodology.html.

Kulkarni, V., Reddy, S. (2003). Separation of Concerns in Model-Driven Development. IEEE Software, Vol. 4, pp. 64-69, septiembre-octubre de 2003.

Marcos, E., Cáceres, P., Vela, B. y Cavero, J.M. MIDAS/DB: a Methodological Framework for Web Database Design (DASWIS 2001). LNCS-2465. Springer Verlag. ISBN 3-540-44122-0. Septiembre, 2002.

Mellor, S.J., Clark, A.N., Futagami, T. (2003) Model-Driven Development. IEEE Software, Vol. 4, pp. 14-18, septiembre-octubre de 2003.

Miller, J. and Mukerji, J. (Eds). (2001) Model Driven Architecture. Document number ormsc/2001-07-01. Retrieved from: http://www.omg.com/mda, 2003.

Sánchez Díaz, J., Pastor López, O. (2001). Generación automática de prototipos de interfaz de usuario a partir de modelos de requisitos. Actas de las Jornadas de Ingeniería de Requisitos Aplicada (JIRA 2001), pp. 83-97, Sevilla (España), Junio de 2001.

Overmyer, S. P. (2000). What's Different about Requirements Engineering for Web Sites? Requirements Engineering, Vol. 5, pp. 62-65, Springer-Verlag, London.

Weneger, H. (2002). Agility in Model-Driven Software Development? Implications for Organization, Process and Architecture. Retrieved from http://www.softmetaware.com/oopsla2002/wenegerh.pdf

Mutación de casos de prueba de JUnit

María del Mar Jiménez, Macario Polo, José Luis Ruiz, Mario Piattini
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo de la Universidad, 4; 13071-Ciudad Real (Spain)
Mar.Jimenez@uclm.es

Resumen

En este artículo se presenta un método y una herramienta para generar u ejecutar de manera automática casos de prueba similares a los de JUnit, pero aplicando operadores de mutación.

1 Introducción.

Las pruebas mediante mutación fueron propuestas originalmente por Hamlet (1977) y DeMillo et al. (1978), y han sido desde entonces muy utilizadas. Dado un programa *P* que se va a probar, se genera un conjunto de copias de *P*, pero introduciendo en cada copia una pequeña alteración, normalmente sintáctica, mediante la aplicación de un "operador de mutación". A estas copias levemente alteradas se las llama "mutantes" de *P*.

De acuerdo con Offut et al. (1996), las pruebas mediante mutación comienzan generando un conjunto de casos de prueba que son pasados al programa original *P*; si la salida de éste es incorrecta para algún caso de prueba, se revisa *P* hasta que las salidas sean correctas. Cuando *P* responde correctamente, se ejecuta el conjunto de mutantes con todos los casos de prueba. Se dice que el mutante *m* vive cuando su salida es igual que la de *P* para todos los casos de prueba; si su salida es distinta para algún caso, entonces se dice que *m* es un mutante muerto. Un mutante puede permanecer vivo por varias causas: bien porque los casos de prueba no hayan ejercitado la instrucción mutada (en este caso, es preciso construir más casos de prueba hasta conseguir matarlo), bien porque el mutante sea *funcionalmente equivalente* al programa original (caso en el que es imposible encontrar un caso de prueba que lo mate, resultando además muy dificultoso detectar dicha equivalencia funcional debido al enorme número de mutantes que se generan y a la mayor o menor complejidad del programa).

La Figura 1 muestra en su lado izquierdo un sencillo programa Fortran (tomado de Offut et al., 1996) y dos mutantes, obtenidos el primero mediante la modificación del lado derecho de una asignación, y el segundo sustituyendo un operador de comparación por otro. Si *P* funciona correctamente para un conjunto grande de casos de prueba, tendremos una seguridad grande de que está bien construido (pero, como es bien sabido, nunca podremos garantizarlo al 100%); sin embargo, si no sólo *P* se comporta correctamente, sino que además los mutantes dan salidas diferentes a *P* para todos los casos de prueba, entonces tenemos aún mayor garantía de que *P* se está comportando correctamente en cada una de las instrucciones mutadas (con lo que, en cierto modo, nos aproximamos a la *resolución del problema irresoluble* de demostrar que un programa es totalmente correcto); además, al ofrecer cada m_i una salida distinta de la dada por *P*, podemos asegurar que se ha ejecutado la instrucción mutada en el programa original,

con lo que podemos conocer la cobertura alcanzada por los casos de prueba. Además, si *P* se comporta correctamente en la instrucción mutada y algún mutante ofrece una salida distinta, entonces sabemos que la sentencia es correcta para ese caso de prueba; si obtenemos resultados similares para otros mutantes, entonces tendremos mucha mayor confianza en la calidad de esa sentencia, de las instrucciones anteriores y posteriores y del programa completo.

P	m_1	m_2

`FUNCTION Min(I,J)`
`  Min=I`
`  IF (J .LT. I) Min=J`
`  RETURN`
	`FUNCTION Min(I,J)`
	`  Min=J`
	`  IF (J .LT. I) Min=J`
	`  RETURN`
		`FUNCTION Min(I,J)`
		`  Min=I`
		`  IF (J .GT. I) Min=J`
		`  RETURN`

Figura 1. Un programa y dos ejemplares de su conjunto de mutantes

La mutación, por tanto, sirve a dos objetivos: (1) proporciona un criterio de adecuación de las pruebas (es decir, de su calidad) y (2) permite detectar fallos en el programa que se está probando.

En este artículo se presenta una técnica de prueba totalmente novedosa, basada en la mutación de casos de prueba JUnit. Los casos son además generados de manera automática mediante una herramienta que extrae mediante Reflexión la estructura de la clase que se va a probar.

2 Mutación de casos de prueba JUnit

Por lo general, para aplicar mutación es preciso utilizar un generador de mutantes que debe comprender la gramática del lenguaje en el que está escrito *P*, debiendo además disponer del código fuente. Existen no obstante excepciones a este punto: Ghosh y Matur (2001), por ejemplo, aplican diferentes operadores de mutación a las operaciones ubicadas en las interfaces de componentes para probar la implementación de éstos: así, por ejemplo, dada una operación en un interfaz que toma dos parámetros *x* e *y*, ejecutan la funcionalidad ofrecida por el componente mediante su interfaz original pasando dos valores cualesquiera, y a continuación los intercambian.

En nuestro trabajo tampoco es preciso disponer de código fuente para aplicar mutación. Partiremos de una clase Java compilada en byte code (con extensión *.class)* de la que extraemos su conjunto de constructores y métodos mediante la API *Reflection* incluida en *java.lang.reflect*. A partir de este conjunto, generamos de manera automática un conjunto de "secuencias de prueba" o, por mantener la notación de Graham et al. (1999), *test scripts*, compuestas cada una por una llamada a uno de los constructores de la clase y una serie de llamadas a sus métodos. Cada *test script* se procesa para generar un conjunto de casos de prueba ejecutables, que consisten en la llamada al constructor y a los métodos contenidos en el test script, pero pasando ahora valores reales a los parámetros de sus operaciones. La Figura 2 muestra un ejemplo de esto: a partir de la clase *Triangulo* pueden generarse multitud de *test scripts* de diferentes longitudes (número de métodos), pero todas ellas consistentes en la llamada a un constructor y a uno o más métodos de los ofrecidos por la clase. A partir del *test script* y de un conjunto de valores de prueba que tenemos previamente almacenados para cada tipo de dato (para el tipo *int*, por ejemplo, podemos almacenar valores límite, como el –327268 o el +32767, el cero, valores próximos a cero y otros valores

aleatorios), generamos un conjunto grande de casos de prueba, pasando las combinaciones de los valores de prueba como parámetros en cada *test case*.

Clase que vamos a probar

```
public class Triangulo implements {
    public Triangulo(){ ... }
    public void setX(int x) { ... }
    public void setY(int y) { ... }
    public void setZ(int z) { ... }
    public void calcularTipo(){ ... }
}
```

Descripción del test script nº 284

```
//TS_284
//public Triangulo()
//public void setY(int x1)
//public void setZ(int x1)
//public void setX(int x1)
//public void calcularTipo()
```

Descripción del 83º test case del test script nº 284

```
Triangulo TS_284_83= new Triangulo();
TS_284_83.setY((int)12455);
TS_284_83.setZ((int)32768);
TS_284_83.setX((int)1);
TS_284_83.calcularTipo();
```

Descripción del 84º test case del test script nº 284

```
Triangulo TS_284_84= new Triangulo();
TS_284_84.setY((int)32768);
TS_284_84.setZ((int)32768);
TS_284_84.setX((int)1);
TS_284_84.calcularTipo();
```

Figura 2. Una clase, uno de sus *test scripts* y dos de los *test cases* generados

Los test case que generamos siguen la misma filosofía que los casos de prueba de JUnit, un entorno para automatizar la realización de pruebas de caja negra de programas Java, que permite el "Desarrollo dirigido por las pruebas", propio de las metodologías ágiles (en particular, de XP) (Beck, 2003). Para crear casos de prueba de JUnit, debe crearse una especialización de TestCase (Figura 3), y añadir a ésta una serie de métodos de prueba, cuyo nombre debe comenzar por test para que sean interpretados y ejecutados (mediante Reflexión) por la herramienta. Cada método añadido constituye realmente un caso de prueba, en el que, a grandes rasgos, se crea manualmente el objeto (resultado) esperado, y mediante llamadas a métodos de la clase que se está probando el objeto (resultado) obtenido. Entonces, ambos objetos son comparados utilizando una serie de métodos assert que están definidos en la clase Assert, de la que también hereda la clase de prueba que hemos construido.

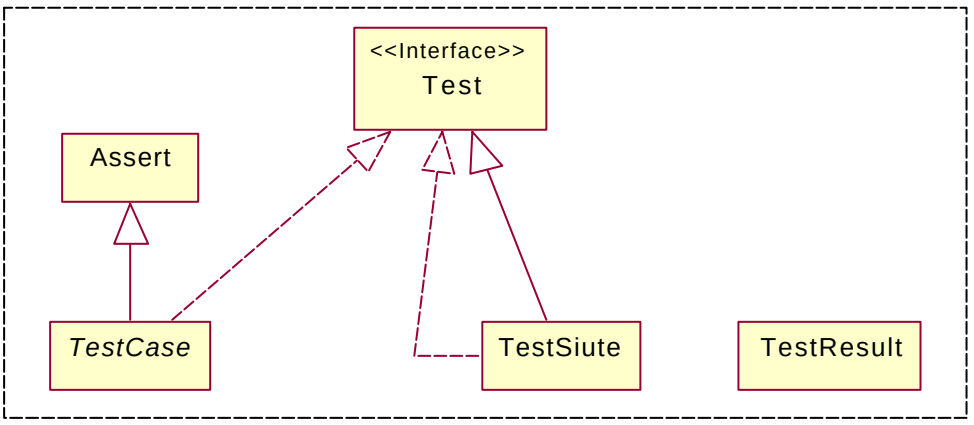


Figura 3. Clases proporcionadas por JUnit para la automatización de la fase de pruebas

En nuestro caso, dada una clase que queremos probar, creamos una clase de prueba a la que añadimos de manera automática un conjunto de métodos de prueba del estilo del mostrado en la Figura 4.

```
public Triangulo testTS_284_23() throws Exception {
       Triangulo TS_284_23= new source.Triangulo();
       TS_284_23.setY((int)12455);
       TS_284_23.setZ((int)32768);
       TS_284_23.setX((int)12455);
       TS_284_23.calcularTipo();
       return TS_284_23;
}
```

Figura 4. Uno de los casos de prueba generados, contenido en la clase *TestTriangulo.java*

Como se observa en la figura, dentro de cada método creamos un objeto (*TS_284_3*) de la clase que estamos probando (*Triangulo)*, sobre la que ejecutamos una secuencia de sus operaciones, instancia que es finalmente devuelta por el método.

La herramienta que genera estos casos de manera automática genera también mutantes de los casos de prueba. Así, podemos aplicar el operador *eliminación de método* (que añade un símbolo de comentario antes de la llamada a un método) u otro de la Tabla 1, en cada una de las llamadas a métodos que realizamos en el ejemplo de la Figura 4, obteniendo los mutantes siguientes:

```
public Triangulo testTS_284_23_m0()
 throws Exception {
    Triangulo TS_284_23= new Triangulo();
    // TS_284_23.setY((int)12455);
    TS_284_23.setZ((int)32768);
    TS_284_23.setX((int)12455);
    TS_284_23.calcularTipo();
    return TS_284_23;
}
```

```
public Triangulo testTS_284_23_m1()
 throws Exception {
    Triangulo TS_284_23= new Triangulo();
    TS_284_23.setY((int)12455);
    // TS_284_23.setZ((int)32768);
    TS_284_23.setX((int)12455);
    TS_284_23.calcularTipo();
    return TS_284_23;
}
```

```
public Triangulo testTS_284_23_m2()
 throws Exception {
    Triangulo TS_284_23= new Triangulo();
    TS_284_23.setY((int)12455);
    TS_284_23.setZ((int)32768);
    // TS_284_23.setX((int)12455);
    TS_284_23.calcularTipo();
    return TS_284_23;
}
```

```
public Triangulo testTS_284_23_m3()
 throws Exception {
    Triangulo TS_284_23= new Triangulo();
    TS_284_23.setY((int)12455);
    TS_284_23.setZ((int)32768);
    TS_284_23.setX((int)12455);
    // TS_284_23.calcularTipo();
    return TS_284_23;
}
```

Figura 5. Algunos de los mutantes generados para el caso de la Figura 4

Identificador	Nombre	Operación
m0	AltOrd	Altera el orden de dos métodos
m1	CambConst	Cambia el constructor
m2	EliMet	Comenta un método
m3	AñaMet	Añade un método
m4	CambPara	Intercambia dos parámetros del mismo tipo
m5	CambValPara	Cambia el valor de un parámetro

Tabla 1. Operadores de mutación para casos de prueba

Los mutantes generados con este operador son, en principio, casos de prueba previamente generados, por lo que estamos repitiendo ejecuciones, sin embargo, lo que nos interesa es ejecutar versiones mutadas del caso de prueba para observar y comparar su comportamiento.

Como se observa, tanto el método de prueba original como los mutantes devuelven un objeto de la misma clase, pero cada uno ha sido construido aplicándole un conjunto diferente de operaciones. Si, al igual que ocurre en mutación tradicional, comprobamos que el método original (Figura 4) ofrece el resultado correcto, podemos ejecutar los mismos casos de prueba sobre los mutantes con el fin de observar la salida de éstos.

Entonces, comparamos el resultado ofrecido por el método original con el que ofrece cada uno de los métodos de prueba. Esto se realiza también mediante la ejecución de otro método, generado también de forma automática, que es el ofrecido en la siguiente figura:

```
public void comprobarTS_284_23() {
    Triangulo TS_284_23=null;
    Triangulo TS_284_23_m0=null;
    Triangulo TS_284_23_m1=null;
    Triangulo TS_284_23_m2=null;
    Triangulo TS_284_23_m3=null;
```

Inicialización de los objetos

```
    try { TS_284_23= testTS_284_23(); }
    catch (Exception e) {}
    try { TS_284_23_m0= testTS_284_23_m0(); }
    catch (Exception e) {}
    try { TS_284_23_m1= testTS_284_23_m1(); }
    catch (Exception e) {}
    try { TS_284_23_m2= testTS_284_23_m2(); }
    catch (Exception e) {}
    try { TS_284_23_m3= testTS_284_23_m3(); }
    catch (Exception e) {}
```

Asignación de los objetos mediante llamadas al método original y a sus mutantes

En las siguientes líneas se compara el objeto "bueno"(el obtenido con la versión no mutada del test case: testTS_284_23()) con los obtenidos mediante mutación, actualizándose las variables que cuentan el número de vivos y muertos

```
    if (TS_284_23==null && TS_284_23_m0==null) mVivos++;
    else if ((TS_284_23==null && TS_284_23_m0!=null) || !TS_284_23.equals(TS_284_23_m0))
        mMuertos++;
    else mVivos++;
    if (TS_284_23==null && TS_284_23_m1==null)
        mVivos++;
    else if ((TS_284_23==null && TS_284_23_m1!=null) || !TS_284_23.equals(TS_284_23_m1))
        mMuertos++;
    else mVivos++;
    if (TS_284_23==null && TS_284_23_m2==null)
        mVivos++;
    else if ((TS_284_23==null && TS_284_23_m2!=null) || !TS_284_23.equals(TS_284_23_m2))
        mMuertos++;
    else mVivos++;
    if (TS_284_23==null && TS_284_23_m3==null)
        mVivos++;
    else if ((TS_284_23==null && TS_284_23_m3!=null) || !TS_284_23.equals(TS_284_23_m3))
        mMuertos++;
    else mVivos++;
```

Por último, se guardan los resultados de forma tabulada en un fichero de texto

```
    String cab="TS_284_23;TS_284_23_m0;TS_284_23_m1;TS_284_23_m2;TS_284_23_m3\n";
    String lin=(TS_284_23==null ? "null;" : TS_284_23.toString() + ";") +
        (TS_284_23_m0==null ? "null;" : TS_284_23_m0.toString() +";") +
        (TS_284_23_m1==null ? "null;" : TS_284_23_m1.toString() +";") +
        (TS_284_23_m2==null ? "null;" : TS_284_23_m2.toString() +";") +
        (TS_284_23_m3==null ? "null\n" : TS_284_23_m3.toString() +"\n");
    try { mF.write(cab.getBytes()); mF.write(lin.getBytes()); } catch (Exception e) {}
}
```

Figura 6. Método generado para comparar los objetos creados por el método original y sus mutantes

3 *testOO*: la herramienta de generación automática de casos de prueba

testOO es una herramienta que extrae mediante Reflexión el conjunto de operaciones de una clase Java ya compilada, y que genera una clase que contiene una lista de métodos *test*, consistentes cada uno en la construcción de un objeto y una secuencia de llamadas a sus métodos (como el mostrado en la Figura 4), un conjunto de mutantes para cada método de prueba (Figura 5), varios métodos para comparar los resultados obtenidos por los métodos "buenos" (no mutados) y los mutantes (Figura 6), así como una función *main* en la que se llama a los distintos métodos de comparación (Tabla 1). Ha sido desarrollada por José Luis Ruiz, coautor de este artículo.

```
        public static void main (String [] args){
                TestTriangulos2 tc=new TestTriangulos2();
                tc.comprobarTS_284_1();
                tc.comprobarTS_284_2();
                ...
                tc.comprobarTS_284_26();
                tc.comprobarTS_284_27();
                String s="Muertos: " + tc.mMuertos + "; Vivos: " + tc.mVivos;
                try { tc.mF.write(s.getBytes()); tc.mF.close(); } catch (Exception e) {}
        }
}
```

Figura 7. Función *main* que ejecuta y muestra los resultados de la prueba, añadida a la clase de prueba

El lado izquierdo de la siguiente figura muestra la ventana inicial de *testOO*, a partir de la cual se carga la clase compilada (también puede cargarse una clase de un modelo Rational Rose con el mismo objetivo) y se muestra en un árbol su lista de miembros. El usuario selecciona la longitud de los *test cases* que desea generar. *testOO* genera todas las secuencias posibles de llamadas (*test scripts*) de longitudes desde 1 hasta la especificada, y las muestra en la ventana que se muestra en el lado derecho. Obviamente, se genera un número muy grande de *test scripts* que no ocurrirán en la ejecución real de la clase (como la mostrada en la figura), por lo que el usuario puede eliminar manualmente las que no desee en la ventana que aparece a la derecha de la Figura 8 (Doong y Frankl, 1994, proponen un método de filtrado de *test scripts* mucho más elegante, que modela las secuencias de llamadas válidas mediante expresiones regulares; estamos trabajando en un método similar, que acepte o rechace secuencias según el diagrama de estados que describe la clase que se está probando).

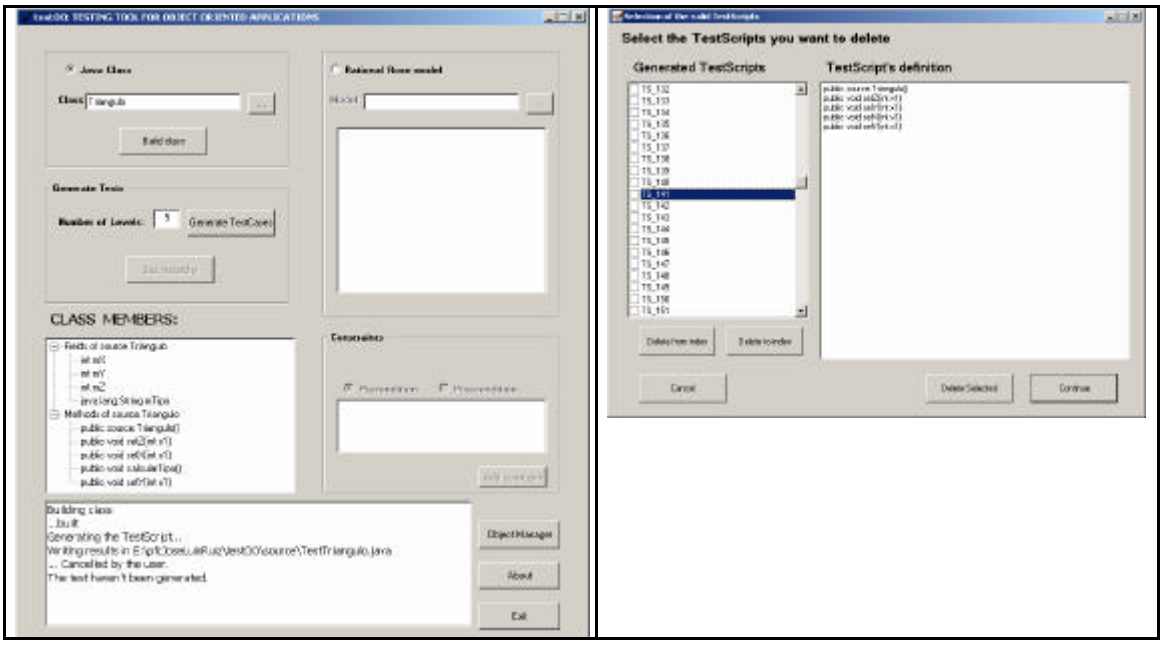


Figura 8. Aspecto de la herramienta testOO

Tras pulsar el botón *Continue* en la ventana del lado derecho de la figura anterior, la herramienta genera un fichero de prueba que contiene el código que ejecutará las pruebas. Para ello, toma los *test scripts* aceptados por el usuario y, para cada uno, genera un conjunto de *test cases* ejecutables dando a los parámetros valores reales (recuérdese el ejemplo mostrado en la Figura 2). Cuando el tipo de los parámetros es primitivo no hay problema en la asignación de los valores; cuando el tipo de alguno de ellos es "complejo" (por ejemplo: en la clase *Tarjeta* de una aplicación bancaria hay una operación *transferir(Cuenta destino, double importe, String concepto)*, en donde *Cuenta* es una clase de la propia aplicación), el objeto usado para dar valor al parámetro debe existir previamente, haber sido serializado y haber sido salvado en un directorio específico en el que *testOO* busca valores de prueba.

La herramienta incluye un editor/visor manual de objetos, que permite crear y serializar instancias. En la Figura 8, el usuario está creando una instancia de *Tarjeta*, que tiene dos campos de tipo *String* (mNumero y mTitular) y otro de tipo *Account*. Para dar valor a los dos primeros, el usuario utiliza la caja de texto que hemos resaltado con una elipse; el tercer campo no es asignable de esta manera, sino que debe asignarse mediante un objeto que ya debe estar creado y guardado en disco, lo que se hace con la caja de texto de debajo, que muestra un cuadro de diálogo para que el usuario navegue y cargue el objeto que quiere asignar. La creación manual de objetos es desde luego bastante costosa, pero los objetos quedan archivados para ser reutilizados cuando sea necesario; esto se consigue utilizando la serialización automática, que me permite guardar objetos en tiempo de ejecución.

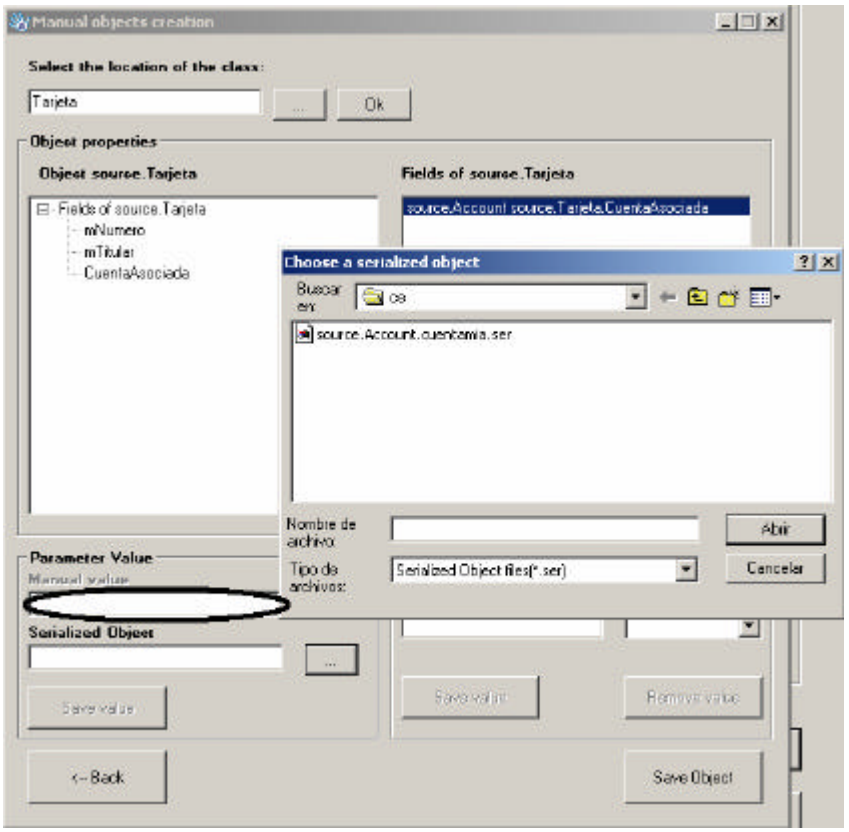


Figura 9. Editor manual de objetos.

4 Conclusiones y trabajo futuro

La técnica y herramienta presentadas ayudan al ingeniero de pruebas a generar de manera automática casos de prueba adecuados para probar programas Java. El coste de los algoritmos de generación es muy elevado, dependiendo fundamentalmente del número de *test scripts* aceptados por el usuario y del número de valores de prueba de cada tipo de dato, por lo que estamos trabajando con ahínco en este sentido.

Igualmente, tenemos pendiente la realización de experimentos (que será sencilla, gracias a la gran cantidad de código abierto existente) para comprobar la validez de la estrategia de generación de casos de prueba y la utilidad del método. Por último, y tras haber comprobado las características de Reflexión ofrecidas por la plataforma .NET. Con ello ampliaremos las posibilidades de la herramienta para realizar pruebas no sólo sobre *byte code* de Java, sino para poder realizar pruebas a partir de la estructura de las clases contenidas en una .dll, un .exe o un .class, adaptaremos *testOO* para que pueda funcionar en esta plataforma.

5 Agradecimientos

Este trabajo está parcialmente financiado por el proyecto MÁS (Mantenimiento Ágil del Software), TIC2003-02737-C02-02, Ministerio de Ciencia y Tecnología.

6 Referencias

Beck K. (2003). Test-drive development: by example. Addison-Wesley. Boston, MA, EE.UU.

DeMillo RA, Lipton RJ y Sayward FG. (1978). Hints on test data selection: help for the practicing programmer. *IEEE Computer*, 11(4), 34-41.

Doong RK. y Frankl PG. (1994). The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101-130.

Fewster M y Graham D. (1999). Software Test Automation. Addison-Wesley, ACM Press Book. Boston, MA, EE.UU.

Ghosh S y Mathur AP. (2001). Interface mutation. *Software Testing, Verification and Reliability*, 11, 227-247.

Hamlet, RG. (1977). Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4).

Offut AJ, Rothermel G, Untch RH y Zapf C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methdology*, 5(2), 99-118.

Experiencias de formación en metodologías ágiles

Patricio Letelier, José Hilario Canós, Mª Carmen Penadés y Juan Sánchez
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{letelier, jhcanos, mpenades, jsanchez}@dsic.upv.es

Resumen

Las metodologías ágiles están acaparando gran interés en la industria del software generando una clara necesidad de formación en este enfoque. El término ágil está estrechamente asociado a un conjunto de ideas pragmáticas para la producción de software, con un marcado énfasis en los aspectos humanos del trabajo en equipo. Por estas características, la formación en metodologías ágiles supone la búsqueda de estrategias innovadoras que permitan motivar al alumno y escenificar los aspectos claves de la filosofía que conlleva una metodología ágil. En este trabajo se describe nuestra experiencia de formación en metodologías ágiles realizada en los dos últimos años con alumnos de informática en la Universidad Politécnica de Valencia. Se explica el esquema de trabajo establecido, fruto de una serie de refinamientos, incluyendo el método docente y el método de evaluación aplicados.

1. Introducción

Es indudable el interés que han despertado las metodologías ágiles para el desarrollo de software. Después de ser vistas en un comienzo como una moda o una mera corriente radical, han pasado a ser un atractivo medio para mejorar el proceso de producción de software, con bastante afinidad respecto de las necesidades actuales de muchos equipos de desarrollo en la industria del software. Este interés industrial ha ido acompañado del reconocimiento en entornos académicos y de investigación, demostrado por el espacio conseguido en conferencias internacionales y revistas de prestigio. Acompañado de esta situación surge la necesidad de contar con estrategias de formación en metodologías ágiles. Sin embargo, los esquemas tradicionales utilizados para llevar a cabo la formación en notaciones de modelado y en herramientas no son apropiados, principalmente porque la formación en una metodología debe orientarse a la aplicación efectiva de los conocimientos en notaciones y herramientas, enfatizando el marco de proyecto de ingeniería y el trabajo en equipo. Los trabajos realizados en este ámbito son escasos y suelen estar más orientados a la evaluación de prácticas ágiles mediante experimentos.

El objetivo de este trabajo es presentar una estrategia de formación en metodologías ágiles y los resultados de nuestra experiencia en su aplicación, llevada a cabo durante los años académicos 2002-2003 y 2003-2004 en la Escuela Técnica Superior de Informática Aplicada (ETSIA) y en la Facultad de Informática (FI) de la Universidad Politécnica de Valencia (UPV). Nuestra iniciativa se enmarca dentro del contexto de programas de ayuda a la mejora de la enseñanza del Proyecto EUROPA [6]. Las asignaturas objeto de la experiencia son: "El proceso del software" (PSW) y "Laboratorio de Desarrollo de sistemas de información" (LDS), ambas optativas de tercer año en la intensificación en Ingeniería del Software de la ETSIA, y "Laboratorio de Sistemas de Información" (LSI), optativa de quinto curso en la FI. El trabajo aquí presentado es la continuación natural de otros previos de los autores [3, 4, 5].

Debido a que eXtreme Programming (XP) [1] es la metodología ágil más popular y que cuenta con abundante información en libros e internet decidimos centrar en ella nuestro estudio.

El resto de este artículo está organizado como se describe a continuación. En la sección 2 se presenta el contexto de aplicación donde se está realizando la formación en metodologías ágiles. En la sección 3 se describen el método docente y de evaluación utilizados. Finalmente en la sección 4 se establecen las conclusiones y el trabajo futuro.

2. Contexto de aplicación

Dado que la enseñanza de una metodología exige que el alumno tenga cierta madurez en cuanto a conocimientos y experiencia en construcción de software, decidimos abrir un espacio a las metodologías ágiles en los contenidos de los últimos cursos, tanto en la titulación de Ingeniero Técnico en Informática (en la ETSIA) como en Ingeniero en Informática (en la FI). Comenzamos el período 2002-2003 en la FI con la asignatura LSI. Este segundo año de aplicación hemos definido también un esquema para la ETSIA, aprovechando la implantación del nuevo plan de estudios que incluía la Intensificación Ingeniería del Software, a la cual pertenecen las asignaturas PSW y LDS.

Las clases de laboratorio se imparten en un aula que dispone de 20 puestos de trabajo acondicionados para 40 alumnos. Las clases de teoría y problemas se realizan en un aula tradicional.

LSI es una asignatura optativa de quinto año con una asignación de 6 créditos, todos ellos son créditos de laboratorio. La matrícula en LSI ha ido en aumento en los últimos años hasta llegar actualmente a copar el máximo establecido de 60 alumnos. LSI viene impartiéndose desde el período 1997-1998 y sus contenidos han evolucionado desde prácticas con notaciones de modelado y herramientas CASE hacia el trabajo en equipo para desarrollar un sistema de información. Hasta hace tres años, sólo utilizábamos Rational Unified Process (RUP) [2] como metodología en LSI. Al introducir XP en LSI nos pareció apropiado mantener también RUP para ofrecer una visión más global en el ámbito de las metodologías. Este año se han definido 8 equipos de trabajo de 6 a 8 integrantes. Así, 4 equipos utilizan RUP y los otros 4 trabajan con XP. Cada alumno asiste a dos sesiones semanales, de 2 horas cada una.

PSW es una asignatura de tercer año de la ETSIA que cuenta con 4.5 créditos de teoría y problemas, y 1.5 de laboratorio. PSW se imparte en el primer cuatrimestre. En los contenidos teóricos, PSW aborda los modelos de proceso para desarrollo de software, metodologías (XP, RUP, Métrica) y estándares en ingeniería de software. En su parte práctica se realiza un caso de estudio aplicando el "Juego de la Planificación" de XP.

LDS se impartirá en el segundo cuatrimestre de tercer año en la ETSIA y dispone de 1.5 créditos de teoría y problemas, y 4.5 créditos de laboratorio. LDS estará dedicada íntegramente a la realización de un proyecto de desarrollo de software trabajando en equipos y utilizando XP como metodología.

Hay que considerar que en tercer año en la ETSIA los alumnos en otras asignaturas comienzan a trabajar con el modelo orientado a objetos y aprenden modelado conceptual de bases de datos. Con lo cual, para PSW y LDS no se hacía recomendable utilizar una metodología tradicional, al menos en la parte práctica, puesto que se presentarían problemas de sincronización difíciles de resolver respecto de los conocimientos y madurez necesaria en notaciones y herramientas. Así, el esquema que hemos depurado en LSI lo aprovechamos y adaptamos para la ETSIA distribuyéndolo en parte en PSW (la parte teórica de metodologías ágiles y un caso práctico aplicando el "Juego de la Planificación") y en LDS (el formato de proyecto de desarrollo trabajando en equipos).

3. Método docente y de evaluación

En esta sección nos centraremos en describir el esquema que hemos establecido en LSI para la enseñanza de metodologías ágiles, en particular XP.

LSI se imparte por dos profesores, de los cuales uno desempeña el rol de cliente y el otro de entrenador para todos los equipos XP. Cada equipo XP tiene un jefe, un *tester/tracker* y entre 4 a 6 programadores. La composición de los equipos y la asignación de los roles se efectúa de forma aleatoria.

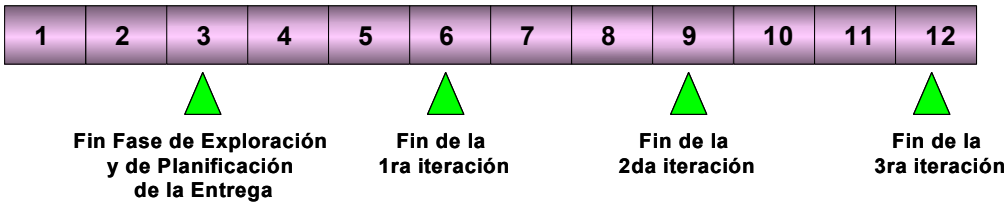


Figura 1: Hitos en la planificación de la entrega

En la Figura 1 se muestra la planificación global de la entrega, preestablecida de acuerdo con las restricciones temporales de las asignaturas LSI y LDS. En base a las 12 semanas con que suele contar un cuatrimestre (podrían ser un par más que se dejarían para otras actividades después del término del proyecto), se definen 4 iteraciones de 3 semanas cada una. La primera cubre las fases XP de Exploración y de Planificación de la Entrega. Durante esta primera iteración en LSI se tiene la oportunidad de introducir RUP y XP de forma muy breve, con lo cual el trabajo del entrenador es fundamental sobre todo al comienzo del proyecto. En LDS no se presenta este inconveniente puesto que la introducción a metodologías ágiles y XP, junto con una práctica del "Juego de la Planificación", se imparten en el primer cuatrimestre en PSW.

Al término de cada iteración los equipos realizan una presentación del resultado obtenido, incluyendo una demo del producto y comentarios de las incidencias en la planificación y otras aparecidas durante el trabajo de la iteración. Se hace hincapié en la detección (encargada al *tracker*) y resolución inmediata de desviaciones en la planificación mediante negociación con el cliente.

Durante todo el desarrollo del proyecto se dedica una sesión semanal exclusiva al trabajo en equipo en el laboratorio. En estas sesiones está presente el cliente. La otra sesión semanal se utiliza para las presentaciones de seguimiento y actividades complementarias trabajando sólo con el entrenador. Además, tanto el cliente como el entrenador están disponibles para los equipos en los horarios establecidos para tutorías u otros que de común acuerdo puedan establecerse. Los equipos se reúnen según sus disponibilidades. Sin embargo, para conseguir la aplicación del equivalente a la práctica de las "40 horas semanales" se estableció un límite de 12 horas de trabajo semanal para cada alumno, incluyendo las 4 horas de trabajo en laboratorio.

Los artefactos con los cuales se trabaja son: Historias de Usuario, Tareas, Casos de Prueba, Plan de la Entrega y Código. Además se deja como opcional el utilizar cualquier técnica de modelado. Normalmente los equipos al menos utilizan un modelo lógico relacional para el diseño de la base de datos.

En la página web de la asignatura (http://www.dsic.upv.es/asignaturas/facultad/lsi/), en el apartado "contenidos", puede verse el detalle de las actividades que se realizan, el método docente para cada una de ellas, el material de apoyo proporcionado al alumno y el tipo de evaluación.

En LSI se aplica un sistema de evaluación continua. Coincidiendo con cada presentación de seguimiento del proyecto se otorgan 3 calificaciones a cada integrante del equipo. El cliente otorga una nota igual para todos los integrantes del equipo basándose en los objetivos establecidos en el plan para el producto, así como en la negociación para resolver las incidencias. El entrenador comunica al jefe un número que debe ser repartido entre los integrantes del equipo; dicho número corresponde a una nota global multiplicada por el número de integrantes. Esta nota global está basada en la presentación de seguimiento, en la dinámica de colaboración que muestre el equipo y en a constatación de cómo se están aplicando las prácticas de XP. Finalmente, cada integrante debe otorgar una calificación al resto de sus compañeros de equipo; estas calificaciones se promedian para dar como resultado una tercera evaluación. Además de estas 12 evaluaciones se realizan otras correspondientes a actividades de apoyo o fuera del contexto del proyecto. Todas estas calificaciones tienen un peso de 80% sobre la nota final. Por último, se realiza un examen escrito que se pondera con un 20%.

En PSW el trabajo de laboratorio se evalúa de acuerdo con una presentación del resultado del la exploración y de la planificación de la entrega. Los equipos deben presentar las Historias de Usuario, Plan de la Entrega, Tareas para la primera iteración y prototipos desarrollados. Este caso de estudio se pondera con un 30% de la nota final. El otro 70% corresponde al examen que cubre los contenidos teóricos de la asignatura.

4. Conclusiones y trabajo futuro

Los alumnos han acogido muy favorablemente la estrategia de trabajo implantada; prueba de ello es su participación, la evaluación positiva que hacen de la asignatura en las encuestas, la masiva asistencia a clases y a tutorías, y el considerable incremento en la matrícula (hasta agotar el cupo máximo). En este sentido la experiencia ha sido gratificante, compensando el esfuerzo adicional que ha supuesto en la dedicación de los profesores.

En cuanto a la recreación eficaz de una situación real de proyecto, pensamos que hay algunos aspectos claves que han contribuido positivamente, entre los que destacaron: a) se cuenta con dos instructores que realizan de forma separada el rol de cliente y el de entrenador; b) se efectúan presentaciones de seguimiento del proyecto y se comenta desde distintas perspectivas (como cliente y como entrenador) el trabajo realizado; c) la utilización de un método de evaluación diversificado en el cual cada integrante tiene una evaluación como parte del equipo desde la perspectiva del cliente, como integrante del equipo evaluado públicamente por el jefe, y como participante, evaluado secretamente por sus compañeros de equipo; y d) la elección de un buen caso de estudio en el cual el cliente pueda desenvolverse con naturalidad y ofrecer material de apoyo como por ejemplo documentos con datos utilizados por el sistema actual. Con respecto a esto último hemos obtenido los mejores resultados trabajando con un sistema de información que existía y había sido desarrollado por el profesor que desempeñaba el rol de cliente.

En XP se han presentado conflictos entre miembros del equipo. Esto tiene su clara explicación en el hecho que los diferentes ritmos de trabajo y capacidades quedan más rápidamente en evidencia en el esquema de trabajo XP, que exige una mayor interacción, participación y protagonismo de los integrantes. Sin embargo, en todos estos casos la situación se apaciguó o al menos se llegaron a

acuerdos prácticos de cara a los objetivos. Esto le dio una perspectiva de realismo no esperada al caso de estudio y como tal fue asumida y aprovechada como experiencia.

En LSI al principio se tiene un gran inconveniente: los alumnos no conocen la metodología y no se dispone de mucho tiempo para dedicar sesiones a desarrollar un ejemplo guiado. Para solventar en parte este problema se realiza una presentación introductoria de métodos ágiles y XP, describiendo cada una de sus prácticas y aportando guías para que se pueda comenzar a trabajar. Sin embargo, el comienzo del proyecto conlleva una carga considerable de trabajo para el entrenador. De manera similar, el cliente durante las fases de exploración y de planificación tiene alta carga de trabajo, puesto que a los alumnos no se les entrega un enunciado del problema, sino que deben comenzar desde cero a establecer los requisitos.

Con las asignaturas PSW y LDS se ha comenzado este año una experiencia similar a la implantada en LSI. En este caso las condiciones parecen incluso más apropiadas para la incorporación de metodologías ágiles. Los alumnos en tercer curso de la ETSIA comienzan a aprender notaciones de modelado (E-R para diseño de bases de datos y UML en orientación a objetos). En segundo curso sólo llegan a ver el enfoque estructurado (DFDs y Diagramas de Estructura) y diseño lógico relacional. Así, la incorporación de una metodología tradicional tendría muchos inconvenientes para ser exitosa. Por otra parte, la sencillez de una metodología ágil como XP permite dar al alumno una visión del trabajo en equipo en un proyecto de desarrollo de software utilizando un enfoque disciplinado, todo ello dentro de las restricciones de tiempo que tiene una asignatura.

Para las estimaciones y el seguimiento de la planificación resultó efectiva la utilización de puntos. Un punto de esfuerzo en una Historia de Usuario equivalía a una semana ideal de trabajo (12 horas en nuestro caso). De esta forma se facilitaba el trabajo de planificación al separarlo del efecto de la dedicación parcial de los alumnos.

Respecto de las prácticas XP, su grado de aplicación varía y es uno de los aspectos que pensamos que se debería intentar mejorar. Entre las prácticas que más dificultades de aplicación presentaron están: pruebas, *refactoring* e integración continua. Estas prácticas requieren experiencia y mucha disciplina, lo cual es difícil de conseguir sin un entrenamiento previo. En el caso de las pruebas, no basta con herramientas del tipo XUnit puesto que particularmente en sistemas de información de gestión gran parte de las pruebas se asocian a probar interfaces gráficas de usuario. La integración continua y el trabajo en grupo necesitan disponer de herramientas adecuadas, que permitan control de versiones y automatización en la integración y pruebas asociadas. Por otra parte, aunque el cliente in-situ es sin lugar a dudas importante, para efectos de formación la disponibilidad que establecimos para el cliente resultó suficiente. Similarmente, en un comienzo pensamos que la configuración del espacio de trabajo que ofrecían nuestros laboratorios podía constituir un inconveniente, pero no fue así. La programación en parejas tampoco presentó problemas destacables, aunque por problemas de coincidencias horarias entre los integrantes no toda la programación se hizo en parejas. El resto de las prácticas de XP fueron aplicadas en un alto grado, particularmente Entregas Pequeñas y Juego de la Planificación.

Como trabajo futuro, además de la continua mejora en el plan de trabajo y en el material de las asignaturas, nos planteamos las siguientes tareas:

- Continuar con la iniciativa de dejar disponible todo el material en la página web de las asignaturas, tal como ya se hace con LSI.

- Hemos comenzado un proyecto para dotar de infraestructura para el trabajo de los equipos. Se trata de la implantación de un servidor de contenidos vía Web que permitirá a los miembros de un equipo de desarrollo de software (alumnos y profesores) colaborar en la realización de un proyecto. Dicho servidor proporcionará un repositorio compartido para los artefactos generados durante el

proyecto y una serie de servicios tales como: administración de usuarios, control de acceso, carga y descarga de artefactos, discusión entre los miembros del equipo, material de apoyo para la realización del proyecto (plantillas, ejemplos, etc.). Además la información resultante del proyecto quedará disponible para cursos posteriores. También se incorporará la evaluación continua del alumno, pudiendo cada alumno conocer sus calificaciones hasta el momento, así como estadísticas de evaluación de su equipos y de los otros equipos. Para la implementación se seleccionarán herramientas Open Source y se desarrollarán las extensiones necesarias para ajustarlas a las necesidades requeridas.

- Estamos realizando trabajos en control de versiones para historias de usuario y en herramientas para integrar patrones de diseño y *refactoring*.

- Finalmente, estamos preparando un seminario en metodología ágiles para desarrolladores de software y que será ofertado a través del Centro de Formación de Postgrado de nuestra universidad. Este taller tendrá una orientación muy práctica siguiendo el esquema de las asignaturas en las cuales hemos experimentado.

Referencias

[1] Beck K.. Una Explicación de la Programación Extrema, Aceptar el Cambio. Addison-Wesley, 2002.

[2] Jacobson I., Booch G and Rumbaugh J. The Unified Software Development Process. Addison-Wesley, 1999.

[3] Letelier P., Canós J.H. Incorporando Extreme Programming como Metodología de Desarrollo en un Laboratorio de Sistemas de Información. Thomson, Actas de las IX Jornadas de Enseñanza Universitaria de la Informática (JENUI 2003), pp. 257-264, Cádiz, Julio 2003.

[4] Letelier P., Canós J.H. An Experiment Comparing RUP and XP. Springer-Verlag, Lecture Notes in Computer Science, Proceedings of IV International Conference on Extreme Programming, XP 2003, pp. 41-46. Génova, Italia, Mayo 2003.

[5] Letelier P., Canós J.H. Working with Extreme Programming in a Software Development Laboratory. Proceedings Fifteenth International Conference on Software Engineering and Knowledge Engineering , SEKE 2003, pp. 612-615. San Francisco, Julio 2003..

[6] Proyecto EUROPA: Una Enseñanza Orientada al Aprendizaje. Vicerrectorado de Coordinación Académica y Alumnado. Específicamente se trata de los programas AME2: "Nuevos Métodos de Enseñanza-Aprendizaje" y AME3: "Mejora de los Sistemas de Evaluación". Universidad Politécnica de Valencia, 2001. www.upv.es/europa

Brief *e*XPERT Approach Description

Teodora Bozheva
European Software Institute
Parque Tecnológico
48170 Zamudio, Bizkaia
Spain
Teodora.Bozheva@esi.es

Abstract: This document introduces a light method for software development applicable to small teams developing projects characterised with often changing requirements, tight schedules, and high quality demands. The method is called *e*XPERT and builds on the principles of eXtreme Programming (XP) and the Personal Software Process (PSP).

*e*XPERT APPROACH

Development Guidelines

All SME developing e-commerce and e-business software have similar business objectives, namely "Faster-Better-Cheaper". Reports from the field show that productivity and software quality increase by applying XP principles. However, even projects that have adopted several or all XP practices meet project management problems, namely related to estimating and planning the project, both in terms of time and costs.

To overcome this obstacle *e*XPERT focuses on combining XP and PSP practices. In particular PSP principles are added to provide the development teams a means to measure their effectiveness, use it to better plan and manage the projects, and as a consequence increase the customer satisfaction. The combination considered the following objectives:

- To maintain the approach comprehensive
- To maintain the approach easy to apply. Calculation of the additional measurements is essential for making the method practical, however the approach has to remain applicable by software developers.
- To provide all data necessary to perform good estimations, project planning and management activities.
- To retain the people motivated to use the approach.

*e*XPERT approach architecture

The *e*XPERT approach builds on XP and PSP principles. The activities described into the *e*XPERT approach are based on the XP practices. Certain modifications to the pure PSP principles are introduced mainly related to measuring the effectiveness of the activities and the defect rates in order to identify problem causes and to eliminate them in the future. The logs for collecting project data follow the templates and the principles of PSP, but are adjusted as to fit the XP method, in particular to reflect the fact that developers work in pairs and the design, testing and coding process are strongly interrelated and executed in parallel. The PROBE method used for estimating time necessary to develop a given customer requirement considers requirement complexity, which is a slight modification of the PROBE method used in PSP.

The *e*XPERT approach is described in terms of processes. There are several reasons for selecting this manner of representation, namely:

- Software development companies, influenced by the well-known models for software process improvement like CMM, ISO and SPICE, are used to think in terms of processes, and it will be easier for them to understand and implement a process-oriented approach.
- A description in terms of processes facilitates the comparison between a current state of the software development activities performed into an organization with their state after the new approach has been implemented and adopted.
- Companies who later will want to assess and/or increase their maturity following models like CMM, ISO or SPICE, will be able to do it easier.

The *e*XPERT approach consists of the following processes, as shown on fig.1:

- Customer Requirements Management (CRM)
- Project Management (PM)
- Design (DS)
- Code (CD)
- Test (TS)

The processes are described in terms of

- Activities,
- Tasks that have to be done to complete an activity, and
- Responsible role for performing a task.

For the sake of brevity the processes are described herein only at the level of activities. The complete description of the eXPERT approach also provides guidelines for its application, definition of the probe method, data recording logs and a set of tools supporting the usage of the method.

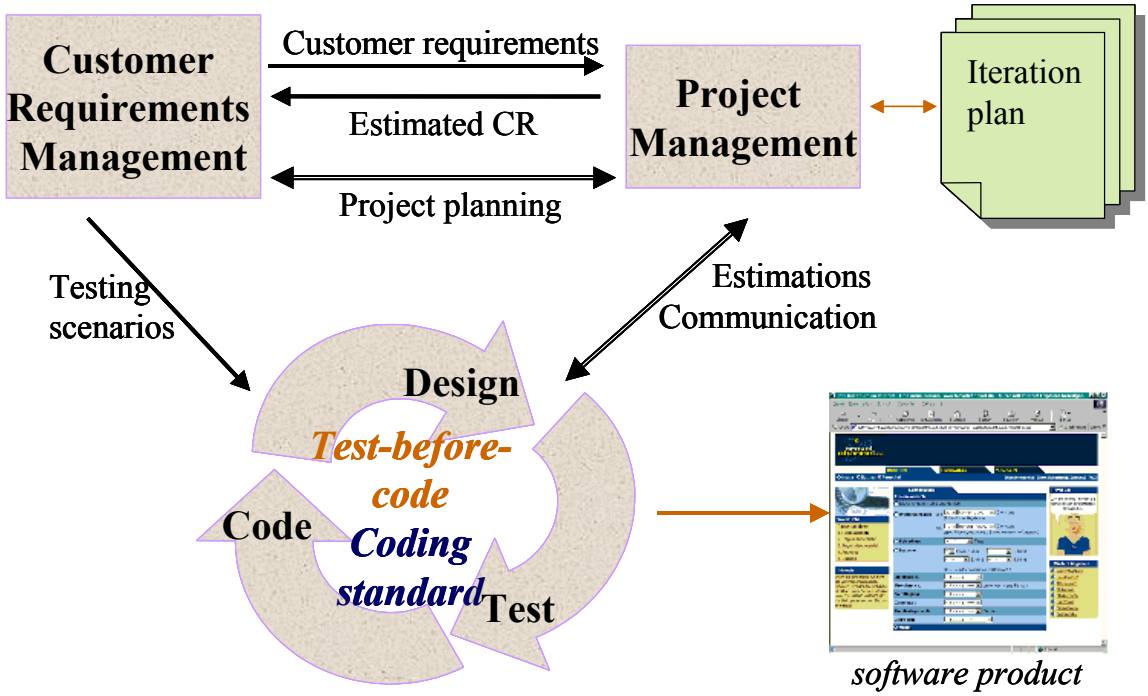


Figure 1 – eXPERT Process Architecture

eXPERT Organisational Structure (Roles)

*e*XPERT organisational structure is very simple. The roles defined are similar to those in XP with some additional responsibilities related to the application of the PSP practices. Of course, one person may execute more than one role in a project, but it is important that this person has the necessary knowledge, skills and time for performing all the responsibilities. In particular, *e*XPERT defines the following roles:

Customer

The customer chooses what will deliver business values, chooses which customer requirements to do first and which ones to defer, and defines the tests to show that the system does what it has to.

It is highly recommended that the Customer stays on-site, and be present with the team full-time.

Customer's responsibilities:

- To determine what will have business value, and the order of building that value
- To express what must be done in terms of customer requirements.
- To prioritise the customer requirements that can be accomplished by the desired delivery date.
- To define releases based on requirements prioritisation and complexity, and considering project velocity.

- To specify tests that prove whether the requirements have been correctly implemented. These *acceptance tests* are later implemented by developers to validate the application.
- To perform the acceptance test before accepting the product from the development team.

Project leader

The Project Leader supervises and guides the application of the eXPERT approach.

Project Leader's responsibility:

- To lead the everyday work: the day-to-day process of planning, designing, testing, coding, releasing.
- To ensure that the entire team takes part into the release and iteration planning. To coordinate the project activities.
- To take measures: project velocity, developer's productivity, defect removal efficiency
- To look for things that slow the team, and to resolve them.
- To resolve scheduling conflicts.
- To keep up-to-date stakeholders which are not directly involved in the project.
- To tune the approach to the project and organisation's needs.

Developer

The developers analyse, design, test, code, and integrate the system. The developers estimate the difficulty of all requirements, and the time they need to develop and deliver the product to the customer.

Developer's responsibilities:

- To estimate requirements complexity
- To split requirements to tasks small enough
- To base the program on simple, clear design, to produce quality software quickly.
- To improve the design using refactoring.
- To keep the system integrated at any time, so that there is a good working version to look at.
- To share the ownership of the code, so everyone feels able to make everything better.
- To follow the accepted coding standards.
- To make sure that the system always works, writing and using comprehensive unit tests, and using customer's acceptance tests.
- When necessary and appropriate to code in pairs, for maximum speed and cross training, in support of shared code ownership and rapid progress.
- To support the Customer in implementing the acceptance tests.

EXPERT Processes

Customer Requirements Management

Overview: This process includes all activities related to eliciting, analysing and controlling the customer requirements for a software product. It builds on the XP practices related to handling user stories. The concept *customer requirements* is used as a synonym of *user stories* from XP. The process is called *customer requirements management* to convey the meaning of organizing, performing and controlling the related activities and taking the responsibility for bringing them to an end.

Process Input: Customer needs and expectation

Activity1: Elicit Customer Requirements

Activity2: Analyse Customer Requirements (CR)

Activity3: Estimate Customer Requirements

Activity4: Measure the process

Process Output: Prioritised CR with complexity and time estimations

Completion criteria: Customer requirements are granulated to such an extent that each of them is estimated to be developed in less than a week

The whole team understands well the customer requirements

Measurements: Effort spent on Customer Requirements Management

Project Management

Overview: The project management process encompasses activities related to planning, tracking and controlling the software development. Project management includes activities that ensure the delivery of a high-quality system on time and within budget.

Estimation of the customer requirements is made into the Customer Requirements Management process because it is a prerequisite for prioritising the customer requirements and deciding which ones of them to include in the following iteration, which is a responsibility of the Customer.

Process Input: Documented customer requirements

Activity1: Define the Project

Activity2: Create Release Plan

Activity3: Create Iteration Plan

Activity4: Monitor and Control the Project

Activity6: Conclude Project

Activity7: Measure the Process

Activity8: Tune process

Process Output: Iteration and release plans

Completion criteria: Project maintained on track, i.e. iterations and releases executed as planned

Measures: Effort spent on Project Management, Project costs. Project velocity

Design

Overview: During the design process the customer requirements are decomposed to product components. The design process is highly interrelated with the Code and Test processes.

Process input: Documented customer requirements

Activity1: Prepare for design

Activity2: Design

Activity3: Document design

Activity4: Make Design Inspection

Activity5: Measure process

Process output: Coding standard, Design document

Completion criteria: simple design well understood by the whole team

Measures: Overall designing effort, Defects found during Design Inspection

Code

Overview: The objective of the Code process is to produce the code of the software product that satisfies the customer requirements. The code is permanently maintained in good condition, i.e. in reusable, testable and working state. The coding process is highly interrelated with the Design and Test processes.

Process input: Documented CR, Design document, Coding standard

Activity1: Implement the product

Activity2: Integrate the code

Activity3: Measure process

Process output: Software product

Completion criteria: Customer requirements and acceptance tests satisfied

Measurement: Implementation effort

Test

Overview: The purpose of the Test process is to validate that the software product produced meets the requirements and satisfy the acceptance criteria defined by the customer. The Testing process is highly interrelated with the Code and Design processes.

Process input: Documented CR, Design document, Coding standard

Activity1: **Prepare for testing**

Activity2: **Implement acceptance tests**

Activity3: **Perform unit testing**

Activity4: **Perform regression testing**

Activity5: **Perform acceptance testing**

Activity6: **Measure process**

Process output: Code without defects

Completion criteria: The test cases defined cover all typical, boundary, and specific cases described by the customer requirements

Measures: Defects rate, Effort spent on acceptance testing

REFERENCES

eXPERT approach description and guidelines for its application, www.esi.es/Expert

Case studies about the 7 trials of the eXPERT approach: www.esi.es/Expert

L. Williams, The Collaborative Software Process, PhD dissertation

Ron Jeffries, Ann Anderson, Chet Hendrickson, The XP Series: EXTREME PROGRAMMING Installed; Addison Wesley

Watts S. Humphrey, A Discipline for Software Engineering; Addison Wesley (1995)

*e*XPERT: Results from seven pilot projects

Teodora Bozheva
European Software Institute (ESI)
Parque Tecnológico #204
48170 Zamudio (Bizkaia)
Teodora.Bozheva@esi.es

1. Introduction

For the past several years, business has been evolving into e-business, commerce into e-commerce, education into e-learning, work into e-work, and the projects related to them all are naturally called e-projects. This development of the e-era requires large amounts of software development with specific requirements and constraints. If in software development projects cost is usually the most important control criteria, in e-project development time-to-market is in most cases the most critical, overriding cost, quality or any other criteria; the reason is that often getting to the marketplace soon would provide a very significant market opportunity.

Several agile methodologies appeared recently whose goal is to serve the needs of e-service providers. They all aim at customer satisfaction, adaptability to changes, delivering high quality products quickly and within an agreed budget frame. They look promising but little actual experience still exists using them in real e-project development (or any other development).

This article presents the results of seven pilot projects, which have applied an agile method based on the principles of extreme programming (XP) and the Personal Software Process (PSP)[1], named *e*XPERT. The goal of combining XP and PSP is twofold: (a) to improve developer's productivity and product quality, and (b) to facilitate the integration of beginner programmers into real software development projects, namely to help them start working in a disciplined manner, to correctly define and estimate their tasks, as well as to provide means for a smoother and more efficient software development.

2. *e*XPERT

The *e*XPERT project[2] was established with two main objectives:

- To define a practical combination of XP[3] and PSP[4] practices, named *e*XPERT approach;
- To experiment the *e*XPERT approach in 7 pilot projects in order to validate its usability, benefits and pitfalls, and to collect and disseminate best practices and lessons learned.

[1] PSP is developed at the Software Engineering Institute (SEI), Carnegie Mellon University.

[2] The *e*XPERT project (http://www.esi.es/Expert) is partially funded by the European Commission through the IST programme (ref. IST-2001-34488) and by the Spanish Ministry of Science and Technology (TIC-2001-5254).

The *e*XPERT approach includes all the XP practices, and three PSP practices: time and effort recording, defect recording, task estimation by means of the Probe method, modified as to reflect customer requirement's complexity instead of LOC.

The *e*XPERT approach is described in terms of 5 processes: Customer Requirements Management, Project Management, Design, Test, and Code. This allows software development companies, influenced by models and standards like CMM(I) and ISO, and used to think in terms of processes, to understand and implement the approach. A description in terms of processes makes it easier to compare the state of the software development activities performed in an organization at different times.

Despite the process-oriented description of the method it preserves the individual orientation of XP and PSP, remains comprehensive and easy to apply, and at the same time provides all the data necessary to perform good estimations, project planning and management activities

3. Experiment approach

The *e*XPERT approach was defined after a thorough study of both models. Two of the piloting teams had little experience in software development, and only one of the other teams had previously applied some XP practices. Every team defined a Baseline project, developed with the companies' traditional approach and a comparable Pilot one, in which the *e*XPERT approach was experimented. All the projects were real ones in the area of e-commerce.

All the pilot projects had the following business goals:

- G1: To increase productivity of the developers by 20%
- G2: To reduce the defect rate by 30%
- G3: To decrease schedule and cost deviations by 15%.

Several metrics (see Table 1) were defined to show goal achievement. Companies, who did not have available data, used part of their Pilot projects as a Baseline. In particular they developed certain modules of the Pilot Project with their traditional approach in order to obtain data and be able to see if there was an improvement or not. To decrease the influence of the experience gained in the domain, different developers took part in the implementation of the Baseline and the Pilot projects. The teams selected, however, had relatively the same professional level. The duration of the pilot projects ranged from 6 to 9 months and they all were in the e-service area.

Three reviews were performed to all the pilot projects. They were realised according to a predefined procedure. After each review lessons learnt by all the experimenting companies were described and documented. Additionally, it was checked, if the project objectives and the additional business goals defined by the companies were reached. The collection of metrics, though tedious in the beginning as mentioned by the companies, has proved to be useful, clearly showing whether the project objectives were fulfilled, and the organisations keep applying them in their following projects.

Metric	Calculation	Unit	Measured	Possible degree of detail
Productivity	Size / Effort	Work/Time	Size [KLOC] Effort [Man month]	Measure Productivity • for the team • for each pair • for each programmer
	Velocity/Effort	-	Velocity Effort [Man month]	
Defect Rate	Number of defects	Number	Number of defects	• Make measurements for different types of defects • Measure Defect Rate both for the team and for each programmer • Measure Defect Rate for the whole project and for each release and/or iteration
	(Effort spent for bug fixing / Effort) x 100	%	Effort spent for bug fixing [Man month] Effort [Man month]	• Measure Defect Rate both for the team and for each programmer • Measure Defect rate for the whole project and for each release and/or iteration
Relative Schedule Deviation	((Real time - Planned time) / Planned time) *100	%	Real time [months] Planned time [months]	Measure the times planned and real for • the whole project • a release • an iteration • each user story/ feature/…
Relative cost deviation	((Real costs - Planned costs)/Planned costs) *100	%	Real costs [K €] Planned costs [K €]	Measure the costs planned and real for • the whole project • a release • an iteration • each user story/ feature/…
Relative project cost change	$(1-Cost_{pp}/Cost_{bp})x100$	%	$Cost_{pp}$ [K €] $Cost_{bp}$ [K €]	Measure Relative project cost change • For the overall project • For selected modules
Customer satisfaction	Customer satisfaction rated between 0-100%	%	Questionnaire	Measure the customer satisfaction with the whole project and/or during the project with each release, iteration etc., depending of the company
Developer satisfaction	Developer satisfaction rated between 0-100%	%	Questionnaire	Measure the developer satisfaction with the whole project and/or during the project with each release, iteration etc., depending of the company

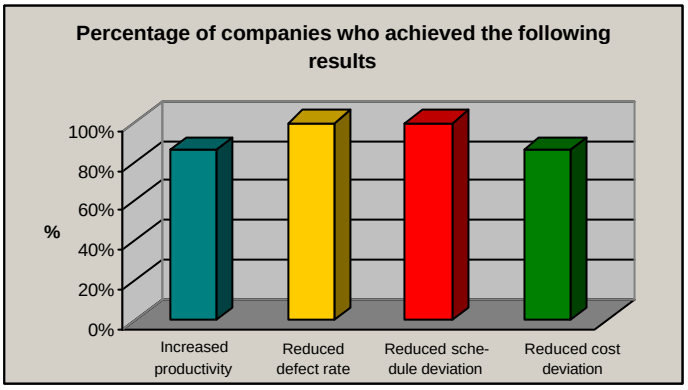
Table 1: Metrics

4. Main results

The following results were obtained within the pilot projects with respect to the project goals:

- G1: Productivity increased up to 73%. One company decreased its productivity by applying eXPERT
- G2: Defect rates reduced between 10% and 83%.
- G3.1: Schedule deviation reduced between 7% and 38%.
- G3.2: Cost deviation decreased up to 31%. Only one company increased its cost deviation.

As a consequence of the achievement of the results with respect to G1-G3, the companies' software products are now at higher quality and developed in a shorter time that is their competitiveness increased too. The diagram on the right shows the percentage of companies, who achieved the defined objectives.



The following correlation between the achievement of the business goals and the application of the XP and PSP practices incorporated in the *e*XPERT approach was observed:

- The factors that mostly influenced the productivity of the developers were
  - the improved discipline, which the *e*XPERT approach established in the pilot projects, in particular the improved responsibility with respect to the team mates, and the effort and defect tracking practices;
  - the maintenance of the code in working state due to the test-driven development, the frequent integration and the small releases;
  - the better task estimations based on the PSP PROBE method and the more effective time spending impacted by the metrics collection.

  The team, which decreased its productivity had been postponing refactoring, which resulted in re-design of a big part of the system implemented.
- The defect rate was mainly reduced due to the application of the test-before-code principles and the requirement to integrate the code at least once a day and to maintain it working.
- The schedule and cost deviations, although not completely removed, were significantly reduced. Key contribution for this had the PSP practices included

in the *e*XPERT approach, namely the estimation with the PROBE method and the effort tracking procedure.

All the teams applying the method found different ways to adopt it with benefits. For the sake of brevity we are going to point out only the most important ones here. Case studies about all the experiments, lessons learnt and a detailed analysis of the trials can be found at the project web site, http://www.esi.es/Expert.

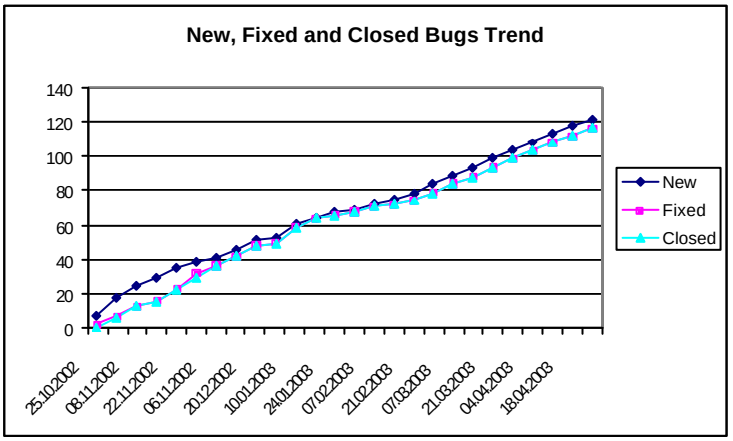
With respect to the work with the customer, since none of the companies managed to have *customer-on-site*, three alternatives of this XP practice were found:

- Developer-on-site: Sending regularly a developer to the customer's office to present the latest state of the developed software and discuss modifications to existing as well as new features.
- Web-demo: Develop a web site with restricted access to it providing the customer possibility to see the current status of the development and provide feedback about it.
- Excellent communication: the key difference from the traditionally recommended good communication with the customer is that *e*XPERT advices to achieve and apply an agreement with the customer that he/she is reachable any time for development-related questions and is going to provide useful feedback.

The software engineering practices have to be adjusted to the customs of the developers and the organisational culture.

All the teams found *pair programming* difficult to apply all day long, but especially useful when resolving key problems in the project. One of the experimenting companies investigated deeper the relationship between the effectiveness of a pair and the qualification of the buddies. The results showed that for difficult tasks, e.g. designing the software architecture, experienced developers have to be paired, while the beginners have to be paired for simpler tasks.

Test-before-code implies a change in the manner programmers are used to develop software and a is key success factor for implementing error-free products (see the diagram on the right).

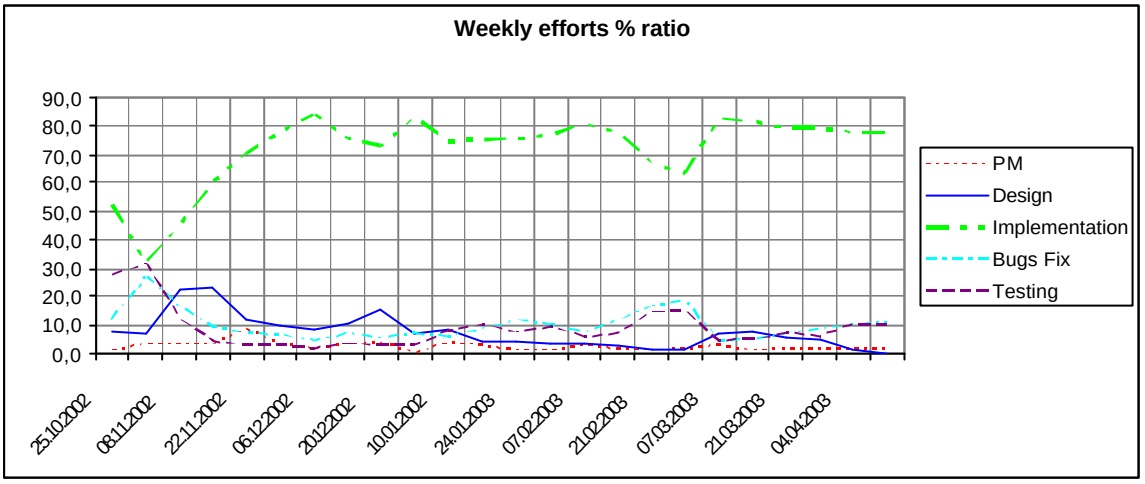


*e*XPERT recommends different approaches to introduce this practice.

One possibility is to begin with implementing test cases for key functionality, e.g. the business logic of the software, and gradually extend the test cases framework to cover more software features. An alternative is to implement the test cases before code

integration. Although this approach is controversial to the principle to write the tests before the code, the experience shows that conscious and continuous testing of recently implemented small pieces of code leads to the same result as testing before coding.

Two ISO-certified organisations took part in the project. It showed up that the application of *e*XPERT increased significantly the agility of their processes and shortened the time-to-market with 30%. Apart from this about 80% of the effort is devoted to Implementation and less than 10% is dedicated to Project Management (see the diagram bellow).



All the alternative approaches to implement *e*XPERT, found during the pilot projects, are generalised in *Guidelines for Applying the eXPERT Approach*, which can be also found at the project web site. Additionally, tools facilitating the method application are recommended in the document.

5. Conclusion

The *e*XPERT approach proved to be helpful and effective for small software teams developing business-critical projects characterised with often changing requirements, exploration and application of new technologies, and need of fast integration of new staff. It is relatively easy to implement and is well accepted by the developers. After successful adoption it decreases the deviations from the schedule and the budget, and improves customer's satisfaction with the final product.

Unlike other agile methods, *e*XPERT offers a set of practices for collection of data, which support the estimation of the projects with respect to time and effort needed for their working out. Additionally guidelines for its application and a list of supporting tools are provided, which make it a very useful and practical package necessary for every team interested in implementing software faster, better, and cheaper.

References

1. Case studies about the 7 trials of the eXPERT approach: http://www.esi.es/Expert
2. L. Williams, The Collaborative Software Process, PhD dissertation

3. Ron Jeffries, Ann Anderson, Chet Hendrickson, The XP Series: Extreme Programming Installed; Addison Wesley
4. Watts S. Humphrey, A Discipline for Software Engineering; Addison Wesley (1995)

Formal Agility. How much of each?

Ángel Herranz and Juan José Moreno-Navarro

Univ. Politécnica de Madrid
Campus de Montegancedo s/n, Boadilla del Monte 28660, Spain
+34 9133674{52,58}
{aherranz,jjmoreno}@fi.upm.es

Abstract. Agile Processes and Formal Methods (FM), water and oil, impossible mixture? Yes at first sight. Nevertheless, being formal methods weight processes and being agile processes informal approaches to software development, it is worth to study how much formal can be an agile process like Extreme Programming (XP) and how much agile can be a formal method. On our view, some XP practices are suitable for a formal approach.

1 Motivation

At first sight, XP [1] and FM [10, 8] are water and oil: an *impossible* mixture. Maybe the most relevant discrepancy is that while one of the strategic motivation of XP is "spending later and earning sooner" FM require "spending sooner and earning later". However, a deeper analysis reveals that FM and XP can benefit their selves.

The use of formal specifications is perceived as improving reliability at the cost of lower productivity. XP and other agile processes focus on productivity so, in principle, using FM following XP practices could improve its efficiency. In particular, *pair programming*, *daily build*, *the simplest design* or *the metaphor* are XP practices that in our view are independent of the concrete development technology used to produce software and the declarative technology and FM is just a different development technology.

On the other hand, one criticism to XP is that it has been called *systematic hacking* and, probably, the underlying problem is the lack of a formal or even semi-formal approach. What XP practices are liable to incorporate a formal approach? We think that *unit testing*, *incremental development* and *refactoring* are three main XP practices where FM can be successfully applied:

- When you write a formal specification you are saying *what* your code must do, when you write a test you are doing the same so one idea is to use formal specifications as tests.
- Incremental development is in some sense similar to the refinement process in FM: specifications evolve to code maintaining previous functionality.
- Finally FM can help to remove redundancy, eliminate unused functionality and transform obsolete designs into new ones, and this is refactoring.

After all, it might be possible to dilute FM in XP. We would like to point out that we are not claiming to *formalise* XP, but just to study how the declarative technology can be integrated in XP and how XP can take advantages of this technology.

2 Formal XP? How?

Most XP practices are technology independent. In our opinion, the XP process could be adopted by using a formal method tool (an specification language and a verified design process) instead of an ordinary programming tool. In other words, we propose to write formal specifications instead of programs. A number of advantages appear:

- Instead of "the code is the documentation", "the specification is the documentation" and business and development would have a high level description through a formal specification of the intended semantics of the future code.
- FM tools (theorem provers, model checkers, etc.) help to maintain the consistency of the specification and the correctness of the implementation.
- Important misunderstandings and errors can be captured in the early stages of the development but close enough to code generation.

While in XP emphasis is on staying light, quick, and under a low ceremony process, the introduction of FM might make XP sometimes heavier, sometimes not. Even in the first case we would have that: i) it can still be considered a light method in the FM area, and ii) the benefits should compensate in many cases the increase of work.

Writing specifications can be perceived as improductive because programmers have to do their work anyway. In order to be productive enough a code synthesiser from specifications is needed. In section 3 we discuss briefly this issue.

The introduction of FM in XP would not impact in all the practices, mainly because most of them are technology independent. What XP practices would be more affected? On our view, Testing, Refactoring an Continuous Integration. Other practices seem to be easily adapted.

2.1 Unit Testing

In XP the role of writing the tests in advance is similar to the role of writing a precise requirement: it is used to indicate *what* the program is expected to do. Tests in XP solves two different problems:

- The detection of misunderstandings in the *intended specifications*.
- The detection of errors in the implementation.

The perspective under both problems is completely different when using FM. The detection of inconsistencies in formal specifications are supported by formal tools, mainly by *a generator of proof obligations* and by *a theorem prover assistant*. With both tools the user get information about possible inconsistencies.

The detection of errors in the implementation is absolutely unneeded thanks to the *verified design process*: a process that ensures that the code obtained from an original specification is correct with respect to it. Notice that the use of tests do not ensure that requirements are satisfied, just "convince" the programmer that it happens. The FM approach overcome this limitation.

Anyway, writing formal specification is not an error free activity. These means that *tests* can be lifted to the specification level by introducing formulas that are, at least, proof obligations. The proof of such formulas allows detecting inconsistencies or misunderstandings.

2.2 Incremental Development

During the Planning Game new customer stories are added in every iteration, these means, in many cases, that those *requirements* are added incrementally. Most valuable stories are then specified and new logical properties must be added to the system and previous properties must still hold.

The formal meaning of a specification can be understood as a theory, a set of formulas. Let us say that the specification of the system after step i is Γ_i, a new story appears at step $i+1$ and the specification of the new story is the set of formulas γ_{i+1}. The the specification at step $i+1$ is then Γ_{i+1}. We call *the combination property at step* $i+1$ to the following property:

$$\Gamma_{i+1} \models \Gamma_i, \gamma_{i+1}$$

Theorem 1. *Proved the* ***combination property*** *at every step* $i \in \{1, \ldots, n-1\}$ *the following property holds:*

$$\Gamma_n \models \gamma_i$$

The proof would proceed by induction on i.

Informally: if after every iteration a new story is correctly specified the system will implement correctly all the customer stories.

The most important consequence of paragraphs above is that we have a method for proving that the specification entails every customer story. Is this method practical enough? To be honest, the answer is *no, it is not*, it is a heavy method. But we would like to point out that the prover technology will help a lot: a great amount of proofs can be automatically done and the rest can be *manually* proved by guiding the prover.

2.3 Refactoring

With respect to Refactoring we think that FM can help in two different ways. First, the system is restructured but the meaning cannot be changed. If the system has been specified, to prove that the behaviour has not changed is possible. Second, declarative technology makes easier to find and remove redundancy, eliminate unused functionality and transform obsolete designs into new ones.

An open possibility is to specify generic patterns and then, whether a specification is an instantiation of such a generic pattern can be proved. The idea is having a relevant collection of generic patterns trust the prover technology of FM are able to *match* specifications with specifications in those patterns. Some works in formalising design patterns [2] have been done using SLAM-SL [3].

3 What FM?

We would like to finish with a brief discussion about which FM would be more suitable for XP. From our point of view, the method should be close to the programming stage, these means that high level languages like Z are not a good choice. A better one is VDM. The VDM abstract syntax is close to the abstract syntax of a programming language.

Since 2001 our group is working in the design and implementation of a specification language that, modestly, we think that it would be a good option: SLAM-SL [6, 9]. The most relevant features of SLAM-SL for this work are:

- It is an object-oriented specification language valid for the *design and programming stages* in the software construction process (a non alien notation).
- It has been designed as a trade-off between the expressiveness of its underlying logic and the possibility of code synthesis. From a SLAM-SL specification the user can obtain code in a high level programming language (let us say Java), a code that is *readable* and, of course, correct with respect to the specification.
- Because the code is readable, it can be modified and, we expect, improved by human programmers.

In [7] SLAM-SL was presented as a useful formal tool for AP. The reader can find a deeper explanation of features above and its relationship with XP practices seen in this work. Other, on our view, interesting autors' work have todo with the synthesis of asssertions for debugging from specifications: [4, 5].

4 Conclusions

We have presented how some XP practices can admit the integration of Formal Methods. In particular, *unit testing*, *refactoring*, and, in a more detailed way, *incremental development* have been studied from the prism of FM. Probably there is more room for FM ideas helping agile methodologies and XP, and we will study this as a future work.

One of the goals of the SLAM system is to make FM and their advantages closer to any kind of software development. Obviously FM are specially needed for critical applications but combining it with rapid prototyping and agile methodologies could make them affordable for any software construction. Up to know we have not equipped SLAM with an automatic interface generator that precludes the use of our system for heavy graphical interface applications. The automatic generation of graphical interfaces is another matter of future work.

References

1. K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Pearson Education, 2000. ISBN 201-61641-6.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
3. A. Herranz, J. Moreno, and N. Maya. Declarative reflection and its application as a pattern language. In M. Comini and M. Falaschi, editors, *11th. International Workshop on Functional and Logic Programming (WFLP'02)*, Grado, Italy, June 2002. University of Udine.
4. A. Herranz and J. J. Moreno. Generation of and debugging with logical pre and post conditions. In M. Ducasse, editor, *Workshop on Automated and Algorithmic Debugging 2000*. TU Munich, 2000.
5. A. Herranz and J. J. Moreno. On the role of functional-logic languages for the debugging of imperative programs. In *Workshop on Functional and Logic Programming 2000*. Universidad Politécnica de Valencia, 2000.

6. A. Herranz and J. J. Moreno. On the design of an object-oriented formal notation. In *Fourth Workshop on Rigorous Object Oriented Methods, ROOM 4*. King's College, London, March 2002.
7. A. Herranz and J. Moreno-Navarro. Formal extreme (and extremely formal) programming. *Fourth International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP'03)*, May 2003.
8. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.
9. The SLAM website. `http://lml.ls.fi.upm.es/slam`.
10. J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.