Introducción a las metodologías ágiles

Otras formas de analizar y desarrollar

Jorge Fernández González

PID_00184468

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 7

1. La necesidad de ser ágiles..... 9
1.1. El "Manifiesto ágil" 10
1.2. Metodologías ágiles más importantes 13
1.2.1. SCRUM 13
1.2.2. Dynamic Systems Development Method 14
1.2.3. Crystal Methodologies 15
1.2.4. Feature-Driven Development 15
1.2.5. Adaptive Software Development 16
1.3. ¿Cuándo puedo aplicar una metodología ágil? 16
1.4. ¿Han muerto las metodologías tradicionales? ¿RUP is RIP? 18

2. EXtreme Programming (XP)..... 19
2.1. Las variables XP: coste, tiempo, calidad y alcance 21
2.2. Los cuatro valores: comunicación, simplicidad, realimentación y coraje 23
2.3. Las doce prácticas básicas de XP 24
2.3.1. Diseño simple 25
2.3.2. Refactorización 25
2.3.3. Test 26
2.3.4. Estándares de codificación 27
2.3.5. Propiedad colectiva del código 28
2.3.6. Programación por parejas 28
2.3.7. Integración continua 29
2.3.8. 40 horas semanales 30
2.3.9. Metáfora del negocio 30
2.3.10. Cliente *in situ*..... 30
2.3.11. Entregas frecuentes 31
2.3.12. Planificación incremental 31
2.4. El ciclo de vida de la metodología XP 31
2.4.1. La fase de exploración 33
2.4.2. La fase de planificación 34
2.4.3. La fase de iteraciones 35
2.4.4. La fase de producción 36
2.4.5. La fase de mantenimiento 37
2.4.6. La fase de muerte del proyecto 37
2.5. Los diferentes roles dentro de XP 37
2.6. El entorno de trabajo 38

2.7. Qué dicen los detractores de XP 39
2.8. Un ejemplo de XP 40
2.8.1. Fase de exploración 41
2.8.2. Fase de planificación 43
2.8.3. Fase de iteraciones 45
2.8.4. Fase de producción 47
2.8.5. Fase de mantenimiento 48
2.8.6. Fase de muerte 49

3. Ser ágiles no es solo programar ágiles 50

Resumen 51

Actividades 53

Ejercicios de autoevaluación 53

Solucionario 54

Glosario 55

Bibliografía 56

Introducción

Algunos de vosotros os estaréis preguntado qué significa eso de "ágil" y si conlleva de forma intrínseca un poco de "hacer mal las cosas" o de "dejarlo a medias". Cierto es que durante la asignatura casi todo lo que hemos intentado enseñar es precisamente a ser metódicos, rigurosos y estrictos científicamente hablando, aunque no menos cierto es que hemos intentado inculcaros también algo de espíritu crítico. Pues bien, es precisamente a este espíritu crítico al que tenemos que apelar en este módulo. Para intentar comprender que ser "ágiles" no significa renunciar a formalismos ni dejar de ser estrictos y rigurosos.

Muchos profesionales que nos dedicamos a los sistemas de información hemos dicho en alguna ocasión algo semejante a esto "eso es en teoría y queda muy bonito pero en la práctica no puedes perder tanto tiempo haciendo el análisis porque cuando acabas el sistema ya ha cambiado y el cliente se ha aburrido". Yo mismo soy consciente de lo fácil que se puede caer en esta dinámica cuando la presión se nos echa encima y los plazos de entrega se acercan.

La alta competitividad actual hace que los sistemas de información se tengan que desarrollar de forma rápida para adaptarse a la organización. Las prisas hacen que lo primero que se deseche en esta carrera "loca" hacia un rápido desarrollo sea un análisis exhaustivo y se sustituye por uno superficial o simplemente se elimina. Éste es sin duda **el gran error**, deseamos tener un sistema desarrollado rápidamente pero con lo que realmente nos encontramos entre las manos es con un sistema lleno de errores, inmanejable y que no se puede mantener.

Es difícil cambiar las reglas del mercado mundial, así que lo que se ha pensado es adaptar las metodologías de especificación y desarrollo a este entorno cambiante y lleno de presiones, en el que obtener un resultado rápido, algo que se pueda ver, mostrar y sobre todo utilizar, se ha vuelto crucial para el éxito de las organizaciones. La metodología necesariamente ha de ser ágil, debe tener un ciclo corto de desarrollo y debe incrementar las funcionalidades en cada iteración del mismo preservando las existentes, ayudando al negocio en lugar de darle la espalda.

Es para este entorno para el que han nacido las metodologías ágiles.

Las metodologías ágiles no son la gran solución a todos los problemas del desarrollo de aplicaciones, ni tan siquiera se pueden aplicar en todos los casos, pero sí que nos aportan otro punto de vista de cómo se pueden llegar a hacer las cosas, de forma más rápida, más adaptable y sin tener que perder la rigurosidad de las metodologías clásicas.

El objetivo de este módulo no es otro que mostrar que no siempre hay un camino único para hacer bien las cosas, no todo es blanco o negro, sino que hay una gran gama de grises. Esos tonos grises también los tenemos en las metodologías de especificación y desarrollo.

En esta *Introducción a las metodologías ágiles* pretendemos mostrar el actual abanico existente de este tipo de metodologías, para pasar luego a centrarnos en una que está teniendo especial éxito entre los profesionales del sector, la Extreme Programming, más conocida por *metodología XP*.

A medida que vayamos viendo esta metodología, iremos haciendo hincapié en las ventajas y en los inconvenientes, concluiremos con un ejemplo de aplicación de XP en un entorno basado en Java. La elección de Java no es arbitraria, responde al hecho de que la idea de "agilidad" en el desarrollo de aplicaciones esta muy ligada a la de "software libre", sin duda debido a que los núcleos impulsores de ambas ideas son muy cercanos.

Finalizaremos mostrando cómo la idea de "ágil" se está introduciendo con fuerza en diferentes ámbitos, como es el caso del diseño de bases de datos y objetos, en la documentación, etc.

Objetivos

El objetivo general de este módulo es que adquiráis una primera visión de las metodologías ágiles para que así tengáis la posibilidad de reconocer cuándo un proyecto es propicio para usarlas. El objetivo general se descompone en los siguientes objetivos parciales, que el estudiante debe conseguir:

1. Adquirir los conceptos del manifiesto ágil.

2. Conocer las principales metodologías ágiles actuales.

3. Desarrollar un espíritu crítico que le permita aplicar la metodología más adecuada a cada proyecto en concreto.

4. Conocer en detalle la metodología ágil que está teniendo mayor éxito: Extreme Programming.

5. Vislumbrar hacia dónde puede ir la metodología de la ingeniería del software en los próximos años.

1. La necesidad de ser ágiles

ágil (del lat. *Agilis*).

1) adj. Ligero, pronto, expedito.

2) adj. Dicho de una persona o de un animal: que se mueve o utiliza sus miembros con facilidad y soltura.

3) adj. Se dice también de estos miembros y de sus movimientos, y de otras cosas. *Luces ágiles*. *Prosa ágil*.

Diccionario de la Real Academia Española

"Que se mueve o utiliza sus miembros con facilidad y soltura". *A priori* no parece un adjetivo muy adecuado para una metodología. Para buscar el origen de ese adjetivo, debemos remontarnos al sujeto inicial que lo poseía que no es otro que la organización o empresa. Es ésta la que debe ser ágil, la que debe mover todos sus "miembros" con facilidad y soltura, en una orquestación que le permita sobrevivir en el mundo altamente competitivo en el que nos encontramos. Para ello necesita de un "cerebro", de un sistema de información que debe crecer según la demanda de necesidades que se le exijan, y que debe crecer de una forma rápida y coherente, de manera que la organización pueda adaptarse rápidamente a los cambios producidos en su entorno.

La competencia cada día es más feroz y los ritmos de cambios son más rápidos. Las organizaciones tendrán que responder de forma inmediata a estos cambios si quieren sobrevivir, tendrán que evolucionar en un plazo corto de tiempo. Los sistemas de información (SI) no se pueden quedar atrás, las organizaciones dependen de ellos para funcionar y no deben ser un lastre, sino una ventaja competitiva. El papel de los SI y su velocidad de adaptación a los cambios marcará la diferencia entre una empresa próspera y una en declive. Los SI tendrán que ayudar a que esta evolución continua no sólo sea una realidad, sino que lo sea con el mínimo coste de recursos posible. Han de permitir que las organizaciones evolucionen de forma eficaz, eficiente y, cómo no, **ágil**.

Así pues, de forma transitiva, si la organización ha de ser ágil, y el SI que la controla ha de ser ágil, que menos que la metodología para desarrollar partes del SI también sea ágil.

Pero cuando realmente nació el adjetivo *ágil* aplicado al desarrollo de software fue en febrero de 2001, en Snowbird Ski Resort de UTAH (EE.UU.). Se reunieron para descansar, esquiar y, ya que estaban, conversar, un grupo de 17 expertos de la industria del software para, precisamente, ver qué se podía

hacer para adecuar las metodologías a las necesidades de la industria. Asistieron representantes de diferentes metodologías como Extreme Programming, SCRUM, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, que ya estaban ofreciendo una alternativa a los desarrollos "tradicionales" caracterizados por la rigidez y la documentación exhaustiva en cada una de las etapas y que se habían mostrado inadecuados para los proyectos de pequeña envergadura (y en ocasiones incluso para los medianos).

Tras esta reunión, se crearon dos cosas que han dado mucho que hablar en los últimos años, la primera de ellas es el "Manifiesto ágil para el desarrollo de software" que recoge la filosofía de las metodologías ágiles.

Webs complementarias

Podéis ver el contenido de "Manifiesto Ágil para el desarrollo de software" y "The Agile Alliance"en las webs siguientes: http://www.agilemanifesto.org/ (http://www.agilealliance.org/)

La segunda de ellas es una organización sin ánimo de lucro "The Agile Alliance", dedicada a promover los conceptos relacionados con el desarrollo ágil de software y ayudar a las organizaciones y empresas a adoptar la agilidad.

1.1. El "Manifiesto ágil"

En el "Manifiesto ágil" se definen los **cuatro valores** por las que se deberían guiar las metodologías ágiles.

> "Estamos buscando mejores maneras para desarrollar software y ayudar a otros a desarrollarlo. En este trabajo valoramos:
>
> - Al individuo y sus interacciones más que al proceso y las herramientas.
> - Desarrollar software que funciona, más que obtener una buena documentación.
> - La colaboración con el cliente más que la negociación de un contrato.
> - Responder a los cambios más que seguir una planificación.
>
> De esta manera, mientras mayor valor tengamos en la parte derecha más apreciaremos los de la parte izquierda."
>
> "Manifiesto ágil"
>
> Firmado por Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

Veamos qué significa cada uno de estos valores:

1) Al individuo y sus interacciones más que al proceso y las herramientas.

Sin duda, la herramienta fundamental de la ingeniería del software y del desarrollo de aplicaciones es el cerebro humano. Unas jornadas maratonianas de catorce horas de trabajo van en detrimento de la calidad del producto final. Pero una persona sola no realiza un proyecto, necesita de un entorno en el que desarrollar su trabajo y de un equipo con el que colaborar. Estas interacciones deben también cuidarse. Un factor clave para el éxito es construir un buen equipo, que se lleven bien entre ellos, y que además sepan cómo construir su propio entorno de desarrollo. Muchas veces se comete el error de construir

primero el entorno y esperar que el equipo se adapte a él. Esto nos resta eficiencia, es mejor que el equipo lo configure sobre la base de sus necesidades y sus características personales.

Además, las interacciones que haga el equipo con el usuario final deberían ser igual de fluidas siendo este usuario un miembro más del equipo, con un objetivo común, que es conseguir que el proyecto funcione y sea útil para él.

Si todo esto funciona bien, es obvio que la elección de las herramientas y el proceso mismo de desarrollo pasan a estar en un plano totalmente secundario en la ecuación del éxito del proyecto.

2) Desarrollar software que funciona más que obtener una buena documentación.

Uno de los objetivos de una buena documentación es poder ir a consultarla cuando haya que modificar algo del sistema. Sin duda es un arma de doble filo, una buena documentación actualizada en cada modificación y bien mantenida al día permite saber el estado de la aplicación y cómo realizar las modificaciones pertinentes, pero son pocos los que con las presiones externas de tiempo y dinero acaban actualizando la documentación. Generalmente, cuando se tiene que arreglar un programa porque algo falla, nos concentramos en que funcione ya que es muy posible que haya usuarios parados (incluso enfadados) y que la empresa o la organización esté perdiendo dinero; en estos casos, ni nos paramos a mirar detenidamente la documentación ni, cuando se acaba de arreglar, nos ponemos a actualizarla.

En la filosofía ágil, lo primordial es evitar estos fallos, centrar nuestro tiempo en asegurar que el software funciona y que se ha testeado exhaustivamente, e intentar reducir la creación de documentos "inútiles", de éstos que acaban no manteniéndose, ni tan siquiera consultándose. La regla no escrita es **no producir documentos superfluos**, y sólo producir aquellos que sean necesarios de forma inmediata para tomar una decisión importante durante el proceso de desarrollo. Estos documentos deben "ir al grano", ser cortos y centrarse en lo fundamental, olvidándonos de los circunloquios que no aportan nada a la comprensión del problema.

3) La colaboración con el cliente más que la negociación de un contrato.

La consultoría informática de los últimos años se ha convertido en una lucha a muerte entre el proveedor del servicio y el cliente que lo contrata. Por una parte, el cliente intenta que se hagan el mayor número de funcionalidades con el mismo dinero, por otra parte, el consultor intenta que por ese dinero sólo se realicen las funcionalidades contratadas inicialmente. En las reuniones de seguimiento de los proyectos es fácil oír frases del tipo "esta modificación no entra en el contrato" respondida generalmente por la tan típica "pues yo ya no tengo más presupuesto y así no podemos trabajar". Al final, este tipo

de proyectos hacen que el consultor informático sea un híbrido entre analista y abogado, que desarrolla habilidades legales para salvaguardarse en caso de conflicto jurídico.

Sin embargo, para que un proyecto tenga éxito es fundamental la complicidad y el contacto continuo entre el cliente y el equipo de desarrollo. El cliente debe ser y sentirse parte del equipo. De esta manera, ambos entenderán las dificultades del otro y trabajarán de forma conjunta para solucionarlo.

4) Responder a los cambios más que seguir una planificación.

Una organización cambia constantemente, se adapta a las necesidades del mercado y reorganiza sus flujos de trabajo para ser más eficiente. Es difícil, pues, que en el desarrollo de un proyecto, éste no sufra ningún cambio, ya que es seguro que las necesidades de información de la empresa habrán cambiado. Y no sólo esto, sino que las posibilidades económicas de la misma pueden influir sobre nuestro proyecto, bien sea agrandándolo o bien sea reduciéndolo. Son muchos los factores que alterarán nuestra planificación inicial del proyecto. Si no la adaptamos a estos cambios, corremos el riesgo de que, cuando acabemos, nuestra aplicación no sirva para nada y el cliente se haya gastado el dinero en vano. La habilidad de responder a los cambios de requisitos, de tecnología, presupuestarios o de estrategia, marca sin duda el camino del éxito del proyecto.

Como consecuencia de estos **cuatro valores**, el Manifiesto ágil también enuncia los **doce principios** que caracterizan un proceso ágil diferenciándolo de otro tradicional donde este enfoque no se había aplicado lo suficiente; siempre se había dejado implícito pero sin hacer hincapié en ellos.

1) La prioridad es satisfacer al cliente mediante tempranas y continuas entregas de software que le aporte un valor.

2) Dar la bienvenida a los cambios incluso al final del desarrollo. Los cambios le darán una ventaja competitiva a nuestro cliente.

3) Hacer entregas frecuentes de software que funcione, desde un par de semanas a un par de meses, con el menor intervalo de tiempo posible entre entregas.

4) Las personas del negocio y los desarrolladores deben trabajar juntos diariamente a lo largo de todo el proyecto.

5) Construir el proyecto en torno a individuos motivados. Darles el entorno y el apoyo que necesitan y confiar en ellos.

6) El diálogo cara a cara es el método más eficiente y efectivo para comunicar información dentro de un equipo de desarrollo.

7) El software que funciona es la principal medida del progreso.

8) Los procesos ágiles promueven un desarrollo sostenido. Los promotores, usuarios y desarrolladores deben poder mantener un ritmo de trabajo constante de forma indefinida.

9) La atención continua a la calidad técnica y al buen diseño mejoran la agilidad.

10) La simplicidad es esencial. Se ha de saber maximizar el trabajo que **no** se debe realizar.

11) Las mejores arquitecturas, requisitos y diseños surgen de los equipos que se han organizado ellos mismos.

12) En intervalos regulares, el equipo debe reflexionar con respecto a cómo llegar a ser más efectivo, y ajustar su comportamiento para conseguirlo.

Llegados a este punto, podemos intuir que las formas de hacer las cosas en la ingeniería del software están cambiando, adaptándose más a las personas y a las organizaciones en las que han de funcionar las aplicaciones.

¿Estamos en la antesala de una revolución? Posiblemente.

1.2. Metodologías ágiles más importantes

Son muchas las metodologías que poseen el calificativo de *ágiles*; algunas de ellas exploran diferentes principios para conseguir el objetivo de satisfacer plenamente las necesidades del sistema de información que se intenta implementar. Veamos algunas de las más importantes.

1.2.1. SCRUM

SCRUM es una metodología que **nace ajena al desarrollo del software**, de hecho sus principios fundamentales fueron desarrollados en procesos de reingeniería por Goldratt, Takeuchi y Nonaka en la década de 1980.

Principio de requisitos indefinidos de Humphrey

Para un nuevo aplicativo, los requerimientos no serán totalmente conocidos hasta que el usuario no lo haya usado.

Podríamos decir que SCRUM se basa en cierto "**caos controlado**" pero establece ciertos mecanismos para controlar esta indeterminación, manipular lo impredecible y controlar la flexibilidad.

Lema de Wegner

Es imposible definir completamente un sistema iterativo.

Se establecen tres fases:

1) pre-juego,
2) juego y
3) post-juego.

En el **pre-juego** se definen y/o revisan las funcionalidades que ha de tener el sistema, en el **juego** se distribuyen las tareas para cada miembro del equipo, se trabaja duro y se intenta conseguir el objetivo. Todos los miembros del equipo han de participar en una reunión diaria que en ningún caso deberá exceder los 30 minutos. En la fase de **post-juego** se evalúa la entrega de funcionalidades, se ven las tareas pendientes, se evalúa el progreso del proyecto y se redefine el tiempo de entrega del mismo si fuera necesario.

El principio de incertidumbre de Ziv

En la ingeniería del software: la incertidumbre es inherente e inevitable en el proceso de desarrollo de aplicaciones.

1.2.2. Dynamic Systems Development Method

Nace en 1994 con el objetivo de crear una herramienta RAD (*rapid applications development*) unificada, mediante la definición de un *framework* de desarrollo (sin propietario ni ánimo de lucro) para los procesos de producción de software.

DSDM

Se originó en Inglaterra y se ha ido extendiendo por Europa (no tanto por EE.UU.). Existe una organización que se encarga de su mantenimiento y desarrollo llamada DSDM Consortium.

DSDM se ha desarrollado teniendo como ideas fundamentales:

- Nada es construido a la perfección a la primera.
- La vieja **regla del 80-20** es cierta (el 80% de las funcionalidades del proyecto se realizan con el 20% del tiempo, y el 20% restante, los detalles, consumen el 80% del tiempo restante).
- Es improbable que alguien conozca todos los requisitos del sistema desde el primer día.

DSMD propone cinco fases, de las que sólo las tres últimas son iterativas a pesar de que hay retroalimentación en todas las fases.

a) Estudio de la viabilidad. Lo primero que se evalúa es si DSDM se puede o no aplicar al proyecto.

b) Estudio del negocio. Se estudia el negocio, sus características y la tecnología.

c) Modelado funcional. En cada iteración se planean los procesos funcionales del negocio sobre el prototipo.

d) Diseño y construcción. Aquí es donde se construye la mayor parte del sistema. El prototipo se vuelve apto para que los usuarios puedan utilizarlo.

e) Implementación. Pasamos de un prototipo a un sistema de producción. Se entrena a los usuarios para que lo usen.

La idea dominante en DSDM es contraria a la intuición inicial. En esta metodología, tiempo y recursos se mantienen como constantes y se ajusta la funcionalidad de acuerdo con ello. Es decir, la idea no es: "¿cuánto me va a costar desarrollar este sistema?", sino que más bien es: "con este tiempo y estos recursos ¿cuántas de las funcionalidades del sistema puedo hacer?".

1.2.3. Crystal Methodologies

No se trata de una única metodología sino de un conjunto de ellas centradas en las personas que tienen que desarrollar el software, **el equipo es la base** de estas metodologías creadas por Alistair Cockburn.

Web complementaria

http://alistair.cockburn.us/crystal/crystal.html

Desarrollar aplicaciones ha de ser como un juego en el que todos cooperan, aportan su parte de invención y se comunican. ¿Por qué olvidamos esta parte?

Crystal establece una serie de **políticas de trabajo en equipo (Methods)** orientadas a fomentar la mejora de estas habilidades. Dependiendo del tamaño del equipo, se establece una metodología u otra designadas por color: Crystal Clear (para 3-8 personas), Crystal Yellow (para 10-20 personas), Crystal Orange (para 25-50), etc.

Ejemplo

No es lo mismo cocinar para cuatro personas que para veinte; no es lo mismo planificar un fin de semana para dos personas que para cuarenta, entonces ¿por qué utilizamos la misma metodología para un grupo de tres desarrolladores que para un grupo de quince?

1.2.4. Feature-Driven Development

Impulsado por Jeff de Luca y Meter Coad, Feature-Driven Development (FDD) se basa en un **ciclo muy corto de iteración**, nunca superior a dos semanas, y en el que el análisis y los desarrollos están orientados a cumplir **una lista de "características"** (*features*) que tiene que tener el software a desarrollar.

Una "característica":

- Ha de ser muy simple, y poco costosa de desarrollar, entre uno y diez días.
- Ha de aportarle valor al cliente y ser relevante para su negocio.
- Debe poderse expresar en términos de <acción> <resultado> <objeto>.

Ejemplos de "características"

- Calcular el balance de situación.
- Conceder un crédito a un cliente de un banco.
- Comprobar la validez del PIN de una tarjeta.

La metodología sigue cinco fases iterativas: Desarrollo/Modificación de un Modelo Global; Creación/Modificación de la lista de "características"; Planificación; Diseño de la característica, e Implementación de la característica.

1.2.5. Adaptive Software Development

Esta metodología parte de la idea de que las necesidades del cliente son siempre cambiantes durante el desarrollo del proyecto (y posteriormente a su entrega). Su impulsor es Jim Highsmith, la novedad de esta metodología es que en realidad **no es una metodología de desarrollo de software**, sino un método (como un *caballo de troya*) a través del cual inculcar una cultura adaptativa a la empresa, ya que su velocidad de adaptación a los cambios marcará la diferencia entre una empresa próspera y una en declive.

Ejemplo

La incertidumbre y el cambio continuo son el estado natural de los sistemas de información, pero parece ser que muchas organizaciones aún no son conscientes de ellos. La idea de "finalizar" un proyecto carece de sentido porque debe seguir adaptándose.

Los objetivos de esta metodología son cuatro:

1) Concienciar a la organización de que debe esperar cambio e incertidumbre y no orden y estabilidad.
2) Desarrollar procesos iterativos de gestión del cambio.
3) Facilitar la colaboración y la interacción de las personas a nivel interpersonal, cultural y estructural.
4) Marcar una estrategia de desarrollo rápido de aplicaciones pero con rigor y disciplina.

1.3. ¿Cuándo puedo aplicar una metodología ágil?

Ya hemos visto una serie de metodologías ágiles, nos hemos dejado en el tintero muchas otras (*lean development, pragmatic programming*, etc.), pero aún no hemos visto cuándo debemos utilizar una metodología ágil.

¿Cómo? ¿No es siempre? Pues no. Como casi todo en esta vida, depende. Depende de cada proyecto en concreto, cada uno necesita de una metodología adecuada a él que le garantice el éxito. Necesita que se adecue no sólo a sus funcionalidades a desarrollar, sino además al equipo de desarrollo, a los recursos disponibles, al plazo de entrega, al entorno socio-cultural, etc.

Un buen profesional no debería cerrarse en una sola metodología de desarrollo, debería analizar el proyecto en concreto y ver cuál de todas las existentes sería la más adecuada a su proyecto. Y si ninguna le satisface plenamente, debería de ser capaz de adaptarlas. Las metodologías son para ayudarnos, pero nosotros debemos decidir cuándo y cómo es mejor aplicarlas.

Aun así, existen algunas reglas básicas, de "sentido común", para discernir si debemos aplicar una metodología tradicional o una metodología ágil. Pero recordad que son simplemente una guía, no un mandamiento.

Primero debéis responder a las siete preguntas siguientes:

1) ¿Cómo es de grande vuestro proyecto? Si es muy grande y/o involucra a muchas personas en el equipo, lo mejor es que se utilicen metodologías más estrictas y que se basen en la planificación y control del proyecto. Si se trata de un proyecto pequeño y/o con un equipo reducido de desarrollo, de tres a ocho personas, tenéis una oportunidad de experimentar con los métodos ágiles.

2) ¿Vuestros requisitos son dinámicos? Si os encontráis ante un ámbito de actuación donde los flujos de negocio están cambiando constantemente y tenéis que adaptarlos, como podría ser una gestión de fuerza de ventas, entonces la metodología ágil os ayudará a adaptaros. Si estáis en un ámbito en el que el dinamismo es escaso, por ejemplo, en la creación de un sistema de contabilidad, quizás deberíais utilizar una metodología orientada al plan que contemple los cambios previsibles y que os asegure minimizar el *reworking*.

Reworking

Son aquellas modificaciones que se realizan en la aplicación debido a los errores cometidos durante la codificación. No se trata de ampliaciones, sino de volver a realizar el trabajo por no cumplir la calidad necesaria.

3) ¿Qué pasa si vuestro sistema falla? Si os encontráis en un sistema crítico en el que cualquier fallo involucra pérdidas humanas o grandes cantidades de dinero, por favor, no utilicéis las metodologías ágiles. Si estáis diseñando un sistema para controlar un bisturí láser, no hay lugar para muchas pruebas, lo mejor es que funcione a la primera y las metodologías clásicas garantizan la calidad.

4) ¿Vuestro cliente tiene tiempo para dedicarlo al proyecto? Si no vamos a conseguir que nuestro cliente se involucre en la creación de un sistema que en el fondo es para él, nos será imposible aplicar una metodología ágil. En este caso lo mejor es utilizar una metodología en cascada con un gran análisis inicial.

5) ¿Cuántos júniors tenéis en vuestro equipo de desarrollo? Las metodologías ágiles requieren de una gran madurez, experiencia y una dosis de talento. Deben ser equipos con gente seniors o semi-seniors. Si vuestro equipo lo constituyen principalmente júniors, lo mejor es que no lo intentéis con las metodologías ágiles.

6) ¿Cuál es la cultura empresarial de la empresa en la que se va a desarrollar el proyecto? Las metodologías ágiles requieren de cierto ambiente "informal", que fomente la comunicación de igual a igual. Si la organización en la que se quiere desarrollar el proyecto tiene un alto grado de ceremonia y jerarquías estrictas, no deberíais utilizar las metodologías ágiles.

7) ¿Os apetece? Hay que saber si realmente se tienen ganas de probar una metodología ágil. Aprender cosas nuevas supone casi siempre un nuevo sacrificio.

Si vuestro equipo es pequeño y está formado mayoritariamente por gente con talento y experiencia, si el cliente final está involucrado y no impone barreras de comunicación, si los requisitos son altamente cambiantes, si no es proyecto crítico y no es demasiado grande, y os apetece probar una metodología ágil, no lo dudéis, es el momento de experimentar esta forma de diseñar y crear aplicaciones.

1.4. ¿Han muerto las metodologías tradicionales? ¿RUP is RIP?

Sin duda, la respuesta es no; siguen muy vivas y muy necesarias para grandes proyectos con grandes equipos de desarrollo o para entornos críticos como hemos visto antes, pero, sin embargo, ahora tienen una serie de competidores en el ámbito de los proyectos más manejables y que requieren de rápida adaptación y de resultados frecuentes.

De las metodologías clásicas la más famosa es RUP (*rational unified process*), algunos podrían aseverar que se opone radicalmente a lo que hemos visto hasta ahora, pero la realidad es que, como ya hemos dicho, cada proyecto tiene su metodología propia y en muchos casos múltiples metodologías pueden coexistir en el mismo proyecto. Rational ha trabajado estrechamente con DSDM Consortium y han creado documentos conjuntos que demuestran la compatibilidad del modelo **DSDM con RUP**.

Existen múltiples estudios de cómo utilizar **UML dentro de la filosofía de eXtreme Programming**; a pesar de que XP desenfatiza el uso de diagramas innecesarios, no hace falta diseñar cada clase, sino las más representativas y/o complejas. El **uso de patrones** no está prohibido en las metodologías ágiles, pero sí que está quizás menos extendido, ya que se prioriza una entrega rápida del producto y el análisis de patrones puede ralentizar el ciclo. Pero sin duda los patrones más repetitivos y extendidos deben utilizarse. Podemos disponer de una implementación de los mismos como parte de las herramientas con las que vamos a construir nuestro sistema a pesar de que no nos pongamos a analizar los patrones específicos del mismo.

Como podéis ver, cada proyecto, e incluso cada parte de un proyecto, debe tener su propia metodología independientemente de que sea ágil, o que sea orientada al plan; lo importante es que nos sirva y que sepamos aprovechar las diferentes cualidades que nos ofrecen.

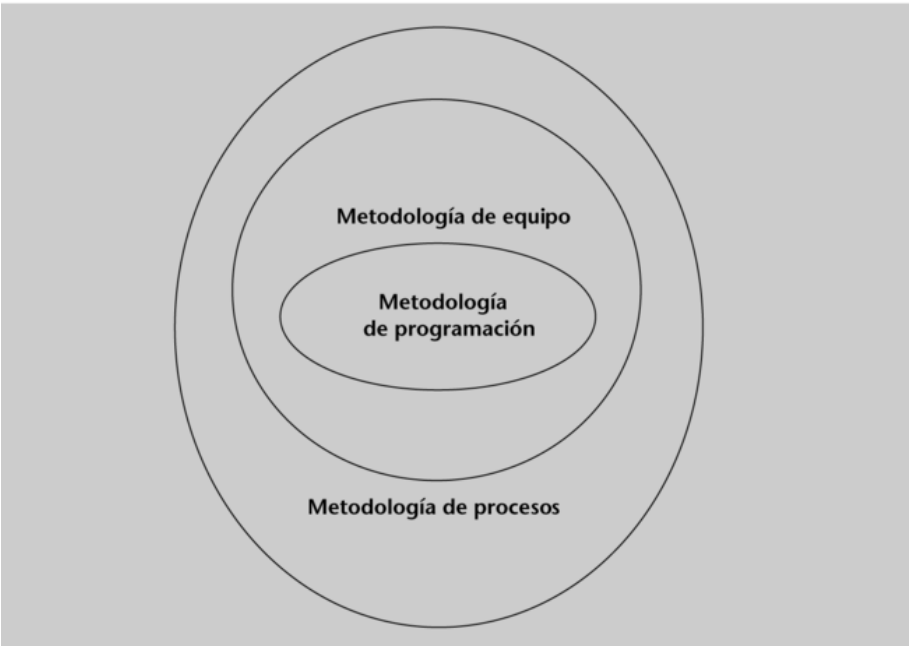
2. EXtreme Programming (XP)

La programación extrema (en adelante XP) puede que marque un antes y un después en la ingeniería del software. En este segundo punto del módulo, intentaremos mostrar todas sus ventajas y desventajas, así como su ambicioso punto de partida: **el software como solución ágil y no como proyectos arquitectónicos**.

Creada por Kent Beck, Ward Cunninghamn y Ron Jeffries a finales de los noventa, la programación extrema ha pasado de ser una simple idea para un único proyecto a inundar todas las "factorías de software". Algunos la definen como un movimiento "social" de los analistas del software hacia los hombres y mujeres de negocios, de lo que debería ser el desarrollo de soluciones en contraposición a los legalismos de los contratos de desarrollo.

¿Estamos viviendo una utopía? Quizás, pero actualmente esta metodología pasa de boca en boca como si de "algo secreto y pecaminoso" se tratase, y muchos jefes de proyectos están buscando la oportunidad para convencer a sus directivos y clientes de ponerla en práctica con algún proyecto piloto. Los resultados dictarán si XP pasa a impregnar el desarrollo del software de las próximas décadas o si simplemente nos quedaremos con la idea de aquella versión utópica de desarrollo de software.

Para alcanzar el objetivo de **software como solución ágil**, la metodología XP se estructura en tres capas que agrupan las doce prácticas básicas de XP:



1) Metodología de programación: diseño sencillo, test, refactorización y codificación con estándares.

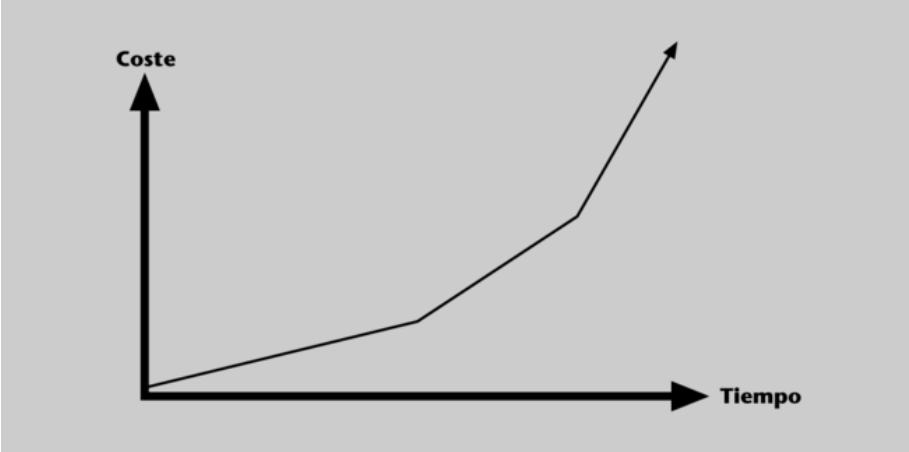
2) Metodología de equipo: propiedad colectiva del código, programación en parejas, integración continua, cuarenta horas semanales y metáfora del negocio.

3) Metodología de procesos: cliente *in situ*, entregas frecuentes y planificación del juego.

Introducir la vertiente de las relaciones sociales dentro de una metodología es lo que hace de XP algo más que una guía de buenas maneras. **Convierte la programación en algo mucho más "humanizado"**, en algo que permite a las personas relacionarse y comunicarse para encontrar soluciones, sin jerarquías ni enfrentamientos. Los analistas y programadores trabajan en equipo con el cliente final, todos están comprometidos con el mismo objetivo, que la aplicación solvente o mitigue los problemas que tiene el cliente. La vertiente social es fundamental en otras áreas del conocimiento: por ejemplo, en las relaciones del equipo médico con el paciente, o en las de bufete de abogados con un cliente, o en las de profesorado y alumnos, cada uno tiene su función pero el objetivo es común.

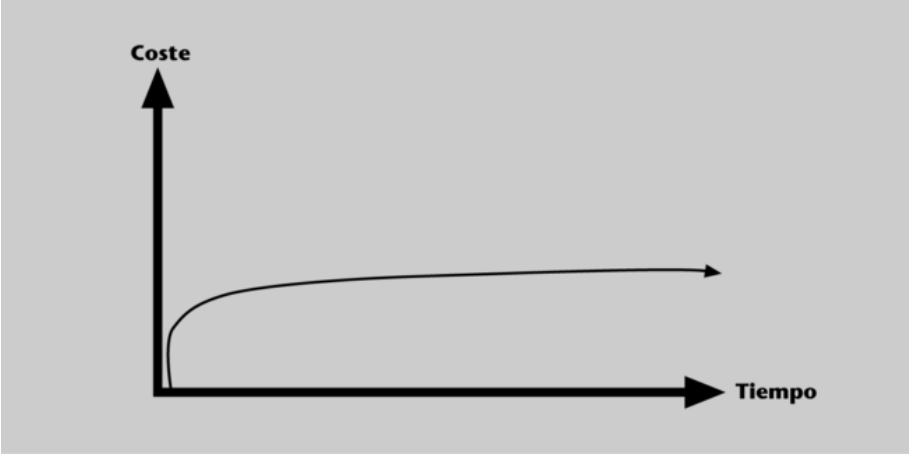
XP también humaniza a los desarrolladores. Un entorno agradable para el trabajo, que facilite la comunicación y los descansos adecuados, forma parte de esta metodología.

Pero donde XP centra la mayor innovación es en desmontar la preconcebida idea del coste del cambio de las metodologías en cascada, es decir, lo que cuesta cambiar alguna funcionalidad de nuestro aplicativo a medida que vamos avanzando en él. La idea generalizada es que cualquier modificación a final del proyecto es exponencialmente más costosa que al principio del mismo; si algo no estaba especificado inicialmente, cuesta mucho introducirlo al final del proyecto.



Lo que XP propugna es que esta curva ha perdido vigencia y que con una combinación de buenas prácticas de programación y tecnología es posible lograr que la curva no crezca siempre de forma exponencial.

XP pretende conseguir una curva de coste del cambio con crecimiento leve, que en un inicio es más costosa, pero que a lo largo del proyecto **permite tomar decisiones de desarrollo lo más tarde posible** sin que esa nueva decisión de cambio implique un alto coste en el proyecto.



Sólo al ver esta gráfica es cuando podemos entender el significado de uno de los lemas más repetidos en la metodología XP "Bienvenidos los cambios".

XP no se olvida de la necesaria rentabilidad de los proyectos, imprescindible en una economía tan competitiva como la occidental. Por eso propone una ecuación de equilibrio entre el coste, el tiempo de desarrollo, la calidad del software y el alcance de funcionalidades del mismo.

2.1. Las variables XP: coste, tiempo, calidad y alcance

El punto de partida de la metodología XP son las variables que utiliza para cada proyecto: **coste** (la inversión económica y en recursos), **tiempo** (el tiempo empleado, determinado por la fecha de entrega final), **calidad** (del código y del aplicativo desarrollado) y **alcance** (Conjunto de funcionalidades).

De estas cuatro variables, sólo tres podrán ser fijadas por el cliente y/o por el jefe de proyectos, la cuarta es responsabilidad del equipo de desarrollo y se establecerá en función de las otras tres.

¿Qué significa esto? Pues que se eliminan las imposiciones imposibles. Se eliminan frases del tipo:

"El cliente quiere estos requisitos satisfechos e implementados en dos meses; el equipo es éste y no habrá más, y el software debe superar todos los test de calidad, ya que es para una farmacéutica".

En este caso el cliente ha establecido dos variables, la de tiempo y los requisitos, y el jefe de proyecto ha fijado el equipo y la calidad. ¿En qué puede derivar esto? Pues simplemente en que seguramente la calidad no será la adecuada o que no se cumplirán todos los requisitos o que el equipo de desarrollo tendrá que hacer jornadas de catorce horas diarias.

El equipo de desarrollo tiene que tener voz y voto y la posibilidad de fijar al menos una de las cuatro variables. De esta manera podremos establecer un proyecto viable.

Obviamente con XP no se establece el valor de todas las variables a la primera de cambio, es un proceso paulatino en el que cada uno de los responsables (cliente, jefe de proyecto y equipo de desarrollo) negocian el valor de las variables que tienen asignadas hasta conseguir una ecuación equilibrada y que satisfaga a todos.

Advertencia: las interrelaciones entre estas cuatro variables no son siempre tan sencillas como parecen.

Ejemplo

Aumentar los recursos de equipos de hardware más eficientes puede que inicialmente disminuya el tiempo de desarrollo, pero seguir aumentando sus capacidades puede que ya no aporte ningún beneficio; o duplicar las personas del equipo de desarrollo no tiene por qué disminuir por dos el tiempo de implementación; es más, en algunos casos incluso puede agravar los problemas de coordinación y comunicación, incrementándolo en lugar de reducirlo.

También hay tareas que ya tienen todas sus variables fijadas de antemano por su propia naturaleza y sobre las que no podemos hacer nada para modificar la ecuación. En este sentido hay una frase muy conocida que refleja esta situación: "nueve mujeres no pueden tener un hijo en un mes". Por mucho que se intente, hay tareas que necesitan su tiempo y sus recursos para hacerlas.

Ejemplo

Otro ejemplo paradójico es el hecho de que en muchos casos aumentar la calidad acaba derivando en una disminución del tiempo de implementación, sobre todo en aquellos proyectos en los que se prevé una gran cantidad de cambios durante el desarrollo. No siempre bajar la calidad del software nos ahorrará costes.

Como podéis ver, las relaciones entre estas cuatro variables no son siempre evidentes, pero sí hay una pregunta que nos haremos siempre con la metodología XP: **¿Qué variable es la que debemos sacrificar?**

Pues depende, pero **la mayoría de las veces es el alcance**. La razón es simple. No siempre tenemos todo el tiempo del mundo, el coste es algo que generalmente se tiende a minimizar, y sin calidad todo el proceso de XP deja de tener significado, no podríamos mantener las aplicaciones si son un caos. Eso nos deja sólo una variable como la candidata más adecuada a sacrificar.

Distinguir entre los requisitos más importantes y los que aportan menor valor al negocio, dando prioridad a los primeros, nos ayudará a conseguir que el proyecto tenga en cada instante tanta funcionalidad como sea posible. De esta manera, cuando nos acerquemos al plazo de entrega nos aseguraremos de que lo principal está implementado y que sólo quedarán los detalles de los que podemos prescindir en el caso de necesidad. El objetivo es siempre maximizar el valor de negocio.

2.2. Los cuatro valores: comunicación, simplicidad, realimentación y coraje

Los creadores de esta metodología quisieron medir su utilidad a través de cuatro valores, que representan aquellos aspectos cuyo cumplimiento nos va a garantizar el éxito en el proyecto: comunicación, simplicidad, realimentación y coraje. Veamos qué significa cada uno de ellos:

a) Comunicación. Debe ser fluida entre todos los participantes en el proyecto; además el entorno tiene que favorecer la comunicación espontánea, ubicando a todos los miembros en un mismo lugar. La comunicación directa nos da mucho más valor que la escrita, podemos observar los gestos del cliente, o la expresión de cansancio de nuestro compañero.

b) Simplicidad. Cuanto más sencilla sea la solución, más fácilmente podremos adaptarla a los cambios. Las complejidades aumentan el coste del cambio y disminuyen la calidad del software. En XP nos vamos a olvidar de frases como "haremos un sistema genérico que...", o "esto lo pongo por si acaso algún día lo necesitamos".

> Sólo se utiliza lo que en ese momento nos da valor, y lo haremos de la forma más sencilla posible.

Alguno de vosotros puede pensar que eso va en contra de toda la filosofía de diseño y utilización de patrones. Nada más alejado de la realidad. En un proyecto XP, el uso de patrones nos va a ayudar a reducir el tiempo de implantación, pero lo que no vamos a hacer es dedicar tiempo a la implementación de patrones que no vayamos a utilizar en este proyecto; sólo haremos los que sean necesarios para éste, no utilizaremos tiempo del proyecto para beneficiar

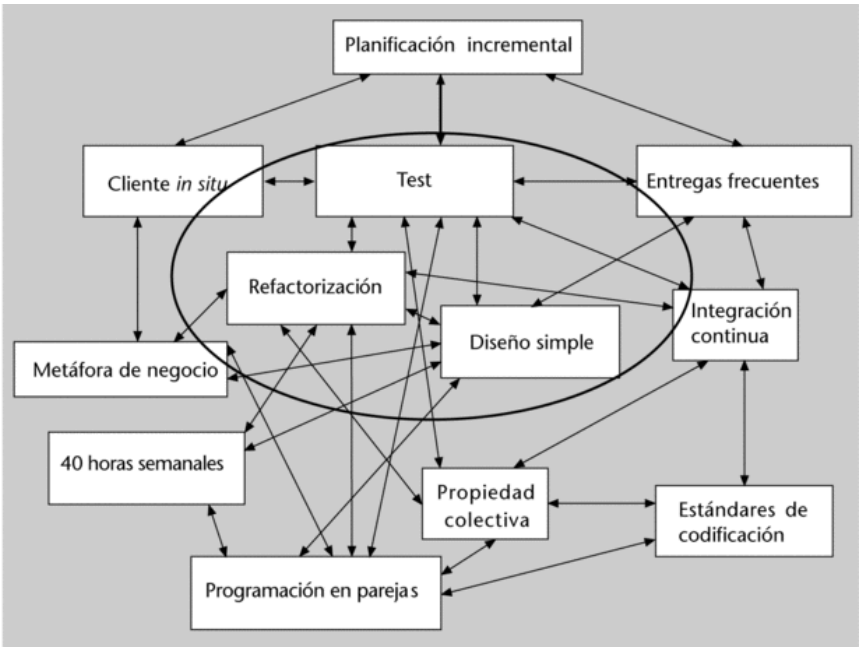
a otro proyecto futuro que quizás no llegue nunca. Por otro lado, nada nos impide desarrollar un proyecto que únicamente se dedique a desarrollar patrones que más tarde se utilicen en proyecto XP.

c) Realimentación. El usuario debe utilizar desde la primera entrega el software desarrollado, dándonos sus impresiones y sus necesidades no satisfechas, de manera que esas historias vuelvan a formar parte de los requisitos del sistema.

d) Coraje. Coraje para vencer la frase más típica de los desarrolladores: "si funciona no lo toques". Con XP debemos tocar continuamente cosas que ya funcionan, para mejorarlas. Hemos de cambiar esta frase por la de: "si funciona, puedes mejorarlo". Y eso, os lo aseguramos, requiere de mucho valor y coraje.

2.3. Las doce prácticas básicas de XP

Kent Beck, Ward Cunninghamn y Ron Jeffries tenían muy claro las prácticas que les habían dado mejores resultados en sus proyectos. Así que intentaron aplicarlas todas juntas dando una vuelta de tuerca. Ése fue el embrión de la metodología XP. Crearon las doce prácticas que se refuerzan entre sí para obtener los mejores resultados. Las relaciones entre ellas las podemos ver en el siguiente gráfico que hemos adaptado del que propuso Kent Beck.



En el centro hemos situado las prácticas que más resultados nos pueden dar al adaptarlas; no son otras que el diseño simple, el test y la refactorización. Incluso si no queremos tomar la totalidad de las prácticas de XP adoptando estas tres a nuestra metodología habitual, podemos tener una sustancial mejora en los resultados obtenidos.

2.3.1. Diseño simple

"Debe ser simple para ser cierto. Si no es simple, probablemente no podremos descifrarlo."

Albert Einstein

Nada tan acertado como esta frase para ilustrar la idea de diseño simple de XP. Si nuestro diseño es simple, cuando alguien lo vea lo entenderá, si es complejo, probablemente no pueda descifrarlo.

El principio es "utilizar el diseño más sencillo que consiga que todo funcione", de esta manera facilitaremos el mantenimiento y minimizaremos los riesgos de modificaciones que se hagan sin "entender" el código.

XP define un "**diseño tan simple como sea posible**" como aquel que:

- No tiene código redundante, ni duplicado.
- Supera todos los tests de funcionalidad, integridad y aceptación.
- No utiliza sintaxis complejas, es decir, que queda clara la intención de los programadores en cada línea de código.
- Contiene el menor número posible de clases y métodos.

2.3.2. Refactorización

Así como en el Génesis nos explican que en un principio todo era caos y luego todo fue orden, cuando programamos generalmente pasa lo contrario. Inicialmente todo son líneas de código bien ordenadas y comentadas, pero a medida que vamos introduciendo cambios, el orden se va perdiendo hasta que aquello deriva en una serie de líneas de código caóticas, llenas de código comentado de las diferentes pruebas y con comentarios obsoletos que no se corresponden con la realidad de la funcionalidad.

El excesivo coste de las modificaciones de las metodologías tradicionales se debe en gran medida a este **deterioro progresivo del código**, tras la acumulación de modificaciones.

Para mantener la curva del coste de cambio tan plana como sea posible, en la metodología XP se aplica **la refactorización**, que no es otra cosa que modificar el código para dejarlo en buen estado, volviendo a escribir las partes que sean necesarias pero siempre desde un punto de vista global a la funcionalidad, independientemente del cambio que hagamos.

El código final debe conservar la claridad y sencillez del original.

Los comentarios son muy importantes para mantener esta claridad y sencillez.

2.3.3. Test

Es el pilar fundamental de las prácticas de esta metodología, sin los test, XP sería un fracaso total, es el punto de anclaje que le da la base metodológica a la flexibilidad de XP.

Si una funcionalidad no se ha testeado, sólo funciona en apariencia, los tests han de ser aplicados tras cada cambio, y **han de ser automatizados**. Si no lo hacemos, podemos incurrir en fallos humanos a la hora de testearlos y eso puede resultar fatal.

El objetivo de los tests no es detectar errores, sino evitarlos, no se trata de corregir errores, sino de prevenirlos.

Para ello, los tests siempre se escriben antes que el código a testear, nunca después. De esta manera estamos obligados a **pensar por adelantado** cuáles son los problemas que podemos encontrar cuando usen nuestro código, a ponernos en el lugar del usuario final, y este hecho de pensar por adelantado evita muchos problemas, previniéndonos sobre ellos en lugar de dejar que aparezcan y luego responder sobre la marcha.

Cuando escribimos el código, ya sabemos a qué se ha de enfrentar y de esta manera los errores se minimizan. El objetivo de nuestro software ya no es cumplir unas funcionalidades sino pasar unos tests.

Es por esto por lo que el usuario final debe ayudar al programador a desarrollar los tests de forma conjunta, ya que en estos tests estará implícita la funcionalidad que queremos que tenga. Otro factor clave es que debe ser el mismo programador el que desarrolle los tests, si no es así, perdemos la ventaja de minimización de errores.

Los tests han de ser de tres tipos: tests de aceptación, tests unitarios y tests de integridad; y siempre han de estar automatizados.

- **Test de aceptación**. Es creado conjuntamente con el cliente final y debe reflejar las necesidades funcionales del primero.

- **Test unitario**. Es creado por el programador para ver que todos los métodos de la clase funcionan correctamente.

- **Test de integridad**. Es creado por el equipo de desarrollo para probar que todo el conjunto funciona correctamente con la nueva modificación.

El uso de los tests nos facilita la refactorización, permitiéndonos comprobar de forma sencilla que los cambios originados por la refactorización no han cambiado el comportamiento del código.

2.3.4. Estándares de codificación

Web complementaria

http://java.sun.com/docs/codeconv/index.html

El equipo de desarrollo debe tener unas normas de codificación común, unas nomenclaturas propias que todos los miembros del equipo puedan entender. Por ejemplo, si el programa ha de escribirse en Java, lo mejor es que todos utilicen las "Code Conventions for the Java Programming Language" de Sun o una adaptación propia de estas normas.

El hecho de utilizar una nomenclatura común permite que cualquier persona del equipo entienda con mayor facilidad el código desarrollado por otro miembro; de esta manera, facilitamos las modificaciones y la refactorización.

Por ejemplo, si decidimos que a la hora de poner nombre a nuestras variables de una función seguiremos la siguiente norma:

1) La primera letra ha de ser una "v" si es variable local o una "p" si nos viene por parámetro de entrada o una "o" si es parámetro de entrada y salida.

2) La segunda letra ha de indicar el tipo de datos "n" si es numérico, "s" si es una cadena de caracteres, "d" si es una fecha y "b" si es booleano.

3) El resto del nombre ha de ser descriptivo con su contenido lógico, separando significados con el "_".

Si yo ahora en el código veo la siguiente sentencia:

```
If (vbPeriodo_no_calculable) { odFinal_Periodo = pdInicio_Periodo;}
```

tengo mucha más información que si hubiese visto la siguiente:

```
If (pnc) { fp = ip;}
```

El funcionamiento es el mismo y ambas sentencias están bien codificadas, pero la primera nos da el valor añadido de la comprensión de la codificación estándar.

2.3.5. Propiedad colectiva del código

Para poder aplicar la refactorización y para asegurarnos de que el diseño es simple y que se codifican según los estándares, tenemos que eliminar otra de las ideas que están muy arraigadas en el mundo del desarrollo de aplicaciones; la "propiedad individual" del código.

Frases como "que lo modifique quien lo hizo que seguro que lo entiende mejor" o "¿quién ha tocado mi función?" dejan de tener sentido en XP, ya que el diseño simple nos garantiza que será fácilmente entendible, la refactorización permite que cualquier miembro del equipo rehaga el código para devolverle su sencillez en el caso de que se haya complicado, los test automatizados nos garantizan que no hemos modificado el comportamiento esperado del código con nuestra modificación, y la codificación con estándares nos da ese grado de comprensión adicional.

> En XP el código es propiedad de todo el equipo y cualquier miembro tiene el derecho y la obligación de modificarlo, para hacerlo más eficiente o comprensible, sin que nadie se tenga por qué sentir ofendido.

2.3.6. Programación por parejas

¿No iríamos más rápido con dos personas programando en lugar de una programando y otra mirando? La verdad es que inicialmente puede chocar, la programación siempre se ha visto como algo solitario, y tener dos personas delante de un solo teclado y de un solo monitor sorprende en un inicio. Pero las ventajas son muchas.

Pensad en cómo funciona la mente humana: nos es muy complicado pensar a nivel abstracto y luego pasar a pensar a nivel concreto, y si lo hacemos de forma continua, acabamos desconcentrados y cometemos errores. Es por esto por lo que, cuando programamos, primero pensamos la estrategia de codificación que vamos a seguir y luego nos ponemos a codificar cada una de las partes, sin pensar de nuevo a nivel estratégico hasta que no finalizamos alguna de las partes o a veces la totalidad del código, y entonces vemos los errores o las cosas que nos hemos dejado en la estrategia de codificación original.

> Pensar a lo grande y en detalle a la vez nos es imposible con un solo cerebro. Pues pongamos dos.

En la programación en parejas uno de los miembros debe esta pensando a nivel táctico y el otro a nivel estratégico de manera que esos dos procesos siempre estén activos reduciendo así los errores y mejorando la calidad del pro-

grama. Obviamente, estos dos roles deben intercambiarse cada poco tiempo entre los miembros de la pareja para abarcar todas las posibilidades tácticas y estratégicas.

El nivel de los miembros de la pareja ha de ser equivalente, no sirve que uno sepa mucho y otro no tenga ni idea, deben de estar equilibrados y obviamente llevarse bien para que tenga éxito.

También la rotación ha de ser muy importante, cada miembro del equipo ha de ser capaz de trabajar en cualquier área de la aplicación que se esté desarrollando. De esta manera no provocaremos cuellos de botella cuando asignemos las tareas.

El hecho de que asignemos las tareas por parejas hace que el tiempo de aprendizaje se reduzca casi a cero, simplemente sustituyendo uno de los miembros por otro nuevo. Así pues, la rotación de áreas y de parejas nos garantiza que podremos hacer un reparto más equitativo del trabajo sin tener que depender de una sola persona para un trabajo específico.

Otro efecto que produce la programación en parejas es el psicológico, disminuye la frustración de la programación en solitario, tienes a alguien que entiende el problema justo al lado, y con el que compartir el problema. Además, muchas veces los problemas vienen por falta de concentración y se tiene la solución delante de las narices y no se ve, y se pierde mucho tiempo. La programación por parejas disminuye la necesidad del "Efecto tótem".

"**Efecto tótem**"

El programador lleva varias horas intentando solucionar un problema, decide llamar a un compañero para ver si él ve la solución. Empieza a explicarle el problema y automaticamente encuentra la solución sin que el segundo compañero haya pronunciado palabra alguna. Recibe este nombre porque podríamos sustituir al segundo compañero por un "tótem" indio con exactamente los mismos resultados.

2.3.7. Integración continua

En XP no esperamos a que todas las partes estén desarrolladas para integrarlas en el sistema, sino que a medida que se van creando las primeras funcionalidades ya se ensamblan en el sistema, de manera que éste puede ser construido varias veces durante un mismo día. Esto se hace para que las pruebas de integración vayan detectando los errores desde el primer momento y no al final de todo. El impacto es mucho menor.

Es responsabilidad de cada equipo publicar lo antes posible cada funcionalidad o cada modificación. La idea es que todos los miembros del equipo trabajen con la última versión del código.

2.3.8. 40 horas semanales

No se pueden trabajar durante 14 horas seguidas y hacerlo con calidad. Las semanas de 70 horas de trabajo son contraproducentes. Los equipos de XP están diseñados para ganar, no para morir en el intento. Al final de la semana se tiene que llegar cansado pero satisfecho, nunca exhausto ni desmotivado. Trabajar horas extra mina la moral y el espíritu del equipo. Si durante dos semanas hay que hacer horas extras, entonces es que el proyecto va mal y se debe replantear alguna de las cuatro variables.

2.3.9. Metáfora del negocio

Para que dos o más personas se puedan comunicar de forma eficiente, deben tener el mismo vocabulario y compartir el mismo significado. El modelo de negocio que entiende el usuario final seguramente no se corresponderá con el que cree entender el programador. Es por esto por lo que en los equipos de XP se debe crear **una "metáfora" con la que el usuario final se encuentre cómodo** y que le sirva al equipo de desarrollo a la hora de modelizar las clases y métodos del sistema.

> La metáfora es una historia común compartida por el usuario y el equipo de desarrollo que describe cómo deben comportarse las diferentes partes del sistema que se quiere implementar.

2.3.10. Cliente *in situ*

En más de una ocasión, cuando estamos programando o analizando el sistema, nos surge una duda y pensamos en que cuando veamos al usuario final se la preguntaremos. Posiblemente tengamos que seguir trabajando sin solventar esa duda y, si nuestra suposición ha sido errónea, mucho del trabajo realizado se puede perder.

XP necesita que el cliente final forme parte del equipo de desarrollo y esté ubicado físicamente en el mismo sitio para que así se agilice el tiempo de respuesta y se puedan validar todas las funcionalidades lo antes posible.

> En XP, el cliente **siempre** tiene que estar disponible para el resto del equipo, formando parte de él y fomentando la comunicación cara a cara, que es la más eficiente de las comunicaciones posibles.

2.3.11. Entregas frecuentes

Se deben desarrollar lo antes posible versiones pequeñas del sistema, que aunque no tengan toda la funcionalidad, nos den una idea de cómo ha de ser la entrega final y que nos sirvan para que el usuario final se vaya familiarizando con el entorno y para que el equipo de desarrollo pueda ejecutar las pruebas de integridad.

> Las nuevas versiones tienen que ser tan pequeñas como sean posibles, pero tienen que aportar **un nuevo valor de negocio para el cliente**.

Dar valor al negocio es lo que hará que la visión final del cliente sea la mejor posible.

2.3.12. Planificación incremental

La planificación nunca será perfecta, variará en función de cómo varíen las necesidades del negocio y en cada ciclo de replanificación se volverán a establecer las cuatro variables de la metodología XP.

Asumir una planificación estática no corresponde con la agilidad que queremos dar, ya que las necesidades del negocio pueden cambiar drásticamente mientras estamos desarrollando el aplicativo. En XP la planificación se va revisando continuamente, de forma incremental, priorizando aquellas necesidades de negocio que nos aporten mayor valor.

> La planificación se plantea como un permanente diálogo entre los responsables de la perspectiva empresarial y de la perspectiva técnica del proyecto.

2.4. El ciclo de vida de la metodología XP

Las doce prácticas habían dado por separado muy buenos resultados a Kent Beck y compañía. Unificarlas en el ciclo de vida de una metodología es lo que dio origen a XP.

La mejor forma de comunicación es la presencial. Cuando tenemos algo importante que decir, nos gusta "decirlo a la cara", para que no haya equívocos. La expresión, la entonación y las miradas son muy importantes.

> El ciclo de vida de XP se organiza como si fuese una conversación cliente-desarrollador.

1) Tengo una necesidad.

2) ¿Puedo ayudarte? ¿Qué necesitas?

3) Lo que necesito es esto.

4) Creo que lo entiendo. ¿Si construyo esto te sirve?

5) Pues sí. ¿Podrías hacerlo?

6) Ahora mismo me pongo a trabajar.

7) ¿Has acabado?

8) Estoy en ello.

9) Necesitaría también esto.

10) ¿Es crucial? Tardaría mucho tiempo.

11) Entonces no lo pongas.

12) Ya tengo esto. ¿Te sirve?

13) No del todo.

14) ¿Y esta nueva versión?

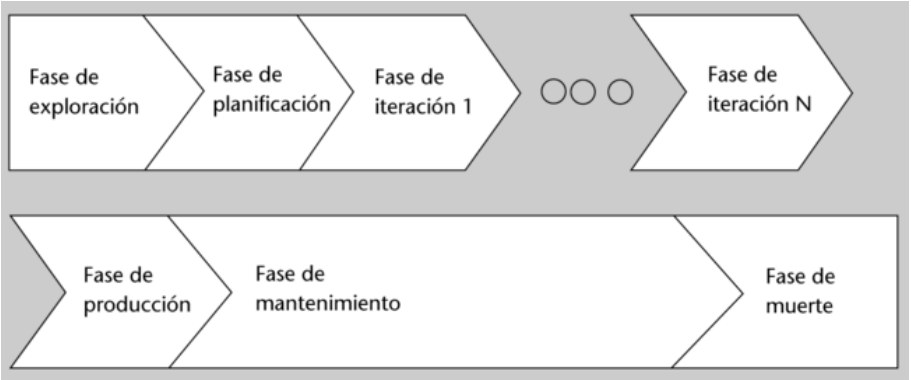
15) Si, es justo lo que necesito. Gracias.

16) De nada. Si tienes problemas, avísame.

Éste sería el desarrollo ideal de un proyecto XP. Para acercarnos a esto, establecemos un ciclo de vida dividido en seis fases.

Ciclo de vida de un proyecto XP

1) Fase de exploración
2) Fase de planificación
3) Fase de iteraciones
4) Fase de producción
5) Fase de mantenimiento
6) Fase de muerte del proyecto



2.4.1. La fase de exploración

La fase de exploración es la primera fase del ciclo de vida de la metodología XP. En ella se desarrollan tres procesos:

1) Las historias de usuario.
2) El *spike* arquitectónico.
3) La metáfora del negocio.

Todo comienza con las "historias de usuario". En esta fase los usuarios plantean a grandes rasgos las funcionalidades que desean obtener del aplicativo. Las historias de usuario tienen el mismo propósito que los casos de uso, salvo en un punto crucial; las escriben los usuarios y no el analista. Han de ser descripciones cortas y escritas en el lenguaje del usuario sin terminología técnica. Estas historias son las que guiarán la creación de los tests de aceptación que han de garantizar que dichas historias se han comprendido y se han implementado correctamente.

No debemos confundir las historias de usuario con el análisis de requisitos, la principal diferencia está en la profundidad de análisis, con los requisitos queremos llegar al último detalle para no "pillarnos las manos ante el cliente", pero en XP el cliente forma parte del equipo y le podemos preguntar más cosas durante la implementación, así que el nivel de detalle en las historias de usuario ha de ser el mínimo imprescindible para que nos hagamos una idea general de la funcionalidad.

Las **historias de usuario** han de ser:

- Escritas por el cliente final; en su lenguaje y sin tecnicismos.
- Descripciones cortas de la usabilidad y funcionalidad que se espera del sistema.

Paralela y conjuntamente se empieza con el *spike* arquitectónico, en el que el equipo de desarrollo empieza a familiarizarse con la metodología, herramientas, lenguaje y codificaciones que se van a usar en el proyecto.

En el ***spike* arquitectónico** del equipo de desarrollo:

- Prueba la tecnología.
- Se familiariza con la metodología.
- Se familiariza con las posibilidades de la arquitectura.
- Realiza un prototipo que demuestre que la arquitectura es válida para el proyecto.

Una vez finalizadas las historias de usuario y el *spike* arquitectónico, se pasa a desarrollar conjuntamente la metáfora del negocio.

Ved también

Como ya vimos en el apartado "Metáfora del negocio" de este módulo, esta metáfora es una de las 12 prácticas básicas de XP.

La **metáfora del negocio**:

- Es una historia común compartida por el usuario y el equipo de desarrollo.
- Debe servir para que el usuario se sienta a gusto refiriéndose al sistema en los términos de ella.
- Debe servir a los desarrolladores para implementar las clases y objetos del sistema.

2.4.2. La fase de planificación

El resultado ha de ser una planificación (recordemos que siempre flexible) del proyecto.

El procedimiento es el siguiente:

- El cliente entrega al equipo de desarrollo las **historias de usuario** que ha confeccionado, pero priorizándolas de mayor a menor importancia.
- El equipo de desarrollo las estudia y **estima el coste** de implementarlas:
  - Si el equipo de desarrollo considera que la historia de usuario es demasiado compleja, entonces el usuario final debe descomponerla en varias historias independientes más sencillas.
  - Si el equipo de desarrollo no ve claro cómo implementar una parte de la historia, el usuario puede realizar un ***spike* tecnológico** para ver cómo se podría implantar y así poder evaluar el coste.
- Una vez tenemos la lista de historias priorizadas junto con su coste de implementación, pasamos a convocar la **reunión del plan de entregas**.

– El plan de entregas se compone de una serie de planes de iteración en el que se especifica qué funcionalidades se van a implementar en cada vuelta de la fase de iteraciones.

– Participan de esta reunión tanto los usuarios como el equipo técnico y cada uno debe aportar su visión del negocio de manera que se obtengan más rápidamente aquellas funcionalidades que den el mayor beneficio para el negocio posible.

– A cada iteración se le asigna un tiempo intentando que todas sean más o menos idénticas (aunque no es necesario).

– Se determina el alcance del proyecto.

2.4.3. La fase de iteraciones

Como hemos dividido el proyecto en iteraciones, esta fase se repetirá tantas veces como iteraciones tengamos. Generalmente, cada iteración suele ser de dos a tres semanas.

El **Plan de iteración** se trata de la manera siguiente:

- Se recogen las historias de usuario asignadas a esta iteración.

- Se detallan las **tareas a realizar** por cada historia de usuario.
  – Las tareas deben ser de uno o tres días de desarrollo. Si son más grandes, deberíamos intentar dividirlas en varias más sencillas.

  – Se estima el coste de cada tarea. Si el total es superior al tiempo de iteración, se deberá prescindir de alguna historia de usuario que se pasaría a la siguiente iteración. Si son muchas las historias de usuario desechadas, entonces hay que volver a estimar las cuatro variables de la metodología y volver a planificar el proyecto.

  – Si el tiempo total estimado de las tareas es inferior al tiempo de iteración, se puede asumir una historia de usuario que correspondiese a la siguiente iteración.

- Se priorizan las tareas que más valor darán al negocio, intentando que se finalicen historias de usuario lo antes posible.

- Se reparten las primeras tareas al equipo de desarrollo y el resto se deja en una **cola de tareas sin asignar** de dónde se irán cogiendo.

- Se convocan **reuniones de seguimiento diarias** para ver si nos vamos retrasando en las estimaciones o nos vamos adelantando a ellas y así poder desechar o incorporar historias de usuario.

Lo más importante es que en cada momento de cada iteración estemos realizando la tarea que más valor posible da al negocio de entre las que tenemos pendientes, de manera que, si tenemos que reducir el alcance del proyecto, sólo afecte a las funcionalidades secundarias de nuestro aplicativo.

2.4.4. La fase de producción

Llegamos a esta fase al alcanzar la primera versión que el usuario final decida que puede ponerse en producción.

Pasaremos el aplicativo a producción cuando alcance las funcionalidades mínimas que aporten un valor real al negocio y una operativa arquitectónica estable.

Es decir, no esperamos a tener todas las funcionalidades implementadas, sino que en cuanto tenemos algo que los usuarios pueden utilizar y que ayuda al negocio, pasamos la primera versión a producción.

Paralelamente, se sigue con las iteraciones finales de proyecto, pero fijaos que antes de que finalice ya estamos dando valor a la organización, el ROI (retorno de la inversión) del proyecto empieza a generarse antes de que éste finalice su versión final.

En la etapa de producción se realizan también iteraciones como en la anterior etapa, pero el ritmo de éstas ya no es de dos a tres semanas, sino mensuales.

Esta fase se mantiene hasta que realizamos la última entrega, con la que finalizamos el ámbito del aplicativo y pasamos al mantenimiento del mismo.

Durante la fase de producción, el ritmo de desarrollo decae debido a que el equipo debe solventar las incidencias de los usuarios. Es por esto por lo que a veces es necesario incorporar nuevo personal al equipo.

2.4.5. La fase de mantenimiento

Una vez el alcance del proyecto se ha conseguido, y tenemos todas las funcionalidades en producción, se revisan con el usuario aquellas nuevas historias de usuario que se han producido tras la puesta en producción del proyecto. Estas nuevas funcionalidades se van incorporando según su valor de negocio y el presupuesto adicional del que se disponga.

El equipo de desarrollo se reduce a la mínima expresión, dejando algunos miembros para el mantenimiento.

2.4.6. La fase de muerte del proyecto

Cuando no existen más historias de usuario para introducir en nuestro sistema o cuando se reduce progresivamente valor de las historias de usuario implementadas en él, el proyecto entra en la fase de muerte.

Se irá desinvirtiendo en él hasta abandonarlo totalmente cuando no aporte valor al negocio o cuando sus historias de usuario hayan sido absorbidas por otro sistema de información.

2.5. Los diferentes roles dentro de XP

Cada rol tiene unas funciones claras dentro de la metodología XP. Cada persona del equipo puede ejecutar uno o varios roles, o incluso cambiar de rol durante las diferentes fases del proyecto.

Son muchas las extensiones que se han hecho de los roles de la propuesta original de Beck, pero los roles que permanecen siempre en cualquier implementación de la metodología XP son los siguientes:

Programador:

- Escribe las pruebas unitarias.
- Produce el código del programa.

Cliente:

- Escribe las historias de usuario.
- Diseña las pruebas de aceptación.
- Prioriza las historias de usuario.
- Aporta la dimensión de negocio al equipo de desarrollo.
- Representa al colectivo de usuarios finales.
- Está siempre disponible para consultas.

Encargado de pruebas (*tester*):

- Ayuda al cliente a diseñar pruebas de aceptación.
- Ejecuta las pruebas de aceptación.
- Ejecuta las pruebas de integración.
- Difunde los resultados entre el equipo de desarrollo y el cliente.
- Es el responsable de las herramientas automatizadoras de las pruebas.

Encargado de seguimiento (*tracker*):

- Se encarga de realimentar todo el proceso de XP, midiendo las desviaciones con respecto a las estimaciones y comunicando los resultados para mejorar las siguientes estimaciones.
- Realiza el seguimiento de cada iteración del proceso de XP tanto en la etapa de iteraciones como en la de producción.
- Revalúa la posibilidad de incorporar o eliminar historias de usuario.

Entrenador (*coach*):

- Se encarga del proceso global.
- Garantiza que se sigue la filosofía de XP.
- Conoce a fondo la metodología.
- Provee guías y ayudas a los miembros del equipo a la hora de aplicar las prácticas básicas de XP.

Consultor:

- No forma parte del equipo.
- Tiene un conocimiento específico de un área en concreto.
- Ayuda a resolver un problema puntual, ya sea de *spike* tecnológico o de valor de negocio.

Gestor (*boss*):

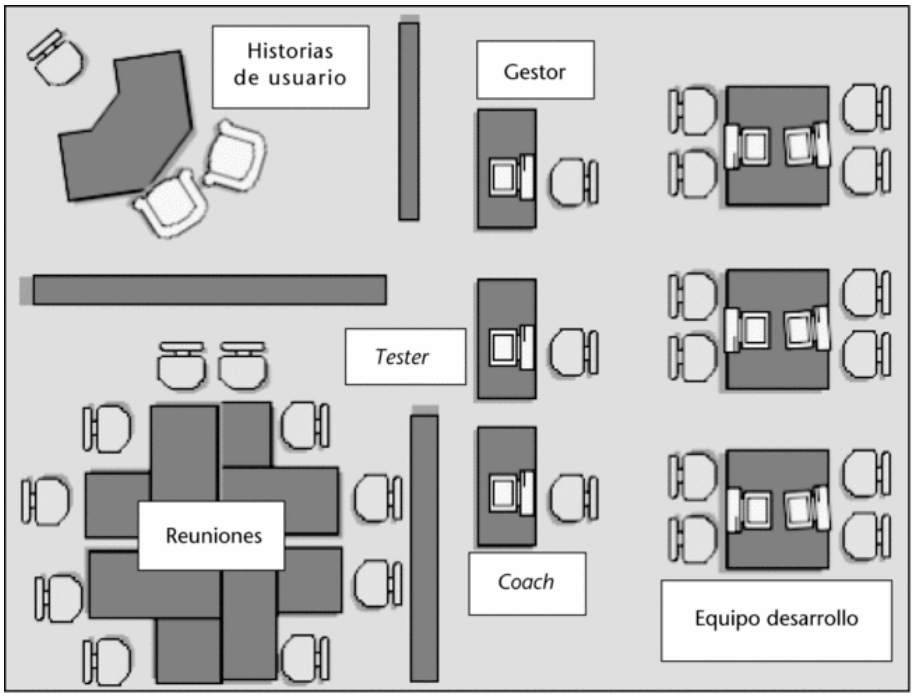
- Es el máximo responsable del proyecto.
- Hace de enlace con los clientes.

Se encarga de coordinar y de garantizar las condiciones necesarias para el desarrollo del trabajo.

2.6. El entorno de trabajo

XP necesita, además, de un entorno que favorezca la comunicación. De nada sirve tener a nuestro equipo disperso por diferentes plantas y edificios, o que el cliente se encuentre a cinco minutos por el pasillo principal de la oficina. Las barreras arquitectónicas y visuales deben eliminarse en la medida de lo posible de un proyecto de XP. La comunicación verbal es fundamental.

Una posible distribución de un equipo de XP podría ser la siguiente:



A este esquema final sería conveniente añadirle una sala de desconexión donde los programadores fatigados pudieran hacer un descanso, tomarse un refresco, picar algo o simplemente mirar por la ventana.

Otro punto importante es que el cliente debe estar cerca y disponible pero debe tener un entorno que le permita desarrollar su trabajo habitual. Por esta razón, en el esquema se encuentra un poco apartado del grueso del equipo.

2.7. Qué dicen los detractores de XP

No todo es un camino de rosas para la metodología XP. Mucho se ha escrito también en contra de esta forma de entender la creación de software. Veamos los principales argumentos en contra.

a) XP sólo funciona con equipos "brillantes" de gente disciplinada y con visión de negocio. Si tenemos a programadores "normales", al aplicar XP obtendremos el efecto contrario, desarrollo mucho más lento y plagado de errores.

b) Las empresas no dejarán que un cliente se integre al proyecto de desarrollo, y si conseguimos que lo consientan, seguramente será aquel usuario final que tenga menor valor para el negocio y que no represente los problemas de la empresa. Así, las historias de usuario y su asignación de prioridades no serán representativas y construiremos nuestro aplicativo sobre una falacia.

c) Cuando se va a hacer un proyecto en una empresa, lo primero que se pide es saber el tiempo y el alcance, antes de aceptar un proyecto. Y con XP ambas son variables a lo largo del proyecto. Sin haberlas prefijado, no se pueden hacer proyectos para terceras empresas y menos optar a un concurso público.

d) XP se basa en las pruebas y los tests, pero en un "entorno real", con la presión que suelen sufrir los proyectos de desarrollo, lo primero que se reduce a su mínima expresión son las pruebas. XP no es aplicable a un entorno de una organización real.

e) La programación en parejas es una utopía. Al final deriva en que uno trabaja y el otro descansa. Además, es de difícil justificación de cara al cliente que paga el software, ya que la medida hora/hombre es la que impera en el mercado. ¿Cómo facturaremos si se ve que de cada dos personas, una sólo apunta cosas a la otra?

f) Los equipos XP han de ser pequeños, no se podría realizar con equipos de proyectos grandes.

g) Deja de lado la documentación, algo fundamental para controlar la alta rotación en las empresas servicios de TI.

Como podemos ver, argumentos no faltan, saber si son sólidos o no, depende ya de vuestro espíritu crítico. Pero eso sí, recordad que, antes de criticar o alabar esta metodología de proyectos, quizás sería conveniente que empezáramos a aplicarla en algún proyecto poco importante, para así podernos formar una opinión fundamentada en nuestra experiencia.

2.8. Un ejemplo de XP

Java es sin duda el lenguaje de programación más accesible a los estudiantes y programadores no profesionales. Además, JEE es la plataforma de desarrollo más extendida entre los movimientos de las comunidades de software libre. Por esta razón, unida a que ambos movimientos están muy vinculados entre sí, hemos creído conveniente realizar un ejemplo de lo que sería un proyecto XP en un entorno puramente JEE.

Para ello tenemos como "voluntaria" a la ya famosa "tienda de mascotas" que junto con el "Hola, mundo" son los dos ejemplos más utilizados en la historia reciente de la informática.

Por desgracia no podemos extendernos demasiado en la descripción detallada de la codificación, pero sí que intentaremos mostrar las decisiones más importantes durante el ciclo de vida del proyecto.

2.8.1. Fase de exploración

El cliente se pone en contacto con nosotros y nos comunica que necesita crear una web de su tienda de mascotas, para lo que cuenta con un presupuesto limitado. El equipo de desarrollo lo compondrán una o dos personas, que realizarán todos los roles dentro de la metodología XP.

Lo primero que vamos a hacer es darle un nombre al proyecto: Proyecto Extremo de Tienda de Animales y vamos a esperar que las siglas no sean un indicativo del resultado final.

El segundo paso es solicitar y ayudar al cliente a escribir las **historias de usuario**.

Historia de usuario 1: Presencia en Internet
Quiero que la gente que busque mi tienda de mascotas pueda encontrarla buscando en el Google, y así hallar mi dirección, mi correo electrónico, y los teléfonos de contacto.

Historia de usuario 2: Catálogo de productos
Además, me gustaría que los clientes pudieran ver mis productos y los precios en la web.

Historia de usuario 3: Pedidos de productos
Y comprarlos.

Historia de usuario 4: Carrito de la compra
No de uno en uno, sino varios a la vez.

Historia de usuario 5: Listas de productos
Y que el programa se acordase de los productos que más compra.

Historia de usuario 6: Productos vinculados
Y que le sugiriese nuevos productos para comprar, según los que haya comprado.

Historia de usuario 7: Ofertas y promociones
Y poder destacar las ofertas de la semana y descuentos al comprar varios productos a la vez.

Historia de usuario 8: Tarjetas cliente
Y que acumulasen puntos por comprar por Internet para descuentos o regalos.

Historia de usuario 9: Pago por móvil
Y que pudieran pagar el pedido con una llamada de móvil en lugar de poner la tarjeta de crédito.

Paralelamente hemos iniciado el ***spike* arquitectónico** basado en JEE.

JEE

Ved JEE, JavaBeans y los Enterprise JavaBeans en la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del Grado de Ingeniería en Informática.

Los dos miembros del equipo de desarrollo se decantan por una arquitectura clásica en tres capas para el proyecto web.

La única duda la tenían en si utilizar sólo JavaBeans o Enterprise JavaBeans a la hora de la persistencia de los datos de las transacciones económicas. Para ello contactan con varios bancos que realizan pagos *on-line* y descubren que en la mayoría de los casos lo único que deben hacer es redireccionar a las páginas propias del banco cuando se vaya a realizar el pago y una vez finalizado, el banco redireccionará el cliente a la página de la tienda que les indiquen. Con esta información, hacen algunas pruebas y se decantan por JavaBeans simplificando así la arquitectura.

Los dos miembros del equipo de desarrollo están familiarizados con las herramientas OpenSource y tras hacer varias pruebas, eligen las siguientes:

Webs complementarias

Para ampliar información sobre las herramientas OpenSource, podéis ver: http://ant.apache.org
http://www.junit.org
http://tomcat.apache.org/
http://www.jboss.org/

- **Ant**. Herramienta OpenSource de codificación y compilación de programas Java.
- **Junit**. Herramienta OpenSource de creación y ejecución automatizada de tests unitarios sobre clases Java y especializada en Extreme Programming.
- **Tomcat**. Como servidor web, OpenSource, donde residirá la aplicación, permite el uso de Java Server Pages. Se ha descartado Jboss, ya que hemos decidido que no necesitaremos EJBs.

Todas las herramientas elegidas son compatibles entre sí y están plenamente integradas.

El último paso es desarrollar la **metáfora del negocio** para que usuario y equipo de desarrollo se sientan a gusto y se sepa que se está hablando en los mismos términos.

- *Dominio*. Nombre por el que se conocerá nuestra tienda en Internet por ejemplo: "tiendadeanimales".
- *Tienda on-line*. Se trata de la página web que será nuestra representación en Internet del negocio de la tienda de mascotas. Es el punto de entrada. Al poner la dirección en el navegador http://www.tiendadeanimales.es aparecerá nuestra tienda.

- *Menú*. Son las diferentes opciones que tendremos a la hora de ver la información de la tienda. Sería como las diferentes puertas de acceso a la tienda normal.

- *Secciones*. La tienda *on-line* está organizada en secciones, igual que la tienda física. En cada sección encontraremos productos.

- *Barra de navegación*. Sería como caminar por la tienda física. En la tienda *on-line* pondremos una zona que permitirá ir al siguiente producto, ver todos los productos de la sección, cambiar de sección, etc.

- *Catálogo de productos*. Lo componen todos los productos que tendremos en la tienda *on-line*. Sería como el catálogo impreso, pero con la capacidad de poder cambiar las páginas fácilmente.

- *Productos*. Corresponden a los productos que venderemos en la tienda *on-line*. Al igual que en la tienda física, un producto puede exponerse en una o más secciones.

- *Carrito de la compra*. Corresponde a la zona de la tienda *on-line* donde iremos depositando los productos que queremos comprar.

- *Proceso de pago*. Su funcionamiento es muy parecido a la siguiente idea. Imagina que en la tienda real los clientes no nos pagan a nosotros en la caja registradora, sino que se van con el carrito de la compra al banco, pagan en el banco, y el banco luego nos lo paga a nosotros. El intercambio de dinero no lo hacemos en nuestra tienda *on-line* sino en la del banco.

- *Demás conceptos*. Que el usuario y el equipo de desarrollo crean necesitar con tantas analogías y/o metáforas como se crea necesarias.

2.8.2. Fase de planificación

El cliente **prioriza** las historias de usuario.

Historia de usuario 1: Presencia en Internet
Historia de usuario 2: Catálogo de productos
Historia de usuario 6: Productos vinculados
Historia de usuario 3: Pedidos de productos
Historia de usuario 7: Ofertas y promociones
Historia de usuario 4: Carrito de la compra
Historia de usuario 8: Tarjetas cliente
Historia de usuario 5: Listas de productos
Historia de usuario 9: Pago por móvil

El equipo de desarrollo **evalúa el coste** de implementación de cada una de ellas en jornadas de ocho horas por pareja de programadores.

Historia de usuario 1: Presencia en Internet (10 jornadas)
Historia de usuario 2: Catálogo de productos (20 jornadas)
Historia de usuario 6: Productos vinculados (4 jornadas)
Historia de usuario 3: Pedidos de productos (6 jornadas)
Historia de usuario 7: Ofertas y promociones (10 jornadas)
Historia de usuario 4: Carrito de la compra (10 jornadas)
Historia de usuario 8: Tarjetas cliente (4 jornadas)
Historia de usuario 5: Listas de productos (4 jornadas)
Historia de usuario 9: Pago por móvil (2 jornadas)

Tras la reunión del **plan de entregas** se establece que la iteración será bisemanal (10 jornadas) y que los miembros del equipo efectivamente serán dos.

La planificación es la siguiente:

- Iteración 1
  Historia de usuario 1: Presencia en Internet (10 jornadas)
  Primer paso a producción

- Iteración 2
  Historia de usuario 2: Catálogo de productos con las secciones más importantes (10 jornadas)

- Iteración 3
  Historia de usuario 2: Catálogo de productos con las secciones restantes (10 jornadas)
  Segundo paso a producción

- Iteración 4
  Historia de usuario 6: Productos vinculados (4 jornadas)
  Historia de usuario 3: Pedidos de productos (6 jornadas)

- Iteración 5
  Historia de usuario 7: Ofertas y promociones (10 jornadas)
  Tercer paso a producción

- Iteración 6
  Historia de usuario 4: Carrito de la compra (10 jornadas)
  Cuarto paso a producción

Quedan fuera del ámbito del proyecto las siguientes historias de usuario:

Historia de usuario 8: Tarjetas cliente (4 jornadas)
Historia de usuario 5: Listas de productos (4 jornadas)

Historia de usuario 9: Pago por móvil (2 jornadas)

Es decir, seis iteraciones de dos semanas cada una, con cuatro pases a producción. En esta planificación a partir de la segunda semana de inicio del proyecto ya estaremos dando valor al negocio, al haber implementado la historia de usuario 1 que era la más importante en la priorización del cliente.

2.8.3. Fase de iteraciones

En el ejemplo actual, esta fase finalizaría en la primera iteración, ya que en ese momento pasaríamos a la fase de producción. Lo más importante de esta primera fase es crear el armazón del aplicativo final, sobre el que iremos construyendo todo; es decir, además de la página web que cubriría la historia de usuario 1, deberíamos crear también las clases Java: Tienda, Sección, Catálogo, Producto, Carrito, y crear sus primeros tests unitarios y sus primeros tests de integración.

Si nos centramos en la clase Producto, una posible primera implementación podría ser esta:

```
/* Clase producto */
/* Versión 1.0 */
package uoc.tienda;
public abstract class Producto {
  public void setPrecio(double precio) {this.precio = precio; }
  public double getPrecio(){ return precio }
  protected double precio;
}>
```

Tras finalizar de implementar el otro miembro de la pareja, podría decir que la función setPrecio no es del todo clara, que sería mejor utilizar la nomenclatura del proyecto.

- La primera letra ha de ser una "v" si es variable local o una "p" si nos viene por parámetro de entrada o una "o" si es parámetro de entrada y salida.

- La segunda letra ha de indicar el tipo de datos "n" si es numérico, "s" si es una cadena de caracteres, "d" si es una fecha y "b" si es booleano.

- El resto del nombre ha de ser descriptivo con su contenido lógico, separando significados con el "_".

Así, la clase Producto quedaría de la siguiente manera.

```
/* Clase producto */
/* Versión 1.1 */
package uoc.tienda;

public abstract class Producto {
  public void setPrecio(double pnPrecio) {this.vnPrecio = pnPrecio; }
  public double getPrecio(){ return vnPrecio }
  protected double vnPrecio;
}
```

Por supuesto esto no se queda aquí, porque recordemos que uno de los dos programadores ha de estar pensando en estratégico, así que ve de forma clara que todos los objetos del proyecto tendrán tres propiedades fijas: Identificador, Nombre y Descripción, e incita a crear un clase con estas tres propiedades de las que herede el resto.

```
/* Clase producto */
/* Versión 1.2 */
package uoc.tienda;

public abstract class Producto extends ObjetoBasico{
  protected double vnPrecio;
  public void setPrecio(double pnPrecio) {this.vnPrecio = pnPrecio; }
  public double getPrecio(){ return vnPrecio }
}
```

```
/* Clase Objeto Básico*/
/* Versión 1.0 */
package uoc.tienda;

public abstract class ObjetoBasico {
  protected int vnIdentificador;
  protected String vsNombre;
  protected String vsDescripcion;

  /* Propiedad Identificador */
  public void setIdentificador(int pnIdentificador)
 throws Exception { this.vnIdentificador =
pnIdentificador; }
  public int getIdentificador(){ return vnIdentificador; }

/* Propiedad Nombre*/
  public void setNombre(String psNombre){ this.vsNombre = psNombre; }
  public String getNombre(){ return vsNombre; }

  /* Propiedad Descripcion*/
  public void setDescripcion(String psDescripcion)
 { this.vsDescripcion = psDescripcion; }
  public String getDescripcion(){ return vsDescripcion; }

}
```

No debemos olvidarnos de hacer los tests unitarios para cada uno de los objetos. Para ello, utilizaremos las clases de JUnit, en especial la clase abstracta Assert que nos permite gran variedad de comprobaciones, y de su implementación en la clase TestCase. Otra clase interesante es TestSuite, que nos permite automatizar una gran cantidad de tests unitarios.

Web complementaria

La jerarquía de la clase Assert la podemos encontrar en:
http://junit.sourceforge.net/javadoc/org/junit/package-tree.html

Siguiendo con el ejemplo, la primera versión del test unitario de la clase Producto sería la siguiente:

```
package uoc.tienda.test;

import.uoc.tienda.Producto;
import junit.framework.TestCase;

public class TestUnitarioProductoextends TestCase {
  public void testProducto( Producto pPTest) {
  // Creo otra instancia de un producto
  final Producto producto1 = new Producto;
  // Compruebo la creación de objetos para ver que se han
  instanciado bien
 assertNotSame(producto1, pPTest);
 assertSame(producto1, producto1);
 assertSame(pPTest, pPTest);
// Le asigno las propiedades a producto 1 los métodos set
  producto1.setPrecio(pPTest.getPrecio());
// correcto y concordante con el método set
  assertEquals(pPTest.getPrecio(), producto1.getPrecio());
//Por último compruebo las propiedades lógicas que ha de cumplir la
clase Producto
// Propiedad 1: Precio nunca ha de ser menor que cero.
  assertFalse(pPTest.getPrecio() < 0);
// Propiedad 2: Precio nunca ha de ser mayor que 999.999.
  assertTrue(pPTest.getPrecio() < 999999);
}
}
```

Al finalizar esta primera iteración, tendríamos que tener la página de entrada inicial y un armazón de todo el aplicativo, así como los tests unitarios y de integración, y las automatizaciones de dichos tests.

La ultima parte de esta primera fase son los **tests de aceptación** creados por el cliente ayudado por el miembro del equipo con el rol de *tester*. Si se pasan con éxito, pondremos esta iteración en producción. En nuestro caso, los tests de aceptación deberían reflejar las necesidades de la Historia de usuario 1: Presencia en Internet.

La creación de tests y los algoritmos de testeo darían para todo un apartado de este libro, pero por desgracia está fuera del alcance de esta introducción.

2.8.4. Fase de producción

Durante esta fase se realizarían las iteraciones 2 a la 6, según el plan de entregas que habíamos diseñado. Hacia la mitad de la segunda iteración, nos damos cuenta de que estamos avanzando más de lo esperado y que en lugar de tardar las 10 jornadas previstas, tardaremos 8 jornadas, de manera que nos planteamos asumir ciertas tareas de la siguiente iteración de la que estimamos ahora que tardaremos también 8 jornadas en lugar de 10. De manera que la planificación queda así.

- Iteración 1
  Finalizada

- Iteración 2
  Historia de usuario 2: Catálogo de productos con las secciones más importantes (8 jornadas)

Historia de usuario 2: Catálogo de productos con las secciones restantes. Parte1 (2 jornadas)

- Iteración 3
  Historia de usuario 2: Catálogo de productos con las secciones restantes. Parte 2 (6 jornadas)
  Historia de usuario 6: Productos vinculados (4 jornadas)
  Segundo paso a producción

- Iteración 4
  Historia de usuario 3: Pedidos de productos (6 jornadas)
  Historia de usuario 7: Ofertas y promociones. Parte 1 (4 jornadas)

- Iteración 5
  Historia de usuario 7: Ofertas y promociones. Parte 2 (6 jornadas)
  Tercer paso a producción

- Iteración 6
  Historia de usuario 4: Carrito de la compra (10 jornadas)
  Cuarto paso a producción

Nos queda ahora una iteración 5 con sólo 6 jornadas y con un hueco para 4 jornadas adicionales. Después de hablar con el cliente, decidimos incluir en ese hueco una de las funcionalidades que habían sido descartadas, la historia de usuario 8: Tarjetas cliente (4 jornadas). Así, la iteración 5 queda de la siguiente manera.

- Iteración 5
  Historia de usuario 7: Ofertas y promociones. Parte 2 (6 jornadas)
  Historia de usuario 8: Tarjetas cliente (4 jornadas)
  Tercer paso a producción

Durante las siguiente iteraciones no hay ninguna incidencia y el proyecto finaliza tal y como se había replanteado en la iteración 2.

2.8.5. Fase de mantenimiento

Durante los dos primeros meses, tras la finalización de la fase de producción se fueron subsanando algunas deficiencias, sobre todo de rendimiento del aplicativo, ya que en XP se prima que las cosas funcionen primero y que luego sean eficientes. Una vez realizadas estas mejoras y debido a que el número de visitas a la tienda real había crecido de forma espectacular, se decide invertir un poco más e implementar la historia de usuario 9: Pago por móvil (2 jornadas).

2.8.6. Fase de muerte

Durante cinco años la web funcionó hasta que la empresa fue absorbida por una multinacional de la venta de mascotas.

3. Ser ágiles no es solo programar ágiles

¿De qué nos serviría poseer una gran elasticidad en las piernas si no pudiéramos doblar las rodillas? Exactamente lo mismo nos pasa en el desarrollo de sistemas de información. De nada nos sirve ser ágiles en la creación de software si no los somos en el resto de facetas organizativas de nuestro trabajo.

La agilidad está llegando a todas las facetas de la ingeniería del software:

- Agile Modeling nos permite realizar modelos que se adaptan a las metodologías aquí vistas. El éxito le viene dado por su gran adaptabilidad, ya que puede ser usada tanto en metodologías RUP, como en XP o cualquier otra metodología de desarrollo de software.

Web complementaria

Agile Modeling (http://www.agilemodeling.com) pretende ser "una guía para el arte de modelar ", nunca para la ciencia de modelar.

- Agile Data Method nos ayuda en la creación y mantenimiento de las bases de datos que dan soporte a los proyectos ágiles en los que el cambio es constante y a veces traumático para los administradores de las mismas.

Web complementaria

http://www.agiledata.org/

- Agile Documentation nos ayuda a centrarnos en la documentación fundamental de los proyectos ágiles que utilicen Agile Modeling.

Web complementaria

http://www.agilemodeling.com/essays/agileDocumentation.htm

- Agile Legacy Integration Modeling nos permite diseñar de forma ágil modelos de aplicaciones que se integren con sistemas heredados tipo host.

- Agile Cualquiercosa. Son muchas las iniciativas que se están desarrollando en torno a los diferentes aspectos que faltan por cubrir, pero sin duda muchos autores están aprovechando el tirón de las metodologías ágiles para poder dotar sus proyectos de la etiqueta de moda. Sólo el tiempo dirá quiénes eran visionarios y quiénes eran simples oportunistas.

Resumen

En este módulo hemos visto cómo cada proyecto (o parte de él) puede realizarse con una metodología diferente. Es responsabilidad nuestra elegir la mejor adaptada a cada situación en concreto. Muchas son las metodologías ágiles, pero hemos visto que todas tienen características y objetivos similares. Unas hacen más hincapié en la comunicación humana de los proyectos, otras en dar valor al negocio lo antes posible, otras en asumir que todo es cambiante, pero todas son útiles en determinados proyectos, cubren una necesidad o carencia que se ha detectado en las metodologías de desarrollo de aplicaciones.

De entre ellas hemos conocido la que más fuerza ha cobrado: la metodología Extreme Programming. Los cuatro valores sobre los que se fundamenta, las doce prácticas que propone y las cuatro variables de los proyectos nos han hecho constatar que se trata de una metodología muy madura, nada despreciable y perfectamente aplicable en una gran variedad de proyectos. También hemos visto que XP es compatible con otras metodologías ya sean ágiles como DSDM, o clásicas como RUP.

Nos hemos asomado a los diferentes ámbitos en los que la agilidad está irrumpiendo con fuerza, siendo el más prometedor de ellos el Agile Modeling. Conocer todos los ámbitos y todas las propuestas nos da un punto de referencia para la reflexión, nos permite saber en qué pueden fallar las metodologías y así poder aplicarlas con mayor criterio.

Con este módulo, hemos vislumbrado todo un nuevo abanico de posibilidades a la hora de diseñar, implementar y gestionar proyectos de ingeniería del software. De vosotros depende ahora ponerlas o no en práctica.

Actividades

1. Describid tres proyectos en los que trabajaríais con metodologías ágiles. Elegid cuál de ellas utilizaríais y comentad en qué fundamentaríais esa decisión.

2. De los tres proyectos de la actividad anterior, elegid uno para desarrollarlo mediante XP. Diseñad las historias de usuario, la metáfora del negocio y planificad las fases.

3. Cread una metodología propia que se adapte al manifiesto ágil y explicad cómo la aplicaríais en otro de los proyectos de la primera actividad.

Ejercicios de autoevaluación

1. ¿Las metodologías ágiles son siempre la mejor opción para la mayoría de proyectos?

2. ¿Las metodologías clásicas y las metodologías ágiles son incompatibles?

3. ¿Cuándo no deberíamos aplicar una metodología ágil?

4. De las doce prácticas de XP, ¿cuáles tienen relación con una metodología de equipo? ¿Cuáles son las tres más importantes?

5. En XP, ¿en qué consiste el *spike* arquitectónico de la fase de exploración?

6. ¿Cuáles son los cuatro valores del "manifiesto ágil"?

Solucionario

Ejercicios de autoevaluación

1. Las metodologías ágiles no son la gran solución a todos los problemas del desarrollo de aplicaciones, ni tan siquiera se pueden aplicar en todos los proyectos, pero sí que nos aportan otro punto de vista de cómo se pueden llegar a hacer las cosas, de forma más rápida, más adaptable y sin tener que perder la rigurosidad de las metodologías clásicas.

2. Rotundamente no. Se pueden compaginar perfectamente, incluso en un mismo proyecto; de hecho, existen múltiples estudios de cómo aplicar DSDM y XP con RUP en un mismo proyecto.

3. En los casos siguientes:

- Con un equipo muy grande y/o formado mayoritariamente por gente sin experiencia.
- Si el cliente final no está involucrado y/o impone barreras de comunicación.
- Si los requisitos son apenas cambiantes.
- Si es un proyecto crítico.
- Si es un proyecto demasiado grande.
- Si no os apetece probar una metodología ágil.

4. Las prácticas que tienen relación con la parte metodológica de trabajo en equipo son: propiedad colectiva del código, programación en parejas, integración continua, 40 horas semanales y metáfora del negocio.

Las tres más importantes de las doce prácticas son el diseño simple, el test y la refactorización. Incluso, si no queremos adoptar la totalidad de las prácticas de XP, adoptando estas tres a nuestra metodología habitual podemos tener una sustancial mejora en los resultados obtenidos.

5. En el *spike* arquitectónico, el equipo de desarrollo:

- Prueba la tecnología.
- Se familiariza con la metodología.
- Se familiariza con las posibilidades de la arquitectura.
- Realiza un prototipo que demuestre que la arquitectura es válida para el proyecto.

6. Los cuatro valores del "manifiesto ágil" son los siguientes:

- Valorar más al individuo y sus interacciones más que al proceso y las herramientas.
- Valorar más el desarrollo de software que funciona que obtener una buena documentación.
- Valorar más la colaboración con el cliente que la negociación de un contrato.
- Valorar más responder a los cambios que seguir una planificación.

Glosario

característica *f* Funcionalidad simple y poco costosa de desarrollar que aporta valor al cliente del software a utilizar.
en feature

historias de usuario *f pl* Descripciones cortas de la usabilidad y funcionalidad que se espera del sistema, escritas por el cliente final, en su lenguaje y sin tecnicismos.

Manifiesto ágil *m* Manifiesto que recoge los valores que deberían cumplir las metodologías de desarrollo de software. Fue firmado por Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland y Dave Thomas.

metáfora del negocio *f* Historia común compartida por el usuario y el equipo de desarrollo que describe cómo deben comportarse las diferentes partes del sistema que se quiere implementar.

metodologías ágiles *f pl* Metodologías que cumplen el manifiesto ágil.

refactorización *f* Modificar el código para dejarlo en buen estado, volviendo a escribir las partes que sean necesarias, pero siempre desde un punto de vista global a la funcionalidad independientemente del cambio que hagamos.

***spike* arquitectónico** *m* Periodo durante el cual el equipo de desarrollo, prueba la tecnología y se familiariza con la metodología que se aplicará durante el proyecto

test de aceptación *m* Test creado conjuntamente con el cliente final que debe reflejar las necesidades funcionales del primero.

test de integridad *m* Test creado por el equipo de desarrollo para probar que todo el conjunto funciona correctamente con la nueva modificación.

test unitario *m* Test creado por el programador para ver que todos los métodos de la clase funcionan correctamente.

Bibliografía

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley.

Beck, K.; Martin F. (2001). *Planning Extreme Programming*. Boston: Addison-Wesley.

Burke, E. M.; Coyner, B. M. (2003). *Java™ Extreme Programming Cookbook*. Sebastopol: O'Reilly & Associates Inc.

Canós, J. H.; Letelier, P.; Penadés, M. C. (2004). *Metodologías ágiles en el desarrollo del software*. Valencia: Universidad de Valencia.

Cockburn, A. (2001). *Agile Software Development*. Boston: Addison-Wesley.

Cronin, G. (2003). *eXtreme Solo: A Case Study in Single Developer eXtreme Programming*. Auckland: University of Auckland.

Highsmith, J. (2002). *Agile Software Development Ecosystems*. Boston: Addison-Wesley.

Hightower, R; Lesiecki, N. (2002). *Java Tools for Extreme Programming*. Nueva York: Wiley & Sons Inc.

Reynoso, C. (2004). *Métodos heterodoxos en desarrollo del software*. Buenos Aires: Universidad de Buenos Aires.

Salo, J. H.; Abrahamson, P.; Ronkainen, J.; Warsta, J. (2002). *Agile Software Development Methods - Review And Analysis*. Universidad de Oulu: VTT Publications.

Schwaber, K.; Beedle, M. (2002). *Agile Software Development with Scrum*. Nueva Jersey: Prentice Hall.

Wake, W. C. (2000). *Extreme Programming Explored*. Boston: Addison-Wesley.

Wallace, D.; Raggett, I.; Aufgang, J. (2002). *Extreme Programming for Web Projects*. Boston: Addison Wesley.