José Carlos Cortizo Pérez, Diego Expósito Gil y Miguel Ruiz Leyva

eXtreme Programming

ÍNDICE

Este libro nació como un simple trabajo sobre lo que en un principio pensamos era una simple metodología, otra más, podríamos decir, pero pronto nos dimos cuenta que no era así. Quizás ahora seamos en parte XP-depedientes, algunos más que otros, eso es evidente, pero lo que si es seguro es que descubrimos que más que una simple metodología, lo que se denomina "Extreme Programming", es más bien una filosofía o una manera de acercarte a los problemas.

En este libro (deberíamos decir escrito, pero la palabra libro nos da la sensación de estar haciendo algo más importante) hablaremos de "Extreme Programming" o símplemente XP, pero nosotros hemos llegado a denominarlo "Visión Extrema", ya que para nosotros XP no aporta una manera de enfocar la programación (incluyendo el análisis, diseño, . . .), sino que nos aporta una visión de cómo afrontar cualquier problema, de hecho, en este libro se han seguido ciertas filosofía de la XP, de hecho en nuestra vida de estudiantes hemos practicado la programación en parejas, la simplicidad en el diseño (cuándo lo hacíamos), hemos seguido lo más posible la filosofía de nunca hacer lo que no es estríctamente necesario, porque siempre da más trabajo y menos beneficios.

Realmente, viendo "XP", aunque solo sea por encima, cualquier persona se da cuenta que es una aglutinación, coherente, de prácticas de aplicación lógica, sobre todo cuando lo que se está haciendo se debería haber terminado ayer. Quizás es lo que más nos ha gustado de esto, algo tan simple y útil que consigue que te plantees el por qué no se te habría ocurrido a ti antes (dejando cuestiones de edad, porque con 8 años estas teorías se nos escapaban un tanto a la razón).

Nosotros confiamos en la "XP" (vamos a denominarla en femenino, porque debería ser la programación extrema en castellano). Y estamos seguros que conseguirá muchos adeptos, quizás porque es una de estas cosas que aportan bastante a nivel técnico (a pesar de ser símple) pero además te aportan una cierta filosofía y manera de vida así como un cambio de las estructuras a existentes. Esto es lo que ha hecho que surgieran los fenómenos Linux, Java, . . . que son sistemas de gran valor técnico, con grandes cualidades, pero presentan un lado totalmente nuevo, intentan cambiar los cimientos de lo que había antes y nos dan el "merchandising" que todos tanto amamos (desde la simples gorras y pegatinas a el poder decir yo soy de...).

También queremos, desde aquí, agradecer a toda la comunidad de "Extreme programmers" que brindan, en internet, una cantidad ingente de información sobre el tema para que cualquiera que desee interesarse en el tema sea capaz de meterse de lleno en él sin necesidad de investigar mucho. Además debemos agradecer la ayuda prestada por los foros de discusión sobre el tema respondiendo a todas nuestras preguntas de una forma bastante rápida y ofreciéndonos conocimiento muy interesante sin el cuál este libro no hubiera sido lo mismo.

Esperando que os guste:

José Carlos Cortizo Pérez
Diego Expósito Gil
Miguel Ruiz Leyva

1.1 EXTREME PROGRAMMING

"Extreme programming" o, castellanizando, la programación extrema es una metodología ligera para la elaboración de software, pero, podemos decir bastante más acerca de ella para acercar, a todo aquel que esté interesado en ella, todo lo que nos puede ofrecer.

XP como metodología La filosofía de la XP

Definir lo que es la XP de una forma formal es tarea dificultosa debido a que su propia definición es rebatida por lo que con ella se intenta lograr. Básicamente se podría decir que la programación extrema es una "metodología ligera[1] o ágil para el desarrollo de software eficiente y altamente efectivo"[2].

La gran controversia en la definición de la programación extrema viene de la mano de la palabra metodología. Muchos defensores a ultranza de la XP consideran que no podemos denominarla metodología porque elimina toda la parte burocrática anteriormente asociada a las metodologías para el desarrollo de software (continua documentación, diseños varios, y papeleos excesivos). Por otro lado, obviar que nos proporciona una guía o un proceso repetitivo para el desarrollo de software sería obviar lo que en sí es. Así pues, se puede tomar como bueno el consenso de denominarla como metodología ligera (o ágil), debido a que elimina la "pesada" carga del papeleo a los desarrolladores.

XP COMO METODOLOGÍA

Básicamente, la programación extrema, busca dos objetivos claramente: hacer un software bien (con calidad) y de la forma más rápida posible. De hecho estos son los objetivos fundamentales de cualquier metodología aplicada al desarrollo de software y a cualquier otro área en general. A pesar de esto, con las metodologías de desarrollo actuales, el 70% de los proyectos fracasan y aproximadamente, también, el 70% de los fallos no son debidos a cuestiones técnicas, son debidos a cambios en la gestión o problemas de comunicación.[3]

Con estos datos es lógico pensar en que las metodologías actuales no son lo suficientemente buenas, porque una tasa de éxito inferior a una tercera parte del total de proyectos no es algo deseable.

Una vez analizado el problema, podemos ver en XP la solución, o al menos un acercamiento. La programación extrema centra su atención en la producción de software con una fuerte arquitectura, intentando sacar productos

[1] La definición como "metodología ligera" se propone en "Why XP?" de Robert C. Martin, presidente de ObjectMentor.
[2] Definición propuesta por Ian Mitchell
[3] Datos extraídos del documento "XP-An Overview"

al mercado rápidamente, con gran calidad y motivando al equipo de trabajo para seguir mejorando esta tendencia.

Como metodología, la programación extrema, presenta muchos puntos comunes con el desarrollo incremental, comenzando por el hecho de que el software desarrollado con XP se realiza de forma incremental. Para ver todas los puntos en que se centra la XP, vamos a dividirlo por fases[4].

- **Codificar:** Trabajar significa que, al final del día, tienes algo que funcione y que proporcione beneficios al cliente. Por tanto, todo el software se produce mediante la puesta a punto de pequeñas versiones incrementales de producción corta.

- **Probar:** Hay que asegurarse de que todo lo que se hace funcione correctamente. Para ello, lo mejor es desarrollar la prueba desde el momento que se conocen los casos de uso (o, según XP, las historias del usuario). Por ello, lo mejor es desarrollar las pruebas antes de generar el código para tener una prueba más objetiva del correcto funcionamiento de éste.

- **Escuchar:** Tanto para diseñar, como para desarrollar pruebas, como para desarrollar, . . . tienes que saber exactamente lo que quieres, para ello, se debe aprender a escuchar muy bien al cliente, al jefe de proyecto y a todo el mundo en general.

- **Diseñar:** El diseño también debe ser incremental y debe estar empotrado en el software, lo cuál quiere decir que la estructura de éste debe ser clara. Hay que diseñar lo que las necesidades del problema requieren, no lo que uno cree que debería ser el diseño. Además, siempre hay que tener en cuenta que diseñar cosas para el futuro es una perdida de tiempo, porque no las vas a necesitar.

XP COMO FILOSOFÍA

La programación extrema llega a ser una metodología un tanto filosófica debido a los valores que promueve entre el grupo de trabajo a la hora de la realización de los proyectos. Quizás esto resulte un tanto estravagante, ya que hablar de algo filosófico nos puede conducir a pensar en algo sin utilidad práctica o algo parecido. Primero veamos los puntos en los que la metodología roza planteamientos más globales que el simple desarrollo de software después veremos con mayor claridad el por qué de este punto.

- **Comunicación:** Comunicación total a todos los niveles de trabajo. Se trabaja en grupos de dos personas por ordenador pero con total comunicación en todos los momentos entre todos los grupos. El total de personas es de 10-12 lo que permite un total entendimiento. Incluso el código debe de ser comunicativo autoexplicativo, ya que

[4] División de las cuatro fases extraída del documento http://c2.com/cgi/wiki?ExtremeProgramming por Ward Cunningham.

se intenta que sea entendible por si mismo y eliminar el engorroso trabajo de documentarlo.

- **Usuario:** El usuario siempre está en mente. Se han de satisfacer sus necesidades pero nada más. Ha que seguir a pies juntillas sus peticiones.

- **Simplicidad:** Lo más simple es lo mejor, funciona mejor, más rápido, es más adaptable, más barato y además es más entendible.

- **YAGNI:** "You aren't gonna need it" (No lo vas a necesitar). No hagas nada que creas que en el futuro puede ser útil porque probablemente no lo vas a necesitar. Es una perdida de tiempo.

- **OAOO:** "Once and only once" (Una única vez). Las cosas se hacen una sola vez.

Quizás de estos puntos, el que aporte mayor valor como aspecto más filosófico es el de la comunicación, ya que consigue que la gente que participa en los proyectos tome una filosofía más comprometida con el proyecto. Además, la filosofía del "pair programming" es algo muy válido para otros aspectos en la vida cotidiana, por lo que es normal que la gente acotumbrada a trabajar así sea más receptiva a según qué situaciones. También cobra especial importancia el hecho de que el cliente se considera como un elemento más del equipo, ya que es el encargado de ofrecer el "feed-back" diariamente y de crear las denominadas "user-stories", algo parecido a los casos de uso.

1.2 MARCO HISTÓRICO

La aparición de la programación extrema tiene mucho que ver con las necesidades de desarrollar software de una forma más eficiente y rápida de lo que hasta el momento de su aparición se podía hacer.

Antes de la XP Aparición de la XP

ANTES DE LA XP

La programación extrema tiene sus fundamentos en un paradigma de la programación, los lenguajes orientados a objetos y en las metodologías que surgieron para ayudar en el desarrollo de proyectos que utilizan este paradigma. Las metodologías anteriores a XP, como ya hemos dicho, eran metodologías muy pesadas, debido a las necesidades de documentación y de generación de una serie de elementos que realmente no prestan utilidad.

Además, todo esto estaba siendo aplicado sobre lo que algunos denominan ahora "vieja economía"[5], ya que los proyectos presentaban una serie de condiciones:

- Gran conocimiento sobre lo que se está haciendo.
- Se proporciona al programador la documentación necesaria para que trabaje sin necesidad de pensar.
- Los programadores no saben nada sobre el "negocio".

En el momento en el que irrumpe la XP, se trabaja con lo que se denomina "nueva economía", que presenta rasgos opuestos a la comentada anteriormente:

- Escaso conocimiento sobre lo que se debe hacer (o nulo).
- Debido a la falta de conocimiento, no ha guías a seguir.

APARICIÓN DE LA XP.

La programación extrema surgió gracias a Kent Beck. Kent, anteriormente a la creación de la XP, había trabajado, sobre todo, con Smalltalk, el primer lenguaje orientado a objetos que realmente se hizo popular.

Siendo un amante de la orientación a objetos, Kent trabajó junto con Ward Cunnigham intentando encontrar un acercamiento al desarrollo de software que permitiese hacer las cosas más símplemente y de forma más eficiente.

A raíz de sus investigaciones, Kent entró a formar parte en un proyecto de DaimlerChrysler, que se denominó C3 (Chrysler Comprehensive Compensation), dónde, debido a las necesidades del propio proyecto, se vió obligado a aplicar sus investigaciones y a ponerlas en práctica, surgiendo lo que denominó "Extreme Programming".

El denominar a esta metodología "Extreme Programming" surge por el énfasis de ésta en tener unas buenas prácticas y seguirlas constantemente, pero seguirlas de una forma extrema, es decir, los principios de la XP deben ser seguidos sin alejarse de ellos ni un ápice, y cualquier otra práctica será ignorada.

[5] Asociación entre "Vieja economía" "Nueva economía" extraída de "XP-An overview" por Ian Mitchell

1.3 ¿POR QUÉ USAR LA XP?

La programación extrema proporciona una serie de ventajas a la hora de afrontar ciertos proyectos, que deben ser tenidas muy en cuenta , porque el uso de la XP en ciertos contextos puede ahorrarnos mucho tiempo y muchos recursos.

Ventajas aportadas por la XP

VENTAJAS

Evidentemente, para que algo esté siendo tomado tan en cuenta como la XP, debe ofrecer una serie de ventajas a la hora de ponerlo en práctica que hagan que el esfuerzo de entender y aplicar sus prácticas, sea insignificante con respecto a los beneficios obtenidos.

- Se consiguen productos usables con mayor **rapidez**.
- El proceso de **integración** es continuo, por lo que el esfuerzo final para la integración es nulo. Se consigue integrar todo el trabajo con mucha mayor facilidad.
- Se atienden las **necesidades del usuario** con mayor exactitud. Esto se consigue gracias a las continuas versiones que se ofrecen al usuario.
- Se consiguen productos más fiables y **robustos** contra los falllos gracias al diseño de los test de forma previa a la codificación.
- Obtenemos código más **simple** y más fácil de entender, reduciendo el número de errores.
- Gracias a la filosofía del "pair programming" (programación en parejas), se consigue que los desarrolladores apliquen las **buenas prácticas** que se les ofrecen con la XP.
- Gracias al "refactoring" es **más fácil el modificar** los requerimientos del usuario.
- Conseguimos tener un **equipo de desarrollo más contento y motivado**. Las razones son, por un lado el que la XP no permite excesos de trabajo (se debe trabajar 40 horas a la semana), y por otro la comunicación entre los miembros del equipo que consigue una mayor integración entre ellos.

- Debido a que se concibe que la **"propiedad" del código es colectiva**, cualquiera puede desarrollar, mejorar, simplificar, . . . cualquier necesidad del proyecto, eso sí, siempre usando sistemas tipo CVS para evitar la duplicación de trabajo usando el "refactoring" si se trata de una modificación.

- Existen muchísimas más ventajas, pero hemos nombrado las más importantes y las más generales, ya que la XP puede ofrecer otro tipo de ventajas en según que entornos se aplique.

1.4 ¿CUÁNDO Y DÓNDE USAR LA XP?

La XP no es algo universal, a que si fuese una herramienta que nos permitiese solventar cualquier tipo de situación, sería la "piedra filosofal" de la informática. Veremos en que situaciones y bajo que criterios es mejor aplicar la programación extrema.

Requerimientos de la XP Para qué nos es útil.

REQUERIMIENTOS DE LA XP

Aunque parezca algo un tanto estúpido, la programación extrema está orientada hacia proyectos con un escaso número de participantes (unos 10/12) trabajando en parejas de dos personas. Por lo que uno de los requerimientos más importantes para poder aplicar esta metodología es el tamaño del proyecto. Sobre esta base, se diseña un entorno tipo para poder practicar la XP. Las necesidades de este entorno son[6]:

- Semanas de **40 horas** de trabajo.

- Todos los desarrolladores deben estar en una **única habitación**.

- Todos los días comienzan con una **reunión de apertura**.

- En la habitación debe haber **comida** (básicamente "snacks para reforzamiento positivo").

- Alta **velocidad** de trabajo.

[6] Las necesidades de la XP recogidas en la WikiWikiWeb.

Al ver estos requerimientos tan poco comunes, podemos pensar que no son algo realmente indispensable para el desarrollo de un proyecto con XP, pero esto no algo acertado. ¿Por qué no podemos saltarnos estos requisitos?, ahora lo veremos:

- El trabajar todos en una misma sala consigue que la mayoría de la **documentación se pueda suprimir** sin mayores consecuencias porque el proceso de comunicación se realiza directamente, de forma hablada.

- El comenzar con una reunión de apertura agiliza, también, la **comunicación** entre los integrantes del proyecto, estándo estos constantemente informados del estado de su trabajo.

- Tanto la existencia de comida ("snacks") como la semana de 40 horas de trabajo consiguen que los integrantes del proyecto estén en todo momento en plenas **facultades para desarrollar** su trabajo, y así se evita el "stress" típico en los desarrolladores.

- La parte de "alta velocidad de trabajo" es tanto un requerimiento como una consecuencia de todo lo planteado aquí. Aunque se trabajen solo 40 horas a la semana, hay que cumplir un trabajo determinado en esas horas al día que se disponen para ello. Esto implica que se trabaja a un ritmo muy fuerte para **evitar** el tener que trabajar **horas extras**.

Para qué nos es útil.

Una vez conocemos el entorno necesario para poder desarrollar las prácticas expuestas por la programación extrema, vamos a ver en qué áreas o en qué proyectos podemos utilizar la XP con éxito.

Como generalidad, la XP, puede ser usada en cualquier proyecto con un tamaño relativamente pequeño, el cual pueda ser desarrollado por un equipo de entre 10-12 personas.

Se pueden aplicar otros criterios, ya que existen muchas opiniones acerca de esto en el mundo de la XP. Nosotros seguiremos básicamente este criterio que es el, a priori, más acertado, porque permite aplicar la XP sin modificar sus principios básicos. También podemos establecer un segundo criterio basado en la dificultad del proyecto. Si nos encontramos ante un proyecto inabordable, en el que no conseguimos avanzar, quizás sea bueno probar un enfoque cercano a la XP, incluso reduciendo el número de personas que participan en el, intentando acercarnos a las necesidades de la XP, pero sin obviar la carga de trabajo. En estos proyectos no podemos aventurar nada acerca de su resolución, pero si podemos decir que el enfoque de la XP puede ser útil si otros acercamientos han fracasado (y cuando menos, no perdemos nada, salvo tiempo).

1.5 XP A VISTA DE ÁGUILA

Para poder llegar a comprender la XP debemos empaparnos de información acerca de ella, algo que en los sucesivos capítulos trataremos de conseguir. Pero nunca es malo tener una visión general, previa a la inmersión total en todo el conocimiento, y esto es lo que vamos a tratar de ofrecer en esta sección.

Esperamos que a esta altura del capítulo, ya esté claro lo que es la programación extrema, en cuanto a concepto. Lo que hasta ahora hemos proporcionado es información acerca de qué es la XP y que ventajas nos aporta. Quizás, no se tenga muy claro el por qué de las ventajas o de los requisitos, pero al ir profundizando en la programación extrema, esto se tendrá más claro.

Básicamente, la XP se basa en cuatro valores:

- **Comunicación** (este punto a ha sido tratado en apartados anteriores).
- **Simplicidad** (también ha sido mencionado anteriormente).
- **Feedback**: Básicamente el continuo contacto con el usuario, al irle entregando las sucesivas versiones, en funcionamiento, del producto, permite que este nos de su valoración y nos comunique, cada vez mejor, lo que realmente quiere en el producto.
- **Coraje**: Básicamente es trabajar muy duro durante las horas dedicadas a ello.

Además de estos cuatro valores, que podríamos denominar como los cuatro pilares de la XP, tenemos un compendio de 12 buenas prácticas, que podríamos denominar como los 12 mandamientos de la programación extrema:

- **La planificación**: se utilizan las "user-stories" ("historias del usuario"), para realizar el análisis, estas "historias", se dividirán en tareas (unidades pequeñas, de 1 a 5 días de trabajo en pareja). Además, se priorizarán las tareas, y cada una de ellas tendrá un desarrollo incremental.
- **Versiones pequeñas**: La primera versión contendrá el conjunto mínimo de requisitos más útiles/necesarios para el sistema global.

- **Sistema metafórico**: Cada proyecto debe tener una metáfora asociada que nos ofrezca unos criterios para nombrar lo que vayamos haciendo de forma fácil.

- **Diseño simple**: Cómo los requerimientos cambian, o pueden hacerlo, diariamente, ha que utilizar los diseños más simples posibles para cumplir los requerimientos que tenemos en la actualidad.

- **Testeo continuo**: Antes de que se implemente cualquier característica de un sistema, se debe escribir un test para ella.

- **Refactoring**: Cuando tenemos que introducir una nueva característica del sistema, si esta tiene mucho en común con otra previa, lo mejor es eliminar el código duplicado, sin miedo a que falle, debido a que el test probará el correcto funcionamiento.

- **Pair programming** ("programación en parejas"): Se trabaja en parejas, cada una utilizando un único ordenador. Así, el código se revisa mientras se desarrolla.

- **Propiedad colectiva del código**: Cualquiera puede modificar cualquier módulo en cualquier momento, nadie tiene la propiedad de ningún módulo.

- **Integración continua**: Todos los cambios se introducen en el sistema, al menos, una vez al día.

- **Semanas de 40 horas** de trabajo: Los programadores se deben ir a casa a su hora.

- **Cliente en su sitio**: Siempre hay un usuario del sistema que es accesible por los miembros del equipo de trabajo.

- **Estandares de codificación**: Todos deben usar los mismos criterios a la hora de programar. De esta forma, no sería posible determinar quién ha realizado una determinada parte de la implementación.

1.6 ANÁLISIS Y DISEÑO EN XP

Con todo lo visto puede ser fácil creer que el análisis y el diseño no tienen importancia en la XP, pero esto no es verdad. Lo único que se intenta es conseguir compactar todo el proceso de desarrollo, pero sin eliminar ninguna de sus fases.

User Stories Modelos de negocio UML

Si hemos considerado a la programación extrema como una metodología para el desarrollo de software, debe ser obvio que no nos vamos a olvidar de las tareas típicas previas a la implementación del código en sí. Tanto el análisis como el diseño son tareas muy importantes en la XP, pero se realizan con un enfoque más ligero y transparente. El análisis es parte fundamental, a que se intentan recoger en él todas las necesidades del usuario. De él surgen las "user stories" que nos servirán para empezar a comenzar a desarrollar nuestro sistema.

USER STORIES[7]

El concepto de las user-stories tiene algo que ver con los famosos use-cases ("casos de uso") utilizados en el ciclo incremental de desarrollo de software. Pero esta similaridad se basa en que su cometido es el mismo, sirven para hacer las mismas cosas, pero no son lo mismo. Nos permiten sustituir unos largos requerimientos por una serie de user stories y además nos permiten hacernos una estimar el tiempo para la reunión de lanzamientos de las futuras versiones de nuestro sistema.

Además de esto, las user stories nos ayudan a crear tests de aceptación. Estos son pruebas que se aplicarán al sistema para ver si cumplen una determinada "historia del usuario", lo cuál viene a ser lo mismo que cumplir un determinado requisito en otros modelos de desarrollo.

Las realiza el propio cliente en forma de 3 sentencias de texto, en las que describe necesidades del sistema con la propia terminología del negocio, sin hacer uso de vocabulario técnico. Existen muchos más detalles acerca de las user-stories, pero serán tratados con mayor profundidad en el capítulo dedicado a la gestión del proyecto.

MODELOS DE NEGOCIO[8]

Algo muy importante para un proyecto de software es que esté correctamente alineado con el modelo de negocio para el que ha sido creado. El problema surge debido a que los modelos de negocio cambian cada vez más rápido, a que las empresas deben actuar cada vez contra más elementos y más variados (respuestas ante nuevas tecnologías, adecuar los procesos de negocio a estas, razones de competitividad, . . .). Debido a esto, nuestro sistema de desarrollo de software debe ser capaz de responder de una manera lo suficientemente rápida a los cambios en las necesidades de nuestro cliente.

UML

La realización de un proyecto mediante técnicas de XP no tiene nada que ver con la utilización o no de UML para el análisis o el diseño. En la

[7] Más sobre "User Stories" en www.extremeprogramming.org
[8] Información recabada en www.xp.co.nz/business_models.htm

programación extrema tenemos las “user stories” para el análisis de requerimientos, pero, debido a que la mayoría de desarrolladores-extremos provienen del mundo de la programación orientada a objetos, suele ser usual utilizar UML para desarrollar diagramas en proyectos con XP.

Pero esto no quiere decir que sea necesario, símplemente es una técnica de modelado de diagramas que puede ser útil o no dependiendo de las necesidades. Como conclusión de este punto se desea reflejar que el uso de XP no implica que no se usen una serie de técnicas, la programación extrema impone una serie de principios básicos que no deben ser violados y suele ser útil no utilizar otros principios, ya que no sería XP, pero dentro de estos márgenes, tenemos muchas herramientas que nos ayudan en nuestro trabajo y no deben ser desestimadas.

2.1 LOS 12 VALORES DE LA XP

Toda metodología (su propio nombre lo indica) esta basada en reglas que permiten de forma sistemática el desarrollo del software. La XP no es menos, y aquí veremos en qué se apoya.

Planificación Versiones Pequeñas Sistema Metafórico. (Metaphor). Diseños simples Testeos Continuos Refactoring Programación en parejas Propiedad colectiva del código Integración continua 40 horas por semana El cliente en su sitio Estándares de codificación

Otras metodologías basadas en la perfecta sincronización entre documentación, diseño y código dividiendo estas tareas entre diversos grupos de programación y contando además con la necesaria vuelta atrás (no siempre sale el software deseado por el cliente a la primera... en realidad casi nunca) y vigilando que no se dispare el presupuesto provocan que la tarea de crear software nuevo sea complicada o al menos no tan estructurada como, en la teoría y a priori nos pueda dar a entender la propia metodología.

La XP siendo como es una metodología o disciplina de programación también proporciona una serie de reglas para la creación de software.

Estas reglas o valores no son fijos ni obligatorios y por esto pueden surgir discusiones sobre la necesidad de una o varias reglas. Por esto cada equipo de creación de software que utilice la XP posiblemente las vea forma diferente o haya concebido alguna adicional que le dio buen resultado en otros proyectos.

2.1.1 PLANIFICACIÓN

Para realizar un buen trabajo es necesario planificarse bien. Esto se traduce en la Xp de la siguiente forma.

- No empezaremos a realizar el análisis hasta que no tengamos las "user-stories" ("historias del usuario"), (ya se vio en capítulos anteriores este concepto).
- Los programadores nunca deberán tomar decisiones que consten en el diseño y se deberán ajustar solo a el.
- Además el equipo de trabajo nunca deberá incluir funcionalidades que no consten en el diseño. Esto es debido a que son funcionalidades que posiblemente gusten al cliente pero que puede que en sucesivas vueltas atrás y en revisiones por parte del cliente del programa decida retocarlas o cambiar su función con lo que hará falta retocar un código que en un principio no debería haber existido hasta que el cliente lo necesitase. Por otro lado puede resultar

trabajo inútil y no reconocido por lo que es mejor no salirse de las especificaciones y no "regalar" esfuerzo y trabajo a nadie.

2.1.2 SISTEMA METAFÓRICO

A la hora de desarrollar código puede surgir el problema de cómo nombrar los métodos, funciones, variables, relaciones etc... de un modo coherente y que todos los componentes del equipo lo entiendan.

No se debe dejar este tema a la improvisación de cada uno.

- Los identificadores deberán ponerse en arreglo a unas ciertas reglas preestablecidas con anterioridad por el grupo.

Con esto conseguimos:

- No perder tiempo valioso de programación en pensar un nombre para un método o similar. Gracias al sistema metamórfico el nombre viene rodado.

- Que el resto del equipo no dude sobre el nombre de un método de un paquete desarrollado por otros componentes.

Entonces como puede verse ahorramos tiempo por dos vías: La de creación y la de consulta. Este fundamento de la XP ayuda a mantener el código limpio y aumenta su simplicidad.

2.1.3 PROGRAMACIÓN EN PAREJAS

La programación del código se realizara en parejas frente a un solo ordenador. Se hace así todo el tiempo y no solo cuando un programador necesite ayuda para realizar tal método o para averiguar la razón de porqué falla el código.

- De esta forma mientras uno desarrolla código el otro realiza un revisión inmediata.

- Se evitaran la mayoría de los errores de programación debido a que habrá en todo momento una persona dedicada íntegramente a buscar errores en tiempo de escritura del código.

- Además la programación será mas fluida pues el desarrollador no tendrá que estar al tanto de encontrar sus errores ya que tendrá a su lado a otro programador que conocerá el diseño de lo que se esta programando y por lo tanto sabrá reconocer un error.

- Por lo tanto el código así generado no contendrá errores sintácticos y posiblemente tampoco errores semánticos y habrá sido realizado en un corto periodo de tiempo.

- Los programadores (y esto es muy importante) permanecerán mas despejados durante el tiempo de trabajo.

2.1.4 40 HORAS POR SEMANA

Este punto esta muy relacionado con el de la dedicación plena en el proyecto durante las horas de trabajo. No hace falta dedicar largas jornadas de duro trabajo, pues esto no es garantía de acabar antes. Posiblemente largas jornadas de trabajo provocaran errores debidos a la concentración y a la motivación del equipo. Es difícil mantenerse en plenitud de facultades durante toda la jornada de trabajo si esta es larga.

La programación extrema se basa en 40 horas por semana. De tal forma se evita que los programadores se fatiguen y que las horas de trabajo sean horas altamente productivas. De todas formas 40 horas de trabajo a la semana pueden repartirse de muchas formas:

- 8 horas cada día durante 5 días (Lunes- Viernes) de forma que se trabaje 4 horas por la mañana y otras 4 por la tarde. El fin de semana descanso merecido.

- 7 horas cada día durante 5 días (Lunes- Viernes). 4 horas mañana y 3 por la tarde. El sábado se trabajan 5 horas, y el domingo se descansa.

2.1.5 VERSIONES PEQUEÑAS

Lo ideal en todo proyecto es que el cliente o usuario tuviese una imagen clara y concisa de lo que quiere y de cómo lo quiere. Desgraciadamente lo mas habitual en todo proyecto es que el cliente sabe lo que quiere en un dominio de su negocio pero carece del conocimiento para expresar esas funcionalidades en un ambiente algo mas técnico o propicio para los desarrolladores.

Para empezar a desarrollar es preferible abarcar pocas funcionalidades, las básicas para crear una aplicación base que funciones y sobre la cual el cliente pueda introducir cambios.

¿Qué son las versiones pequeñas? Son versiones de tamaño reducido que incluyen funcionalidades básicas requeridas por el cliente que forman un conjunto mínimo de tal forma que la aplicación funciona.

Estas versiones pequeñas tienen varias ventajas:

- El cliente puede ver regularmente como evoluciona el software introduciendo continuos cambios en los requerimientos de tal forma que nunca se lleve a engaños y se sienta "jefe" del proyecto.
- La vuelta atrás siempre será mas fácil con versiones cortas que con versiones que contienen un gran numero de modificaciones que las anteriores.

- Evita que el código se descontrole rápidamente y facilita la claridad y la sencillez tanto del diseño como del código del programa.

2.1.6 DISEÑOS SIMPLES

El desarrollo de software de calidad siempre se basa en un buen diseño. Por ello el diseño es la base para el buen funcionamiento del equipo. Un diseño correcto en todo momento es aquel que cumple los siguientes requisitos reflejadas en el programa resultante:

- Ejecuta correctamente todo el diseño de pruebas.

- Esto significa básicamente que el programa funciona correctamente y muestra los resultados esperados.

- No contiene código redundante y aprovecha al máximo las capacidades de la maquina.

- En XP la idea es que el programa final sea la forma mas simple de aglutinar todos los requerimientos.

2.1.7 TESTEOS CONTINUOS

Los test son los encargados de verificar el funcionamiento del programa. En XP se producen continuos testeos del código del programa. No se debe programar nada de código si haber realizado antes los diseños de pruebas. Las pruebas deben ser capaces de verificar el buen funcionamiento de todos los requerimientos solicitados por el cliente. Contando además con las versiones pequeñas que se realizan en la XP hacer testeos continuos proporciona seguridad en las versiones del software creado con lo que se puede seguir ampliando el sistema con la certeza de su buen funcionamiento.

Siempre es mas fácil solucionar un problema cuando el código no es muy extenso o cuando se hace cada poco tiempo. Esto ya se vio anteriormente.

2.1.8 REFACTORING

Según el refactoring el código debe ser simple y claro. Esto provoca el huir del código duplicado por ejemplo. El código duplicado complica la tarea de programación por diversas causas.

Una de ellas es que "ensucia" el programa haciéndolo mas largo de lo estrictamente necesario (con lo que no satisfacemos el fundamento de los Diseños Simples). Además aumentan las posibilidades de tener errores, y hace incomprensible o de difícil lectura el programa.

2.1.9 PROPIEDAD COLECTIVA DEL CÓDIGO

El código que el equipo genere para llevar a cabo un proyecto es propiedad de cada uno de los componentes del equipo.

- De esta forma cualquier componente del equipo podrá modificar un modulo o porción de código generada por otro componente si lo cree conveniente.
- Dentro de un equipo no existen derechos de autor ni prohibiciones de ningún tipo a la hora de modificar el código de un compañero.
- Todo el código desarrollado por cada componente del equipo es cedido para el bien del propio equipo y poniéndolo a su disposición para cualquier mejora o comentario.
- La propiedad colectiva del código permite al grupo trabajar mas rápidamente y rendir de una forma mas eficaz debido a que no se producen los retardos o esperas ocasionados por la propiedad privada de un código y la incapacidad de modificarlo sin pedir permiso al miembro del equipo "creador" del código.

2.1.10 INTEGRACIÓN CONTINUA

La idea de la XP es la de tener siempre versiones simples y manejables del sistema. Esto también se puede aplicar a los cambios que se deben introducir en versiones pasadas.

- El tratar de retrasar la inserción de cambios en nuestro código lo único que produce son problemas. Si ya sabemos que un código claro y simple es mas fácil de manejar y de entender también deberíamos saber que los cambio que se acumulan pueden convertirse en un obstáculo difícilmente superable.

- Los cambios deberán ser integrados siempre continuamente y no acumularlos y integrarlos de un golpe todos . La integración continua favorece no solo la legibilidad del código sino además la prevención de errores y si se producen favorece su localización rápida y sencilla.

- Además permite tener conciencia en todo momento del trabajo que se lleva desarrollado y también permite tener a mano cuando el cliente lo solicite una versión perfectamente funcional y con los últimos avances.

- La forma de llevar a cabo la integración continua es introducir todos los cambios producidos a lo largo del día a final de la jornada.

2.1.11 EL CLIENTE EN SU SITIO

Una de las principales características de la XP es la Comunicación. Al Principio del capitulo se vio como la comunicación constante entre los miembros del equipo influye en la calidad del código final.

Las funcionalidades que debe cumplir el proyecto son dictadas por el cliente. Estas funcionalidades puede que no queden claras desde un principio. La XP propone, entonces, que el propio cliente pase a formar parte del proyecto, que se involucre en la producción del software desde sus posibilidades que no son escasas. El equipo de producción deberá estar en continua comunicación con el cliente que deberá aclarar en la medida de lo posible los requerimientos que necesitara utilizar.

De esta forma se soluciona o al menos se facilita la compresión de las funcionalidad que en ciertos casos suele ser la etapa mas difícil , mas cara y mas lenta del desarrollo del proyecto.

2.1.12 ESTÁNDARES DE CODIFICACIÓN

Ya se ha visto como la XP promueve la codificación en parejas (es mas rápida, mas eficiente, y menos candidata a comentar errores) ,la propiedad colectiva del código (de forma que todo el código es de todos y ningún integrante del equipo posee plenos derecho sobre alguna parte), el testéo continuo, la integración continua...

- Pues bien, todas estas cualidades que hacen de la XP una metodología eficaz de programación para ciertos proyectos, se vendrían abajo si además no contásemos con los estándares de codificación.

- Este concepto se puede equiparar al de Sistema metamórfico. Recordemos que el sistema metamórfico decía que era necesario establecer un criterio fijo que proporcionase reglas para la creación de nombre para variables y métodos del tal forma que ninguno de los

integrantes del equipo dudase en ningún momento del nombre que debiera poner a un método o del nombre que tiene que utilizar para llamar a un método desarrollado por otra pareja de programadores del equipo.

- Según esto también será necesario para el buen termino del proyecto establecer un estándar de codificación de tal forma que los programadores sigan los mismos criterios para desarrollar código.

- Con este Estándar además de las ventajas que se ganaban con el Sistema metamórfico (rapidez, claridad...) se gana eficacia al decidir de antemano el mejor estándar para satisfacer los requerimientos propuestos por el cliente.

3.1 VALORES PRINCIPALES DE XP:

Un valor es una descripción de cómo debe enfocarse el desarrollo de software. Vamos a tratar a lo largo de este capitulo los cuatro valores fundamentales acerca de XP, estos valores son:

Comunicación Simplicidad FeedBack Coraje

3.1.1 COMUNICACIÓN

Hay pocas actividades donde sea tan importante una buena comunicación entre el equipo de desarrolladores. Un equipo de XP es una mezcla de clientes y equipo técnico encargado de desarrollar dicho proyecto.
XP anima a que se produzca una comunicación extrema, si el cliente necesita ver el programa funcionando para especificar requerimientos entonces lo que se debe hacer es reunirlo junto con el equipo de desarrollo para trabajar juntos en las versiones, en períodos entre 2 y 4 semanas.

Un equipo de programadores trabaja en pareja: es decir dos personas por cada equipo. La pareja suele efectuar rotaciones de puesto regularmente, lo que implica que el código es propiedad de todo el equipo no es parte de una sola pareja. Esto promueve comunicación de conocimiento técnico por parte de todo el equipo, esto permite que en el momento en que un problema técnico aparece entonces todo el equipo puede formar parte en la resolución de este.
La programación por parejas es también excelente para la gente que posea diferentes habilidades lo que permite que aquellas personas que posean menos nivel sean guiadas por personas de una mayor experiencia, así el riesgo de añadir código de personas menos experimentadas se reduce drásticamente.

3.1.2 SIMPLICIDAD

Este valor se basa en el hecho de que el futuro a nivel de requerimientos no forma parte del proyecto sólo nos debemos ocupar de cubrir las necesidades inmediatas, partiendo del hecho de que la predicción de hechos futuros carece de fiabilidad lo que se transforma automáticamente en pérdida de dinero.

El cliente es el que tiene la última palabra en XP. Haga lo que el cliente necesita tan simple como sea posible y nada más.

Todo el código debe ser Refactorizado tan a menudo como sea posible. Refactoring para mejorar la estructura sin cambiar su funcionalidad. Ya que

Refactoring produce una alta conectividad entre los diversos objetos de nuestro proyecto, objetos que son más fáciles de probar, fáciles de usar, más flexibles, y por consiguiente, más manejables.

3.1.3 FEEDBACK

En general lo principal es encontrar un error y arreglarlo al mismo en que fue introducido por el equipo. Xp se esfuerza en realizar el feedback tan rápido como sea posible en todos los aspectos de nuestro proyecto.

Las unidades de test son escritas para la mayoría de los módulos de código. Dichos test deben ser ejecutados en su totalidad para que nuestro módulo sea registrado en nuestro proyecto. Cuando una producción debe ser cambiada los efectos colaterales deben ser analizados. Después de que nuestro código es analizado entonces es inmediatamente integrado con los últimos cambios del resto de los miembros del equipo.

Se escriben pruebas de aceptación con el cliente para verificar que la aplicación esta haciendo aquello que el cliente necesita de la aplicación. La programación en parejas provoca constantes revisiones de código, no habrá más reuniones para la revisión del código, hay que prestar la máxima atención cuando el código se esta escribiendo. La colectividad del equipo y las rotaciones que estos efectúan aumentan la cantidad y calidad del código revisado.

3.1.4 CORAJE

Algunas veces XP adopta la miope visión de que todos encuentran el software tan fascinante como sus desarrolladores. Pero muchos usuarios finales no desean que sus aplicaciones luzcan y funcionen diferentemente cada vez que las usan, y prefieren no recibir frecuentes pequeños releases. En resumen, para administradores de proyectos, decidiendo usar XP en su próximo proyecto pueden de hecho sentirse como saltar al vacío desde un puente. debe estar lo suficientemente seguro de sí mismo para abandonar un buen trato de control, ya que no tendrá más la autoridad para determinar qué quiere y cuándo lo quiere. Aunque la metodología está basada en las duras realidades y los escollos del desarrollo de software, y ofrece muchas soluciones de sentido común, XP puede no ser la palabra final en la evolución del desarrollo de software, aunque certeramente es un paso en la dirección correcta.

4.1 EL NUEVO MANAGEMENT

Si la programación extrema implica cambios radicales en cuanto a las tareas típicas de analizar, diseñar y programar en sí, los cambios afectan de igual manera en cuanto a las labores del jefe de proyecto, ya que, evidentemente, tiene que adaptarse a nuevas circunstancias.

Adaptándonos a las nuevas necesidades "Plannig" temporal

Según las propias palabras de Kent Beck, el manager, según la XP no es la persona que crea el proceso, que lo mueve, sino es la persona que lo suaviza al estar entre los dos frentes siempre abiertos en cualquier proecto informático: el cliente y los programadores. Por tanto, un jefe de proyecto en XP tiene un rol de gestión muy marcado, dejando, en parte, las labores más técnicas que debe realizar al seguir otras metodologías.

ADAPTÁNDONOS A LAS NUEVAS NECESIDADES

Démonos cuenta de la nueva situación para el manager recién llegado. Con la XP, el análisis lo realiza el cliente en bastante mayor proporción que en otras metodologías al usar las "user stories"; el diseño es una labor muy ligada a la programación y son los propios programadores los encargados; hay que documentar muy poco, por lo que gran parte de toda la burocracia que acarrea un proyecto la tenemos resuelta. Pero todo esto no significa que el jefe de proyecto sea prescindible o que no tenga nada que hacer. Surgen nuevas necesidades, nuevas tareas y nuevos riesgos que el manager debe asumir y combatir.

El nuevo manager debe alejarse del camino que toman los desarrolladores en sí. Debe preocuparse por todo lo que les rodea y abandonar la preocupación de lo que están haciendo en sí. Debe ser un buen gestor, lo cuál implica detectar aquellos puntos negros que enlentecen el desarrollo del sistema, combatir aquello que se detecta como fuente de problemas, controlar el buen funcionamiento de las infraestructuras y de todo aquello que necesitan usar los desarrolladores, . . . resumiendo, conseguir que todo esté en orden para que los desarrolladores puedan terminar su trabajo a tiempo.

Además de esta parte de gestión más bien de recursos físicos, el jefe de proyecto debe acometer otro tipo de gestión, la de planificación de recursos humanos. Una vez centrados en el día a día, el manager es el encargado de acometer las reuniones de apertura diarias y de asignar a los miembros del equipo aquellas labores que deben hacer. Una vez asignadas esas labores, el manager no debe ir a más bajo nivel, conseguirá hacer un buen trabajo si tiene todas las labores que se deben hacer en la jornada de trabajo asignadas y a todos los miembros del equipo trabajando, lo cuál es bastante complicado.

Como se puede apreciar claramente, el día a día del manager es el preparar la reunión diaria, ya que la otra gestión es algo más puntual, o que se puede englobar dentro de la tarea diaria al tener encargados de mantenimiento. Realizar esta reunión diaria es una tarea tediosa, ya que hay muchos puntos que tener en cuenta[9]:

- Planificar la fecha, el lugar, coordinar a todo el mundo para que asista y reservar el lugar adecuado, teniendo en cuenta de reservar el material necesario, tanto bebidas y "snacks" como la pizarra para exponer las ideas y desarrollar el planning. (Estas labores suelen ser ejecutadas por segundas personas, pero el manager debe ser quién invoque a estas segundas personas a ejecutarlas).

- Contactar con el usuario para que asista y cree nuevas "user stories" que proporcionarán nuevo trabajo a los desarrolladores. Además, se les debe proporcionar ayuda en dicha labor.

- Después de las reuniones se debe proporcionar la documentación necesaria para aquellas partes externas al desarrollo del proyecto que la necesiten.

- Coordinar todas las labores a todo el equipo.

Además de toda la parte dedicada a la planificación del proyecto en sí, un buen jefe de proyecto debe asegurarse de la buena cooperación entre los diferentes componentes del equipo, intentando que exista el menor número de conflictos internos posibles.

"PLANNING" TEMPORAL

Una de las labores más arduas para el jefe de proyecto es la división temporal de este, intentando que los plazos de tiempo prefijados se cumplan. Para poder tener una idea de cómo realizar esta labor, conviene tener muy claro cuál es el ciclo de trabajo seguido con la XP.

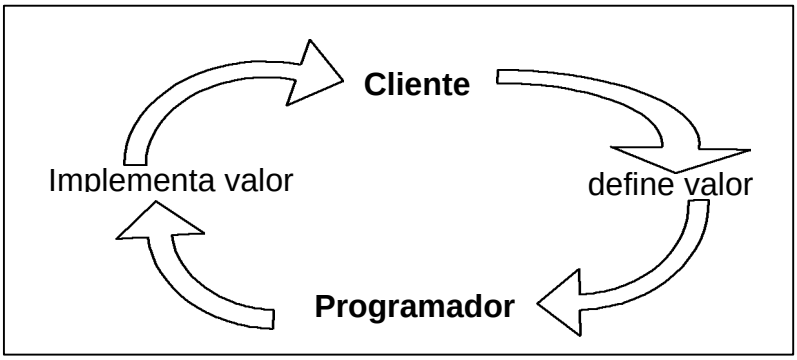


Figura 1[2]

[9] Las necesidades de las reuniones de apertura han sido extraidas del libro XP-Installed de Kent Beck

[2] Figura 1, 2 y 3 extraidas de "XP Installed"

Desde un punto de vista simplista, el ciclo de desarrollo de la XP se basa en un bucle en el cuál el cliente define sus necesidades y el desarrollador las implementa. Esto viene reflejado por la asignación de "user-stories" a los desarrolladores en las reuniones de apertura, creando gran dinamismo. Esto es el ciclo de vida a un nivel muy simple, por lo que podemos redefinirlo, dándonos cuenta de otras actividades que el desarrollador/programador debe realizar.

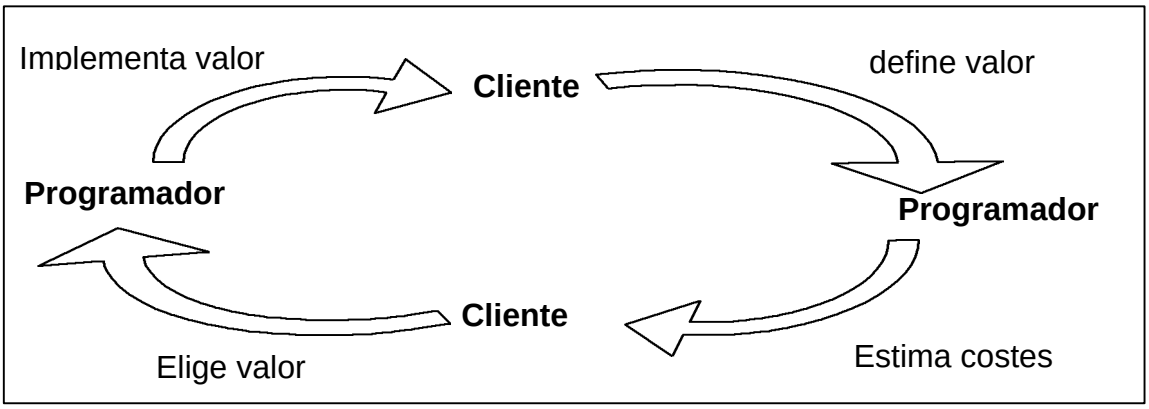


Figura 2

En la figura 2 podemos apreciar que una de las labores del programador o desarrollador es la de estimar los costes ante una petición del usuario. De esta manera, el cliente enuncia sus necesidades y éstas son costeadas. Una vez conociendo el coste de sus necesidades, el cliente puede evaluar si le conviene pagar el precio determinado por ellas o no, eligiendo, así, el conjunto de valores/requisitos que desea que sean implementados en el producto final. Esta información acerca del coste, es ofrecida por el propio programador, ya que su experiencia le permite evaluar la dificultad y el tiempo que va a necesitar para implementar aquello que le ha sido requerido. Evidentemente, esta es una ayuda de gran calidad para el jefe de proyecto.

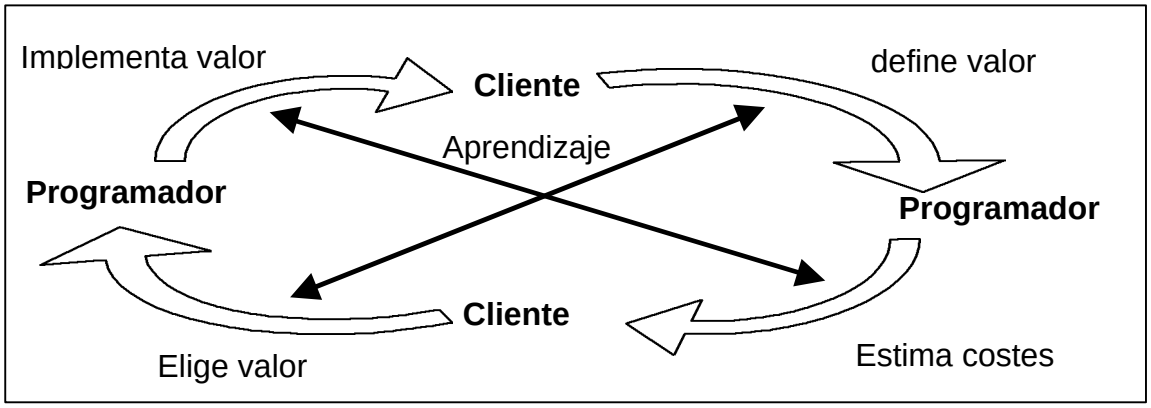


Figura 3

La experiencia proporciona el poder realizar un buen trabajo, tanto a la hora de estimar los costes como a la hora de implementar aquello que es requerido por el usuario. Entonces, cuanta más experiencia acumulada, más se tenga aprendido, más fácil será alcanzar las necesidades del usuario. Y esta experiencia que es requerida para acercarnos mejor a las necesidades del usuario es lo que nos proporciona el propio trabajo. En cada iteración, tanto el programador/desarrollador como el cliente aprenden. El desarrollador acumula experiencia al enfrentarse con nuevas situaciones con nuevos clientes y el cliente aprende a definir cada vez mejor sus necesidades. Así, se puede lograr un estado de equilibrio, o de facilidad, para poder consolidar mejor los objetivos comunes al proyecto (cliente y equipo).

Todo esto puede parecer más una ayuda al programador que al manager, pero tener una buena visión de las iteraciones para implementar los valores del cliente, nos permite darnos cuenta de ciertos factores que pueden ayudar al manager a definir las fechas de entrega de las distintas versiones del proyecto:

- El tiempo destinado a cada necesidad depende en parte de la dificultad y en parte de la capacidad del desarrollador. En la mayoría de las situaciones es el propio programador el más consciente de su capacidad, y el que mejor puede estimar el tiempo necesario para cada problema planteado.

- Es bueno conseguir que tanto el equipo como los clientes se den cuenta que dependen unos de otros. Lograr una buena cooperación implica lograr un mayor ritmo de trabajo que ayuda a cumplir los objetivos.

- Hay que encontrar ciertos compromisos entre valores complementarios. Un programador bajo presión es capaz de cumplir las fechas de entrega, pero la calidad se consigue rebajando el nivel de presión, para que se pueda pensar mejor y con mayor calidad. Para conseguir productos de calidad en el tiempo adecuado hay que compensar estos dos valores en su justa medida.

Para estimar el tiempo de un proyecto, comenzaremos realizando estimaciones aproximadas, contando con la ayuda de los programadores. Una vez tengamos una serie de ellas implementadas, la mejor manera de conocer el tiempo a dedicar a cada una de las restantes es estableciendo comparaciones con las ya desarrolladas, lo cuál nos dará una mejor visión del tiempo que necesitemos emplear.

También es relativamente frecuente encontrarnos con tareas que son dificilmente planificables, ya que no conocemos mucho acerca de las necesidades. Para poder ofrecer una visión más o menos acertada acerca del coste temporal, conviene realizar un pequeño prototipo de las necesidades, implementando en unas horas o, como mucho, un día, una parte básica de las necesidades de la tarea.

Una vez podemos establecer con mayor precisión el coste temporal de las restantes, se debe recomponer la escala de tiempo para el proyecto de

forma global para poder ofrecer al cliente una mejor aproximación a la escala temporal final.

4.2 ANÁLISIS Y DISEÑO EN XP

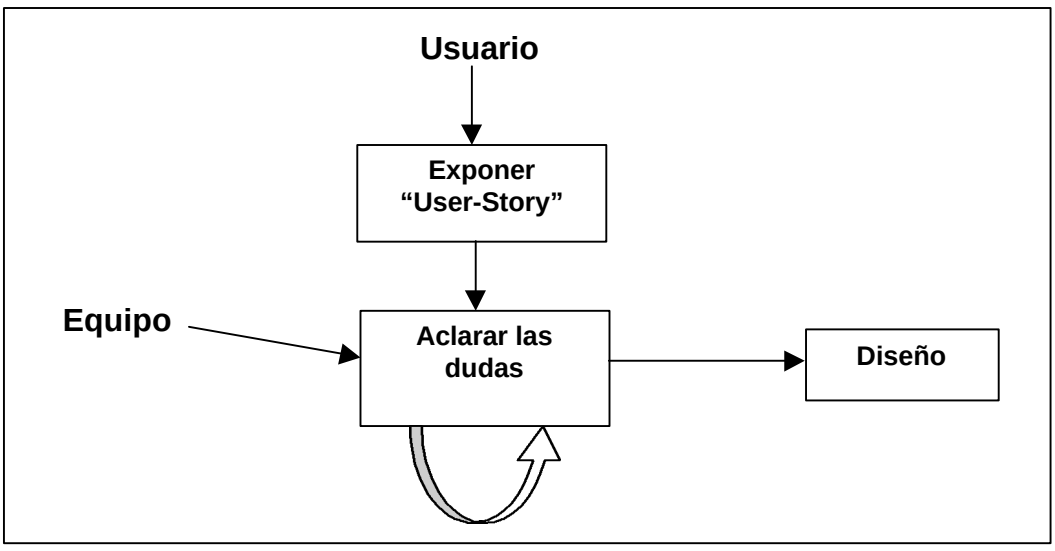
En el ciclo de desarrollo visto en el punto anterior, vemos que el esquema óptimo de trabajo es aquel en el cual se define, se evalúa, se elige y se implementa. Como se puede apreciar, la implementación es la última parte del ciclo y previamente se han realizado otras labores no menos importantes, como son Análisis y Diseño.

"User Stories"

En la informática, como en la vida, cuando se intentan aplicar métodos que a simple vista son muy prácticos, que van muy "directos al grano", se suele desestimar la parte más teórica. Esto llevado al mundo informático, más concretamente a la programación, implica que una metodología como la programación extrema puede hacer pensar a los desarrolladores en olvidarse de la parte más tediosa y por regla general, menos gustosa, que es la del análisis y diseño.

En la realidad, esto no es así. La XP se orienta a la parte más práctica, intentando eliminar todo aquello innecesario y poniéndonos a implementar lo más pronto posible. Aún así, y pese a que la parte de análisis es mucho más ligera, la parte de diseño sigue siendo parecida a las labores de diseño más o menos habituales en otras metodologías.

El análisis en la programación extrema es una labor común a los miembros integrantes del equipo y al propio cliente. Debido a los propios requisitos de la XP, un equipo pequeño, la parte de análisis se transforma en la exposición por parte del cliente de las "user-stories" que elabora en tarjetas físicas (de esto hablaremos más adelante) y pasa a exponer en las reuniones de apertura con el equipo. De esta forma, lo que se intenta es que todo el equipo tenga claro lo que se quiere hacer.



Una vez todos los miembros del equipo han comprendido lo que se va a realizar, podemos decir que la primera parte del análisis ha concluido. Realmente ha finalizado la parte de análisis más tradicional. A continuación le sigue una parte de diseño global del sistema, en el que se profundiza hasta el nivel necesario para que los desarrolladores sepan exáctamente que van a tener que hacer.

Esta parte de diseño global se realiza mediante "brainstorming", intentando lograr entre todos un cuadro global del sistema. En este brainstorming, los miembros del equipo intentan detectar todas las tareas necesarias para desarrollar la "user-story". Por regla general, nos encontramos con que el equipo ha encontrado una solución correcta, que implica una extensión de las funcionalidades de la última versión desarrollada. Otras veces, encontramos la existencia de varias aproximaciones, por la que el equipo debe elegir la más simple, acorde con la filosofía que siempre se sigue en XP. En otras ocasiones, no se encuentra ninguna solución factible a priori. Estas son las ocasiones típicas en las que se debe iniciar una iteración experimental, que nunca debe durar más de un día o dos, intentando ver cuál es una posible solución. Aquí nunca se resolverá el problema, se debe encontrar únicamente la manera, pero sin profundizar más allá de lo necesario para saber qué hacer.

Una vez se tiene una visión global del sistema a desarrollar en la iteración en cuestión, se dividen las tareas en grupos de dos personas, que iniciarán un ciclo como el visto en la figura 3 del punto anterior, estimando su tarea, de manera que ayudan al jefe de proyecto a la hora de la estimación del tiempo y consiguen cierta libertad al desarrollar en un plazo de tiempo en el que ellos creen.

USER-STORIES

No vamos a introducir nada nuevo si decimos que las "user-stories" son la forma en la cuál el usuario entrega los requerimientos del sistema al equipo de trabajo. Básicamente, podemos resumir lo que son las "user-stories" como "la descripción del sistema desde el punto de vista del usuario"[10].

Para la realización de las "user stories", necesitaremos tarjetas de cartulina sin líneas en ellas, de un tamaño determinado, usualmente 4x6 o 5x8. Es conveniente proporcionar o tener disponibles muchísimas tarjetas en blanco para trabajar sobre ellas. Han de ser tarjetas físicas, no valiendo tenerlas almacenadas en el ordenador ni en ningún otro medio, deben ser tangibles, algo al alcance de cualquiera, para trabajar con mucha mayor maniobrabilidad y manejándolas fácilmente .

A la hora de crearlas, el jefe de proyecto debe proporcionar una mesa, generalmente se utiliza la mesa dónde se realizan las reuniones de apertura. En esta mesa se sentará el cliente o los clientes (generalmente dos) un grupo de dos programadores. Se dispondrán las tarjetas en blanco, en las que todos escribirán mientras dialogan acerca de las necesidades del sistema. Así, el proceso consiste en que el cliente enuncia una necesidad y se habla sobre ella, intentando que no exista ningún punto oscuro. Para ello, se irán anotando en las tarjetas lo que cada uno cree que puede ser una buena definición informal

[10] Descripción proporcionada por Kent Beck

del requerimiento. Una vez escritas las tarjetas, se evaluarán entre todos, refinándolas, para lo cuál se usarán nuevas tarjetas. El objetivo es conseguir una serie de tarjetas en las que, en cada una, encontremos una "user-story" detallada, en 4 o 5 líneas, de forma que la carga de trabajo para desarrollarla sea de más o menos una semana.

Si la tarjeta presenta una tarea complicada, que requiera un tiempo mayor, se procederá a la creación de nuevas tarjetas que serán el resultado de dividir la tarea inicial en tareas más pequeñas. Para realizar esta división o partición en nuevas tareas, se contará con la ayuda del usuario, ya que todo este proceso se realiza conversando con el diálogo, en una fase de análisis informal.

Las tarjetas contendrán 3, 4 o 5 líneas, no más, pero durante las conversaciones con el usuario, se elaborarán una serie de documentos incluyendo toda la información acerca de la tarea en cuestión, que será anexada para ser utilizada en las fases de diseño, creación de las pruebas, implementación, . . .

Las "user-stories" servirán como unidades básicas de planificación de nuestros proyectos. Para poder realizar una escala de tiempo para el proyecto en global, se debe tener la información del coste en tiempo para cada tarea, información que será ofrecida por los programadores, que son los que tienen el conocimiento técnico y la experiencia en este sentido. Evidentemente, el tamaño variará según el proyecto, por lo que la mejor técnica para conocer el tamaño adecuado para las "user-stories", es el diseñar un par de ellas al principio del proyecto analizar su tamaño con los programadores para que estimen el tiempo necesario para desarrollar la tarea.

El número de "stories" que tendremos en nuestro proyecto será de unas 60-120 para un periodo de 6 meses. Si tenemos menos de este número, convendrá particionar las existentes. Si tenemos más, tampoco ha problemas, lo único que tenemos que hacer es conseguir implementar el mayor número de tareas en el menor tiempo posible.

También ha que tener en cuenta que las cosas no son ni mucho menos estáticas, todo varía. La solución para esto es muy fácil, símplemente se debe cambiar la "user-story" al principio de una iteración y pasar a desarrollarla.

4.3 LABORES DEL JEFE DE PROYECTO

El jefe de proyecto es la cabeza visible de un proyecto, por lo que debe asumir una serie de responsabilidades bastante importantes, entre ellas, siendo una de las más importantes, que todo se cumpla de acuerdo a lo previste. Básicamente las responsabilidades del manager son los derechos del cliente.

Causar Coordinar Reportar Recompensar Eliminar obstáculos

El jefe de proyecto es la cabeza visible del proyecto, esto ha sido siempre así, sin depender de la metodología que se use. Para poder visualizar las labores, obligaciones y responsabilidades[11] del jefe de proyecto, vamos a comenzar tomando una visión del jefe de proyecto como defensor de los derechos del cliente, lo cual es cierto, ya que debe ser quién controle que se cumplan ciertos requisitos.

- El manager debe ofrecer al cliente una visión global del sistema, para que éste sepa que es lo que se va a hacer, cuándo, y a que coste.
- El manager debe conseguir que se ofrezca, por cada semana de trabajo, la mayor parte de tareas implementadas posibles.
- El manager debe ofrecer dinamismo al cliente, para que este vea un sistema en evolución.
- El manager debe ofrecer al cliente la posibilidad de cambiar los requerimientos a un coste no exhorbitante.
- El manager debe tener informado constantemente al cliente del calendario del proyecto, así como de los posibles retrasos, para que este pueda realizar las modificaciones que considere oportunas si desea rescatar el calendario original.

CAUSAR

La razón de ser de un jefe de proyecto, en primera instancia, es la de "causar", conseguir que las cosas ocurran siendo el desencadenante de todos los procesos necesarios para el desarrollo del proyecto. En alguna forma, el manager debe ser la chispa que haga que se inicie cualquier tarea en el proyecto.

Si un jefe de proyecto tiene en cuenta esto, conseguirá darse cuenta que debe estar alejado, en la medida de lo posible, del desarrollo del proyecto, ya que debe ser quién proporcione una medida objetiva o pseudo-objetiva del desarrollo del proyecto al tener una perspectiva lo más exterior posible a lo que es el desarrollo en sí. Debe tener más parte de cliente que de desarrollador, intentando ofrecer la información al equipo sobre lo que deben hacer y corrigiendo todo aquello que considere sea un fallo, o que interrumpa el transcurso normal del proyecto.

COORDINAR

Es evidente que el manager debe coordinar a todo el equipo; además, el que sea el causante de cualquiera de las acciones que deba emprender el equipo lo convierte en el coordinador inequívoco. El es quien debe distribuir en

[11] Las responsabilidades del jefe de proyecto están esquematizadas en www.xp.co.nz/management.htm

cierta medida la carga de trabajo, evitar que se realicen trabajos más de una vez, . . .

Además de coordinar a nivel de trabajo, debe controlar que el equipo esté bien no existan fisuras internas, intentando arreglar cualquier problema entre los miembros del equipo.

REPORTAR

A pesar de que la programación extrema proponga que se deben eliminar todos los papeleos que no sean útiles al 100%, siempre queda una parte de documentación que es necesaria.

Para la XP, la documentación más preciada es aquella sobre las distintas versiones, en la cuál se definen cuantas y qué "user stories" han sido implementadas en la versión y cuales serán implementadas en la futura versión.

Además de esta documentación, a veces, es necesario generar documentación para terceras parates, bien por requerimientos de cliente o por otro tipo de necesidades. Siempre se debe minimizar la carga de trabajo que necesiten estas actividades por ser consideradas como no productivas, pero nunca se deben obviar.

RECOMPENSAR

El equipo es una pieza clave, ha que conseguir lograr un buen ambiente en él, sobre todo si todo va bien, se cumplen los objetivos y se gana dinero, que es el objetivo del jefe de proyecto, de los desarrolladores de todos los que trabajan, en general. Por ello, el manager debe planificar las recompensas que se ofrecerán en caso de éxito en caso de un éxito absoluto, casos en los cuales el trabajo ha sido excelente.

Esta parte no se debe dejar a un lado, porque es algo tan importante como muchos de los otros puntos que se tratan, a que a cualquier trabajador le satisface ver recompensado su trabajo y más especialmente si este trabajo ha sido realizado con brillantez.

ELIMINAR OBSTÁCULOS

Otra labor importante del jefe de proyecto es la de eliminar cualquier obstáculo al equipo de desarrollo. Con esto se intenta que los programadores se dediquen a su labor y no se tengan que preocupar por terceras partes o por cualquier cosa ajena a lo que es su trabajo.

Por obstáculos entendemos cualquier cosa que entorpezca al programador, desde la tozudez del usuario, sus prioridades, hasta problemas debidos a infraestructura, problemas con la red, . . .

4.4 EL CLIENTE[12]

En gran medida, el jefe de proyecto es el "defensor del cliente", en otra gran medida es "defensor del equipo". Esto es así, porque debe intentar hacer ver a su equipo las necesidades del cliente y por otra parte debe limitar las exigencias del cliente a lo posible. Esta labor es dificil, pero se consigue teniendo al cliente como objetivo final a satisfacer, el cliente es el jefe del manager, en cierta medida.

El Cliente es el objetivo último, él es a quién debemos satisfacer. Tener esto claro nos ayuda a resolver muchos problemas.

Básicamente, el cliente debe estar en todas las partes de nuestro proyecto, desde el comienzo hasta el final. Para empezar, el análisis se hace con él, los test se deben enseñar al usuario para que éste de su aprobación y el Cliente recibirá todas las versiones del producto que se vayan produciendo.

Además, el cliente debe resolver cualquier duda de cualquier desarrollador acerca de lo que éste quiere conseguir con una determinada tarea. El cliente es la persona idonea, más bien, es la única que tiene el conocimiento, para resolver estas dudas, por lo que nunca se debe dudar el consultarle sobre cualquier tema relacionado con el proyecto, a nivel de negocio.

Como conclusión, se podría decir que la clave para un desarrollo correcto al 100% es la coordinación entre los programadores y el cliente.

4.5 CONTRATOS[13]

Otra labor del manager que resulta realmente escabrosa. Tanto fijar una fecha para el fin del proyecto, como fijar un presupuesto para él, son labores que nos pueden conducir al fracaso como jefe de proyecto. En cuanto al presupuesto, es muy fácil intentar hacer mucho dinero con un proyecto, pero estamos en malos tiempos para esto debido a la gran competencia. El otro extremo es peligroso, porque cuando estamos perdiendo dinero, es que la cosa va realmente mal.

[12] Podemos encontrar la figura del cliente en "XP-All the Rest" de Ian Mitchell en www.xp.co.nz/on-site_customer.htm

[13] La principal fuente de información sobre contratos es "Optional Scope Contracts" de Kent Beck Dave Cleal

A cualquier programador se le puede pedir que desarrolle un proyecto determinado en un tiempo determinado, siempre que el plazo de tiempo se razonable, sea cuál sea la dificultad del proyecto, el programador se pondrá a hacerlo, con más o menos gusto. Pero si a este programador le pedimos que evalúe los costes del proyecto, seguramente le estemos dando un gran disgusto.

En el mundo de la informática, los contratos para los proyectos suelen establecer tiempos de espera y costes fijos, o, por lo menos, con un margen muy pequeño de posibles modificaciones. Esto es así porque es la manera que casa mejor con las necesidades del cliente y del equipo.

- Para el cliente, ofrece:
  - ✓ Costes Predecibles.
  - ✓ Calendario predecible.
  - ✓ Entregas predecibles.
- Para el proveedor/equipo, ofrece:
  - ✓ Ingresos predecibles.
  - ✓ Demanda predecible.

Todo esto son características visibles a priori, el problema viene a la hora de trabajar en proyectos reales, a que muchas veces, encontramos que algún requerimiento es más complicado de implementar que lo que a priori parecía. En estos casos, como el cliente no va a pagar más ni va a permitir que alarguemos la entrega, probablemente, entreguemos una parte con un nivel de calidad inferior, debido a tener que trabajar más, en menos tiempo y a desgana. Con este cuadro, el equipo esta perdiendo dinero, pero, a largo plazo, también lo pierde el cliente de forma indirecta al perder calidad en el producto.

Con todo lo visto, nos podemos dar cuenta que estamos ante una filosofía que pone al cliente y al proveedor en contra uno de otro, lo cuál se verá mejor, todavía, en el siguiente cuadro.

Cliente	*Proveedor*
Interpreta los requerimientos a lo ancho, es decir, intenta conseguir muchas más características por el mismo precio	Interpreta los requerimientos a lo estrecho, intentando reducir los recursos necesarios.
Quiere el trabajo lo antes posible.	Intenta tener el trabajo hecho justo en la fecha prevista para tener el siguiente contrato preparado.
Quiere una calidad excelente.	Intenta ofrecer la calidad justa que cree que se merece lo pagado por el cliente.

Los programadores estresados no son su problema.	Quiere que sus trabajadores estén bien, para que estén preparados para el siguiente trabajo.

Para evitar muchos de estos problemas, debemos tener en cuenta una serie de variables que afectan a los proyectos informáticos:

- Tiempo.
- Coste.
- Ámbito de acción.
- Calidad.

Ahora, la mejor manera de fijar un contrato es tener en cuenta estas variables. Además, incluso podemos eliminar la variable de calidad, porque, hoy por hoy, los programadores quieren ofrecer calidad, y usan estándares diversos para ello, porque son conscientes que su trabajo lo requiere.

Vamos a ver una buena manera de hacer contratos variables teniendo en cuenta las variables mencionadas. Un ejemplo, podría ser algo parecido o basado en lo siguiente.

"...Se pagará una cantidad de 75.000 euros al mes durante los próximos dos meses. Cualquier software que se produzca deberá cumplir los estándares de calidad acordados. Existen una serie de estimaciones iniciales en el Apéndice A, pero no representan nada a tener en cuenta...."[14]

Aunque parezca un contrato sin sentido, a que no se fijan objetivos claros, tenemos ventajas por ambos lados. Para el lado del equipo, tenemos que no se fijan unos requerimientos en un determinado tiempo, se deja trabajar en relativa tranquilidad. Por el lado del cliente, se está arriesgando poco, porque los proyectos suelen ser más largos, de un año, como poco, por lo que solo arriesga 1/6 parte del total del coste del proyecto, además, sabe que el equipo lo quiere hacer bien para seguir trabajando con él.

Vamos a revisar ahora el cuadro que enfrentaba anteriormente a cliente a proveedor.

Cliente	*Proveedor*
Interpreta los requerimientos a lo ancho, es decir, intenta conseguir muchas más características por el mismo precio	Encantado de aceptar cambios de interpretación en los requerimientos.
Quiere el trabajo lo antes posible.	Quiere el trabajo hecho a tiempo. Pero está encantado de poder ofrecer la mayor funcionalidad posible siempre

[14] Ejemplo extraído de "Optional Scope Contracts" de Kent Beck Dave Cleal.

	que no se pierda calidad.
Quiere una calidad excelente.	Quiere calidad ante todo.
Quiere que los programadores trabajen para el otros dos meses más (al menos hasta acabar todo el trabajo.	Quiere que sus trabajadores tengan éxito en este proyecto para conseguir más proyectos

El cuadro que antes reflejaba un enfrentamiento, ahora refleja un gran acercamiento de posturas, y no hemos perdido casi nada de lo que ofrecían los contratos fijos, solamente, las entregas predecibles, pero esto es algo muy dificil de cumplir, por lo que en los proyectos reales, tampoco se ofrece al 100%. Por tanto, todo son ventajas.

5.1 COMUNICACIÓN EN EL EQUIPO.

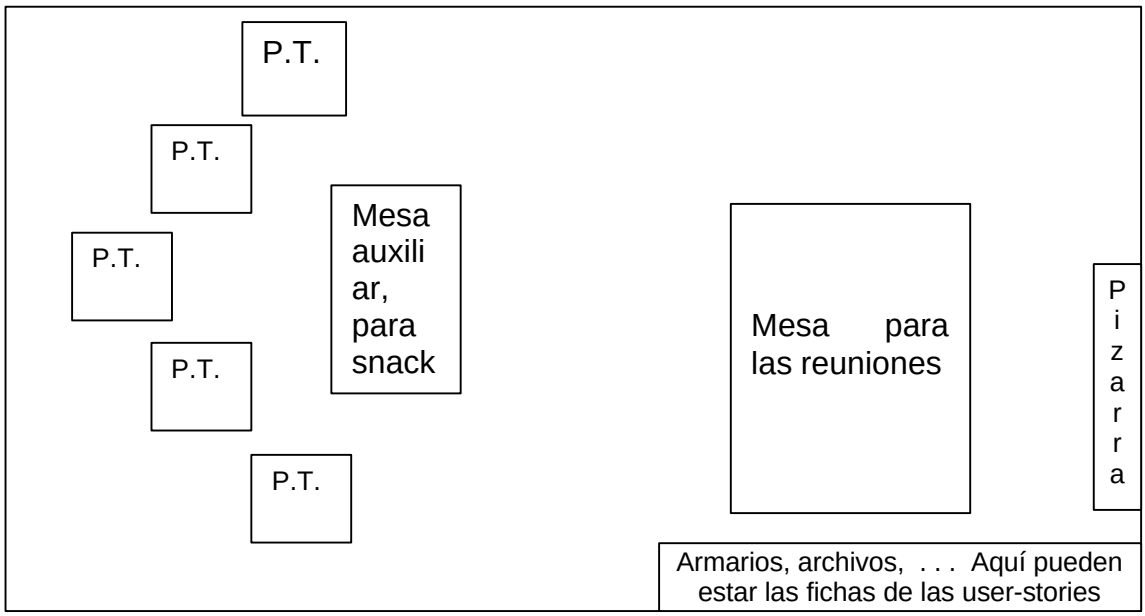
La comunicación está considerada como uno de los valores principales de la XP ya que con ella logramos eliminar gran cantidad de pasos intermedios, como la documentación, a la hora de desarrollar un proyecto software.

Un entorno de trabajo orientado a la comunicación.

El lector que halla seguido el orden de capítulos marcado por el libro, a estas alturas de la lectura, no se verá sorprendido por encontrarse un apartado dedicado a la comunicación dentro del equipo[15].

UN ENTORNO DE TRABAJO ORIENTADO A LA COMUNICACIÓN[16].

En este apartado vamos a describir lo que consideramos como un entorno de trabajo orientado a la buena comunicación entre los componentes del equipo. Asumimos unos 10-12 componentes, más el jefe de proyecto. Todo lo que se exponga aquí, es una referencia puede ser llevado a otras necesidades un tanto más concretas.



[15] Considerando, siempre, al usuario como uno más del equipo.
[16] Esta es una orientación. Original de José Carlos Cortizo.

Este esquema para trabajar nos aporta varias cosas. La primera, los grupos de trabajo(parejas), se sientan cerca unos de otros, intentando formar un semicírculo abierto, por dos razones, la primera, el poder verse unos a otros. Esto permite un gran contacto visual, y, en muchas ocasiones, logra un ambiente más distendido, impidiendo el que cada grupo se cierre a su trabajo. La segunda ventaja que se logra es que todos tienen a vista la pizarra dónde se anotan las tareas a realizar, en la reunión de apertura.

Evidentemente, la disposición de los puestos de trabajo puede ser distinta, pero una distribución similar a esta nos permitirá ganar bastante, sobre todo contando con grupos de 10-12 personas, porque, si no, es mucho más dificil el conseguir esto en una habitación no muy grande. Si la habitación es muy grande, se pierde el contacto con la pizarra y muy facilmente el contacto visual entre componentes.

El que la pizarra se encuentre en la zona de trabajo presenta las ventajas que ya hemos visto, pero lo no tan claro es que la mesa de reuniones esté en el mismo sitio. Otra posibilidad sería tener una pizarra móvil y tener las reuniones en otra sala. Nuestro punto de vista es que la primera opción es mejor, ya que perdemos menos tiempo al no tener que andando de una habitación a otra (lo cuál no es una tontería si contáramos las horas que se pueden llegar a perder por movimientos entre habitaciones, con sus consecuentes pérdidas de tiempo al encontrarse a alguien en el pasillo, . . .) y además conseguimos que cuando el cliente venga a exponer sus user stories ya sea en una reunión de apertura o para tratarlas a nivel más particular con algunos programadores, él se integre en el ritmo de trabajo del equipo, el equipo tenga una visión más global del proyecto.

Otro punto importante es tener los "snacks" en un lugar cercano a los puestos de trabajo. Esto sirve para que cuando los programadores decidan tomarse un respiro por un momento, beber agua, . . . no se rompa el ritmo de trabajo al estar en contacto con el resto, perdiendo una cantidad mínima de tiempo, y, además, logramos que no se cohiban de tomarse un momento de alivio al tener el derecho reconocido las facilidades dispuestas para ello.

5.2 PAIR PROGRAMMING[17]

El pair programming es una filosofía de trabajo que no solo nos es útil en programación, si no es aplicable a muchos campos. Se basa en un trabajo por parejas y aunque parezca deshechar recursos, esto no es así.

¿Qué es? Historia Beneficios Inconvenientes

[17] La fuente de información sobre Pair Programming usada es Laurie Williams de la Universidad de Carolina del Norte, con los documentos "Why have two do the work of One?" y "The Costs and Benefits of Pair Programming" junto con Alistair Cockburn.

Hemos hablado mucho acerca del "pair programming" y es muy probable que el simple nombre nos de una idea cercana de lo que en realidad es. Hay que tener mucho cuidado acerca de lo que se dice del "pair programming", porque no es el simple hecho de programar en parejas.

¿QUÉ ES EL PAIR PROGRAMMING?

El "pair programming" es una manera de trabajar según la cual dos desarrolladores se responsabilizan de una parte del desarrollo y lo realizan los dos juntos, en el mismo puesto de trabajo y usando un único ordenador, por lo que uno lo utilizará mientras el otro revisará lo que hace y ambos discutirán acerca de cómo afrontar cada problema.

Hay que tener cuidado, porque en esto no es solo para la programación en sí, si no que es útil para todo el desarrollo de software, desde el análisis, diseño, pruebas, . . .

Puede que parezca que dos personas estén haciendo el trabajo de uno solo, pero esto no es así. Vamos a ver una serie de críticas al "pair programming" desde los distintos puntos de vista en un proyecto.

- Para los jefes de proyecto, los programadores son el recurso más valioso, y por tanto, se ve como algo contra natura el necesitar el doble de gente para hacer la misma labor.

- Desde el punto de vista del programador, es difícil asumir el trabajar con otra persona porque la programación es una tarea tradicionalmente individual además, muchos programadores creen que su código es algo personal que tener otra persona al lado les enlentece.

Tenemos que ver otro punto de vista, para poder evaluar la situación con mayor criterio.

- Muchos programadores "respetables", prefieren trabajar en parejas, haciendo del "pair programming" su modalidad de trabajo preferida.

- Las parejas que han trabajado juntas durante cierto tiempo, aseguran que el trabajar en parejas consigue el alcanzar objetivos mayores que el doble de trabajo para uno solo.

- Por regla general, el diseño que se realiza en parejas es mejor, más simple y más fácil para reutilizar.

- Permite el entrenamiento de programadores novatos y el que los expertos aprendan de estos, también.

HISTORIA DEL "PAIR-PROGRAMMING"[18]

Aunque el "pair programming" se asocia a la XP, es una práctica totalmente independiente y aplicable a cualquier metodología. Además, la programación en parejas es una actividad mucho anterior a la programación extrema (1996), a que sus orígenes se fijan en torno a 1953.

En 1953, Fred Brooks otra gente ya programaban en parejas, aunque todavía no lo denominaban así, lo veían como una actividad normal.

Ya en 1995, Larry Constantine publicaría el libro "Constantine on Peopleware", en el que se trata acerca de los Duos Dinámicos que producen código más rápidamente y con menos fallos. También sobre esta fecha aparece el libro de Jim Coplien, "Developing in Pairs", en el cuál se puede leer "....juntos, pueden producir más código que la suma de lo que harían individualmente...".

En 1996, en la base aérea de Hill, se comienza a usar la programación en parejas, con bastante éxito, ya que se consiguen unos resultados de 175 líneas por persona y mes mientras que antes se conseguían unas 77 líneas por persona y mes; además, se redujeron los errores en tres órdenes de magnitud. En estas pruebas, también se concluyó que tenía unas ventajas añadidas, al observar fenómenos como energía focalizada, "brainstorming", continuo diseño codificación la vez, . . .

A partir de esta fecha, se comienzan realizar muchísimos ensayos acerca de la programación en parejas, sobre todo en universidades y con estudiantes. La mayoría de ellos concluyen que el "pair programming" es rentable, sobre todo en téminos de calidad.

BENEFICIOS[19]

El trabajar en parejas tiene muchas más virtudes que defectos, por lo que vamos a entrar de lleno en todos los puntos a favor del uso de esta técnica.

- Satisfacción: La gente que h trabajado en parejas, ha encontrado la experiencia como algo más divertido que el trabajar solo.

- Calidad en el diseño: Las parejas producen programas más cortos, con diseños mucho mejores.

- Revisiones continuas: El trabajar codo con codo hace que las parejas estén continuamente pensando en mejorar ciertas cosas ya realizadas, por lo que conseguimos unos ratios de eliminación de defectos bastante buenos.

[18] Historia del "pair programming" extraida de "Why have two do the work...."
[19] Figuras 1 y 2 extraidas de "Why have two do the work....." de Laurie Williams

- Solución de problemas difíciles: Según varios estudios, las parejas son capaces de resolver problemas difíciles en unos tiempos más que aceptables, mientras que los programadores solos, tardarían mucho si es que encontraran la solución.

- Aprendizaje: Evidentemente en las parejas se aprende unos de otros.

- Trabajo en equipo: Se aprende a trabajar en equipo.

- Gestión del proyecto: Se consigue mejor calidad en menor tiempo, evidentemente esto ayuda a la gestión del proyecto.

Ahora veremos las ventajas de la programación en parejas acorde con las típicas exigencias de un cliente:

- El cliente siempre exige calidad, vamos a ver un gráfico que determina una comparativa en cuanto a calidad entre los trabajos hechos individualmente y los realizados en parejas.

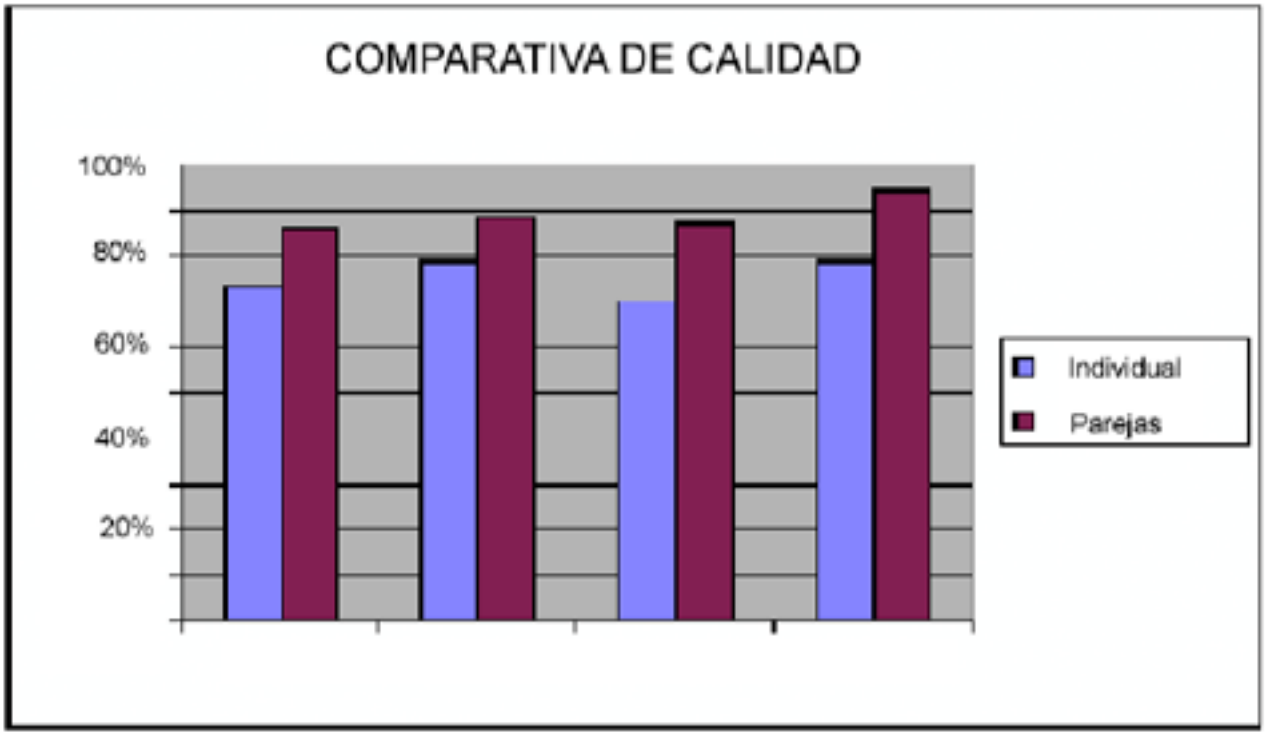


Figura 1

- El cliente también necesita rapidez, quiere las cosa, tipicamente, “para ayer”, vamos a ver una comparativa en cuanto al tiempo empleado.

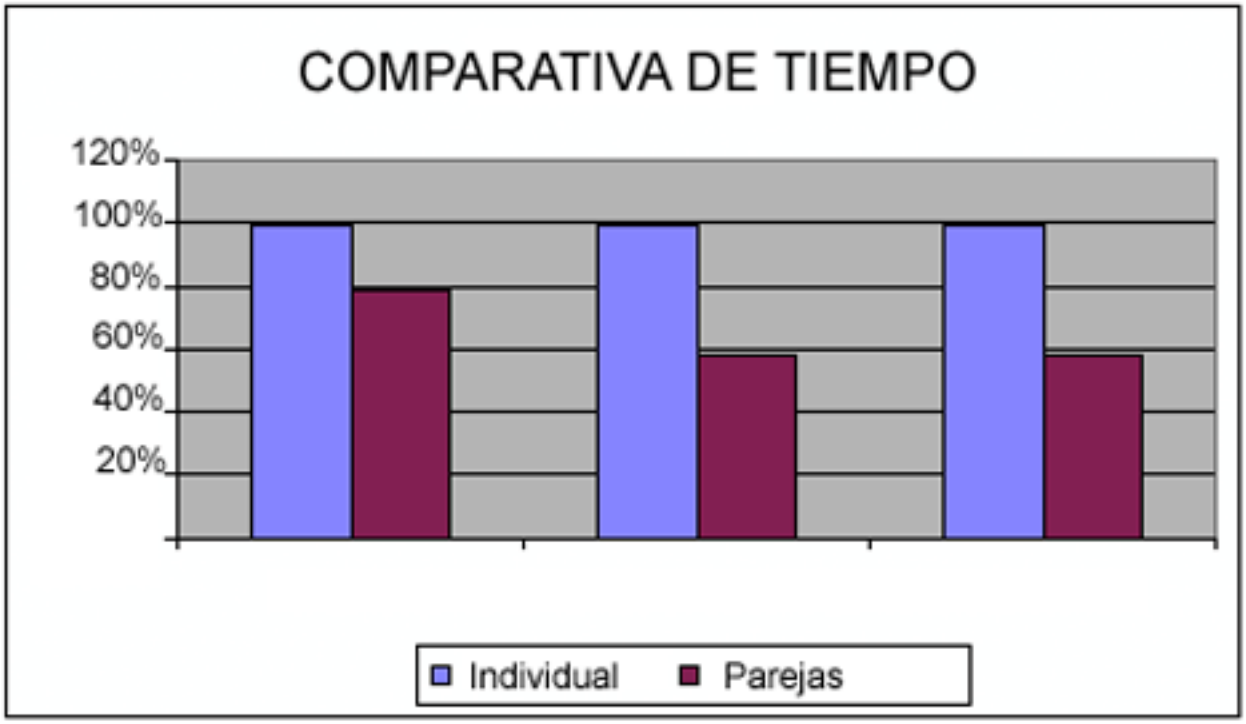


Figura 2

INCONVENIENTES

Las mayores desventajas acerca de la programación en parejas, vienen de la mano de su implantación. Es decir, de por si, la programación en parejas es una actividad ventajosa, pero conseguir su implantación en un equipo es algo que puede resultar dificultoso, sobre todo si el equipo no está acostumbrado a este tipo de técnicas.

6.1 REFACTORING

Refactoring se refiere a un grupo de técnicas de codificación usadas para mejorar código defectuoso escrito por otros programadores "y de vez en cuando" por los propios creadores de código. Se ha de confiar en las pruebas a las que someteremos nuestro código para mantener la confianza de que el "Refactoring" no ha perjudicado algo más y los cambios producidos han sido demasiado pequeños.

Modelos de Refactorización

A veces cuando una nueva "historia de usuario" va a ser implementada es normal que los objetos existentes deban ser estructuralmente cambiados o incluso debamos crear nuevos objetos. Se discutirá entonces la necesidad de refactorizar por parte del equipo, que confirmara la validez o no de dicho proceso.

Se trabajará en la nueva estructura del objeto y cuando se este conforme será publicado al equipo de programación. Una vez decidida su aprobación entonces una pareja comenzara con la implementación del nuevo objeto, el código existente de ese nuevo objeto será incluido de nuevo en el objeto, donde la nueva funcionalidad no esta todavía incluida.

Cuando la nueva unidad funcional esta terminada entonces la pareja comenzará con la fase de pruebas; después se deben empezar a cambiar todas las invocaciones y ejecutar todos los métodos de integración de ambos componentes (para producir lo que en un futuro será el nuevo objeto) , que serán analizados por la propia pareja de creación de la nueva funcionalidad. Entonces y solo entonces los nuevos cambios del código serán "publicados" al resto del equipo.

Entonces las nuevas funcionalidades son adheridas al proyecto y la secuencia de codificación y análisis normal continua su curso.

Debemos señalar la importancia que en el refactoring se le da a la fase de pruebas, a que son sometidos los cambios que efectuemos a nuestro nuevo código ya que se puede entender que la modificación de código no es nada aleatoria y cualquier fallo podría tener consecuencias catastróficas de cara a la operatividad de nuestro software.

Por esto después de cada paso son analizados y probados los nuevos cambios efectuados para verificar su funcionalidad y robustez.

Modelos de Refactorización

Los modelos de refactorización son actualmente objeto de estudio y están en constante evolución, sirva de ejemplo la siguiente selección:

- Encadenación de los constructores
- Reemplazo de constructores múltiples por métodos de fábrica
- Encapsulamiento de subclases con métodos de fábrica
- Creación de clases
- Reemplace cálculos condicionales con "Strategy"
- Reemplace Construcciones Primitivas con Compuestas
- Encapsule Compuestos con Constructores
- Extraiga la Lógica de los caso-uso hacia "Decorators"
- Reemplace las Notificaciones de código por "Observadores"
- Transforme Acumulaciones a un Parámetro Colectivo
- Métodos Compuestos

Dada la extensión de este punto, intentaremos no extendernos demasiado en la explicación de estos modelos.

6.1.1 Contructores Encadenados

Si tienes varios constructores que contienen código duplicado, el objetivo es unir los constructores para obtener menos código duplicado.

Veamos el siguiente ejemplo:

```
public class Loan {
...
public Loan(float notional, float outstanding, int rating, Date expiry) {
this.strategy = new TermROC();
this.notional = notional;
this.outstanding = outstanding;
this.rating =rating;
this.expiry = expiry;
}
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
this.strategy = new RevolvingTermROC();
this.notional = notional;
this.outstanding = outstanding;
this.rating = rating;
this.expiry = expiry;
this.maturity = maturity;
}
public Loan(CapitalStrategy strategy, float notional, float outstanding,
int rating, Date expiry, Date maturity) {
this.strategy = strategy;
this.notional = notional;
this.outstanding = outstanding;
this.rating = rating;
this.expiry = expiry;
this.maturity = maturity;
}
}
```

```
public class Loan {
...
public Loan(float notional, float outstanding, int rating, Date expiry) {
this(new TermROC(), notional, outstanding, rating, expiry, null);
}
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
this(new RevolvingTermROC(), notional, outstanding, rating, expiry, maturity);
}
public Loan(CapitalStrategy strategy, float notional, float outstanding,
int rating, Date expiry, Date maturity) {
this.strategy = strategy;
this.notional = notional;
this.outstanding = outstanding;
this.rating = rating;
this.expiry = expiry;
this.maturity = maturity;
}
}
```

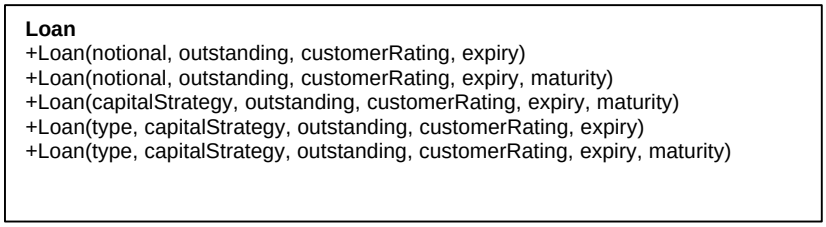
El código que esta duplicado en varios de los constructores de la clase es una invitación para tener problemas.

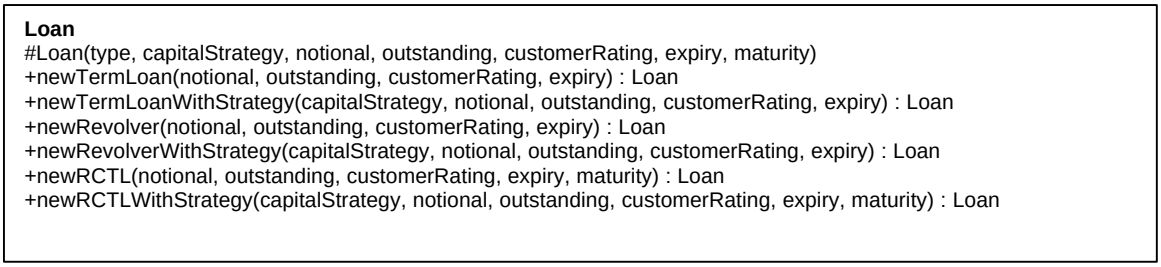
Alguien que cree una nueva variable a la clase , actualiza el contructor para inicializar la variable , pero abandona sin querer la actualización en los otros constructores, y bingo ya tenemos nuestro error asegurado. Por lo tanto es una buena idea reducir estos constructores que no hacen más que infringir un peligro.

Si tienes el constructor al final de la cadena, tenemos varias ventajas además de evitar el fallo previamente comentado, llamaré a todos los constructores de la cadena; incluso esta composición acepta más parámetros que los otros constructores y pueden o no ser privados o protegidos.

6.1.2 Reemplazo de constructores múltiples por métodos de fábrica.

Muchos constructores en una clase hacen difícil decidir cuales debemos usar durante el desarrollo. Entonces debemos reemplazar esos constructores por métodos.





Los programadores son a menudo motivados a usar este modelo para hacer su software más flexible. Pero no hay otra motivación más importante que la de hacer el código más fácilmente comprensible para su posterior uso. Una forma de hacer esto es encapsular los constructores que no se entienden tan bien como los métodos.

Pongamos un ejemplo, digamos que queremos instanciar una manzana de una clase de varias formas:

- Con o sin pepitas
- Pelada o no
- Pais de origen
- Familia de la manzana

Estas opciones presentan varios tipos de manzanas diferentes, aunque no estén definidas como una subclase de manzana. Para obtener una instancia como nosotros queremos tenemos que llamar al constructor correcto, pero hay varios que podrían servirnos, podemos intentar mejorar esta situación encapsulando los constructores de la clase manzana con métodos parecidos a estos:

```
public Apple createSeedlessAmericanMacintosh();
public Apple createSeedlessGrannySmith();
public Apple createSeededAsianGoldenDelicious();
```

Estos métodos son más fácilmente comprensibles que los constructores, pero surge un pequeño problema, ¿ que pasa si yo quiero crear una manzana de España? Pues lógicamente pasando parámetros a los constructores, véase el ejemplo:

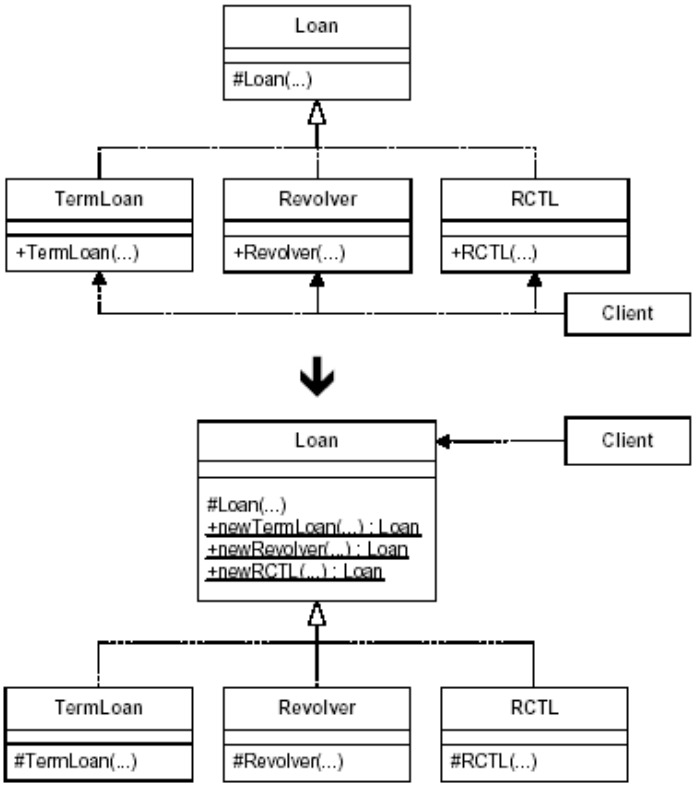
```
public Apple createSeedlessMacintosh(Country c);
public Apple createGoldenDelicious(Country c);
```

Una cosa importante de esto es que llega a ser fácil de ver y de usar, aunque también algunos de estos métodos podemos pensar que no serán de utilidad del todo.

El problema reside principalmente en los constructores ya que hay determinadas cosas que no permiten realizar y deberíamos tener muchos constructores para implementar todos los posibles casos.

6.1.3 Encapsulamiento de subclases con métodos de fábrica

Subclases implementan un interfaz común pero son construidas de modos diferentes.



El motivo para hacer esta refactorización es ocultar los detalles de implementación para aquellas personas que no necesitan conocerla.

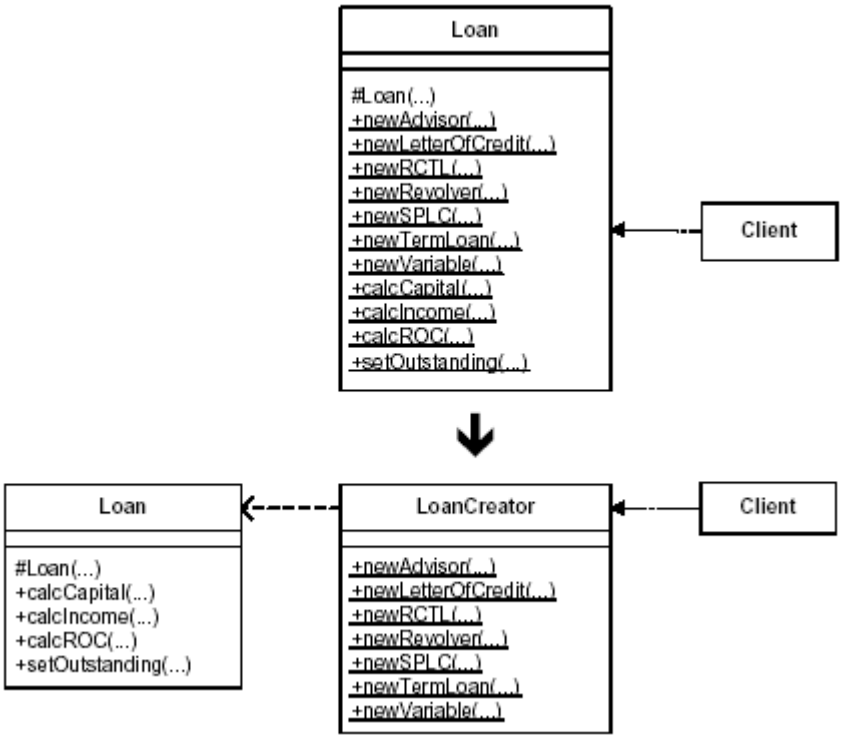
Consideremos un diseño en el que tenemos numerosas clases que implementan un interfaz común. Ahora bien si no protegemos los constructores

de las subclases, se permitirá a los clientes hacer una instanciación de los constructores.

Una vez que los que los programadores escriben código que llama directamente a una subclase en lugar de hacerlo a través del interfaz común, el camino esta abierto para cambiar el interfaz de la subclase en respuesta a las necesidades del código Cliente, esto desgraciadamente sucede una y otra vez, una vez que esto sucede la subclase alterada es diferente del resto de subclases y de su clase base: esta tiene métodos especiales en ella que la restringen sólo a la interfaz y no a la interfaz común. Puede parecer que no sea algo muy importante, pero esto conduce a la creación de código poco vistoso ("feo").

6.1.4 Creación de clases

Demasiados métodos de construcción dentro de una clase la hacen cumplir tareas que no están dentro de sus responsabilidades, el objetivo es mover dichos métodos a clases auxiliares que cumplan dichas tareas.



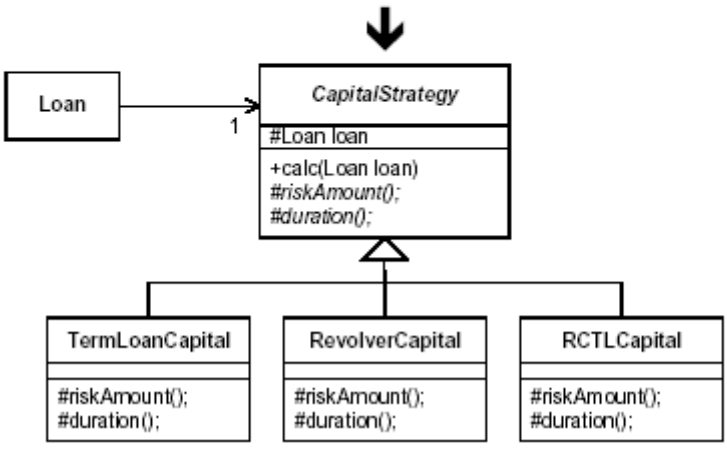
Esta reforma es esencialmente capturar todos aquellos métodos de creación y pasarlos a una clase creada especialmente para ello . No hay nada de malo en tener unos pocos métodos de creación en una clase, pero como el numero de ellos crece las responsabilidades de nuestra clase pueden ser "contaminadas". A veces es mejor restaurar la identidad de la clase extrayendo todos estos métodos de creación a otra clase.

Estas clases son habitualmente implementadas como clases que contienen métodos estáticos los cuales solamente crean instancias de objetos, del mismo modo que fueron especificadas por la propia clase.

6.1.5 Reemplace cálculos condicionales con Strategy

Usamos demasiadas condiciones lógicas para realizar calculos, debemos delegar esta misión a objetos de estrategia.

```
public class Loan ...
  public double calcCapital() {
    return riskAmount() * duration() * RiskFactor.forRiskRating(rating);
  }
  private double riskAmount() {
    if (unusedPercentage != 1.00)
      return outstanding + calcUnusedRiskAmount();
    else return outstanding;
  }
  private double calcUnusedRiskAmount() {
    return (notional - outstanding) * unusedPercentage;
  }
  private double duration() {
    if (expiry == null)
      return ((maturity.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else if (maturity == null)
      return ((expiry.getTime() - start.getTime())/MILLIS_PER_DAY)/365;
    else {
      long millisToExpiry = expiry.getTime() - start.getTime();
      long millisFromExpiryToMaturity = maturity.getTime() - expiry.getTime();
      double revolverDuration = (millisToExpiry/MILLIS_PER_DAY)/365;
      double termDuration = (millisFromExpiryToMaturity/MILLIS_PER_DAY)/365;
      return revolverDuration + termDuration;
    }
  }
  private void setUnusedPercentage() {
    if (expiry != null && maturity != null) {
      if (rating > 4) unusedPercentage = 0.95;
      else unusedPercentage = 0.50;
    } else if (maturity != null) {
        unusedPercentage = 1.00;
    } else if (expiry != null) {
        if (rating > 4) unusedPercentage = 0.75;
        else unusedPercentage = 0.25;
    }
  }
```



Muchas condiciones lógicas pueden disimular algún calculo, incluso aunque solo sea una. Cuando esto ocurre, tus calculo pueden ser mal interpretados .

Nuestro modelo es una ayuda para realizar bien este tipo de situaciones.

Un objeto del contexto obtiene un objeto de la Estrategia y entonces delega uno o varios cálculos a esa Estrategia.

Esto aclara el contexto de la clase moviendo este tipo cálculos condicionales a una pequeña colección de "estrategias" de cálculos independientes, cada uno de los cuales puede manipular una de las varias formas de hacer los cálculos.

Podemos pensar que muchos de los cálculos lógicos que efectuamos no son lo suficientemente complicados como para justificar el uso de esta estrategia; podríamos pensar también que aunque hubiera suficientes condiciones lógicas deberíamos considerar la opción de si usar métodos de plantilla no sería una mejor opción, pero usando esto asumimos que podemos poner el “esqueleto” de nuestros cálculos en una clase base que tiene subclases suministrando alguno o todos de los detalles del cálculo. Esto puede o no ser posible dada nuestra situación. Por ejemplo si tienes subclases y varias formas de calcular algo no será fácilmente encajable en esas subclases.

Puedes ver que situando cálculos en clases separadas, limitaras tu habilidad de cambiar un tipo de calculo por otro, esto podría significar que cambiando el tipo del objeto con el que estas trabajando, sería preferible antes que sustituir un objeto de "Strategy" por otro.

Una vez que decidimos refactorizar, debemos considerar cómo los cálculos situados en cada clase tendrán acceso a las variables que necesitan para realizar sus cálculos. Para conseguir esto debemos pasar como parámetro la clase base como referencia al objeto Strategy, y sean cuales sean las variables que necesite el objeto Strategy sean accesibles por medio de métodos públicos.

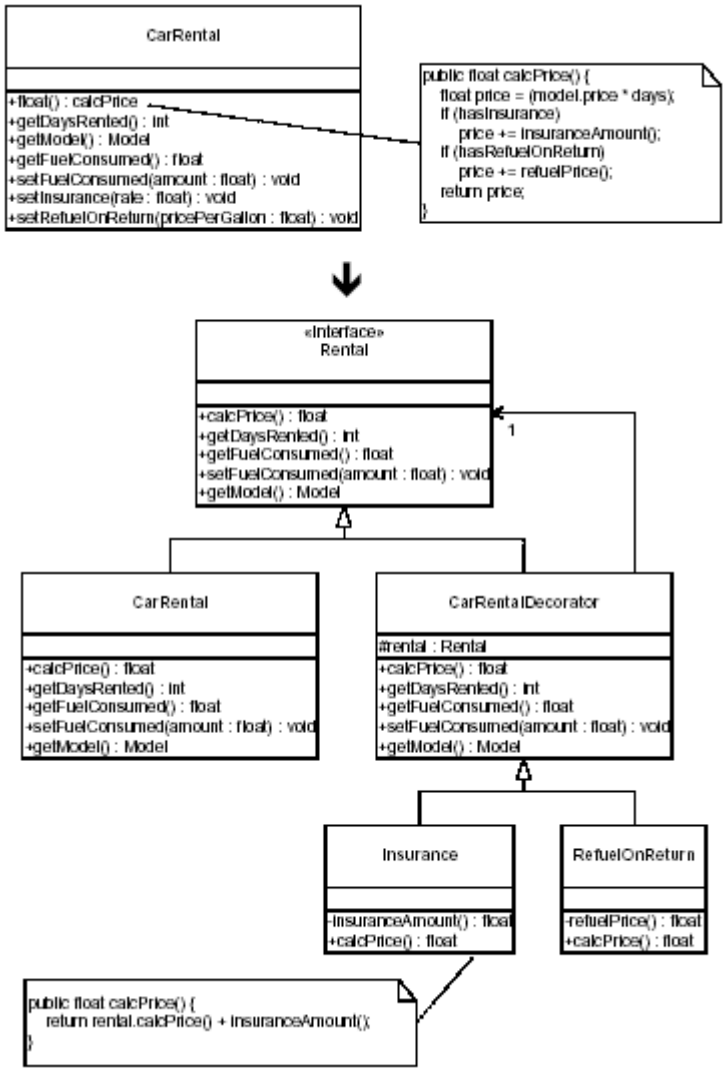
El punto final se basa en ver como la clase final obtendrá la estrategia que debe usar. Siempre que nos sea posible debemos proteger al código cliente de tener que preocuparse acerca de las instanciaciones a Strategy y de los accesos al constructor.

Para ello podemos usar métodos: simplemente definiremos un método que devuelva una instancia, propiamente equipado con la Estrategia apropiada, para cada caso.

6.1.6 Extraiga la Lógica de los caso-uso hacia Decorators

Nuestras clases o métodos tiene opcionales o casos especiales que tratan la lógica.

Debemos conservar la esencia de esta lógica pero debemos exportar los casos especiales hacia "Decorators".



Podríamos decir que puede ser uno de los más simples y elegantes pero muchas veces debemos resistirnos a usarlo ya que muchos de los problemas a los que nos enfrentamos son excesivamente sencillos para su uso y más que una ayuda implicaría un retraso, ya que normalmente las soluciones sencillas suelen ser las mejores, no obstante siempre dispondremos de una oportunidad para comprobar la claridad y simplicidad que puede proporcionar a nuestro código.

¿Entonces en que tipo de circunstancias debemos usar nuestro modelo?, bueno para ello lo mejor es ver algún ejemplo. Consideremos una clase factura que es responsable de guardar los datos de una factura realizada para un

cliente, la mayoría de las facturas son simples; principalmente la cantidad adeudada y los objetos o conceptos de los cuales calculamos dicha cantidad, pero ¿ qué pasa cuando la cantidad debida se retrasa o si queremos aplicar un descuento al cliente porque es un cliente especial?, estas son dos condiciones especiales que el método calcAmountOwed tendrá que encargarse . Nosotros probablemente no necesitamos todavía un "Decorator" para encargarse de estas condiciones simples.

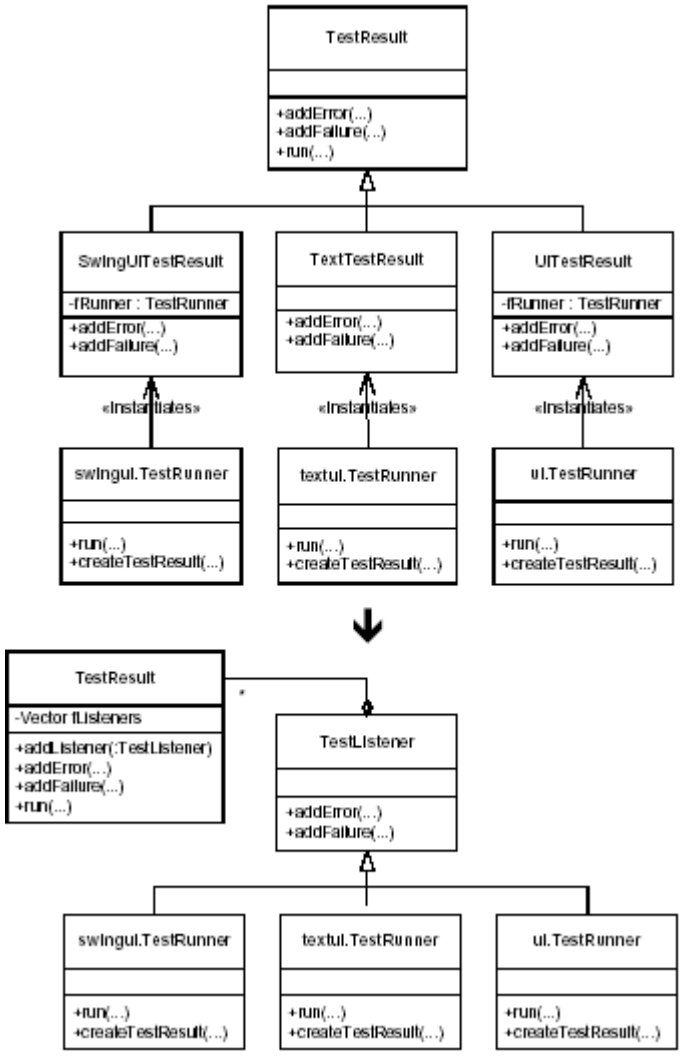
Pero, ¿qué sucede si añadimos cada vez más y más condiciones a nuestro método?, entonces nuestro método comenzará a complicarse, cada vez necesitamos más variables auxiliares y más métodos … lo que complica nuestra clase indebidamente.

Ahora nuestra clase es muy compleja, pero debemos tener en cuenta que es posible que la mayoría de la veces nuestro método ejecute el caso base, con lo que debemos preguntarnos si merece la pena haber complicado tanto nuestro método, manteniendo esta lógica de trabajo debemos darnos cuenta que nuestro método se ha vuelto pesado de leer y de comprender y es aquí en donde entra en juego nuestro modelo de refactoring.

Existen algunos trabajos encargados de implementar esta refactorización, en Java refactorizar a un "Decorator" involucra crear una clase una clase "Decorator" y algunas subclases para esta clase base. Esto implica una buena cantidad de trabajo así que sólo tendrá sentido hacer esto si tenemos "buenos trozos de código" para los que merezca la pena realizar esta implementación.

6.1.7 Reemplace las Notificaciones de código por Observadores

Nuestras clases o subclases suelen realizar normalmente notificaciones entre ellas en diversos momentos.



Este modelo es bien conocido por los programadores, ya que se suele emplear habitualmente, pero el truco esta en saber cuando debemos usarlo y cuando no, a continuación podemos ver algunas de las razones para ello.

- Cuando una abstracción tiene dos aspectos, uno dependiente del otro, encapsulando éstos aspectos en objetos separados le permiten variarlos y rehusarlos independientemente.
- Cuando un cambio en un objeto necesita cambiar otros y no estamos seguros de que objetos necesitan ser cambiados.
- Cuando un objeto debe poder notificar a otros objetos sin hacer suposiciones de quien son esos objetos.

Entonces que pasa cuando sabemos que objetos debemos actualizar y no necesariamente tenemos “contacto con ese objeto”.

Por ejemplo supongamos que una clase “A” necesita actualizar objetos de la clase “B”, puesto que esto es una responsabilidad de la notificación entonces podemos querer llevar a cabo el correspondiente uso de este modelo , pero realmente estamos seguros de llegar tan lejos o por el contrario esto puede ser algo excesivo para cubrir nuestras necesidades. ¿Qué pasaría si escribimos código en la clase A que informe a los objetos de la clase B cuando sea necesario?.

Ciertamente esto podría trabajar bien pero nuestra solución se complica si además la clase C debe recibir cambios por parte de nuestra clase A entonces debemos aumentar el tamaño de nuestra clase para cubrir nuestras necesidades; lo que podemos percibir aquí es el aumento de tamaño y de complicación que puede surgir si debemos notificar a un numero de clases algo más elevado cambios que se produzcan en nuestra clase A.

La respuesta a todas estas preguntas debe ser la opción de optar por elegir nuestro modelo, pero su uso nos debe llevar a la implementación de un código más simple, pequeño y fácil de leer.

6.1.8 Transforme Acumulaciones a un Parámetro Colectivo

Si tienes un método simple que almacena información en una variable, debemos dividir en submétodos el proceso de actualizar nuestra variable que pasaremos por parámetro a dichos métodos.

```
class TagNode. . .
public String toString() {
   String result = new String();
   result += "<" + tagName + " " + attributes + ">";
   Iterator it = children.iterator();
   while (it.hasNext()) {
      TagNode node = (TagNode)it.next();
      result += node.toString();
   }
   if (!tagValue.equals(""))
      result += tagValue;
   result += "</" + tagName + ">";
   return result;
}
```

```
class TagNode. . .
public String toString() {
   return toStringHelper(new StringBuffer(""));
}
private String toStringHelper(StringBuffer result) {
   writeOpenTagTo(result);
   writeChildrenTo(result);
   writeEndTagTo(result);
   return result.toString();
}
```

Un parámetro colectivo es un objeto que pasamos por parámetro a diversos métodos para que dichos métodos realicen una actualización del valor de nuestro objeto. Una buena razón para usar nuestro modelo es cuando queremos descomponer un método en métodos más pequeños y necesitamos obtener información de cada uno de dichos métodos. Debemos hacer que cada uno de los métodos devuelva un valor que más tarde usaremos para componer el resultado final con los resultados obtenidos por el resto de submétodos de nuestro procedimiento.

6.1.9 Métodos Compuestos

No es fácil de entender normalmente la lógica de los procedimientos que creamos, debemos transformar la lógica de nuestros procedimientos en un numero pequeño de pasos pero mismo nivel de detalle.

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.getX()).intValue();
   int locY = new Double(p.getY()).intValue();
   boolean completelyWithin =
               (locX >= coords[0] &&
               locY >= coords[1] &&
               (locX+CardComponent.WIDTH) <= coords[2]) &&
               (locY+CardComponent.HEIGHT) <= coords[3];
   if (completelyWithin) return true;

   locX = locX+CardComponent.WIDTH;
   locY = locY+CardComponent.HEIGHT;
   boolean partiallyWithin =
               (locX > coords[0] &&
               locY > coords[1] &&
               (locX < coords[2]) &&
               (locY < coords[3]));

   return partiallyWithin;
}
```

```
public boolean contains(Component c) {
   return completelyWithin(c) || partiallyWithin(c) ;
}

private boolean completelyWithin(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.x).intValue();
   int locY = new Double(p.y).intValue();
   return (locX >= coords[0] &&
            locY >= coords[1] &&
           (locX + CardComponent.WIDTH) <= coords[2]) &&
           (locY + CardComponent.HEIGHT) <= coords[3];
}

private boolean partiallyWithin(Component c) {
   Point p = c.getLocation();
   int locX = new Double(p.x).intValue() + CardComponent.WIDTH;
   int locY = new Double(p.y).intValue() + CardComponent.HEIGHT;
   return (locX > coords[0] &&
            locY > coords[1] &&
           (locX < coords[2]) &&
           (locY < coords[3]));
}
```

Kent Beck dijo una vez que algunos de sus mejores modelos son aquellos de los cuales el pensó que alguien podría reírse de el por escribirlos. Un método Compuesto es pequeño, simple y muy fácil de entender. ¿ Escribe usted muchos métodos compuestos?. Cuando mis programas tienen muchos métodos compuestos tienden a ser fácil de usar, de leer y de entender.

6.1.10 Reemplace una o muchas distinciones con compuestos

Si tenemos código separado para manipular simples elementos y colecciones de esos elementos , debemos combinar el código para manipular dichos elementos.

```
public class Product...
  protected Vector singleParts = new Vector();
  protected Vector collectedParts = new Vector();

  public void add(Part part) {
    singleParts.addElement(part);
  }
  public void add(PartSet set) {
    collectedParts.addElement(set);
  }
  public float getPrice() {
    float price = 0.0f;
    Enumeration e;
    for (e=singleParts.elements(); e.hasMoreElements();) {
      Part p = (Part)e.nextElement();
      price += p.getPrice();
    }
    for (e=collectedParts.elements(); e.hasMoreElements();) {
      PartSet set = (PartSet)e.nextElement();
      price += set.getPrice();
    }
    return price;
  }
```

```
public class Product...
  protected Vector parts = new Vector();

  public void add(Part p) {
    parts.addElement(p);
  }
  public float getPrice() {
    float price = 0.0f;
    for (Enumeration e=parts.elements(); e.hasMoreElements();) {
      Part p = (Part)e.nextElement();
      price += p.getPrice();
    }
    return price;
  }
```

Una buena razón para refactorizar con compuestos es con intención de librarse de código que distingue entre objetos simples y colecciones.

6.1.11 AJUSTE DE INTERFACES

Mi clase implementa algún interfaz pero sólo usamos algunos de los métodos que implementa. Entonces movamos los métodos a un "Adapter" y hagamos que este sea accesible desde un método de Factoria.

```
public class CardComponent extends Container implements MouseMotionListener ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(this);
  }
  public void mouseDragged(MouseEvent e) {
    e.consume();
    dragPos.x = e.getX();
    dragPos.y = e.getY();
    setLocation(getLocation().x+e.getX()-currPos.x,
                getLocation().y+e.getY()-currPos.y);
    repaint();
  }
  public void mouseMoved(MouseEvent e) {
  }
```

```
public class CardComponent extends Container ...
  public CardComponent(Card card,Explanations explanations) {
     ...
     addMouseMotionListener(createMouseMotionAdapter());
  }
  private MouseMotionAdapter createMouseMotionAdapter() {
    return new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        e.consume();
        dragPos.x = e.getX();
        dragPos.y = e.getY();
        setLocation(getLocation().x+e.getX()-currPos.x,
                    getLocation().y+e.getY()-currPos.y);
        repaint();
      }
    };
  }
```

Estos métodos vacíos son un fastidio, y simplemente están ahí para satisfacer un "contrato" por el hecho de implementar un interfaz, pero sólo necesitamos algunos de estos métodos.

Este modelo proporciona una forma agradable de refactorizar este tipo de código.

7.1 TEST BEFORE PROGRAMMING

Para verificar el correcto funcionamiento del código programado es necesario probarlo de algún modo. El conjunto de pruebas cumple esta misión. El Test before Programming en XP se basa en elaborar y ejecutar un completo conjunto de pruebas para un módulo del programa incluso antes de que el módulo este acabado con la finalidad de trabajar siempre en un entorno seguro, libre de errores.

¿Qué es el Test before programming? Beneficios del Test before programmnig ¿Cuándo es el mejor momento para encontrar un error en el código? ¿Cómo ejecutar nuestro conjunto de pruebas? Cualidades de las pruebas Situaciones entre código y conjunto de pruebas. Conclusión

Como ya se vió anteriormente un programador que hace uso de la programación extrema nunca programará nada que no este en las especificaciones facilitadas por el cliente de un modo u otro.

Nunca programará nada de código pensando en que "así quedara más bonito" o "yo creo que funcionara mejor así". Esto es debido a que la XP promulga que el programa deberá estar finalizado cuanto antes cumpliendo solo lo requerido por las especificaciones.

Para asegurar el correcto funcionamiento del programa cuanto antes y trabajar en un entorno fiable y libre de fallos la XP hace uso de TEST BEFORE PROGRAMMING (diseñar pruebas antes de programar).

7.1.1 ¿QUÉ ES EL TEST BEFORE PROGRAMMING?

En realidad de lo que se trata es de realizar un exhaustivo conjunto de pruebas para un modulo del programa antes incluso de que el modulo este terminado. Solo realizando estas pruebas sobre el código y obteniendo resultados esperados se tendrá la certeza de que el código es correcto.
Entonces entendemos como test el conjunto de pruebas a realizar en el código junto con los resultados esperados asociados a cada una de las pruebas anteriores.

La XP cuenta con Pair Programming (como se vio en los fundamentos). La programación se realiza por parejas donde un programador pica el código y otro lo va revisando en tiempo real ideando mejoras y buscando errores. Esta forma de programación en parejas ayuda sobremanera a la correctitud del código en tiempo de programación por lo que también favorece al periodo obligado de testeo.

7.1.2 BENEFICIOS DEL TEST BEFORE PROGRAMMING

Es raro encontrar a un programador al que le guste realizar pruebas para una aplicación. Es mas, es raro encontrar un programador que no deteste realizar dichas tareas. Sobre todo cuando el código que esta probando lo ha realizado el mismo. Esto es debido quizás a que a nadie l e gusta encontrar fallos en su trabajo, pues las pruebas lo que revelan son fallos en la programación o en la interpretación errónea de los requerimientos funcionales del programa.

Sin embargo un buen programador debe hacerse cargo de la importancia que tienen las pruebas sobre el normal desarrollo de un proyecto. Todo el mundo se equivoca y es imposible realizar un código de cierta envergadura a la primera sin cometer errores y por esto un buen programador sabe de la importancia de realizar continuas pruebas de su código y continuas depuraciones y mejoras.

Para la XP las pruebas (testeos) del programa son mas importantes si cabe que para otras metodologías. La programación modular, es decir, crear un programa a partir de módulos (paquetes, objetos...) lo mas independiente posible entre ellos, es otro aspecto importante en la XP. Gracias al testeo continuo la labor de modularizar y empaquetar el proyecto en subprogramas totalmente funcionales se ve fuertemente favorecida.

Esta demostrado que la mayoría de los errores cometidos en un bloque o modulo de nuestro programa traen consigo otros errores localizados en distintos módulos que dependan del bloque erróneo.

7.1.3 EL MEJOR MOMENTO PARA ENCONTRAR UN ERROR EN EL CÓDIGO

Cualquier momento es bueno si sirve para solucionar un error pero como es lógico pensar cuanto antes nos demos cuenta de un error cometido mejor nos irá en el proyecto.

Esto es debido a que cuanto más fresco tengamos el código mas fácil nos será interpretar el error, descubrir sus causas, sus dependencias, y sus consecuencias en el resto del código.

Por esta razón el momento ideal para probar un código es justo después de haber terminado de programarlo. Aparte de las pruebas realizadas en función de los requerimientos funcionales del proyecto cuando acabemos de programar el código posiblemente se nos ocurran distintas pruebas en función del código realizado y que ayudaran a que el conjunto de pruebas sea completo.

- **Entre los errores mas comunes se pueden distinguir dos tipos:**

Los errores en tiempo de compilación : Estos errores rompen el ritmo del programador y le pueden hacer perder un tiempo precioso pues suelen ser habitualmente producidos por un fallo en la escritura del código incumpliendo la sintaxis, la semántica o la faceta léxica de un lenguaje. Estos errores no son complicados de encontrar pues dan error al compilar el programa y se pueden evitar en su mayoría con La Programación en Parejas (ya vista anteriormente en Los fundamentos de la XP).

Los errores en tiempo de ejecución: Estos errores son los mas peligrosos para el correcto desarrollo del proyecto. Producen resultados inesperados en el programa y pueden ser muy complicados de encontrar.
Los errores en tiempo de ejecución pueden, a su vez, dividirse en errores conceptuales, que son los errores cometidos en los que se puede llamar el concepto o contenido que debe cumplir un programa, y en errores funcionales derivados del lenguaje, y que son derivados de un mal uso, en tiempo de ejecución, de las funciones que nos facilita el lenguaje de programación.

Veamos un sencillo ejemplo:

Queremos realizar un código que nos devuelva el área de una circunferencia utilizando la formula Area = pi * Radio * radio . Si por algún motivo el programador se equivoca y en vez de definir el área de un circulo como se ha visto lo hace de esta forma : Area = pi * radio + radio. El programa compilara y se podrá ejecutar sin problemas pero sus resultados serán incorrectos en todos los casos excepto cuando el radio valga 2. Así el programador deberá repasarse todo el código línea por línea trazándolo mentalmente o de alguna otra forma y realizando un verificación exhaustiva para encontrar el posible error. Cuando el código es tan simple como el del ejemplo no hay problema pero cuando el código sea el de una aplicación seria y medianamente profesional puede convertirse en un castigo.
Imaginemos que además nuestro pequeño programa que nos dice el área de una circunferencia forma parte de un código mas complejo dividido en otros módulos independientes y dependientes del nuestro.
Cuando ejecutemos la aplicación y se produzca el error ¿dónde empezaremos a buscar?. ¿Como saber en que modulo se ha producido el error?.

Los testeos continuos y exhaustivos de los que hace uso la XP bajo el nombre de Test before Programming llevan consigo un conjunto bastante significativo de beneficios y ventajas respecto a otras metodologías o técnicas de programación que no deben pasarse por alto.

Realizar pruebas continuas sobre nuestro código nos reporta para empezar una mayor **fiabilidad y estabilidad** en el trabajo de programación realizado hasta el momento. Esto significa que al realizar las pruebas correctamente y con resultados esperados sobre nuestro código tendremos como resultado un modulo entero programado de forma correcta, que funciona ahora y que funcionara siempre independientemente de donde lo incluyamos. Esto nos proporciona confianza y elimina posibles candidatos a contener error cuando conjuntemos todos los módulos. No hace falta decir que el correcto

funcionamiento de todas las partes de un todo conllevan que el todo funcione correctamente siempre y cuando todas las partes estén bien ensambladas.

Además de la fiabilidad el testeo continuo de nuestros modulo adelanta trabajo y lo facilita, acelerando el proceso necesario para la creación del proyecto.

Al crear módulos individualmente validos nos ahorramos el pesado trabajo de buscar errores si se hubieran producido cuando anexionemos todos los módulos. Siempre es mejor realizar módulos pequeños y realizar pruebas constantes sobre ellos para que sean fiables que realizar módulos grandes sin probarlos y luego pretender que el conjunto de todos los bloques del programa funcionen correctamente a la primera.

7.1.4 EJECUTAR NUESTRO CONJUNTO DE PRUEBAS

El funcionamiento de un programa realizado con XP (y, en general, con otras metodologías) nunca puede depender del entorno o lenguaje de programación usados para su creación. Tampoco debería depender en gran medida de la maquina donde pretenda correr. Nuestro programa deberá depender única y exclusivamente de los requerimientos pretendidos por el cliente.Por esta razón es posible realizar el conjunto de pruebas antes incluso de programar el código. Además de depurar nuestro código las pruebas nos ayudaran a comprender el buen funcionamiento del programa si han sido diseñadas antes de picar el código.

Ahora bien, no siempre las pruebas pueden ser un apoyo o ayuda para la realización del código. Un conjunto de pruebas es tan importante como el código que pretenden testear. Las pruebas deben estar bien diseñadas y bien redactadas pues es peor lanzar pruebas erróneas sobre nuestro programa que el no lanzar ninguna. Recordemos ahora el problema del área de la circunferencia que vimos en el apartado anterior. En el ejemplo vimos como habíamos cometido un error en la formula para hallar al área de la figura.
También vimos la necesidad de realizar pruebas para encontrar el error de ejecución que se producía. Las pruebas que hubiéramos realizado sobre el programa nos habrían conducido a encontrar nuestro error. Pero ¿qué habría sucedido si nuestras pruebas hubieran también sido erróneas?. El error del programa podría haber pasado camuflado entre las pruebas. Hubiera sido arrastrado pensando, el programador, que su código era correcto, hasta el momento en que se fusionasen todos los módulos que componen la aplicación final. En este momento hubieran salido a la luz, o lo que es peor, en este punto las pruebas también podrían estar erróneas. Entonces podemos ver la importancia de una pruebas bien hechas en el resultado final de la aplicación y del proyecto.

7.1.5 CUALIDADES DE LAS PRUEBAS

Además de que las pruebas estén bien construidas en el sentido del código a probar, esto es que las pruebas sean **correctas**, deben cumplir otras condiciones. El conjunto de pruebas también debe ser **completo**, es decir, cumplir con su cometido y probar todos los casos conflictivos posibles que se puedan dar.

7.1.6 SITUACIONES ENTRE CÓDIGO Y CONJUNTO DE PRUEBAS

Podemos distinguir varias situaciones basándonos en lo visto anteriormente:

- **Código correcto # Pruebas correctas**.

Este es el caso ideal con el que todo desarrollador quiere encontrarse.
El código del programa es correcto por lo que pasara sin problemas las pruebas realizadas que también estarán bien diseñadas.

- **Código correcto # Conjunto de Pruebas incompleto**.

Este caso como el anterior no dará ningún problema pero no hay que llevarse a engaño. Un buen diseño de pruebas debe ser completo y probar todos los casos posibles. En esta situación nos hemos quedado casos por probar aunque afortunadamente nuestro código era correcto y no se crearan mas problemas. No es una situación fiable pues en cuanto el código tenga algún pequeño error (algo bastante común) lo podemos obviar en nuestro conjunto de pruebas.

- **Código correcto # Conjunto de Pruebas incorrecto**.

Cuidado con este caso. Después de haber realizado el código cuidadosamente para evitar cometer errores. Después de haber programado en parejas (pair programming) y haber obtenido un código correcto realizamos un conjunto de pruebas incorrecto con consecuencias imprevisibles.
Podemos tener suerte y que las pruebas no abarquen todo el rango de casos (estarían mal diseñadas ya que no serian completas) y que no se produzcan errores (un caso muy raro).
Podemos no tener tanta suerte y descubrir (equivocadamente) que nuestro código es erróneo cuando en realidad lo erróneo son las pruebas. Esto nos llevaría a revisar el código en busca de posibles candidatos a errores y a encontrarlos donde en realidad no existen. (en teoría un buen programador conoce a la perfección los requerimientos del software que desarrolla y no se equivocaría cuando buscase errores donde no los hay).
Cambiar un código correcto (pensando que es incorrecto) a un código incorrecto (con el convencimiento de que ahora es correcto) nos conducirá a una situación de caos donde el código, a la hora de realizar las pruebas finales frente al cliente (este nunca se equivoca), fallara de forma escandalosa. Nadie

sabrá cómo ni porque pues se corrigió frente a las pruebas y las paso en una segunda pasada.
Pero si por fortuna el error verdadero surge en revisiones anteriores tendremos la oportunidad de arreglar el entuerto. Este arreglo posiblemente nos robe tiempo de otros asuntos y nos retrase los plazos.

- **Código incorrecto # Conjunto de Pruebas correcto**.

Este es, posiblemente, el caso mas común que se puede dar. Aunque el código contenga errores no deben suponer mucho problema ya que las pruebas son correctas y completas. El tiempo necesario para corregir los errores dependerán de la clase de errores. Como norma general el tiempo no será grande y los errores serán solucionados rápidamente. Cuando hayamos revisado el código solucionando los errores deberemos exponerlo de nuevo a otra pasada del conjunto de pruebas y así sucesivamente hasta haber solucionado los errores que pudieran haber aparecido al arreglar los antiguos.

- **Código incorrecto # Conjunto de Pruebas incorrecto**.

Puede ser peligroso encontrarse con este caso. Las pruebas incorrectas posiblemente avisasen de errores en el código pero estos errores no tendrían que ver necesariamente con los verdaderos por lo que esta situación podría desembocar en no arreglar los errores realmente existente y dañar el código en zonas donde estaba bien programado.

Como ya vimos en el caso del código correcto y conjunto de pruebas incorrecto estas situaciones nos retrasaran sobremanera en el mejor de los casos y de crear un software completo cuyos módulos funcionan incorrectamente en el peor de los casos.

7.1.7 CÓDIGO DIFÍCIL DE PROBAR

Algunas áreas de nuestro código no serán fáciles de verificar mediante pruebas y deberemos poner el máximo empeño en programarlas libres de errores.
Estas áreas son:

- Acceso a Bases de datos
- Sistemas basados en la WEB. (sistemas que se sirvan de internet para realizar alguna función).
- Procesos que utilicen Threads (hilos, procesos paralelizados).

7.1.8 CONCLUSIÓN

Testear un programa es realizar sobre él un conjunto de pruebas diseñadas con intención y obtener los resultados esperados para cada una de ellas.Si el conjunto de pruebas es realizado en su mayoría o hasta donde sea posible antes de codificar el programa a probar nos ayudaran a codificar el programa mas rápidamente teniendo cuidado en los puntos que advierten las pruebas.Realizar pruebas continuas sobre nuestro código ayuda a que sea modularizado y libre de errores, además nos reporta una mayor fiabilidad y estabilidad en el trabajo de programación realizado hasta el momento.La fase de pruebas tiene la misma importancia que la fase de codificación y de ella depende el buen funcionamiento del proyecto.

8.1 COMO VENDER LA XP A UN CLIENTE:

La metodología XP nos propone una serie de reducciones y técnicas que persiguen un único fin: *la satisfacción del cliente****.***

Introducción Problemas Típicos

INTRODUCCIÓN

Cualquier desarrollador, analista, técnico y, en general, cualquier persona que haya participado en el diseño o desarrollo de un proyecto informático, seguramente haya 'sufrido' el yugo que supone estar supeditado al cumplimiento de una determinada metodología.

Las metodologías que se aplican suelen ser rígidas; dejan poco margen de maniobra y, lo que es peor, intentan ser demasiado generalistas.

Además son, en la mayor parte de los casos, metodologías obsoletas, poco ágiles y que suponen una tremenda traba para el tipo de proyectos que actualmente se desarrollan: proyectos pequeños, ágiles, cambiantes y orientados hacia las nuevas (y también cambiantes) tecnologías web.

Aplicar estas metodologías supone tener clientes descontentos, jefes de proyectos confundidos, analista más confundidos y programadores desconcertados. Son difíciles de asimilar y de aplicar y, por tanto, pierden su propio sentido rápidamente.

La XP suple, al menos en gran parte, las deficiencias de las tradicionales, ya que aplica, a grandes rasgos, técnicas de usabilidad a la ingeniería del software.

PROBLEMAS TÍPICOS

A la hora de realizar un proyecto de software hay que decidir muchas cosas. Hay que tener en cuenta entre otros el presupuesto, la cantidad de recursos con los que se cuenta, los requerimientos, los plazos, el entorno de trabajo en el que se usará el software, el lenguaje de programación, los roles necesarios, el cliente, y, por supuesto, la metodología que se usará. Estos dos últimos (el cliente y la metodología) están fuertemente ligados. El cliente es el que proporciona el presupuesto (el que paga) y todo deberá estar a su gusto incluido la metodología que se use en el desarrollo de su software. Una metodología es una forma de trabajar y abordar el proyecto por lo que si el cliente está conforme con ella las cosas serán siempre mucho más fáciles.

Pongámonos en el caso más difícil: Un cliente que no sabe absolutamente nada de XP, no conoce sus fundamentos ni sus resultados por lo que le resultará difícil comprender como funciona si no se le vende bien.

Además como caso general cuando un cliente contrata los servicios de un grupo de desarrolladores para elaborar un software suele pasar que piensa que cuando entregue sus requerimientos su trabajo habrá terminado y que como es el que paga tiene derecho a obtener un programa terminado y que funcione a su gusto sin más implicación que la que ha hecho (requerimientos iniciales (y dinero)). En algunas metodologías esto puede funcionar pero como caso general este planteamiento falla. En especial cuando hablamos de XP, en la cual el cliente debe funcionar como un miembro más del grupo de desarrolladores exponiendo en todo momento sus ideas y cambios de parecer en cuanto a requerimientos y tratando de explicar cualquier duda que surja por parte de los programadores y diseñadores. Su función es por lo tanto importante en un proyecto ya que es la guía que lleva al desarrollo por el buen camino y que sirve de salvavidas en muchos momentos.

Si el cliente se siente parte del proyecto será mas fácil que le guste tanto el resultado final como el esfuerzo que mostraron los miembros del equipo en su desarrollo.

Pero vender la XP a un "ateo" puede resultar muy complicado. Fundamentos de la XP como "pair programming", 40 horas semanales, etc son difíciles de explicar a alguien que no sepa en realidad como funciona la XP.

Por ejemplo:

Ya vimos lo que significaba Pair Programming (dos programadores trabajando juntos).

EL cliente, que no sabe nada de XP ni de las ventajas del pair programming, ve a dos Programadores a los que esta pagando mucho dinero (para el cliente lo poco o mucho que pague siempre le resultará caro) y lo piensa es que uno de ellos está perdiendo el tiempo cuando podría estar programando y adelantando trabajo. Seguro que esto ni lo entiende ni le gusta.
Si se entera de que se trabaja 40 horas semanales y el proyecto aun no está terminado aunque vaya según los plazos estipulados tampoco le gustará pues pensará que se está perdiendo el tiempo y que se podría adelantar trabajo (eso sí las horas extras no la paga él).

Además de esto la XP es una metología relativamente nueva. Nadie duda de los beneficios que aporta al desarrollo del proyecto (La XP proporciona resultados rápidos y por lo tanto aumenta la productividad de los desarrolladores) pero siempre surgen dudas de su autentico potencial. Ya hemos visto para que casos es mejor aplicar la XP (proyectos no demasiado grandes, grupos de trabajo reducidos...). Es en estos casos cuando elegir la XP reporta todos sus beneficios y ventajas con respecto a otras metodologías.

La XP además enfatiza el trabajo en grupo, un grupo formado por los jefes de proyecto, desarrolladores y clientes.

Se simplifican los diseños, se agiliza el desarrollo eliminando especificaciones superfluas, se incrementa la comunicación entre desarrolladores y clientes, se acortan los períodos de implantación (es mejor implantar poco y pronto que mucho y tarde).

Pero a nadie le gusta experimentar con su dinero y arriesgarlo en una empresa que puede no salir bien. Puede que el cliente haya tenido otras experiencias negativas anteriormente con otros proyectos de software y esto no ayudará a vender un software con XP.

El mayor fracaso de las metodologías tradicionales ha sido, y es, el no mantener contentos a nuestros clientes. Todo ello porque las metodologías

tradicionales no son capaces de dar respuestas rápidas a las cambiantes especificaciones de un proyecto en marcha.

Los retrasos se acumulan al mismo tiempo que se amontona el trabajo: demasiadas reglas, demasiados documentos, demasiados formalismos. El cliente se impacienta, el jefe de proyecto se desespera, los analistas y programadores no son capaces de tener a tiempo los cambios requeridos.

En este momento (sólo cuando las cosas se ponen muy malas se inicia el proceso del cambio) es cuando hay que hacerle ver las ventajas de la XP y convencerle de que su inversión está más que segura. Cuando todo marcha bien, es decir el cliente tuvo otras experiencias buenas con otras metodologías será difícil hacerle ver la necesidad del cambio porque seguro que preferirá no hacer pruebas raras cuando sabe que algo le ha funcionado.

Hay, entonces, que hacer ver al cliente que la XP con todos sus fundamentos y todas sus particularidades puede conseguir que su proyecto esté terminado antes y con mejores resultados. Y para esto el propio cliente deberá fusionarse al grupo de trabajo y aportar su esfuerzo.

La XP debe ser vendida como lo que es: Una metodología que aporta dinamismo al proceso de desarrollo de software, que lo acelera, lo facilita y a la vez lo abarata.

8.2 EJEMPLOS DE PROYECTOS REALIZADOS CON XP:

Ejemplo 1 Ejemplo 2

8.2.1 EJEMPLO 1[20]:

En es te punto veremos cómo integrar la XP con un lenguaje como C++, que no nos da tantas ventajas como otros, cómo Java.

Introducción Fundamentos de a XP usados en el proyecto Conclusión

[20] http://objectmentor.com/XP/xpwitch.html

INTRODUCCIÓN

El proyecto que se trato de sacar adelante con XP estaba destinado a el "Educational Testing Service".

La primera dificultad se la encontraron al intentar embeber las enseñanzas de la XP con el lenguaje C++.

¿ puede la XP ser usada con C++?

C++ es un lenguaje fuertemente interdependiente. Esto significa que contiene muchas dependencias internas con librerías,(# includes), con otras partes del código etc...

Debido a esta interdependencia surge la duda de cuanto podrá soportar nuestra aplicación el Refactoring sucesivo al que sometida. Como el Refactoring es un pilar importantísimo en la XP... ¿hasta que punto puede ser compatible con C++?

La forma que idearon fue la de crear nuevas reglas o principios con los que evitar el Refactoring tan exhaustivo que obliga la XP. El proyecto fue desarrollado por el grupo "Object Mentor" desde 1993 hasta 1997.

Consistía en obtener un conjunto de aplicaciones que almacenaran y evaluaran en base a unos principios básicos de arquitectura, proyectos de arquitectos. Los proyectos de los arquitectos contendrían planos y un conjunto grande de documentos destinados a describir plenamente sus intenciones y razones.

El programa fue creado en C++ para correr sobre Windows95. Necesitó de 250000 líneas de código aproximadamente y se invirtieron 4 años en su desarrollo y 5 ingenieros.

FUNDAMENTOS DE LA XP USADOS EN EL PROYECTO:

Se usaron varios fundamentos de la XP. Casos de Uso, Sistema metamórfico, Iteraciones Frecuentes, Integración continua, Programación en parejas. Cuando desarrollaron el proyecto aun no conocían como tal la XP ni sus principios por lo que decir que usaron varios fundamentos de la XP como la conocemos ahora no es totalmente cierto.

Para empezar los programadores estaban físicamente separados por lo que se hacía imposible la programación en parejas. Para suplir esta carencia utilizaron una variante de la programación en parejas a la cual llamaron "el sistema de compañero". Esta variante se desarrollaba de la siguiente forma: Un ingeniero programaba una parte del código y se la enviaba a su compañero que la revisaba intensamente. Además ejecutaría pruebas sobre el código e inventaría pruebas nuevas como si de su código se tratase.

Con esto no se consiguen todos los beneficios del Pair Progamming pero al menos se obtienen algunos.

Otro principio de la XP como es el hacer las cosos una vez y sólo una vez (reutilización máxima) tuvo vital importancia en el proyecto. Al estar físicamente separados se corría el riesgo de perder comunicación y por lo tanto de desarrollar el trabajo por líneas diferentes.

Para evitar caer en este error se dispusieron a identificar estructuras y algoritmos que podrían tener en común los subprogramas de los que se componía el programa final.

Una vez encontrados los desarrollaron y se dedicaron a exportarlos a los distintos subprogramas.

El problema de la flexibilidad también tuvo que ser resuelto de forma especial.Al principio se trató de crear un código que pudiera ser integrado en código ya hecho para tratar de ahorrar tiempo y esfuerzo. El problema del asunto esta en que C++ no presenta flexibilidad por lo que reutilizar el código resulta difícil. Esto es porque en C++ modificar el código y las dependencias es muy costoso. Para solucionar esto se tuvo que rescribir el código pero tratando de seguir ciertas reglas que permitiesen que posteriormente se pudiera reutilizar.

Durante la reescritura del código se utilizaron una serie de practicas para mantener C++ más flexible y que así pudiese soportar mejo el refactoring.
Más adelante en otros proyectos se volvieron a utilizar estas practicas debido a su éxito en este proyecto.
Sin embargo al tratar de aplicar estos conceptos nuevos a la XP pueden surgir ciertos problemas. Podemos violar ciertos principios como hacer las cosas lo más simple posible.

Por fin el proyecto fue llevado a buen termino no sin alguna otra dificultad.El grupo de desarrollo (Object Mentor) utilizó todo lo aprendido en este proyecto(flexibilidad de C++, fundamentos XP) posteriormente en otros trabajos.

CONCLUSIÓN:

Aunque, quizás este proyecto no era el más idóneo para desarrollar con XP,el tamaño resultó ser excesivo, se alargó mucho en el tiempo, el lenguaje C++ era poco flexible... se consiguió adaptar a ella y finalizarlo con éxito.

La XP puede usarse entonces conjuntamente con C++ mientras se mantenga el código lo más flexible posible. Sin esta flexibilidad hacer refactoring en el código sería imposible.

Quizás la principal cualidad de la XP es la plasticidad (hasta ciertos límites) de su fundamentos. Como es lógico para obtener todos los beneficios incluidos en estos fundamentos hay que tratar de usarlos tal cual son. Sin embargo hay ocasiones donde se hace imposible utilizar todos y cada de ellos. En estos casos hace falta echarle imaginación y transformar en la medida de lo posible los fundamentos hasta obtener una XP a nuestra medida.

8.2.2 EJEMPLO 2[21]:

Repo Margining System: Aplicando XP en la Industria Financiera

El entorno Datos del proyecto XP y sus fundamentos Lecciones Aprendidas Resultados Obtenidos Conclusión

EL ENTORNO:

En la industria financiera la habilidad de desarrollar rápidamente software lo suficientemente flexible como para adaptarse a los continuos cambios de la industria el vital. La existencia de un entorno de trabajo en el cual es común el estrés y la presión complica la tarea ya de por si difícil de llegar a obtener software de calidad y se prefiere habitualmente el software rápido aunque contenga errores. Lo que se trato al desarrollar este software fue el conseguir una aplicación buena, de desarrollo rápido y útil. Estos eran claramente preceptos de la XP. El grupo de desarrollo se llamaba Klondike.

DATOS DEL PROYECTO:

Introducción:

El proyecto Repo Margining System fue escogido como tal durante el Sun Education *"SUNBIM" programa* debido a su no muy grade importancia, su relativa completitud y el poco dominio del equipo en el campo del desarrollo software para la industria financiera.

Descripción del proyecto:

El sistema debería permitir el comercio a través de un "navegador" y mantener informadas a ambas partes del trato. Además capturará las ofertas que se hagan y las mostrará en un margen especialmente dedicado a ello.

Estructura organizativa del proyecto:

El equipo estaba formado por 7 componentes. El espacio de trabajo estaba habilitado para favorecer la comunicación entre los componentes. Se empezó a desarrollar en JAVA utilizando herramientas para crear las clases, para realizar las pruebas y para hacer refactoring. El trabajo no se hizo desde cero ya que el equipo contaba con código de otros trabajos anteriores y que pudo reutilizar.

[21] http://www.xp2001.org/xp2001/conference/program.html

XP y sus fundamentos:

Las siguientes prácticas fueron usadas por completo: The Planning Game, versiones pequeñas, sistema metamórfico, diseños simples, creación del conjunto de pruebas antes de la codificación, Refactoring, Programación en parejas, Propiedad colectiva del código, estándares de codificación. Aunque el cliente no estuvo presente siempre se mantuvo la comunicación con el en todo momento. Las 40 horas semanales fueron ampliadas a 50 horas semanales por decisión conjunta de todo el grupo de las cuales 14 se dedicaron a estudiar e informarse del entorno de desarrollo.

"Lecciones aprendidas":

Refactoring – Hay que prestarle atención (puede llegar a ser el 50% del esfuerzo prestado al proyecto) está pensado para incrementar la simplicidad y comunicación en el desarrollo del proyecto. Permite que el equipo incluya nuevas funcionalidades o realice cambios a las ya creadas, según el gusto del cliente, manteniendo el mismo nivel de calidad y en menos tiempo.

Pair Programming – Esta prática permitió resolver errores más rápidamente y que todo el equipo tuviese buena cuenta de como funcionaba el programa en todo momento.

El plan de trabajo – El equipo notó que el cliente se centra en el reparto del valor. El cliente está satisfecho incluso cuando el equipo ofrece partes que han sido dejadas?.

Tracking and planning – Partiendo de 30 minutos de trabajo se hace la planificación teniendo en cuenta la vuelta atrás y permite estimar de forma exacta las necesidades actuales.

Metrics – Es necesario tener en cuenta varios factores y variables para decidir si es necesario refactorizar.

Testing – Los test propòrcionan una vía de tranquilidad para realizar cambios en el código sin crear problemas mayores.

Resultados obtenidos:

Más o menos el trabajo estuvo concluido en 4 iteraciones, lo que llevó 7 semanas. El sistema ya tomó forma a partir de la tercera iteración. El propósito de desarrollar bien, rápido y barato fue progresivamente considerado como posible. Esto llevó al equipo a estar convencidos de que construían software de calidad y de que disfrutaban en la tarea. El equipo también llegó a controlar muy bien los tiempos de diseño, programación pruebas y a tener una estimación aproximada de cuanto llevaban y de cuanto les faltaba.
Los errores que se cometieron fueron mínimos y resueltos con facilidad.

En realidad no fueron todo lo rápidos y baratos que se podía esperar pero el propio equipo reconoció que con un poco más de experiencia en el uso de la XP obtendrían una velocidad superior a unos costes más bajos.

9.1 BENEFICIOS Y DESVENTAJAS

Nos vamos a ocupar en este apartado de explicar brevemente algunos de los beneficios que implica el uso de XP y algunos de las desventajas que provoca.

Beneficios y Desventajas

BENEFICIOS Y DESVENTAJAS

La propuesta de XP reconoce que el desarrollo de software es lo suficientemente difícil y diferente de otros tipos de proyectos corporativos como para ameritar su propia metodología. XP promete ahorrar tiempo y dinero, mientras ayuda a los desarrolladores a crear aplicaciones de mejor calidad.

XP puede ser aplicada a cualquier lenguaje de programación, aunque es mejor aplicado al comienzo del ciclo de desarrollo. Los desarrolladores son impulsados a trabajar en ciclos de release más rápidos y múltiples reiteraciones.

Mediante el enfásis en la frecuencia, los pequeños releases, la propuesta de XP le da la flexibilidad para ajustar sus aplicaciones a través de todo el ciclo de vida del desarrollo.

Pros:

- Mejora la comunicación.
- Introduce eficiencias en planificación y pruebas.

Contras:

- Puede no siempre ser más fácil que el desarrollo tradicional.
- Los usuarios pueden no querer frecuentes pequeños releases.
- Requiere rígido ajuste a los principios XP.

9.2 COMPARACIÓN CON ASD

Existen otras metodologías que están floreciendo en un mercado orientado hacia internet y la denominada "economía movil", basada en dispositivos más diversos, portátiles y con mayor facilidad de uso.

ASD XP Comparación

ASD

ASD es una nueva metodología de software que se dirige hacia "La economía de Internet" esta economía es de una gran velocidad y de grandes cambios. La gran velocidad y los grandes cambios necesitan de una nueva metodología que no puede ser manejada por metodologías convencionales, este mercado imprevisible y los métodos de desarrollo no pueden ser llevados a cabo por las tradicionales tendencias de control de procesos.

El primer objetivo de cualquier organización de desarrollo es poder responder con rapidez a los cambios que se puedan producir. La adaptabilidad no puede ordenarse, debe fomentarse. Este fomento es realizado a través de un modelo de dirección que Highsmith llamó Adaptive Leadership-Collaboration. Esto lleva a un ambiente en el que la colaboración y la adaptación crecen para que un orden local pueda ser establecido. Fomentando conductas adaptables en cada nivel, el nivel global llega a ser adaptable.

Highsmith recomienda dos estrategias para crear un ambiente adaptable y colaborador. La *primera estrategia* pide a los directores que den un menor enfoque al proceso y mayor al producto final ya que al fin y al cabo es el resultado de la colaboración. Aunque se debe aplicar rigor al resultado de los procesos. La *segunda estrategia* solicita a los directores proveer de las herramientas y técnicas necesarias para desarrollar una auto-organización por el equipo virtual. El equipo virtual es un equipo que esta distribuido por el mundo. Esta segunda estrategia solo se necesita si la ASD es aplicada al desarrollo de software de gran potencia.

El enfoque hacia los productos solicita un acercamiento reiterativo, porque el resultado de cada iteración llega a ser la entrada principal para dirigir el proceso. Cada iteración consiste principalmente en tres fases: teorías, colaboración y aprendizaje. Las teorías en el producto se refieren a la discusión y consecuente definición de que es lo que se quiere conseguir en cada iteración. Se sigue a una fase en la que los miembros del equipo colaboran hacia una fase que incorpora rasgos como sugirió la fase previa. En la fase final el resultado es repasado y la próxima iteración esta siendo preparada.

Cada ciclo tiene las propiedades siguientes:

- Ver que misión se maneja basada en la visión global del proyecto.
- Es más un componente que una tarea base.
- Tiene un límite de tiempo.
- Posee riesgo pero debe estar controlado.
- Es tolerante a los cambios.

El cambio se ve como una oportunidad de aprender y obtener una ventaja en lugar de verlo como un detrimento al proceso y a sus resultados.

XP

Remitiendonos a la descripción de Beck podemos separar la Xp en varias partes:

- Valores: un valor en referencia a la XP es una descripción de cómo el desarrollo de software debe enfocarse. XP esta basado en los siguientes cuatro valores: Communication, Simplicity, Feedback, and Courage.

- Principios: Un principio en XP es algo que nosotros usamos para determinar si un problema debe ser planteado usando esta metodología. Beck presenta los siguientes cinco principios derivados de los valores: Rapid Feedback, Assume Simplicity, Incremental Change, Embrace Change, and Quality Work.

- Actividades : Beck ve las siguiente cuatro actividades como los pilares básicos de desarrollo de software: Codificación, Test, Escucha, Diseño.

- Práctica: Una práctica es una técnica que los miembros del proyecto usan para llevar a cabo cualquiera de los objetivos encomendados.

- Estrategias: finalmente para ejecutar las prácticas en el mundo real, Beck presenta varias estrategias y heurísticas para lograr esto.

COMPARACIÓN

Desde sus inicios ASD y XP parecen ser muy similares. Ambas tendencias de desarrollo de software se orientan hacia proyectos de cierta incertidumbre y donde se respira un ambiente de cambio continuo.

- Motivación: ASD es motivado por lo inapropiado de las tradicionales metodologías en el ámbito de la creciente economía. Para la XP es más orientado por las necesidades que surgen, mediante la experiencia de los desarrollos convencionales de software.

- Técnicas: ASD cree que las técnicas son importantes, pero no son ningún pilar fundamental. Mientras que XP confía fuertemente en una combinación especifica de técnicas a aplicar.

- Orientación: ASD está dirigido tanto a ámbitos pequeños como grandes, por el contrario XP esta más orientado proyectos de unas dimensiones no demasiados grandes. Aún así ASD y XP parecen tener un centro común de temas y creencias, que identificaremos como un sistema de valores. Si cogiéramos cualquiera de las dos metodologías para aprender de la otra, deberíamos proporcionar una base para comparar sistemas de valores para determinar la compatibilidad entre ambas. De momento se cree que no existe tal base.

Un Modelo de Sistema de Valores: para poder efectuar una comparación entre ambas metodologías vamos a definir las características de un pequeño modelo de Sistema de Valores que posteriormente usaremos para establecer una comparación más certera.

La esencia del modelo se puede resumir en:

- Papel de las personas en el desarrollo de software: ¿ Qué es un "papel"(misión) y cual es la contribución de las personas al proyecto?. Esto incluye el papel de clientes, desarrolladores, y directores.

- Relaciones humanas en el desarrollo software: Trata de cómo de importante es la comunicación, colaboración y competencia.

- Relaciones entre personas y tecnología: ¿Domina la tecnología a los desarrolladores o viceversa?

- Objetivos del desarrollo software: ¿ Porqué llevamos a cabo un desarrollo de software específico?. Para hacerlo lo más rápido

posible y cobrar, realizando y software que cubra las necesidades del cliente.

En base al modelo anterior vamos a establecer una tabla comparatoria de XP y ASD.

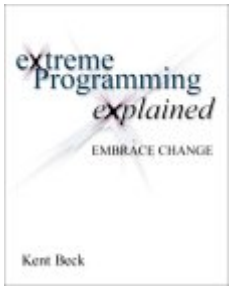
	ASD	XP
PAPEL DE LAS PERSONAS	Indiscutible	Indiscutible
* Desarrolladores	dirigen el producto,	dirigen proceso y producto y usan
	adaptandolo a los requerimientos	técnicas predefinidas para hacerlo
* Directores	dirigen proceso y producto	dirigen proceso y producto y usan
	adaptandolo a los requerimientos	técnicas predefinidas para hacerlo
* Clientes	proporcionan entradas al producto	proporcionan entradas al producto
PAPEL DE LAS RELACIONES	principal foco de innovación	principal foco de innovación
HUMANAS	habilidad para adaptar	satisfacción del trabajo
* Comunicación	es clave para emergencias	es fundamental para el trabajo
* Cooperación	es clave para emergencias	es fundamental para el trabajo
* Competición	es clave en emergencias	no hay
PERSONAS Y TECNOLOGIA	personas controlan tecnologia	personas controlan tecnologia
	es una herramienta no un problema	es una herramienta no un problema
* Aplicación de tecnologia	las personas la usan	las personas la usan sin un
		predeterminado entorno de trabajo
* Tipos de tecnología	es mejor si es más ligera	es mejor si es más ligera
OBJETIVOS	supervivencia y creciente organiz.	entrega del producto-trabajo hecho
* Entrega	es fundamental	es fundamental
* Apoyo	posible	es fundamental

R.1 BIBLIOGRAFÍA

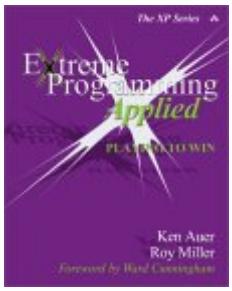
La forma más antigua de almacenar la información a través del tiempo sigue siendo una gran fuente de la cuál extraer todo aquello que necesitemos conocer.

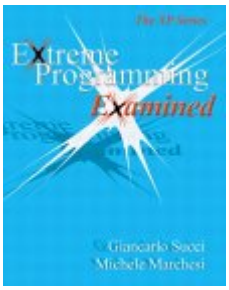
Vamos a hacer un repaso a los libros que consideramos más adecuados para el estudio de la programación extrema y las áreas de estudio que rodean a ésta.

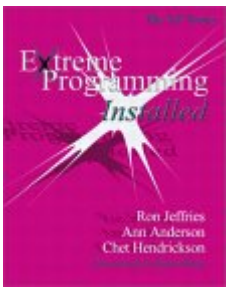
Extreme Programming

	**Extreme Programming Explained: Embrace Change.** **Autor**: Kent Beck **Qué incluye**: metodología XP, principios, roles del equipo, facilidad de diseño, testeo, refactoring, el ciclo de vida del software en XP y adoptando XP. **Nivel**: Medio.

	**A Practical Guide to eXtreme Programming.** **Autor**: David Astels, Granville Miller, Miroslav Novak **Qué incluye**: iniciación a la programación extrema desde un enfoque muy básico. **Nivel**: Básico

	**Extreme Programming Applied: Playing to Win.** **Autor**: Ken Auer, Roy Miller. **Qué incluye**: explica los fundamentos de la XP. **Nivel**: Básico.

	**Extreme Programming Examined.** **Autor**: Giancarlo Succi, Michele Marchesi. **Qué incluye**: 33 documentos sobre XP sacados de la conferencia de Junio del 2000 sobre programación extrema, abordando diversos puntos de vista. **Nivel**: Avanzado.

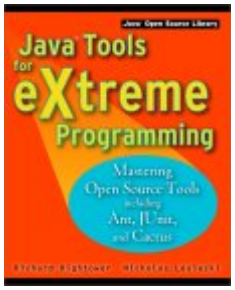
	**Extreme Programming Installed.** **Autor**: Ron Jeffries, Ann Anderson, Chet Hendrickson, Kent Beck, Ronald E. Jeffries. **Qué incluye**: Aborda todo lo necesario para utilizar esta metodología, con terminología no muy técnica. **Nivel**: Medio.

Otros

- Beck, Kent. *Smalltalk Best Practice Patterns.* Upper Saddle River, N.J.: Prentice Hall, 1997.
- Fowler, Martin. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable*
- *Object Oriented Software.* Reading, Mass.: Addison-Wesley, 1995.
- Kent Beck and Erich Gamma. JUnit Testing Framework. Available on the Web (http://www.junit.org).
- Brooks, Frederick P., Jr., *The Mythical Man-Month*, Anniversary Ed., Addison Wesley, ISBN 0-201-83595-9, 1995.
- McConnell, Steven C., *Code Complete*, Microsoft Press, ISBN 1-55615-484-4, 1993.

Herramientas útiles para la XP

	Java Tools for Extreme Programming: Mastering Open Source Tools Including Ant, JUnit, and Cactus. **Autor**: Richard Hightower, Nicholas Lesiecki. **Qué incluye**: Aborda la parte más práctica de la XP, refactorización, tests, . . usando Java como herramienta. **Nivel**: Avanzado.

R.2 PÁGINAS WEB

Desde hace relativamente pocos años contamos entre nosotros con otra herramienta para el intercambio de información entre personas a lo largo del tiempo, esta herramienta es Internet, que, para nosotros, ha sido la principal fuente de información.

A lo largo del desarrollo del trabajo hemos indagado en cientos, quizás cerca de mil, páginas web relacionadas de una forma u otra con la XP o alguno de sus puntos o valores. Vamos a ver aquellas que nos han resultado más útiles.

Extreme Programming

- **Extreme programming: A gentle introduction**
  http://www.extremeprogramming.org

- **An eXtreme Programming Resource**
  http://www.xprogramming.com

- **XP: The New Zealand Way**
  http://www.xp.co.nz/

- **Wiki Wiki Web**
  http://c2.com/cgi/wiki?ExtremeProgramming

Extreme Programming “FAQ’S”

- **Extreme programming FAQ**
  http://www.jera.com/techinfo/xpfaq.html

- **Extreme programming FAQ from jGuru**
  http://www.jguru.com/faq/printablefaq.jsp?topic=XProgramming

- **Extreme programming FAQ**
  http://www.aka.org.cn/Magazine/Aka6/light/Extreme%20Programming%20FAQ.htm

Pair Programming

- **PairProgramming.com**
  http://pairprogramming.com/

- **Development times two**
  http://www.infoworld.com/articles/mt/xml/00/07/24/000724mtpair.xml

- **Some papers about pair programming**
  http://www.cse.ucsc.edu/classes/cmps012a/Winter01/sbrandt/supplements/pairProgramming.html

Refactoring

- **Refactoring Home Page (Martin Fowler)**
  http://www.refactoring.com/

- **“Refactoring, Reuse & Reliability”**
  http://st-www.cs.uiuc.edu/users/opdyke/wfo.990201.refac.html

- **“Lifecycle and Refactoring”**
  http://www.laputan.org/lifecycle/lifecycle.html

Pruebas

- **"Test before programming"**
  http://www.xprogramming.com/xpmag/Reliability.htm

Management

- **XP management**

http://www.xp.co.nz/Management.htm

R.3 OTROS RECURSOS

Internet nos brinda muchísima información y diversas formas de comunicación, algunas de ellas nada tienen que ver con una página web o un libro, pero nos son de mucha utilidad.

Existen multitud de listas de correo y foros abiertos en internet sobre temas de la extreme programming. Aquí pretendemos ofrecer una selección de ellos para todos los que quieran acercarse por primera vez a la programación extrema.

Listas de correo.

- XPDENVER
  http://groups.yahoo.com/group/xpdenver/

- Extreme Programming
  http://groups.yahoo.com/group/extremeprogramming/

- XP User Group
  http://groups.yahoo.com/group/xpusergroups/

- The Refactoring Group
  http://groups.yahoo.com/group/refactoring/

Documentos on line

- "Resilience vs. Anticipation”
  http://www.forbes.com/asap/97/0825/056.htm
- “The new Methodology”
  http://www.martinfowler.com/articles/newMethodology.com
- “Future of Software Development”
  http://www.technetcast.com/tnc_play_stream.html?stream_id=227