Scrum Manager

En busca de la excelencia del Código

Raúl Herranz

Rev. 1.0

Scrum Manager
En busca de la excelencia del Código

Formación

Rev.1.0.0 Febrero-2012

Título

En busca de la excelencia del Código

Autor

Raúl Herranz Serrano

Imagen de Portada

Raúl Herranz Serrano

Revisiones y ampliaciones

1.0.- Febrero de 2012

Más información, y última versión en: http://www.scrummanager.net

Derechos

Derechos registrados en Safe Creative.
Condiciones de uso, copia y distribución en: http://www.safecreative.org/work/1202091053784

Contenido

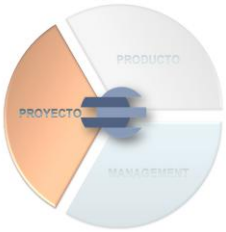
Prólogo

Apuntes de formación Scrum Manager

Este es un libro de texto para formación en el área de Ingeniería del marco de gestión "ScrumManager®".

Es un recurso educativo abierto (OER) y forma parte de la plataforma Open Knowledge Scrum: http://scrummanager.net/oks/

Se puede emplear de forma gratuita para consulta y auto-formación a título personal.



Si desea ampliar la formación asistiendo a un curso presencial de Scrum Manager sobre excelencia de código, solicitar un curso para su empresa, o una certificación académica de este temario, puede consultar el calendario de cursos presenciales (http://scrummanager.net/cursoscalendario) o contactar con Scrum Manager.

Sobre certificación académica Scrum Manager: http://scrummanager.net/certificacion

Plataforma abierta para consulta y formación profesional Scrum Manager

ScrumManager Member

Open Knowledge Scrum es una plataforma de acceso libre para consulta y formación, está disponible en http://www.scrummanager.net/oks/ donde encontrarás la última versión de este curso, además de otros materiales, foros, talleres, etc.

Un punto abierto en la Red para consultar y compartir conocimiento, y mantenerte profesionalmente actualizado.

Servicios profesionales para formación y asesoría Scrum Manager

ScrumManager Certified

Puede contactar con empresas certificadas para servicios profesionales de formación y asesoría en la implantación y mejora de Scrum Management, en el directorio de centros de formación autorizados Scrum Manager: http://scrummanager.net/centros

Contacto e información:

http://scrummanager.net/
formacion@scrummanager.net

Agradecimientos

Mi más sincero agradecimiento a los miembros y colaboradores de Scrum Manager por las sugerencias y aportaciones ofrecidas durante la revisión del contenido del libro y, en especial a Claudia Ruata y a Juan Palacio por la confianza que han depositado en mí desde mis primeros pasos como colaborador en los cursos de la plataforma Open Knowledge (http://www.scrummanager.net/oks), sin la cual no me hubiese animado a escribir este libro.

También agradezco todo el apoyo que me han prestado mi familia, mis amigos y todos aquellos que me acompañan en mi trabajo, ya que ellos mejor que nadie conocen y soportan mis peculiaridades.

Y no quiero dejar de dar gracias a Patricia, porque ella es quien me brinda día a día la posibilidad de alcanzar, entre tanta locura, algunos momentos de cordura.

A todos, gracias por hacer posible este libro.

Raúl Herranz

Introducción

¿A quién va dirigido este libro?

Este libro va dirigido a vosotros, miembros de aquellos equipos de desarrollo de software interesados en la mejora de la calidad de sus proyectos, entendiendo la calidad como el conjunto de propiedades que le confieren la capacidad para satisfacer las necesidades implícitas o explícitas para las que se ha desarrollado, y que para ello quieran conocer y aprender diferentes técnicas y herramientas que les permitan alcanzar la excelencia de su código.

Además, este libro también resultará útil a cualquier otra persona involucrada en un proyecto de desarrollo de software, como pueden ser los miembros de la gerencia de las empresas u organizaciones que dedican sus recursos a este tipo de proyectos o los clientes de las mismas.

Todos vosotros podréis encontrar en estas páginas información útil para conocer estas técnicas y herramientas, así como los beneficios de animar a los equipos de desarrollo a utilizarlas en sus proyectos.

¿Qué es la excelencia del Código

?
Intentar definir la excelencia del código puede resultar, a priori, un objetivo muy ambicioso. Sin embargo, quizá resulte más sencillo establecer una serie de características que aseguran que el código desarrollado no puede calificarse de excelente:

- **un código que no cumple la funcionalidad esperada, *no es un código excelente***.

  Ante una definición como la siguiente: “Cuando un usuario se de de alta en el sistema se enviará un correo electrónico de bienvenida. Además, si tras el alta el usuario no accede durante un periodo parametrizable, el sistema le enviará otro email recordando los datos de acceso, y notificará este hecho al administrador.”

  Estaríamos ante una situación de código no excelente en cualquiera de los siguientes casos (o una combinación de ellos):

  - si un usuario no pudiera darse de alta,
  - si a un usuario que se diera de alta el sistema no le enviase un email de bienvenida,
  - si el periodo de tiempo para enviar el email al usuario recordándole sus datos de acceso no fuera parametrizable,
  - si pasado el periodo indicado no se enviase el email al usuario recordándole sus datos de acceso, o
  - si pasado el periodo no se notificase al administrador que el usuario no ha accedido al sistema.

- **un código que provoca errores, *no es un código excelente***.

  Partiendo de la definición del sistema del ejemplo anterior, tampoco sería un código excelente, por ejemplo, aquel que al enviar un email a un usuario para recordarle sus datos de acceso pasado el periodo indicado cruzase los datos con los de otro usuario.

- **un código difícil de entender, *no es un código excelente***.

  Existen muchos factores que dificultan a otros desarrolladores, o incluso al desarrollador original pasado un determinado tiempo, comprender qué es lo que hace el código, y por qué. Algunos ejemplos podrían ser:

  - la utilización de valores arbitrarios como constantes,
  - la escritura de líneas de código de longitud excesiva,
  - el uso de expresiones condicionales muy complejas, o
  - la ausencia de documentación o sus defectos (documentación incompleta, no actualizada o con errores)

- **un código que no utiliza la potencia de la orientación a objetos, *no es un código excelente***.

  Un ejemplo, por desgracia muy común, es el abuso del “copy+paste” frente a los mecanismos de reutilización proporcionadas por el paradigma de la orientación a objetos basados en la herencia, la abstracción y el encapsulamiento o el uso de librerías, frameworks y/o componentes.

- **un código que podríamos reducir, *no es un código excelente***.

  Cuando hay código muerto (código que no es accesible desde ningún camino de ejecución posible), grandes cantidades de código comentado o clases/métodos que varían muy poco entre ellos, resulta sencillo reducir el

tamaño del código del proyecto ya sea eliminando lo que ya no se usa o sacando factor común al código similar. Si no se realizan estas sencillas acciones el código no podrá calificarse de excelente.

Así pues, a partir del anterior listado se podría aventurar una definición de la excelencia del código concluyendo que, para poder calificarlo como excelente:

> ***al menos será necesario que el código cumpla con la funcionalidad esperada sin errores, siendo fácilmente comprensible para cualquier desarrollador, utilizando los mecanismos proporcionados por la orientación a objetos y minimizando su tamaño***

¿Qué encontrarás en los siguientes capítulos?

En los siguientes capítulos se realizará un repaso sobre una serie de actividades que resultan de gran ayuda en el objetivo de lograr el desarrollo de un código excelente:

- la **Automatización de Tareas**,
- las **Revisiones del Código**,
- las **Pruebas**, y
- la **Refactorización**.

La explicación de estas técnicas vendrá apoyada por ejemplos basados en el paradigma de la Orientación a Objetos y, más concretamente, en la plataforma de desarrollo JEE. Sin embargo, muchos de estos conceptos son fácilmente trasladables a otros lenguajes o plataformas, e incluso a otros paradigmas de programación. Por eso, aunque tus proyectos no estén basados en esta plataforma o en este paradigma en concreto, estoy seguro de que la lectura de este libro te seguirá resultando de gran ayuda.

Automatización de Tareas

Introducción

En todos los proyectos de desarrollo de software existen una serie de tareas habituales (o que deberían ser habituales) que pueden ser automatizadas de un modo relativamente sencillo:

- *construcción del proyecto*
- *ejecución de pruebas*
- *verificaciones del código*
- *despliegue*
- *generación de la documentación*

La sencillez de la automatización de estas tareas viene dada por su **repetibilidad**. De hecho, esta cualidad hace que, cuando no se automatizan, estas tareas consuman un tiempo muy valioso del equipo de desarrollo que lo aleja de su principal objetivo: desarrollar el código que permita proveer al cliente de un producto que satisfaga sus necesidades; una situación nada deseable, ya que podría desembocar en el desánimo del equipo.

Para evitar esta situación, los equipos de desarrollo deben perseguir la automatización de estas tareas repetitivas. Con el objetivo de asegurar el éxito de esta automatización puede ser recomendable seguir la secuencia de pasos que se describe a continuación:

1. establecer unas **convenciones de desarrollo**, donde al menos se incluyan:
   - las herramientas (tanto de desarrollo como de automatización) y librerías externas que se utilizarán
   - la distribución de los artefactos generados
   - las reglas de estilo del código y de su documentación
2. **colectivizar el código fuente**,
3. escribir los **scripts de automatización**,
4. implantar un **sistema de integración frecuente**.

En los siguientes apartados se tratará cada uno de estos pasos en detalle.

Convenciones de desarrollo

Las convenciones para el desarrollo del proyecto son una primera herramienta muy útil para los equipos de desarrollo que quieren automatizar sus tareas. Además, esas convenciones facilitan la creación de un entorno controlado y amigable en el que los miembros del equipo pueden desarrollar su trabajo, logrando aumentar paralelamente la cohesión del equipo.

Como ya se comentó en la introducción, las convenciones de desarrollo deben contener, al menos, los siguientes aspectos:

- **un catálogo de herramientas y librerías externas**
- **unas normas de distribución de los artefactos del proyecto**
- **unas reglas de estilo para el código y su documentación**

En los siguientes epígrafes se describen en detalle estos contenidos.

Catálogo de herramientas y librerías externas

Establecer y mantener un catálogo de las herramientas utilizadas durante el desarrollo del proyecto, así como de las librerías utilizadas es la base a partir de la que se construirá un entorno de desarrollo común en el cual será posible detectar, frente a determinadas tareas y/o problemas, formas de actuar comunes y repetitivas llevadas a cabo por los diferentes miembros del equipo de desarrollo. Además, el entorno de desarrollo de los equipos que cuentan con este catálogo de herramientas y liberías externas gozará de un mayor grado de estabalidad que el de aquellos equipos que no dispongan de él.

Para que este catálogo resulte efectivo, lejos de considerarse inmutable debe entenderse como algo vivo, ya que a lo largo de la vida del proyecto podrán darse situaciones que hagan necesario ampliar el conjunto de herramientas o librerías, así como realizar actualizaciones a nuevas versiones.

Ahora bien, y sobre todo cuando se habla de nuevas versiones, los cambios no deben realizarse sin valorar previamente las consecuencias de los mismos y, en su caso, será imprescindible realizar las pruebas necesarias antes de aceptarlos, buscando evitar de este modo posibles errores o problemas debidos a incompatibilidades con las nuevas versiones candidatas.

Aunque cada equipo de desarrollo debe crear este catálogo a partir de sus necesidades concretas, a modo de ejemplo se muestra un sencillo catálogo de herramientas y liberías que

podría utilizarse para el desarrollo de un proyecto web JEE:

HERRAMIENTAS		
Nombre	**Versión**	**Notas**
JDK	1.6.0_04	**Ubicación:** /usr/local/jdk1.6.0_04
Eclipse	Helios	**Ubicación:** /usr/local/eclipse **Plugins:** • Web Tools • Mylyn
…		
LIBRERÍAS		
Nombre	**Versión**	**Notas**
IceFaces	2.0.2	**Ficheros:** icefaces202.jar
…		

Otra alternativa interesante es la creación de máquinas virtuales (utilizando herramientas como VMWare, VirtualBox o Hyper-V) con una versión "limpia" del entorno de desarrollo, de modo que cualquier miembro del equipo pueda disponer en cualquier momento de una máquina completamente configurada y lista para instalarla en su equipo y poder comenzar a trabajar sin esperas.

Distribución normalizada de artefactos

En el ejemplo mostrado en el apartado anterior, el ojo atento ya se habrá percatado, por las notas asociadas a las herramientas utilizadas, de la importancia de especificar la ubicación de los distintos artefactos (en el caso del ejemplo, de las herramientas).

Estas normas que establecen la ubicación o distribución de los artefactos en directorios comunes a los diferentes miembros del equipo -o incluso a diferentes proyectos de una misma empresa u organización- incrementa notablemente la facilidad de automatización de las tareas repetitivas, ya que el comportamiento de las acciones deberá ser el mismo sin que la máquina utilizada para ejecutar los scripts de automatización sea una variable a tener en cuenta.

Además, la estandarización de los entornos disminuye la curva de aprendizaje para los posibles nuevos desarrolladores que pasen a formar parte del equipo del proyecto, especialmente cuando la distribución normalizada es común a toda la empresa/organización. Gracias a esta disminución de la curva de aprendizaje se facilita a estos nuevos miembros el sentirse cómodos, por la familiaridad de los entornos, al incorporarse a proyectos que ya se encuentren iniciados. Y esta comodidad, acaba facilitando su rápida y completa integración en el equipo.

Como punto de partida, un equipo de desarrollo puede partir, a la hora de establecer una distribución de directorios en algún estándar de la industria. Como ejemplo, un equipo podría utilizar la distribución de artefactos propuesta por la herramienta Maven (http://maven.apache.org):

Directorio	Contenido
/	build.xml, ficheros de propiedades, README.txt
/src/main/java	Código fuente
/src/main/resources	Recursos
/src/main/resources/sql	Scripts SQL
/src/main/config	Ficheros de configuración
/src/main/webapp	Ficheros web (páginas, imágenes, hojas de estilo)
/src/test/java	Código fuente de los tests
/src/test/resources	Recursos de los tests
/src/test/config	Ficheros de configuración de los tests
/target	Salida de la construcción del proyecto
...	...

Siguiendo esta distribución, el directorio raíz contendrá los ficheros build.xml, README.txt (en el caso de que exista contendrá información útil para otros desarrolladores, especialmente información que trate de cambios o personalizaciones sobre los estándares establecidos), ficheros de propiedades y, aunque se permiten otros ficheros y directorios de metadatos (como los creados por herramientas como el CVS y Subversion), sólo se crearán explícitamente dos directorios:

- **target:** ubicación donde, tras ejecutar el proceso de construcción del proyecto, se almacenarán todos los productos generados.
- **src:** directorio que contendrá los fuentes, recursos y ficheros de configuración que permitirán construir el proyecto. Contendrá un subdirectorio main para el artefacto principal del proyecto y, además, podrá contener otros subdirectorios como la carpeta test que contedrá los fuentes de los tests, sus ficheros de configuración, etc.

Reglas de estilo del código y su documentación

El último punto que debería incorporarse (como mínimo) a las convenciones de desarrollo de los proyectos serían unas normas de estilo tanto para el código como para la documentación técnica asociada al mismo. Estas normas de estilo reportan una serie de beneficios indiscutibles a los equipos de desarrollo de software:

- mejoran la comprensión del código desarrollado por otros miembros del equipo,
- facilitan la portabilidad y la reutilización del código desarrollado, así como su mantenimiento,
- ayudan a la integración de nuevos desarrolladores al equipo, acelerando su integración en el flujo de trabajo, y
- proporcionan una imagen de profesionalidad del trabajo realizado por el equipo.

En la industria del desarrollo de software existen convenciones más o menos estándar. Cada equipo de desarrollo debe considerar si seguir alguna de estas recomendaciones o crear unas normas propias adaptadas a su estilo. Sea cual sea la opción elegida, la coherencia es vital:

> ***Una vez establecidas las normas estas deben ser conocidas y adoptadas por todo el equipo.***

Tan sólo en casos excepcionales, por exigencias del proyecto, podrán obviarse de manera consensuada en un momento dado.

Java Code Conventions

Como ejemplo de convenciones o reglas del código y la documentación para proyectos java, los equipos de desarrollo pueden utilizar las Java Code Conventions, que siguen siendo un referente a pesar de que fueron publicadas en septiembre de 1997, y que están accesibles en:

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf
(Sun Microsystems, 1997)

Tras comenzar con una breve explicación sobre los beneficios que aportan las convenciones del código fuente, este documento hace un repaso de aspectos tan variados como el nombrado y organización de los ficheros, el uso de la sangría o indentación y las lineas en blanco para la mejora de la legibilidad o el formato y contenido de los comentarios entre otros. A pesar de la validez contrastada de este documento, a continuación se describen algunos aspectos que quizá resulte interesante valorar:

3.1.1 Beginning Comments
All source files should begin with a c-style comment that lists the programmer(s), the date, a copyright notice, and also a brief description of the purpose of the program.

Actualmente, la información relativa al versionado (fecha, desarrollador, versión...) puede resultar innecesaria cuando se utiliza un sistema de control de versiones.

4. Indentation
Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified. Tabs must be set exactly every 8 spaces (not 4).

Siempre es preferible el uso del espaciado frente al tabulador por el distinto tratamiento que diferentes sistemas hacen del mismo.

En el caso de usar el tabulador, siempre que sea posible es preferible establecerlo a un ancho de 4 caracteres (los mismos caracteres que se recomienda usar con el espaciado).

6.2 Placement
Put declarations only at the beginning of blocks.

En ocasiones puede resultar interesante situar las declaraciones próximas al código donde se utilizan las variables ya que así se puede facilitar una posterior extracción de dicho código a un método propio.

9. Naming Conventions
Variable names should be short yet meaningful.

En aquellos casos en los que un nombre más largo elimine la necesidad de un comentario explicativo adicional, podría obviarse esta norma.

10.5.4 Special Comments
Use XXX in a comment to flag something that is bogus but works. Use FIXME to flag something that is bogus and broken.

Existen alternativas[1] al uso de XXX como:
- KLUDGE
- KLUGE
- HACK

[1] tan sólo hay que hacer una búsqueda por el término “XXX” en internet y echar un vistazo a los resultados obtenidos para entender la razón.

Colectivización del código fuente

Otro de los aspectos importantes para lograr una automatización efectiva de las tareas repetitivas del proyecto es la colectivización del código fuente, o lo que es lo mismo:

> ***Se debe permitir que todos los miembros del equipo de desarrollo del proyecto puedan realizar modificaciones sobre cualquier porción del código fuente.***

Aunque la primera vez que se nos presenta esta idea pueda resultar normal pensar que la adopción de esta práctica se convertirá en un camino directo al caos en el proyecto, la comunicación fluida entre los miembros del proyecto junto al uso de un sistema de control de versiones evitará este fatal destino, facilitando que se puedan disfrutar de los innumerables beneficios que aporta, entre los que se pueden destacar los siguientes:

- **se eliminan cuellos de botella**

en aquellos equipos de desarrollo en los que varios miembros están desarrollando diferentes módulos que hacen uso de otro módulo "central" que es mantenido por un único desarrollador, cuando varios desarrolladores precisan modificaciones que afectan a dicho módulo "central", su propia naturaleza hace que automáticamente dicho módulo se convierte en un cuello de botella para el desarrollo del proyecto.

- **el conocimiento se comparte**

partiendo del caso del ejemplo anterior, si el desarrollador que mantiene el módulo "central" causase baja, cualquier modificación que fuese necesario realizar en el módulo "central" supondría un grave problema ya que implicaría que otro miembro del equipo tendría que enfrentarse a un código totalmente desconocido por él.

- **el diseño se vuelve vivo**

si el diseño del sistema surge únicamente del trabajo previo de un arquitecto y cada desarrollador únicamente se hace cargo de una pequeña porción del mismo, cualquier propuesta de cambio tendrá que ser revisada por el arquitecto, que será la única persona que tendrá la visión en conjunto. Sin embargo, cuando el código es de naturaleza colectiva, como el conocimiento se reparte entre todos los miembros del equipo, estos pueden aportar mejoras al diseño en función de las necesidades concretas que vayan surgiendo durante el desarrollo, convirtiéndolo por tanto en un diseño vivo al servicio de los objetivos del proyecto.

Sistemas de control de versiones

Los sistemas de control de versiones, como se ha adelantado, son herramientas que facilitan la colectivización del código debido a que, básicamente, son sistemas que permiten compartir información de una manera ordenada.

El concepto clave de estos sistemas es el **repositorio**, el cual es un almacén de ficheros, que puede estar centralizado (CVS, Subversion, ...) o distribuido (Git, Mercurial, ...), y que permite almacenar información mediante la típica jerarquía de un sistema de ficheros en forma de árbol de directorios.

A este repositorio pueden conectarse un número indeterminado de clientes para leer o escribir de tal manera que:

- al escribir datos, un cliente posibilita el acceso a la información a otros clientes, mientras que
- al leer datos, un cliente recibe la información compartida por el resto de clientes.

Además, el repositorio de un sistema de control de versiones posee la capacidad de recordar los cambios que se realizan tanto sobre los ficheros como sobre la estructura de directorios. Entre estos cambios que el repositorio es capaz de recordar se incluyen la creación, la actualización, el borrado e incluso el cambio de ubicación de los ficheros y los directorios.

Gracias a esta característica, los clientes pueden consultar el estado en un determinado instante tanto del repositorio en general como de un elemento en particular, accediendo a su histórico, y obteniendo entre otros datos el contenido en dicho momento, quien o quienes realizaron modificaciones y cuales son los cambios que se llevaron a cabo respecto a una determinada versión.

Gestión de las modificaciones

Aunque un sistema de control de versiones pueda parecer sencillo, en realidad debe enfrentarse a un problema para nada trivial:

¿Cómo prevenir que, de manera accidental, las modificaciones de unos clientes sobre-escriban los cambios introducidos por otros?

Para hacer frente a este problema, las primeras herramientas optaban por un **modelo basado en la exclusividad** a la hora de realizar modificaciones, es decir, sólo permitían que una persona, en un momento dado, modificase un fichero.

Para gestionar esta exclusividad, estos sistema de control de versiones usaban la técnica del bloqueo: cuando un desarrollador ha bloqueado un fichero, condición indispensable para poder editarlo, ningún otro miembro puede bloquearlo ni, por tanto, tampoco realizar cambios sobre él. A lo sumo podría leer el fichero y esperar a que su compañero finalizase sus modificaciones y liberase el bloqueo.

Este modelo de bloqueo-modificación-desbloqueo presenta una serie de inconvenientes:

- **problemas administrativos**

  En determinados casos, por ejemplo si por un descuido se deja bloqueado un fichero cuando un miembro del equipo se toma unas vacaciones, puede ser necesaria requerir la ayuda de un administrador del sistema para liberar un bloqueo. Además de causar un retraso en el proyecto, este bloqueo habría provocado que diversos actores tuviesen que malgastar su tiempo y su esfuerzo para resolverlo.

- **causa procesos en serie innecesarios**

  A pesar de que diversos trabajos pueden realizarse en paralelo (por ejemplo, cuando se modifican dos métodos diferentes de una misma clase) el sistema de bloqueos impide esta forma de trabajo.

- **causa una falsa sensación de seguridad**

  Cuando se trabaja con bloqueos se tiende a pensar que nuestros cambios no afectarán (ni se verán afectados) al trabajo de otros miembros del equipo de desarrollo. Sin embargo, si dos miembros del equipo editan ficheros diferentes, ¿qué ocurrirá si estos ficheros dependen el uno del otro, y los cambios realizados en cada uno son incompatibles?

Para hacer frente a estos problemas, surge otro modelo de versionado: el **modelo basado en la fusión**. Un modelo de copia-modificación-fusión en el que cada cliente mantiene una copia local del proyecto que puede modificar de forma independiente, tras lo cual debe fusionar los cambios con los contenidos del repositorio.

Este modelo puede parecer caótico, pero en la práctica su funcionamiento presenta pocos contratiempos. Los usuarios pueden trabajar en paralelo, sin tener que esperar los unos por los otros e, incluso cuando trabajan en los mismos ficheros, la mayoría de las veces los cambios concurrentes no se solapan, de manera que los conflictos son poco frecuentes. El esfuerzo empleado en resolver estos conflictos normalmente será mucho menor que el tiempo malgastado con los bloqueos.

En cualquier caso, el correcto funcionamiento de cualquier sistema de control de versiones depende de un factor crítico:

La comunicación entre los usuarios

Cuando los usuarios tienen una comunicación pobre, los conflictos sintácticos y semánticos se incrementan, mientras que cuando la comunicación es rica y fluida, los conflictos disminuyen.

Conceptos clave

Cuando se trabaja con sistemas de control de versiones, hay una serie de conceptos, aparte del concepto de repositorio, que juntos conforman su propio vocabulario:

- **Copia de trabajo**

  Árbol de directorios en el sistema de archivos del cliente que contiene una copia del estado, en un determinado instante, del repositorio o de una parte del mismo. El cliente puede editar estos archivos como desee ya que su copia de trabajo es su área de trabajo privada.

  Mientras el cliente trabaje con la copia de trabajo, el sistema de control de versiones nunca incorporará los cambios que se suban al repositorio, ni hará que sus cambios estén disponibles para los demás. Para ello, se deberá pedir al sistema de forma explícita.

- **Revisión**

  Cada vez que un cliente publica en el repositorio los cambios realizados el repositorio crea un nuevo estado del árbol de archivos, llamado revisión. A cada revisión se le asigna un número natural único y mayor que el de la revisión anterior.

- **Tag (etiqueta)**

  El concepto de "tag" (etiqueta) representa una fotografía de un proyecto en un momento dado.

  En sistemas de control de versiones como Subversion, cada revisión del proyecto podría tomarse como una etiqueta (puesto que precisamente cada revisión se trata exactamente de una fotografía del proyecto en un momento determinado), sin embargo, incluso en estos sistema de control de versiones se sigue usando el concepto de "tag", puesto que al crear una etiqueta es posible asignarle un nombre que sea más fácil recordar (y asociar con un hito del proyecto) que el número de la revisión correspondiente de la linea principal de desarrollo del proyecto.

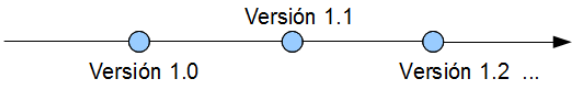


Ilustración 1 – Tag (etiqueta)

- **Branch (rama)**

  Una rama o "branch" es una linea de desarrollo que existe de manera independiente a otra linea, pero con la que comparte una historia común a partir de un punto en el pasado. Una rama siempre comienza como una copia de algo, momento a partir del cual comienza a generar su propia historia.

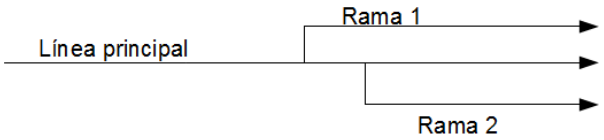


Ilustración 2 – Branch (rama)

- **Merge (fusión)**

  Este concepto representa el acto de replicar los cambios efectuados en una rama (branch) a otra.

  En función de la cantidad de cambios realizados en cada rama a fusionar, así como de la complejidad de los mismos, la fusión podrá ser más sencilla o más compleja.

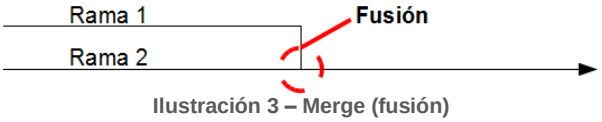


Ilustración 3 – Merge (fusión)

Estrategias de ramificación y fusión

A la hora de crear y fusionar ramas es posible optar por un sinfín de estrategias. Estas estrategias implican la búsqueda del equilibrio entre el riesgo, la productividad y el costo ya que se enfrentan:

- la seguridad de trabajar de forma aislada,
- el aumento de la productividad que proporciona el trabajo en paralelo, y
- el aumento del costo debido al esfuerzo adicional de realizar las fusiones entre ramas.

El primer paso a la hora de determinar la estrategia a seguir consiste en decidir qué representará cada rama. Habitualmente las ramas se suelen alinear con aspectos correspondientes a:

- la estructura de descomposición del producto (EDP o PBS -del inglés *Product Breakdown Structure*) donde se especifica la arquitectura o la funcionalidad del sistema,
- las unidades organizativas en las que se descompone el equipo de desarrollo, o
- la estructura de desglose del trabajo (EDT o WBS -del inglés *Work Breakdown Strutcture*) que define las tareas o fases del proyecto.

Un método para validar la opción escogida es considerar escenarios de cambio: si se baraja alinear las ramas con la arquitectura (por ejemplo una rama por cada componente) se deberá imaginar un escenario con un número significativo de cambios arquitectónicos que obliguen a re-estructurar las ramas y los procesos y políticas asociados a las mismas. Así, en función de la probabilidad de que ocurra ese escenario, se podrá determinar si dicha opción es, o no, la más acertada.
A continuación se exponen de manera breve algunas de las estrategias de ramificación y fusión más habituales. Sin embargo, más importante que conocer al detalle cada una de ellas, lo importante es comprender la importancia que tiene la elección de la estrategia de ramificación y fusión y conocer la amplia gama de

posibilidades que puedes adoptar en función de la naturaleza de los proyectos:

- **Rama por versión**

  Una de las estrategias más común es alinear cada rama con una versión del producto. Ocasionalmente puede ser necesario fusionar las actualizaciones desde una versión a otra pero, normalmente, las ramas nunca se fusionan.

  En este caso las ramas siguen vivas hasta que dejan de usarse por dejar de dar soporte a la versión a la que corresponde.

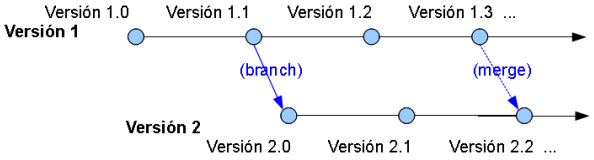


Ilustración 4 – Rama por versión

- **Rama por fase del proyecto**

  Otra aproximación muy común es alinear las ramas con los niveles de promoción de los artefactos de software. A una versión específica de Desarrollo le corresponde una rama de Test, donde se realizarán las pruebas de integración y de sistema. Una vez completadas las pruebas, los artefactos de software pasan a una nueva rama de Producción, desde la cual podrán ser desplegados.

  Durante las pruebas se puede actualizar el código al encontrar y solucionar errores. Estos cambios se fusionarían con el código de la rama de Desarrollo cuando se promocionasen los artefactos a la rama de Producción.

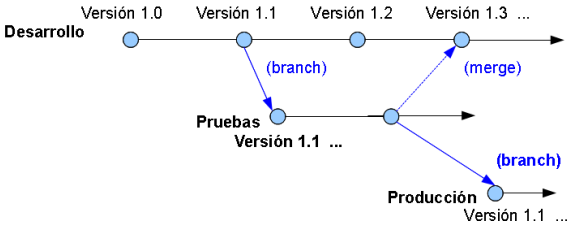


Ilustración 5 – Rama por fase del proyecto

- **Rama por tarea**

  Otra opción, que busca evitar una pérdida en la productividad debido al solapamiento entre tareas, es la de aislar cada tarea en ramas separadas. Estas ramas serán ramas con una esperanza de vida corta, ya que tendrán que fusionarse con la rama principal tan pronto como se complete la tarea correspondiente. En otro caso, el esfuerzo requerido para realizar la fusión excederá los beneficios de crear una rama para cada tarea.

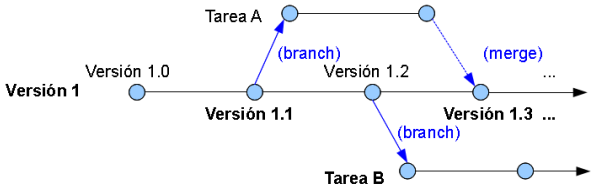


Ilustración 6 – Rama por tarea

- **Rama por componente**

  Otra posibilidad es alinear cada rama cada componente/subsistema del sistema, dejando una rama (habitualmente la principal) como rama de integración. El equipo de desarrollo decidirá, en función de la evolución de cada componente, cuando realizar las fusiones de los mismos con la rama de integración.

  Esta estrategia funcionará bien si la arquitectura del sistema está definida correctamente y cada componente ha definido de igual forma sus interfaces. El hecho de desarrollar cada componente en una rama separada posibilita un control de grano fino sobre el desarrollo de los distintos artefactos software.

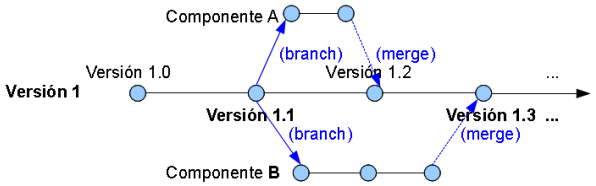


Ilustración 7 – Rama por componente

- **Rama por tecnología**

  Esta estrategia está, de igual forma, alineada con la arquitectura del sistema. Sin embargo las ramas no hacen referencia a la estructura interna del producto, si no a las diferentes plataformas tecnológicas para las cuales se desarrolla, permitiendo así mantener el código común en una rama y el código propio de cada plataforma en ramas específicas.

  En este caso, debido a la naturaleza de los artefactos desarrollados en cada rama, lo más probable es que estas ramas nunca se fusionen.

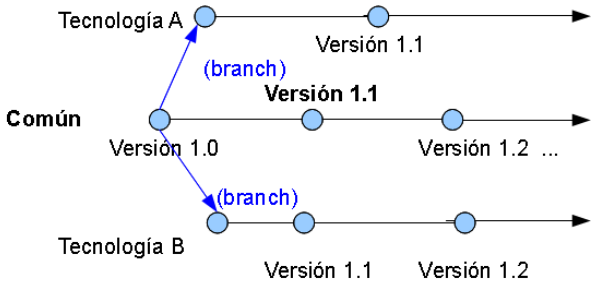


Ilustración 8 – Rama por tecnología

- **Otras estrategias**

  Además de las estrategias descritas, existe una gran variedad de opciones adicionales a la hora de estructurar las ramas del sistema de control de versiones, algunas de las cuales puede ser una combinación de algunas de las estrategias que se han visto. Por poner un ejemplo, se podría optar por una rama por cada componente, donde además por cada componente se estableciese una estrategia de rama por tarea, todo ello combinado con una rama por fase del proyecto (Desarrollo, Test, Producción).

Scripts de automatización

Hoy en día, los entornos de desarrollo integrado (por ejemplo Eclipse) permiten realizar de manera automática o semiautomática algunas de las tareas habituales en el entorno local de un desarrollador. Sin embargo, esta solución de automatización presenta el problema de que queda limitada a dicho entorno local, por lo que no es posible automatizar tareas en los servidores de integración.

Entre las tareas a automatizar en los servidores de integración destacan:

- borrar o volver a crear el directorio de clases y los ficheros temporales,
- realizar la compilación,
- copiar los ficheros binarios y los recursos a las carpetas correspondientes,
- empaquetar el entregable como archivo jar, war o ear,
- generar la documentación javadoc del código fuente,
- desplegar en el servidor de pruebas,
- ejecutar los tests automatizados del proyecto,
- ...

Para dar respuesta a estas necesidades, existen en el mercado un gran número de opciones. Habitualmente, los equipos de desarrollos, en el mundo del desarrollo de aplicaciones JEE, se decantan por:

- usar un **lenguaje de script de propósito general** (shell en LINUX, batch en Windows...).

- usar **Ant**, una librería Java y una herramienta por línea de comandos cuya misión es dirigir diferentes procesos, generalmente con el objetivo de construir aplicaciones Java, descritos en unos ficheros de configuración.

  Esta herramienta proporciona una gran flexibilidad ya que no impone convenciones de codificación ni de distribución de los artefactos del proyecto.

- usar **Maven**, una herramienta de gestión de proyectos de software basado en el concepto de Modelo de Objetos de Proyecto (POM - Project Object Model) a partir del cual se puede administrar la construcción del proyecto así como la elaboración de informes y documentación a partir de una especificación única. Es una apuesta clara de convención sobre configuración, por lo que para trabajar de manera eficiente es necesario comprender y adoptar las convenciones que utiliza. Una clara ventaja frente a Ant es que permite gestionar las dependencias del proyecto.

Sistemas de Integración Frecuente

Los Sistemas de Integración Frecuente son herramientas que facilitan a los miembros de un equipo de desarrollo de software integrar su trabajo individual, frecuentemente, de manera que puedan realizarse pruebas y verificaciones del sistema completo (habitualmente de manera automatizada) permitiendo detectar errores de integración en etapas tempranas, reduciendo de este modo los problemas y logrando desarrollar un software bien cohesionado desde el principio.

Estos sistemas surgieron para hacer frente al problema al que se enfrentaban un gran número de equipos de desarrollo que eran incapaces de determinar el esfuerzo que les llevaría integrar un sistema en las fases finales del desarrollo, aumentando de ese modo el estrés en una situación ya de por sí habitualmente tensa.

Además, el uso de sistemas de integración frecuente suele estar asociado a una reducción del número de bugs:

> ***El hecho de poder realizar pruebas sobre el sistema integrado desde las fases iniciales del desarrollo permite solucionar los errores según se van encontrando.***

De esta manera, se evita que en las etapas finales del desarrollo exista una gran acumulación de errores que, a la dificultad intrínseca de su resolución, añadiría la dificultad asociada a la gestión de un elevado número de bugs.

Buenas prácticas

Para obtener los mayores beneficios de estas herramientas, es conveniente aplicar los siguientes consejos:

- **Actualizar de manera frecuente el repositorio**

  Los beneficios obtenidos de la integración frecuente son directamente proporcionales a la frecuencia con la que se actualiza el repositorio. De hecho, lo ideal es que si la estrategia de ramificación y fusión seleccionada lo permite, todos los miembros del equipo de desarrollo actualicen la rama principal con una frecuencia, al menos, diaria.

  Sin embargo, estas actualizaciones deben realizarse teniendo en cuenta, al menos, el siguiente conjunto de normas de etiqueta:

  - las actualizaciones no provocarán errores de compilación
  - las actualizaciones del código fuente se realizarán al mismo tiempo que las actualizaciones de los test correspondientes

- **Mantener un proceso de construcción rápido**

  Se debe mantener un proceso de construcción rápido, de modo que el feedback que obtengan los desarrolladores al actualizar el código del repositorio sea lo más inmediato posible.

  Para lograr un proceso de construcción rápido se suele separar la automatización en fases:

  - una primera fase en la que se realiza la construcción y se ejecutan los tests unitarios, de manera automática, cada vez que hay cambios en el repositorio.
  - una segunda fase en la que se ejecutan otras tareas secundarias (ejecución de pruebas de integración y aceptación automatizadas, generación de documentación, despliegue en entornos de pruebas, revisiones del código, informes de cobertura de los test unitarios, …) ya sea en procesos nocturnos o bajo demanda.

- **Notificar convenientemente los resultados**

  Todos los miembros del equipo de desarrollo deben recibir notificaciones con información sobre los resultados de los procesos de construcción (especialmente cuando hay errores). Además, siempre que sea posible, debe permitirse que aquellos otros interesados que lo deseen (gestores, personal de marketing, clientes, usuarios) tengan acceso a esta misma información.

Resumen

- En los proyectos de desarrollo de software existen tareas que pueden ser fácilmente automatizadas gracias a su repetibilidad.

- Para asegurar el éxito de la automatización se recomienda:
  1. establecer unas convenciones de desarrollo,
  2. colectivizar el código fuente,
  3. escribir los scripts de automatización,
  4. implantar un sistema de integración frecuente.

- Las convenciones de desarrollo deberían incluir:
  - catálogo de herramientas y librerías externas,
  - normas de distribución de los artefactos, y
  - reglas de estilo.

- La colectivización del código implica que todo el equipo de desarrollo puede modificar cualquier porción de código del proyecto.

- Los sistemas de control de versiones y la comunicación entre los miembros del equipo facilitan la colectivización del código.

- En el contexto de los sistemas de control de versiones se usan los siguientes conceptos:
  - Repositorio,
  - Copia de trabajo,
  - Revisión,
  - Tag (etiqueta),
  - Branch (rama), y
  - Merge (fusión).

- Hay un gran número de estrategias de ramificación y fusión, que deberán escogerse en función de las características del proyecto (*PBS*, organigrama, *WBS*), como por ejemplo:
  - Rama por versión,
  - Rama por fase del proyecto,
  - Rama por tarea,
  - Rama por componente, o
  - Rama por tecnología.

- A la hora de automatizar las tareas se puede usar:
  - Lenguaje de script de propósito general,
  - Ant, o
  - Maven.

- Los Sistemas de Integración Frecuente facilitan integrar el trabajo individual de los desarrolladores con frecuencia, permitiendo realizar pruebas y verificaciones del sistema al completo de manera automatizada.

- Para obtener el máximo beneficio de la integración frecuente se recomienda:
  - Actualizar de manera frecuente el repositorio,
  - Mantener un proceso de construcción rápido, y
  - Notificar convenientemente los resultados.

Revisiones de Código

Introducción

Bajo la denominación de revisiones de código se aglutinan una serie de técnicas que tienen por objetivo la localización de porciones del código que pueden/deben ser mejoradas.

Una posible clasificación de estas revisiones tiene en cuenta el momento en el que estas se llevan a cabo. Siguiendo esta clasificación se pueden diferenciar técnicas de revisión del código en paralelo al desarrollo del código y técnicas que se utilizan una vez finalizado:

- **en paralelo al desarrollo**
  - programación por parejas
- **a posteriori**
  - revisión por pares
  - análisis de métricas de calidad del código

Actualmente, la programación por parejas goza de una mayor aceptación entre los equipos de desarrollo que muchas veces consideran las técnicas de revisión a posteriori innecesarias y poco productivas debido, principalmente, a su origen en las metodologías clásicas o predictivas de gestión de proyectos. Sin embargo, bien entendidas y correctamente aplicadas, estas técnicas resultan una gran ayuda ya que la información que genera su aplicación permite establecer una sólida base sobre la que trabajar en la mejora, tanto del código ya desarrollado como del que se vaya a desarrollar. Además, el uso de estas técnicas facilita que el código desarrollado sea más comprensible para cualquier desarrollador y disminuye la curva de aprendizaje para nuevos desarrolladores que se incorporen al equipo.

Programación por Parejas

La Programación por Parejas es considerada como uno de los pilares en las que se asienta el éxito de la Programación Extrema (XP - eXtreme Programming), una metodología ágil de desarrollo ideada por Kent Beck, Ward Cunningham y Ron Jeffries que se dio a conocer cuando, gracias a su aplicación,la compañía Chrysler pudo lanzar el sistema Chrysler Comprehensive Compensation en Mayo de 1997 a pesar de los problemas iniciales que estaba teniendo su desarrollo siguiendo una metodología tradicional.

La Programación por Parejas es una técnica relativamente sencilla de aplicar en la cual dos programadores trabajan codo con codo, en un único ordenador, para desarrollar el código de un sistema de información. En este contexto, la persona que teclea recibe el nombre de *conductor*, mientras que la persona que revisa el código recibe el nombre de *observador* o *navegante*. Estos dos roles, *conductor* y *observador/navegante*, en ningún momento deben ser considerados como roles estáticos.

> ***Los miembros de la pareja deben intercambiar sus roles de manera frecuente durante toda la actividad de desarrollo.***

Si no se cumple este sencillo requisito, no se podrá hablar de una verdadera Programación por Parejas.

Programación por Parejas vs. Mentoring

Antes de presentar algunas variantes de la Programación por Parejas resulta conveniente marcar las diferencias existentes entre la Programación por Parejas y el Mentoring, otra técnica de trabajo en parejas con la que habitualmente se confunde.

El Mentoring puede explicarse como una práctica por la cual dos personas, una de ellas con sobrada experiencia (que tomaría el rol de *maestro*) y la otra con un talento que es necesario fomentar (que tomaría el rol de *aprendiz*), se comprometen durante un periodo limitado de tiempo a compartir su trabajo con el objetivo de contribuir juntos al desarrollo profesional del *aprendiz*.

En el caso del Mentoring los roles son estáticos: durante el periodo estipulado una persona actuará como *maestro* y la otra como *aprendiz*. Por contra, en la Programación por Parejas los roles (*conductor* y *observador/navegante*) deben intercambiarse de manera frecuente.

Otra diferencia entre ambas técnicas es el objetivo que se busca con su aplicación:

> ***Para el Mentoring, la meta es contribuir a mejorar el talento del aprendiz.***
>
> ***Para la Programación por Parejas, el objetivo es mejorar la calidad del código, o dicho de otro modo, la búsqueda de la excelencia del Código.***

Variantes de la Programación por Parejas

Si bien es cierto que la mejor manera de Programar por Parejas es **sentarse juntos frente al mismo monitor, intercambiando el teclado y el ratón continuamente**, y concentrados ambos en el código que se está escribiendo, este no es el único modo en la que puede aplicarse la Programación por Parejas.

Hoy en día no es extraño encontrar situaciones en la que se aplica, o se puede aplicar, esta técnica en entornos o equipos de desarrollo distribuidos. Para ello, los dos miembros de la pareja tendrán que usar herramientas como los editores colaborativos en tiempo real, los escritorios compartidos o los plug-ins para la **Programación por Parejas Remota** para el entorno de desarrollo integrado (IDE) que utilicen.

Esta Programación por Parejas Remota, sin embargo, introduce una serie de dificultades añadidas ya que es preciso un esfuerzo extra en las tareas de coordinación y además trae consigo una serie de problemas relacionados con la pérdida de la comunicación verbal cara a cara.

Otra posibilidad, a la hora de aplicar la Programación por Parejas, es la **combinación con otras técnicas como el desarrollo de pruebas unitarias** del Desarrollo Dirigido por Test (TDD - *Test Driven Development*). Esta variante se conoce con el nombre de Programación por Parejas 'Ping Pong', ya que, como en una partida de *ping pong*, los dos miembros de la pareja se 'pasan la pelota' a través del desarrollo de pruebas unitarias y del código que permita ejecutarlas sin errores a través del siguiente protocolo:

1. el *observador/navegante* escribe una prueba unitaria que falla y el conductor desarrollaría el código para que la prueba se ejecute sin errores.
2. se repiten estas dos tareas de forma cíclica hasta que el *observador* sea incapaz de escribir nuevas pruebas unitarias que fallen.
3. el *conductor* y el *observador/navegante* intercambian los roles, y se vuelve al punto 1.

Buenas prácticas en la Programación por Parejas

Todo se comparte

La Programación por Parejas va más allá de compartir un teclado, un ordenador y un monitor. Cuando se está aplicando esta técnica, los dos miembros de la pareja, sin importar el rol que estén desempeñando en un determinado momento, son los autores del código. Por ello, no deben permitirse recriminaciones del estilo "*este error ocurre en tu parte del código*" y, en su lugar, ambos tendrán que aprender a compartir sus éxitos y sus fracasos, expresándose mediante frases como "*cometimos un error en esta parte del código*" o, mucho mejor, "*hemos pasado todas las pruebas sin errores*".

Observador activo

Como ya se ha explicado, en la Programación por Parejas los dos desarrolladores intercambian de manera frecuente sus roles pasando de *conductor* a *observador/navegante* y viceversa. Cuando se produce este cambio, no debe caerse en el error de confundir el rol de *observador* con disponer de un periodo de descanso, realizar una llamada o contestar a un mensaje de texto recibido en el móvil. Es decir, no se debe caer en la tentación de convertirse en un *observador pasivo*.

Frente a este *observador pasivo* tenemos, sin embargo, al *observador activo*, que es aquel que ocupa su tiempo realizando un proceso continuo de análisis, diseño y verificación del código desarrollado por el *conductor*, tratando en todo momento de que la pareja sea capaz de alcanzar la excelencia del código.

Deslizar el teclado

Cuando se vaya a aplicar la Programación por Parejas se debe tener en cuenta una sencilla regla:

> ***Deslizar el teclado, no cambiar las sillas.***

Esta regla nos indica que el espacio de trabajo ocupado por la pareja debe estar dispuesto de modo que no sea necesario intercambiar el lugar frente al teclado (cambiar las sillas) para poder intercambiar los roles. Con el simple movimiento de deslizar el teclado para que éste cambie de manos la pareja debe ser capaz de intercambiar

los roles de *conductor* a *observador/navegante* y viceversa.

La importancia del descanso

La aplicación de la técnica de la Programación por Parejas puede resultar extenuante si no se realizan descansos de manera periódica. Estos descansos ayudarán a la pareja a retomar el desarrollo con la energía necesaria, por lo que es imprescindible que, durante los mismos, los dos desarrolladores mantengan sus manos y su mente alejados de la tarea que estén desarrollando en pareja.

Algunas tareas que se pueden realizar durante estos periodos de descanso son: revisar el correo electrónico personal, realizar una llamada de teléfono pendiente, contestar a los mensajes de texto recibidos en el móvil, navegar por la red, tomarse un tentenpié o, incluso, leer un capítulo de este libro.

Tiempo para el trabajo personal

Además de los descansos es conveniente que un porcentaje del tiempo de trabajo diario sea dedicado al trabajo personal.

Algunas de las actividades que se podrán beneficiar de este trabajo personal son:

- el prototipado experimental, es decir, comprobar si una idea surgida durante el trabajo en pareja es factible antes de ponerla en común con el compañero, o
- la resolución de problemas que requieren un nivel alto de concentración, ya que en estos casos, el trabajo en parejas puede ser contraproducente.

El coste de la Programación por Parejas

Habitualmente los gestores de una empresa de desarrollo de software y los gestores de proyectos, cuando escuchan por primera vez hablar de la Programación por Parejas, piensan de forma natural que el coste del desarrollo de un proyecto en el que se aplica será sensiblemente superior (como mínimo, piensan, el doble) al coste que tendría si no se aplicase esta técnica. Sin embargo, son muchos los estudios que defienden una realidad bien diferente:

- En su ensayo "The Costs and Benefits of Pair Programming", Alistair Cockburn y Laurie Williams (Cockburn & Williams, 2001) concluyen que el incremento en el costo del desarrollo "es aproximadamente del 15%", subrayando además que este incremento se ve retornado por "el abaratamiento de las pruebas, las tareas de aseguramiento de la calidad y el soporte". Además, destacan los siguientes beneficios al aplicar la Programación por Parejas:
  - muchos errores son detectados mientras se desarrolla en vez de durante un periodo posterior de pruebas, durante las tareas de aseguramiento de calidad o una vez desplegado en el entorno de producción,
  - el número de defectos decrece frente a proyectos donde no se usa esta técnica,
  - surgen mejores diseños y el código generalmente resulta más corto,
  - los problemas se resuelven antes,
  - los desarrolladores aprenden más, tanto sobre el proyecto como sobre desarrollo de software,
  - los proyectos finalizan con un conocimiento mejor repartido sobre cada pieza del sistema,
  - los miembros del equipo aprenden a trabajar como un equipo,
  - el flujo de información y la dinámica del grupo mejoran, y
  - los miembros del equipo disfrutan más de su trabajo.

- En el estudio "Analyzing the Cost and Benefit of Pair Programming" de Frank Padberg y Matthias M. Müller (Padberg & Muller, 2003), se concluye a su vez que si "el time to market es decisivo para el éxito de un proyecto, la aplicación de la Programación por Parejas puede acelerar el proyecto y aumentar su valor de negocio a pesar del aumento en los costes de personal. Esto se debe al hecho de que de los programadores por parejas se puede esperar que tengan una mayor productividad y calidad del código en comparación con esos mismo programadores trabajando de manera individual".

Cuándo aplicar la Programación por Parejas

Como cualquier técnica o metodología existente, la Programación por Parejas no debe ser entendida como una bala de plata o como la piedra filosofal que convertirá en oro nuestro proyecto en cualquier situación. Como indican Kim Man Lui y Keith C.C. Chan en su estudio “Pair programming productivity: Novice–novice vs. expert–expert” (Man Lui & Chan, 2006), existen dos principios claves que debemos tener en

cuenta a la hora de plantear la aplicación de la Programación por Parejas:

1. **Una pareja es mucho más productiva** en términos de tiempo **y puede obtener una solución mejor** en términos de calidad y mantenimiento del software que dos desarrolladores trabajando de manera individual **cuando el problema planteado es nuevo** para ellos y, por tanto, su resolución requiere un mayor esfuerzo para producir su diseño, su algoritmo y/o su implementación.
2. **La Programación por Parejas sufrirá una substancial disminución en términos de productividad** en términos de tiempo, que no de calidad, **cuando la pareja tenga sobrada experiencia** en el tipo de tarea requerida.

> ***Cuanta mayor sea la experiencia que tenga la pareja en el tipo de problema a resolver, se obtendrá un menor beneficio de la aplicación de la Programación por Parejas.***

Por tanto, quizá el mejor escenario que podemos plantear sea el de fusionar la Programación por Parejas para tareas con cierto grado de novedad, para aquellas que tienen un riesgo alto, al comienzo del proyecto cuando la solución arquitectónica o el diseño es muy reciente o cuando se adopta una nueva tecnología, y la programación individual para tareas que ya se hayan convertido en habituales para nuestro equipo de desarrollo. De este modo tendremos más posibilidades de mantener un alto nivel de productividad en todas las tareas del proyecto.

Y ante la duda, quizá deba recordarse que **no hay evidencias de que la Programación por Parejas sea contraproducente** para los equipos o para los proyectos, y por el contrario sí hay, cada vez más, evidencias de la utilidad de esta técnica. Además, el uso de esta técnica ya es considerada, en muchos foros, como una buena práctica de gestión de proyectos. Quizá estas razones sean suficientes para considerar una buena idea permitir aplicar la Programación por Parejas a aquellos miembros de los equipos de desarrollo que lo soliciten, y dejar que sigan desarrollando de manera individual los que no.

Revisión por Pares

La revisión por pares (o *peer review*, en inglés) puede explicarse, en una primera aproximación, como la revisión del código realizada por otro miembro del equipo diferente al autor original del código, y cuyo objetivo principal es la búsqueda de defectos y la propuesta y evaluación de soluciones alternativas o de mejoras en el diseño y los algoritmos utilizados. Además, la revisión por pares sirve como facilitador para la difusión del conocimiento a través del equipo y, en su caso, de la organización.

Esta técnica, aunque proviene del ámbito de la publicación de trabajos científicos donde desde mediados del siglo XX forma parte integral de dicho proceso, ha calado en el ámbito del desarrollo de software, donde pueden verse referencias a ella en publicaciones tan dispares como la Guía de CMMI (Chrissis, Konrad, & Shrum, 2007), en el Área de Proceso de Verificación, o el libro La Catedral y el Bazar de Eric S. Raymond (Raymond, 1997).

Es aconsejable tener en cuenta las siguientes consideraciones básicas a la hora de implantar esta técnica en un equipo de desarrollo:

- las revisiones por pares buscan la identificación temprana de errores, por lo que se deben realizar de forma incremental según se van completando etapas en el desarrollo (mejor varias revisiones pequeñas que una única revisión al finalizar el desarrollo).
- el foco de las revisiones por pares debe ser siempre el código, no la persona que lo ha desarrollado.

A partir de estos principios, es posible construir sistemas más o menos complejos de revisión entre pares: sistemas básicos donde la revisión se realiza por un único desarrollador que toma el rol de revisor, o donde la revisión se realiza por un grupo de desarrolladores/revisores.

Tipo	Ventajas ✓	Inconvenientes ✘
Individual	Muy sencilla de implementar.	Puede producir pactos tácitos de no agresión. Puede provocar conflictos entre autor y revisor.
Grupal	Son más efectivos que los sistemas de revisión individual. Se fomenta la colectivización del código.	Puede surgir la figura de líder que reste efectividad al método. Algunos autores se sienten incómodos ante este tipo de revisión.

Técnicas de revisión

Revisón síncrona

Este es la técnica más sencilla de implementar y consiste básicamente en que, una vez finalizada una tarea de desarrollo, otro desarrollador (o un grupo de desarrolladores) revisa el código que el desarrollador original (el autor) va mostrando.

A continuación se muestra el proceso:

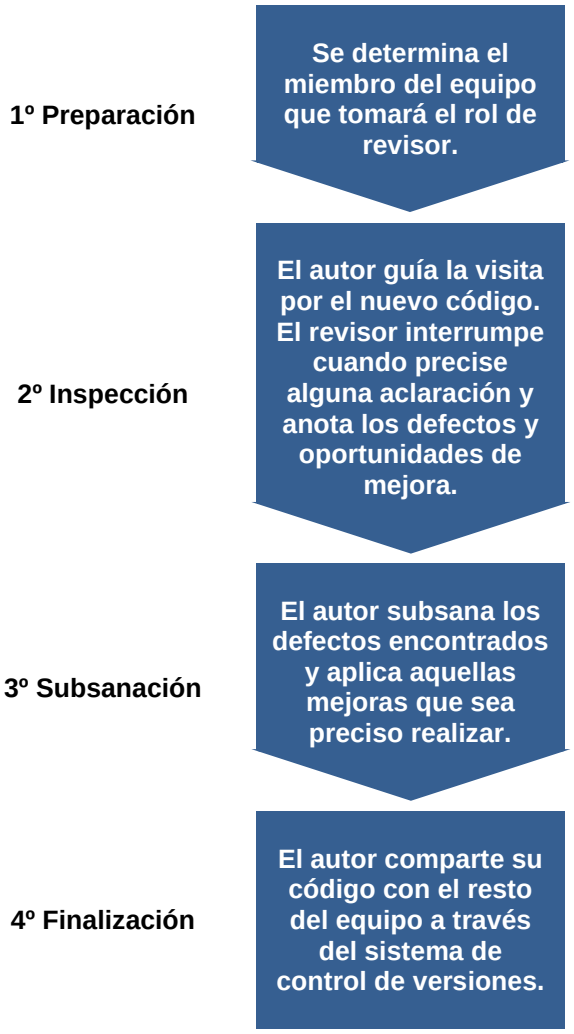


Ilustración 9 – Revisión síncrona

Habitualmente, el autor guía la revisión situándose al mando del teclado y el ratón, abriendo varios ficheros, apuntando a los cambios realizados y explicando qué es lo que se hizo. Si el/los revisor/es detecta/n algún defecto o propuesta de mejora menor, el autor podrá resolverlo sobre la marcha, pero cuando los cambios requeridos sean de mayor tamaño, será preferible su resolución una vez finalizada la revisión completa.

Revisión asíncrona

Esta técnica de revisión de código consiste en que el autor pone el código desarrollado a disposición del resto del equipo para que cada desarrollador pueda tomar el rol de revisor y enviar al autor los defectos u oportunidades de mejora encontrados.

A continuación se muestra el proceso:

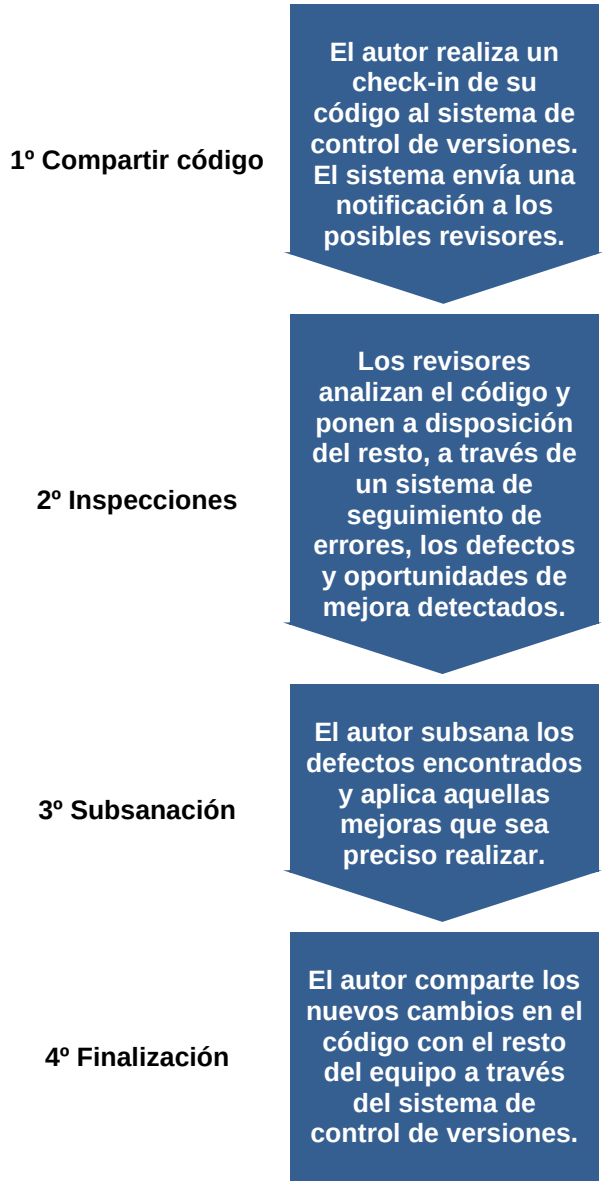


Ilustración 10 – Revisión asíncrona

La principal desventaja frente a la revisión síncrona, presentada con anterioridad, radica en la habitual falta de comunicación cara a cara. Frente a dicha desventaja, este tipo de revisiones permite la participación de todos los miembros del equipo sin que sea necesario ponerse de acuerdo en el momento y lugar en el que llevar a cabo la revisión.

Centrarse en lo importante

El valor de las revisiones por pares reside en detectar el impacto de nuevos desarrollos en el sistema. Por ello, durante las revisiones se debe mantener el foco en los aspectos realmente importantes:

- ¿se siguen los principios de la Programación Orientada a Objetos?
- ¿se utilizan correctamente las librerías y frameworks? ¿se está utilizando alguna no estándar?
- ¿hay refactorizaciones claramente necesarias que mejorarán la legibilidad del código?
- ¿se pueden prever problemas de rendimiento, memory leaks...?
- ¿se están usando correctamente las excepciones, el log, ...?
- ...

Otros aspectos superfluos, como el cuestionarse la visibilidad de un atributo o método de una clase, o indicar las deficiencias de formato del código; podrán obviarse, especialmente cuando estos puedan ser solucionados de manera automática (por ejemplo, usando un formateador de código).

Análisis de Métricas de Calidad

Las métricas de calidad del código de un proyecto podrían definirse como los valores asignados a ciertos atributos del código a partir del estudio o la observación de su naturaleza desde la perspectiva de los parámetros de calidad.

Las métricas de calidad del código ayudan en la toma de decisiones, mostrando la dirección a tomar en el camino hacia la excelencia del código, a partir de la obtención y el análisis posterior de una información que permite evaluar, de manera objetiva, el progreso.

No se recomienda, en ningún caso, la adopción de métricas de calidad por otras razones tales como ser habituales o recomendadas por determinado modelo o metodología o para poder usarlas como herramienta de poder en las negociaciones con miembros del equipo de desarrollo.

¿Qué métricas tomar?

Como la obtención y el análisis de los datos supone un sobre-esfuerzo a los equipos de desarrollo, para lograr el menor perjuicio el análisis de las métricas de calidad no debe circunscribirse únicamente a un análisis de los datos obtenidos, si no que debe comenzar desde las primeras etapas, cuando se decida qué métricas se van a tomar.

Una técnica que puede servir de ayuda para determinar las métricas a tomar en un proyecto es la conocida como GQM (*Goal, Question, Metric*). Esta técnica se basa en la definición del modelo de métrica en tres niveles diferentes:

i. **Nivel conceptual** (Objetivo – *Goal*)

   Se establece un objetivo para cada elemento que se desea medir, considerando el producto, el proceso y los recursos desde diferentes puntos de vista.

ii. **Nivel operativo** (Pregunta – *Question*)

   Tomando como base los objetivos definidos a nivel conceptual, se establecen un conjunto de preguntas que permiten caracterizar la evaluación/logro de un objetivo específico.

iii. **Nivel cuantitativo** (Medida – *Metric*)

   Se asocian un conjunto de datos a medir, a cada pregunta, que permitan dar una respuesta cuantitativa a los objetivos.

Tomando como base estos tres niveles, se establecen habitualmente los siguientes pasos para llevar a cabo GQM en un determinado proyecto:

1. Desarrollar un conjunto de objetivos desde el punto de vista de la organización y del proyecto.
2. Generar preguntas cuyas respuestas nos permitan definir dichos objetivos de manera cuantificable de la manera más completa posible.
3. Especificar las medidas a recolectar para poder dar respuesta a las cuestiones planteadas en el punto 2 y así poder realizar un seguimiento del nivel de conformidad del proyecto respecto a los objetivos.
4. Desarrollar los mecanismos para la recolección de los datos.
5. Recolectar, validar y analizar los datos “en tiempo real” para proporcionar feedback al

proyecto y que puedan tomarse medidas correctivas en caso de necesidad.
6. Analizar los datos “post mortem” para evaluar la conformidad con los objetivos y formular recomendaciones para aplicar mejoras en proyectos o iteraciones futuros.

Los tres primeros pasos de este proceso buscan definir las métricas a tomar en el proyecto, mientras que los últimos tres tratan sobre la toma de datos en sí, y los análisis posteriores de los mismos.

Clasificación de las Métricas

Existen diferentes clasificaciones para las Métricas de Calidad de un producto Software. Una de estas clasificaciones, que es la que veremos a continuación, es la que siguen las normas ISO/IEC 9126 (ISO, 2001) e ISO 25000 (ISO, 2005):

- **Métricas Internas**
- **Métricas Externas**
- **Métricas en Uso**

En los siguientes epígrafes se describirá cada uno de estos tipos de métricas en detalle.

Métricas Internas

Las métricas internas miden las propiedades intrínsecas del software como el tamaño, la complejidad o la conformidad con los Principios de la Orientación a Objetos.

Los datos de estas métricas se obtendrán del propio código y permitirán medir la calidad de los entregables intermedios y estimar la calidad final del producto en base a los resultados parciales obtenidos. De este modo, cuando los resultados esperados estén por debajo de unos umbrales pre-establecidos, incluso desde etapas tempranas el equipo podrá iniciar acciones correctivas que permitan elevar el nivel de calidad del software desarrollado.

En el mercado existen gran cantidad de herramientas que facilitan la recolección de las métricas internas del código. CodePro AnalytiX, FindBugs, PMD, CheckStyle o los plugins Eclipse Metrics Plugin y X-Ray Plugin para Eclipse son algunos ejemplos.

Algunos ejemplos de métricas internas del código serían:

- Ratio clases por paquetes/módulos
- Número de constructores por clase
- Número de propiedades por clase
- Número de métodos por clase (normalmente, excluyendo setters y getters)
- Número de parámetros por método
- Líneas de código por clase
- Líneas de código por método
- Profundidad de los bloques por método
- Complejidad ciclomática por método
- Acoplamiento aferente
- Acoplamiento eferente
- Nivel de abstracción
- Profundidad de la jerarquía de herencia
- Número de subclases
- Número de clases comentadas
- Número de métodos comentados

Métricas Externas

Las métricas externas miden el comportamiento del software en producción y estudia diferentes atributos como el rendimiento en una máquina determinada, el uso de la memoria o el tiempo de funcionamiento entre fallos.

Esto implica que, para poder comenzar a tomar estas métricas, es necesario disponer de una versión ejecutable del software que, una vez instalado en un equipo con unas características definidas permitan estimar cual será el comportamiento en el entorno de producción real.

Una práctica habitual para extraer métricas externas es someter el ejecutable a pruebas automatizadas de carga que ejecuten una serie de scripts de acciones que simulen un número determinado de usuarios. Estas pruebas se podrán diseñar con un número de usuarios simulados creciente en orden de magnitud, de modo que sea posible estimar el grado de degradación del proyecto en función de la carga de trabajo a la que se vea sometido.

Como ejemplos de métricas externas se pueden tomar:

- Tiempos de respuesta
- Tasa de errores
- Uso de memoria
- Uso del procesador
- Número de accesos al sistema de ficheros
- Uso de la Red
- Conexiones a BBDD

Métricas en Uso

Las métricas en uso miden la productividad y efectividad del usuario final al utilizar el Software.

Es por tanto indispensable, para poder recoger estas métricas, que los usuarios finales (o un subconjunto representativo de los mismos)

utilicen el software en una serie de escenarios de uso que permitan probar el software desde el punto de vista de las diferentes tareas operativas asociadas al mismo -instalación y configuración, back-ups... - como desde las distintas funcionalidades, priorizando aquellas relacionados con las tareas más comunes para cada tipo de usuario.

Algunas de las métricas que puede ser interesante recoger son:

- % de tareas completadas.
- % de tareas completadas en el primer intento.
- % de usuarios que completan todas las tareas.
- Nº de veces que se acude a la ayuda.
- Tiempo empleado en cada tarea.
- Tiempo empleado en recuperarse de los errores.
- Número de acciones requeridas para completar cada tarea.

Además, puede ser interesante pedir a los usuarios que completen un cuestionario que recoja información subjetiva respecto a diversos factores relacionados con la usabilidad:

- ¿Le ha parecido fácil instalar el software?
- ¿Le ha parecido fácil trabajar con el software?
- ¿Le han parecido adecuadas las ayudas?
- ¿Considera que el software se adapta a sus necesidades (bueno), o que él tiene que adaptarse a las restricciones del software (malo)?
- ¿Cómo se siente al terminar la tarea (bajo tensión, satisfecho, molesto, ...)?
- ¿Qué calificación le otorga respecto a versiones anteriores o respecto a software de la competencia?

El análisis de estos tres tipos de métricas en conjunto será el que permita descubrir el mayor número posible de defectos y oportunidades de mejora.

El lado oscuro de las métricas

Por el nombre de “Metric abuse” se conoce a un anti-patrón de la gestión de proyectos que suele darse en aquellas organizaciones que implementan, por primera vez, un proceso de análisis de métricas de calidad:

- en ocasiones se realiza un auténtico sobre-esfuerzo para tratar de medir absolutamente todo en los proyectos, llegando al extremo de imposibilitar la extracción de información relevante de los datos recogidos y, por tanto, se imposibilita de igual modo la ejecución de aquellas acciones que permitan alcanzar mejoras tangibles, debido al exceso de información con el que se encuentran.
- en otros casos, los datos obtenidos se utilizan con fines destructivos (por ejemplo: para atacar a los miembros del equipo de desarrollo) en vez de usarlos para los fines constructivos para los que deberían haberse tomado: mejorar el desarrollo hasta conseguir la excelencia del código.

El resultado de la aparición de este anti-patrón en el proyecto tiene como consecuencias más reseñables:

- que la prioridad del equipo se convierta en obtener buenos resultados en las métricas en detrimento del desarrollo del código que permita proveer al cliente de un producto acorde a sus necesidades,
- que surja el miedo en el equipo de desarrollo como consecuencia del uso destructivo de los resultados del análisis de las métricas, y
- que se dificulte el seguimiento del proyecto por el aumento de la incertidumbre respecto a su estado real.

Resumen

- Las revisiones de código son técnicas que permiten localizar porciones de código que pueden/deben mejorarse.

- Las revisiones de código se pueden clasificar teniendo en cuenta cuando se realizan:
  - en paralelo al desarrollo (programación por parejas), o
  - tras el desarrollo (revisión por pares, análisis de métricas de calidad del código).

- En la programación por parejas dos desarrolladores trabajan en un único puesto intercambiadno con frecuencia los roles de conductor y observador/navegante.

- No se debe confundir la programación por parejas con el mentoring, otra técnica de trabajo en pareja cuya meta es contribuir a mejorar el talento de uno de los miembros gracias a la experiencia del otro.

- El beneficio de aplicar la programación por pareja es inversamente proporcional a la experiencia en el dominio del problema.

- La revisión por pares (*peer review*) es la revisión del código, a posteriori, realizada por otro miembro del equipo diferente al autor.

- La revisión por pares se puede clasificar en:
  - Individual o grupal, y
  - Síncrona o asíncrona

- A partir de la obtención y el análisis de las métricas de calidad del código es posible evaluar objetivamente el progreso hacia la excelencia del código.

- Medir es costoso. Se deben escoger con cuidado las métricas a tomar. GQM (*Goal, Question, Metric*) es una técnica que facilita esta elección.

- Las métricas de calidad del código se clasifican en:
  - Métricas Internas,
  - Métricas Externas, y
  - Métricas en Uso.

- Se debe evitar caer en el antipatón “metric abuse” que se plasma:
  - en la dificultad para obtener información y poder utilizarla debido a un número excesivo de métricas,
  - en el uso con fines destructivos de los resultados del análisis.

Pruebas

Introducción

Probar el software no es, ni de lejos, una actividad sencilla. Y más aún cuando se tiene en consideración las palabras de Edsger Dijkstra, al que posiblemente conocerás por el "Algoritmo de Dijkstra" (solución al problema del camino más corto entre dos nodos de un grafo), la "Notación Polaca Inversa" o por cualquier otra de sus geniales contribuciones al mundo de la computación; quien ya proclamaba, en el año 1969:

> ***"Las pruebas del software son capaces de mostrar la presencia de errores, pero no su ausencia."***
>
> *("Testing shows the presence, not the absence of bugs").*

(Cunningham, 1992)

A pesar de lo dura que resulta esta afirmación, especialmente para los Ingenieros de Pruebas o los miembros del equipo que dedican su esfuerzo a las tareas de testing en un proyecto, las pruebas del software son un elemento crítico en la garantía de la calidad del software desarrollado. Es preciso, por tanto, asegurar en nuestros proyectos el diseño de aquellas pruebas que permitan detectar el mayor número de posibles errores cometidos empleando para ello la menor cantidad de tiempo y esfuerzo.

Además, frente a la opción de considerarlas como una etapa diferenciada y cercana al final el ciclo de vida del proyecto, es preferible que las pruebas sean ejecutadas de manera paralela al resto de actividades del proyecto. Se deben probar la funcionalidad, la estabilidad y el rendimiento desde los primeros prototipos o versiones iniciales desarrollados ya que, cuanto antes comiencen las pruebas y antes se detecten errores, menor esfuerzo costará solucionarlos.

Preparar el terreno

Para asegurar la fiabilidad de los resultados de las Pruebas es conveniente establecer una serie de entornos controlados. Además del entorno de desarrollo local, el entorno que cada miembro del equipo de desarrollo dispone en su puesto para realizar su trabajo, lo habitual, por su utilidad, es disponer al menos de los siguientes entornos:

- **Entorno de Integración**

  En este entorno, muy cambiante, generalmente se prueban tanto los resultados de la integración del código desarrollado por los diferentes miembros del equipo interactuando entre sí, como aquellas propuestas de solución resultado de un proceso de investigación[2] o de haber iniciado el desarrollo de una Prueba de Concepto en la forma de un spike[3] o de una bala trazadora[4].

- **Entorno de Pre-producción**

  Debe ser un entorno equivalente al de producción (a ser posible una copia lo más fidedigna posible) de modo que sea posible probar, en un entorno controlado, los posibles bugs detectados por los usuarios sin que dichas pruebas afecten a los datos reales de producción.

  Este entorno, además, se usa habitualmente para realizar la batería completa de pruebas previa al paso a producción de nuevas versiones.

- **Entorno de Producción**

  Este entorno, también conocido como entorno de explotación, es el entorno real donde los usuarios interactúan con el sistema.

En función de la naturaleza del proyecto podrá resultar interesante establecer otros entornos.

Clasificación de las Pruebas

Hay multitud de opciones a la hora de clasificar las pruebas. Una de las más comunes es diferenciar las pruebas teniendo en cuenta el conocimiento que se tiene del sistema a probar:

- **Pruebas de caja negra**

  Las pruebas de caja negra demuestran que cada función es completamente operativa desde el punto de vista de las entradas que

[2] Técnica que examina un gran número de alternativas permitiendo obtener conocimiento de base que habilite al equipo a realizar la estimación, o al menos que les ayude a determinar sobre qué se debe llevar a cabo un spike.

[3] Un desarrollo rápido y 'a sucio' realizado para obtener conocimiento sobre alguna herramienta o técnica de desarrollo y que será desechado una vez obtenido dicho objetivo.

[4] Una implementación light, pero desarrollada siguiendo los estándares de calidad del equipo, que facilita la introducción de una pequeña parte de una epopeya en una iteración, disminuyendo la incertidumbre facilitando su integración con otros módulos del software previa al momento en que se aborde su implementación completa.

recibe y las salidas o respuestas que produce, sin tener en cuenta su funcionamiento interno.

- **Pruebas de caja blanca**

  Las pruebas de caja blanca aseguran que la operación interna se ajusta a las especificaciones y que todos los componentes internos se han comprobado de forma adecuada.

  Para realizar estas pruebas se suelen aplicar técnicas específicas que, por ejemplo, recorran todos los posibles caminos de ejecución (cobertura), que prueben las expresiones lógico-aritméticas del código o que comprueben el correcto comportamiento de los diferentes bucles para valores extremos de sus contadores como 0,1, máx.-1, máx. y máx.+1 iteraciones.

Otra posible clasificación, que será la que se utilice en el desarrollo del presente capítulo, es aquella que tiene en cuenta los objetivos específicos de cada prueba. Aplicando este criterio, se puede hablar de:

Pruebas	Objetivo
Unitarias	verificar la funcionalidad y estructura de cada **componente individual**
de Integración	verificar el **correcto ensamblaje** de los distintos componentes
de Sistema	verificar las especificaciones funcionales y técnicas del **sistema en su conjunto**
de Implantación	conseguir la **aceptación del sistema** por parte del **usuario de operación**
de Aceptación	conseguir **la aceptación final** por parte del **Cliente**
de Regresión	verificar que los cambios en un componente **no causan daños colaterales** en otros componentes no modificados

Pruebas Unitarias

Las Pruebas Unitarias son aquellas que permiten comprobar el funcionamiento de cada componente del Sistema de Información por separado, lo que aplicado al Paradigma de la Orientación a Objetos es equivalente a hablar de pruebas que permiten comprobar el funcionamiento correcto de una Clase de modo que, para cada uno de sus métodos, o al menos **para cada método no trivial**[5], se pueden ejecutar casos de prueba independientes.

Algunas de las ventajas del uso de las pruebas unitarias son:

- **dan soporte a la refactorización y a la implementación de mejoras en el código**

  Si se dispone de una batería de pruebas unitarias completa, y tras efectuar modificaciones en el código se pasan las pruebas unitarias sin errores, aumentará nuestro nivel de confianza respecto a los cambios implementados ya que podremos asegurar que el resto del código no se habrá visto afectado negativamente por dichas modificaciones.

- **simplifican la integración del proyecto**

  Los errores, en caso de aparecer durante dicha integración entre los componentes del Sistema de Información, estarán más acotados y, por tanto, será más fácil solucionarlos.

- **sirven como documentación sobre el uso del código implementado**

  Las mismas pruebas unitarias sirven como ejemplo del uso correcto del mismo.

Cada vez se ven más proyectos en los que se hace uso de las pruebas unitarias para verificar el código desarrollado. Sin embargo, no todas las pruebas unitarias que se implementan se consideradas correctas desde un punto de vista académico. Las buenas pruebas unitarias se basan en una serie de características, entre las que se pueden destacar las siguientes:

- **Independencia**

  La ejecución de una prueba unitaria nunca podrá depender de las ejecuciones previas de otras pruebas.

- **Automatización**

  Debe ser posible integrar las pruebas unitarias en un sistema de integración frecuente.

[5] En ocasiones en nuestros diseños incorporamos métodos que no aportan lógica del negocio, es decir, *métodos triviales*. Un ejemplo de este tipo de métodos son los métodos setX y getX que, en la mayoría de los casos, simplemente se utilizan para establecer el estado de una instancia de una determinada Clase, o lo que es lo mismo, permiten establecer los valores de sus propiedades.

- **Completitud**

  Los Casos de Prueba deben comprobar el mayor número de casos posibles (por ejemplo, para verificar el código de un método con parámetros, los casos de prueba deben comprobar el uso de valores habituales, valores extremos, valores nulos...).

Ejemplo

Dada una clase Matematicas como la siguiente:

```
public class Matematicas {
    public int suma(int a, int b) {
        return a + b;
    }
}
```

Se podría desarrollar la siguiente prueba unitaria:

```
import junit.framework.*;

public class TestMatematicas extends
TestCase {
    public void testAdd() {
        int num1 = 3;
        int num2 = 2;
        int total = 5;
        assertEquals(Matematicas
          .suma(num1, num2), total);
    }
}
```

Esta sencilla prueba permite comprobar que al realizar una llamada a la función suma de la Clase Matematicas, el valor de retorno es el esperado.

Pruebas Unitarias y Desarrollo Dirigido por Pruebas (TDD)

Aunque las pruebas unitarias están íntimamente relacionadas con el Desarrollo Dirigido por Tests (TDD - Test Driven Development), por ser este tipo de pruebas la técnica en la que se sustenta la aplicación de dicha metodología, el uso de pruebas unitarias en un proyecto no implica la aplicación de dicha metodología en el mismo.

Para ilustrar la diferencia entre el uso de esta técnica y la aplicación de TDD, a continuación se presenta de modo resumido el proceso[6] en el que se basa esta metodología:

1. **Escribir la prueba**

   La prueba deberá considerarse más como un ejemplo o especificación que como una prueba propiamente dicha, puesto que al ser esta la primera actividad de la metodología, aún no estará disponible el código a probar.

   No es necesario diseñar todos los test antes de comenzar el desarrollo, sino que se deben escribir uno a uno siguiendo de forma iterativa los pasos del algoritmo TDD.

2. **Implementar el código**

   A la hora de implementar el código debe mantenerse la máxima de no implementar nada más que lo estrictamente necesario para cumplir la especificación actual, es decir, para que la prueba escrita en el paso anterior se ejecute sin errores.

3. **Refactorizar**

   Una vez se logra la ejecución sin errores de la prueba, se deben rastrear fragmentos duplicados en el código así como revisar aquellos módulos o clases que no cumplan con los Principios Básicos del Diseño Orientado a Objetos. Una vez detectados los aspectos a mejorar, se deberán aplicar aquellas técnicas de refactorización necesarias para solucionarlos.

En contraposición a este proceso, en un proyecto que no siga esta metodología las pruebas unitarias no tienen por qué escribirse antes que el código a probar, y por tanto, como ya se había avanzado en párrafos anteriores, esto implica que:

El hecho de que se apliquen pruebas unitarias en un proyecto no implicará que se esté aplicando la metodología TDD.

Pruebas de Integración

Las Pruebas de Integración son aquellas que permiten probar en conjunto distintos subsistemas funcionales o componentes del proyecto para verificar que interactúan de manera correcta y que se ajustan a los requisitos especificados (sean estos funcionales o no).

Este tipo de pruebas deberán ejecutarse una vez se haya asegurado el funcionamiento correcto de

[6] Habitualmente se conoce a este proceso como el "ciclo *red, green, refactor*", donde red (rojo) hace referencia al fallo del test unitario y green (verde) a la ejecución correcta de dicha prueba.

cada componente implicado por separado, es decir, una vez se hayan ejecutado sin errores las pruebas unitarias de estos componentes.

La aplicación de este tipo de pruebas en un proyecto proporciona una serie de ventajas, especialmente relacionadas con la prevención de la aparición de errores ocasionados por:

- un mal diseño de los interfaces de los componentes
- un mal uso de los componentes

Además, al igual que las pruebas unitarias, las pruebas de integración sirven como documentación del uso de los componentes puesto que en definitiva, de nuevo podrán considerarse como ejemplos de su uso.

Artefactos específicos para pruebas

Aunque desde un punto de vista puramente académico resulta sencillo distinguir las pruebas unitarias de las pruebas de integración, en los proyectos reales la frontera entre estos tipos de pruebas resulta, en muchos casos, difusa, ya que son pocas las ocasiones en que se puede probar una Clase o Sujeto en Pruebas (SUT – *Subject Under Test*) de forma totalmente aislada. En la mayoría de los casos, el SUT debe recibir por parámetros instancias de otras Clases, debe obtener información de configuración de un fichero o de una base de datos, o incluso su lógica puede depender de otros componentes del Sistema de Información.

Si se vuelve sobre la definición de las pruebas unitarias, donde se decía que *las Pruebas Unitarias son aquellas que permiten* ***comprobar el funcionamiento de cada componente del Sistema de Información por separado***, se podría pensar que aquellos componentes que dependen de otros para su funcionamiento no podrán probarse de manera unitaria. Sin embargo, para evitar estos problemas se ha diseñado un abanico de artefactos específicos para pruebas que permiten desarrollar y ejecutar pruebas unitarias en estos casos haciendo creer al SUT que está trabajando en un entorno real:

- **Objetos ficticios**

  Son objetos enviados como parámetros a los métodos del SUT, pero que en realidad nunca se usan. Normalmente este tipo de objetos ficticios sólo se usan para completar la lista de parámetros del método que se está probando.

- **Objetos falsos**

  Son objetos que tienen implementaciones totalmente funcionales, pero que incorporan algún tipo de atajo que los invalida para su uso en producción (por ejemplo, hacen uso de una base de datos en memoria).

- **Objetos Stub**

  También conocidos como objetos auxiliares, son objetos que proporcionan respuestas predefinidas a las llamadas recibidas durante la prueba, y que generalmente no proporcionarán ningún tipo de respuesta a nada que no se haya programado en la misma, aunque puedan almacenar información relacionada con las llamadas recibidas para su posterior análisis por parte de los Ingenieros de Pruebas.

- **Objetos Mock**

  Son objetos que están pre-programados con expectativas que, en conjunto, forman una especificación de las llamadas que se espera recibir. A diferencia del resto de artefactos específicos para pruebas, basados en la verificación del estado de los objetos, los Objetos Mock se basan en la verificación de su comportamiento.

En definitiva, y formalmente hablando:

> ***Si es preciso realizar pruebas unitarias de Clases que dependen de otras, tendremos que usar algún artefacto específico para pruebas, ya que en otro caso se estarán aplicando pruebas de integración.***

Ejemplo

A partir de las siguientes clases e interfaces:

```
public class Persona {
    private int id;
    private String nombre;
    public Persona(int id,
      String nombre) {
        this.id = id;
        this.nombre = nombre;
    }
    …
}
```

```
public class LogicaPersona {
    private PersonaDAO dao;

    public Persona(
      PersonaDAO dao) {
        dao = dao;
    }


public Persona buscaPersona(int id) {
        ...
       return dao.busca(id);
    }
    ...
}

interface PersonaDAO {
    Persona  busca(int id);
    ...
}
```

Se puede desarrollar el siguiente test para realizar una prueba unitaria de la clase LogicaPersona:

```
import junit.framework.*;

public class LogicaPersonaTest
  extends TestCase {
    public void testBuscaPersona()
      throws Exception {
        //TODO: inicializar DAO
        PersonaDAO dao;

        LogicaPersona logica =
          new LogicaPersona(dao);

        assertNotNull(
          logica.buscaPersona(1));
    }
}
```

Tal y como indica el comentario (*TODO*), es necesario inicializar la variable dao con un objeto de una Clase que, implementando el Interface PersonaDAO, se pueda pasar como parámetro al constructor de LogicaPersona.

Para poder ejecutar el test se puede crear la siguiente Clase auxiliar:

```
public class PersonaDAOStub
  implements PersonaDAO {
    public Persona  busca (int id) {
        return new Persona(1, "ana");
    }
    ...
}
```

Esta clase auxiliar, con el método "busca" que **proporciona una respuesta predefinida a la llamada recibida durante la prueba** sea cual sea el valor que reciba por parámetro, es un claro ejemplo de *Stub*.

Pruebas de Sistema

Se conoce con el nombre de pruebas de sistema a aquellas pruebas que toman el Sistema de Información al completo y lo prueban tanto en su conjunto como en sus relaciones con otros sistemas con los que se comunique.

Estas pruebas, que pueden verse como un subconjunto de las pruebas de integración, deben ejecutarse una vez se haya finalizado el desarrollo de la versión entregable del sistema, y una vez se hayan logrado ejecutar, sin errores, las diferentes pruebas unitarias y de integración del proyecto.

Las pruebas de sistema permiten verificar que tanto las especificaciones funcionales como las técnicas se cumplen para el entregable. Además, si el entorno en el que se realizan estas pruebas es equivalente al de producción, permiten obtener una visión sobre su comportamiento que podrá extrapolarse a dicho entorno.

El conjunto de pruebas de sistema está formado por un amplio abanico de pruebas, entre las que destacan:

- **Pruebas Funcionales**

  Permiten determinar si el Sistema de Información hace lo que se requiere de él.

- **Pruebas de Comunicación**

  Buscan detectar posibles problemas de comunicación de la aplicación con otros Sistemas de Información.

- **Pruebas de Carga**

  Permiten conocer el comportamiento del Sistema de Información cuando trabaja con grandes volúmenes tanto de usuarios como de datos.

- **Pruebas de Recuperación**

  Permiten determinar si la aplicación es capaz de recuperarse de un fallo sin comprometer la integridad de los datos.

- **Pruebas de Accesibilidad**

  Comprueban si es posible acceder al Sistema de Información desde los entornos para los que ha sido diseñado.

- **Pruebas de Usabilidad**

  Ayudan a determinar el grado de facilidad y comodidad de los Interfaces de Usuario de la aplicación para sus usuarios objetivo.

- **Pruebas de Seguridad**

  Comprueban si los mecanismos de control de acceso al Sistema de Información evitan alteraciones indebidas en los datos.

- **Pruebas de Operación**

  Prueban la posibilidad de realizar las diferentes operaciones de mantenimiento (arranques, paradas, copias de seguridad, ...) sin comprometer el nivel de servicio esperado para la aplicación.

Será responsabilidad del equipo establecer el conjunto de pruebas que debe pasar el Sistema de Información antes de su entrega al cliente.

Pruebas de Implantación

La ejecución de las pruebas de sistema permite comprobar que el Sistema de Información desarrollado, en su conjunto, responde a sus requisitos. Sin embargo, estas pruebas se habrán ejecutado en un entorno controlado (hablando en términos deportivos, se podría decir que se ha "jugado en casa").

Las pruebas de implantación, sin embargo, serán aquellas que, una vez implantado el Sistema de Información en el entorno real de explotación del Cliente (o en un entorno casi real, comúnmente conocido como entorno de pre-explotación) permitirán al equipo de Sistemas/Operaciones del propio Cliente comprobar el funcionamiento sin errores coexistiendo con el resto de los sistemas existentes y que conforman el ecosistema del nuevo Sistema de Información. Además, se deben volver a repetir, siempre que sea posible, aquellas pruebas que verifiquen los requisitos de rendimiento, seguridad, operación...

Como resultado de la ejecución de estas Pruebas de Implantación se obtiene la aceptación de la instalación del sistema en el entorno del Cliente, desde un punto de vista operativo, por parte de su Responsable de Sistemas/Operaciones.

Pruebas de Aceptación

Una vez implantado el Sistema de Información en el entorno real (o de pre-explotación) del Cliente, y tras recibir la aceptación del Responsable de Operación si se han realizado las pruebas de implantación correspondientes, sería el momento de llevar a cabo las denominadas pruebas de aceptación, unas pruebas que tienen como objetivo obtener la validación por parte del Cliente, desde el punto de vista de la funcionalidad y del rendimiento, de que el Sistema cumple con sus requisitos y expectativas.

Para evitar incómodas sorpresas durante la realización de estas pruebas, es de vital importancia que desde las primeras etapas del proyecto se haya alcanzado un acuerdo con el Cliente sobre los criterios de aceptación del Sistema de Información, y que estos criterios se mantengan siempre presentes durante el desarrollo del proyecto.

Para establecer estos Criterios de Aceptación y, por tanto, para ejecutar la batería de pruebas correspondiente, se deben tener en cuenta diferentes aspectos como la criticidad del sistema, el número de usuarios esperados y el tiempo que se vaya a disponer para llevar a cabo estas pruebas.

Ejemplo

Si partimos de una Historia de Usuario como la siguiente:

"Como Usuario quiero poder acceder al Sistema de Información para poder usar sus funcionalidades"

Se podrían establecer los siguientes Criteros de Aceptación:

1. El Usuario introduce un par [Nombre de Usuario, Contraseña] que no es válido → El Sistema de Información deniega el acceso y muestra el mensaje de error: "El Nombre de Usuario o la Contraseña no son válidos".
2. El Usuario introduce un par [Nombre de Usuario, Contraseña] válido → El Sistema de Información concede el acceso y muestra un menú de opciones que da acceso a las distintas funcionalidades del mismo.

Otra opción de Criterios de Aceptación para esta misma Historia de Usuario, en el caso de preferir una redacción en forma de script, podría ser:

1. Se dará de alta un Usuario con los siguientes datos:
   - Nombre de Usuario: usuario
   - Contraseña: 12345678

2. Se probará el acceso con los siguientes pares [USR, PWD], obteniendo las [Respuestas] del Sistema de Información que se indican a continuación:

USR	PWD	Respuesta
USUARIO	12345678	**Acceso denegado.** Se muestra el mensaje de error: "*El Nombre de Usuario o la Contraseña no son válidos*".
uSuArIo	12345678	
usuario	87654321	
usuario	123456789	
usuario	123456789	
	12345678	
usuarios	12345678	
Usuario	12345678	**Acceso concedido.** Se muestra un menú de opciones que da acceso a las distintas funcionalidades del Sistema de Información.

Pruebas de Regresión

Bajo el nombre de pruebas de regresión se engloban todas aquellas pruebas cuyo objetivo es el de eliminar los temidos daños colaterales causados por los cambios realizados en el código, tanto en los casos en que estos cambios hayan sido efectuados para añadir nuevas funcionalidades como en aquellos en que lo han sido para reparar algún error (mantenimiento correctivo) del Sistema de Información.

Estos daños colaterales causados por las modificaciones en el código pueden incluir desde comportamientos no deseados en alguna funcionalidad del sistema a todo tipo de errores tanto en aquellos componentes del Sistema de Información que han sido modificados como en aquellos que no sufrieron cambios.

Habitualmente, estas pruebas de regresión suelen implicar la repetición de la batería completa de pruebas realizadas a lo largo de toda la vida del proyecto, es decir, la repetición de las pruebas unitarias, de integración, de sistema... Este hecho pone de manifiesto la necesidad/conveniencia de la automatización de todas aquellas pruebas que sea posible.

Esta automatización, además de lograr una disminución del esfuerzo requerido para poder realizar las pruebas cada vez que se realizan modificaciones en el código, vendrá acompañada de una notable disminución del número de errores causados por daños colaterales tanto durante el desarrollo de los distintos componentes del Sistema como a lo largo de su mantenimiento.

Resumen

- "*Las pruebas del software son capaces de mostrar la presencia de errores, pero no su ausencia*" (Dijkstra).

- Para asegurar los resultados de las pruebas es conveniente establecer al menos los siguientes entornos:
  - Entorno de Integración
  - Entorno de Pre-producción
  - Entorno de Producción

- Las pruebas unitarias permiten verificar la funcionalidad y estructura de cada componente individualmente. Deben ser:
  - Independientes
  - Automatizables
  - Completas

- Usar pruebas unitarias no es sinónimo de aplicar TDD. TDD se basa en el "ciclo *red, green, refactor*".

- Las pruebas de integración permiten verificar el correcto ensamblaje de los distintos componentes.

- En ocasiones la única diferencia entre pruebas de integración y unitarias es que estas últimas precisan de artefactos específicos:
  - Objetos ficticios
  - Objetos falsos
  - Stubs
  - Mocks

- Las pruebas de sistema permiten verificar las especificaciones funcionales y técnicas del sistema en su conjunto.

- Las pruebas de implantación permiten que el usuario de operación acepte el sistema al

realizarse la implantación en el entorno de producción o en el de preproducción.

- Las pruebas de aceptación permiten conseguir la aceptación final por parte del Cliente.

- Las pruebas de regresión permiten verificar que los cambios en un componente no causan daños colaterales en otros componentes no modificados. Generalmente se repiten todas las pruebas, por lo que es importante que estas estén automatizadas.

Refactorización del Código

Introducción

La refactorización es una técnica que permite reestructurar, de manera disciplinada, el código existente de un componente o de un Sistema de Información. Su uso permite modificar la estructura interna sin modificar el interfaz a partir de un conjunto de pequeñas transformaciones que preservan el comportamiento.

Aunque cada transformación, conocida a su vez como refactorización, realiza únicamente un mínimo cambio sobre el código (volviendo a remarcar que cada uno de estos pequeños cambios no modifica el comportamiento del código), esta técnica alcanza toda su potencia cuando se aplican en secuencia todo un conjunto de refactorizaciones. De esta forma, es posible obtener importantes beneficios gracias a una profunda reestructuración del código.

El hecho de que cada transformación sea pequeña reduce el riesgo de problemas debidos a este proceso. De hecho, tras la aplicación de cada refactorización, el Sistema de Información deberá seguir manteniéndose completamente funcional.

En definitiva, la refactorización puede entenderse como un **proceso de mantenimiento de un Sistema de Información cuyo objetivo no es ni arreglar errores ni añadir nueva funcionalidad, si no mejorar la comprensión del código** a través de su restructuración aplicando, por ejemplo, los Principios de la Orientación a Objetos, algún Patrón de Diseño o simplemente cambiando el algoritmo utilizado o eliminando código muerto, para facilitar así futuros desarrollos, la resolución de errores o la adición de nuevas funcionalidades.

Refactorización y Pruebas

Una de las claves para que la refactorización tenga éxito es que el código desarrollado, y que va a ser objeto de la reestructuración, **tenga asociadas sus correspondientes pruebas**. Además, otro factor clave de este éxito será la **automatización de dichas pruebas**, ya que dicha automatización permitirá comprobar, tras cada aplicación de esta técnica, que el Sistema de Información sigue cumpliendo con los requisitos que implementaba sin añadir nuevos errores, sin que esas comprobaciones repercutan de manera negativa en el ritmo del equipo de desarrollo.

¿Cuándo se aplica?

Existe una gran variedad de estrategias a la hora de aplicar la refactorización. En función de las características intrínsecas y del contexto del proyecto en el que se vaya a aplicar esta técnica habrá que decidir entre una u otra:

- **Cuando se vayan a introducir nuevas funcionalidades** en un determinado módulo, paquete o clase.
- **De manera continua**. Se alterna el desarrollo de funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad, como en el caso del Desarrollo Dirigido por Pruebas a través del proceso red/green/refactor.
- **Bajo demanda**. El uso de la refactorización viene marcado por los resultados de las revisiones del código o por el uso de un diario de deuda técnica.

La deuda técnica

El término deuda técnica surge de una analogía utilizada por Ward Cunningham en 1992 en la que realiza la equivalencia entre un desarrollo de software que presentase determinadas carencias debido, generalmente, a la necesidad de realizar una entrega temprana, con la petición de un crédito financiero.

Esta analogía ha sido ampliamente extendida ya que de igual forma que dicho crédito permitiría obtener liquidez a corto plazo a costa de generar unos intereses a liquidar durante un período de tiempo prolongado, las carencias introducidas durante el desarrollo del software proporcionan un beneficio a corto plazo (la entrega a tiempo de una versión del software) pero tarde o temprano requerirán gastar el tiempo inicialmente ahorrado sumado a un esfuerzo extra (los intereses), es decir, dichas carencias generarán una deuda técnica con los mismos aspectos negativos que los intereses del crédito financiero solicitado.

Y aunque las deudas técnicas se pueden clasificar en advertidas o inadvertidas, y en prudentes o imprudentes, atendiendo al reconocimiento o no de la deuda en el momento en que esta se da y a la realización o no de un análisis de costo frente a beneficios, y a pesar de que cada equipo tendrá que atender sus propias carencias durante el desarrollo de sus proyectos de software, a continuación se listan algunos ejemplos típicos de deudas técnicas:

- Errores conocidos y no solucionados.
- Mejoras en el código no implementadas.

- Insuficientes pruebas o pruebas de mala calidad.
- Documentación desactualizada, incompleta o inexistente.

El diario de deuda técnica

Un diario de deuda técnica es una herramienta que permite conocer la deuda técnica advertida, acumulada en cualquier momento del proyecto.

Una opción muy extendida para llevar este tipo de diarios es utilizar las listas de tareas que habitualmente incorporan los entornos de desarrollo, las cuales escanean automáticamente el código en busca de comentarios con etiquetas en un formato particular y que, a partir de los hallazgos encontrados, generan un lista. Habitualmente, los entornos de desarrollo suelen aceptar las siguientes etiquetas:

- **FIXME**: permite marcar una porción de código problemático (se sabe que en determinadas circunstancias falla o puede fallar) y que por tanto requiere una especial atención y/o revisión posterior.
- **NOTE**: permite indicar el funcionamiento interno de código o posibles dificultades que no están documentadas.
- **TODO**: indica posibles mejoras a realizar en un futuro más o menos próximo.
- **XXX**, **KLUDGE**, **KLUGE** o **HACK**: cualquiera de ellos advierte de código ineficiente, poco elegante o incluso insondable, pero que, sin embargo, funciona.

Para evitar el riesgo de que estas anotaciones se acumulen con el tiempo (o lo que es lo mismo, de que la deuda técnica se acumule) puede resultar útil incluir la fecha al comentario asociado a la etiqueta para facilitar su seguimiento. Además, también puede resultar interesante anotar el número de veces que esa porción de código ha provocado algún tipo de problema.

Con toda esta información será posible determinar, en cada momento, la cantidad de deuda técnica que acumula un proyecto para determinar, así mismo, cuando ha llegado el momento de detenerse a resolver cada uno de los problemas de los que se compone.

Refactorizaciones básicas

En la actualidad, los entornos de desarrollo suelen ofrecer herramientas para aplicar algunos métodos de refactorización de manera automática, actualizando correctamente las dependencias (siempre que sea posible) o marcando los elementos afectados, simplificando en gran medida este proceso.

Entre el conjunto de las refactorizaciones básicas ofrecidas por estos entornos es habitual encontrarse, al menos, las siguientes:

- **Renombrar una clase o un método**

  El cambio de nombre de una clase o de un método puede resultar útil para favorecer la comprensión del código o para adecuar el código a un determinado estándar (del equipo de desarrollo, de la organización, del cliente o de la industria).

- **Mover una clase**

  Es habitual, cuando el diseño se realiza de forma iterativa, tener en un momento dado la necesidad de mover una clase a otro paquete de código (módulo, espacio de nombres...) de nueva creación, permitiendo agrupar de este modo clases con comportamientos similares.

- **Cambiar el número, orden y tipo de los argumentos de un método**

  Un caso habitual es cuando se necesita un nuevo parámetro o, en el caso contrario, cuando un parámetro ya no tiene sentido.

  Además, también puede resultar útil cuando un determinado número de parámetros se han encapsulado en una nueva clase (por ejemplo, en el caso de tener los parámetros fechaInicio y fechaFin común a un gran número de métodos, se podría crear una clase RangoDeFechas que los encapsulase) y es necesario cambiar convenientemente diferentes métodos y las llamadas a los mismos.

- **Extraer un método del código de otro**

  Cuando se tiene un método excesivamente complejo, o cuando partes del mismo son comunes a otros métodos, debería valorarse el dividirlo convenientemente, extrayendo código a un nuevo método.

- **Extraer una constante**

  Gracias a este método de refactorización es posible extraer literales que se encuentran desperdigados por el código, agrupándolos, por ejemplo, en una clase de constantes.

- **Extraer una clase o mover métodos a otra clase**

  Si una clase tiene más de una responsabilidad (más de una razón para cambiar) deberían extraerse algunos de sus métodos a otra clase (ya existente, o una nueva que se cree en ese momento) para seguir el Principio de Responsabilidad Única (principio básico de la POO).

- **Generar un interface a partir de una clase**

  Esta refactorización resulta muy útil cuando el diseño inicial se va ampliando, permitiendo seguir el Principio de Inversión de Dependencia (principio básico de la POO).

Refactorizaciones avanzadas

Martin Fowler, en su libro Refactoring (Fowler, Beck, Brant, Opdyke, & Roberts, 2000), la referencia clásica cuando se habla de Refactorización, incluye un catálogo más o menos completo de refactorizaciones. Como referencia se puede acceder, en modo 'resumido', a través de la web en http://www.refactoring.com/catalog/.

Entre el conjunto de refactorizaciones propuestas se observan algunos casos de refactorizaciones avanzadas. A continuación se destacan algunas:

- **Descomponer un condicional**

  Si se tiene una sentencia condicional compleja, puede ser recomendable extraer la condición y cada uno de los bloques dependientes a otros métodos.

Ejemplo

Para mejorar la legibilidad del siguiente código:

```
if (date.after(SUMMER_START)
        && date.before(SUMMER_END)) {
    price = quantity * SUMMER_RATE;
} else {
    price = quantity * WINTER_RATE
            + WINTER_SERVICE_CHARGE;
}
```

...sería recomendable crear los siguientes métodos:

```
boolean isSummer(Date date) {
    return date.after(SUMMER_START)
        && date.before(SUMMER_END);
}

double summerCharge(int quantity) {
    return quantity * SUMMER_RATE;
}

double winterCharge(int quantity) {
    return quantity * WINTER_RATE +
WINTER_SERVICE_CHARGE;
}
```

...de modo que el código inicial se reescribiese como sigue:

```
if (isSummer(date)) {
    price = summerCharge(quantity);
} else {
    price = winterCharge(quantity);
}
```

- **Consolidar una expresión condicional**

  Si se tiene una secuencia de condicionales de verificación que retornan el mismo resultado, puede resultar interesante agruparlo en una única expresión condicional y extraerla a un método.

Ejemplo

El siguiente código es susceptible de aplicar la técnica de Consolidar una expresión condicional:

```
double eVote(Citizen ctzn) {
    if (ctzn.getAge() < FULL_AGE) {
        return -1;
    }
    if (ctzn.getNationality()!=SP) {
        return -1;
    }
    if (!eVotingRequested(ctzn)) {
        return -1;
    }
    //TODO: operaciones e-voto
}
```

Para ello se puede crear el siguiente método:

```
boolean isAbleToVote(Citizen ctzn) {
    boolean returnValue = true;
    if (ctzn.getAge() < FULL_AGE
            || ctzn.getNationality()!=SP
            || !eVoteRequested(ctzn)) {
        returnValue = false;
    }
    return returnValue;
}
```

De modo que el código inicial del ejemplo finalmente resultaría como sigue:

```
double eVote(Citizen ctzn) {
    double returnValue;
    if (isAbleToVote(ctzn)) {
        //TODO: operaciones e-voto
    } else {
        returnValue = -1;
    }
    return returnValue;
}
```

- **Introducir variables explicativas**

  Cuando hay una expresión (o partes de una expresión) compleja, se recomienda utilizar variables temporales que tengan nombres representativos de modo que sea posible hacer más legible dicha expresión compleja.

Ejemplo

La siguiente expresión condicional es bastante compleja:

```
if (ctzn.getAge() < FULL_AGE
        || ctzn.getNationality()!=SP
        || !eVoteRequested(ctzn)) {
    //TODO: realizar operaciones
}
```

Para facilitar su comprensión se propone usar variables convenientemente nombradas:

```
boolean isUnderAge =
        citizen.getAge() < FULL_AGE;

boolean isForeign =
        citizen.getNationality() != SP;
```

Y mediante su uso, la anterior expresión condicional resulta mucho más comprensible a simple vista:

```
if ( isUnderAge || isForeign
  || !eVoteRequested(citizen)) {
    //TODO: realizar operaciones
}
```

- **Introducir extensiones locales**

  Cuando se precisan nuevas funcionalidades de una clase que no podemos modificar (por ejemplo, por ser una clase de la API del lenguaje o de una librería de terceros), se propone crear una nueva clase local que contenga dichos métodos extra y que herede o contenga a la clase original.

Ejemplo

En una aplicación se quiere proveer a los usuarios de una utilidad que escriba sus nombres de usuario intercalando mayúsculas y minúsculas.

El desarrollador encargado de resolver esta funcionalidad, que echará en falta en la clase String proporcionada por la API del lenguaje java un método que proporcione esta utilidad, podría desarrollar la siguiente clase que hereda de dicha clase String:

```
class AdvancedString extends String {
    public AdvancedString
            intercalateUpperLowerCase() {
        StringBuffer returnString =
            new StringBuffer();
        for (int i=0;i<this.length;i++) {
            if (i % 2 == 0) {
                returnString.append(
                    this.chartAt(i)
                    .toUpperCase());
            } else {
                returnString.append(
                    this.chartAt(i)
                    .toLowerCase());
            }
        }
        return new AdvancedString(
            returnString.toString());
    }
}
```

- **Eliminar dobles negaciones**

  Se deberían evitar condicionales cuya sentencia de test sea una doble condicional, para ello es recomendable escribirlas como una sentencia de test positiva.

Ejemplo

Las dobles negaciones, en general, son difíciles de entender. Por eso, ante un código como el siguiente:

```
if (!person.hasNotChild()) ...
```

...será preferible reescribirlo para dejarlo como sigue:

```
if (person.hasChild()) ...
```

- **Introducir objetos nulos**

  Cuando se tienen repetidos chequeos de valores *null* se aconseja reemplazar los valores *null* por objetos nulos (con implementaciones 'vacías' de los métodos) para eliminar dichos chequeos.

Ejemplo

Se parte de una factoría que permite obtener instancias de objetos 'manejadores' para tratar unos objetos de unos 'tipos' determinados.

```
class Factory {
    Map<String, Handler> handlers = ...;
    getHandler(String typeId) {
        return handlers.get(typeId);
    }
}
```

Debido al comportamiento del método get del interface Map, si el parámetro *typeId* es *null* o no está mapeado en *handlers*, este código retorna *null*. Por ello, en cada punto donde se quiera iterar sobre una colección de este tipo de objetos se tendrá un código similar al siguiente para evitar las temidas *NullPointerException*:

```
Handler handler = null;
for (Element element:elementList) {
    handler = factory
        .getHandler(element.getTypeId());
    if (handler!=null) {
        handler.operation(element);
    }
}
```

Sin embargo, es posible modificar el código de la factoría para que retorne un objeto nulo cuando no tenga mapeado un determinado 'typeId':

```
class Factory {
    ...
    getHandler(String typeId) {
        Handler handler =
            handlers.get(typeId);
        if (handler == null) {
            handler = NullHandler();
        }
        return handler;
    }
}
```

Y gracias a esta modificación, el código de iteración sobre las colección de 'manejadores' resultará mucho más sencillo:

```
for (Element element:elementList) {
    factory.getHandler(element
      .getTypeId()).operation(element);
}
```

Refactorizaciones 'a la carta'

Además del uso de aquellas refactorizaciones más extendidas en la industria del desarrollo de Software, los desarrolladores pueden/deben diseñar o adaptarlas en función de las necesidades que tengan en un momento concreto. A continuación se muestra una refactorización extraída de un caso real:

- **Reemplazar condicional por factoría de estrategias**

El problema que trata de resolver esta refactorización podría resumirse a través del siguiente ejemplo, teniendo en cuenta que dicho código se encuentra repetido[7] en varios lugares del código original:

```
for (Element elmnt:elementList) {
    if (elmnt instanceOf Type1) {
        //operaciones para el tipo Type1
    } else if ...
      ...
    } else if (elmnt instanceOf TypeN) {
        //operaciones para el tipo TypeN
    }
}
```

Para resolver este problema se propone desarrollar la siguiente jerarquía de estrategias:

```
public interface Strategy {
  void operation(Element elmnt);
}

private class NullStrategy
        implements Strategy {
    public void operation(
            Element elmnt) {}
}

private class Type1Strategy
        implements Strategy {
    public void operation(
            Element elmnt) {
        //operaciones para el tipo Type1
    }
}
...
private class TypeNStrategy
        implements Strategy {
    public void operation(
            Element elmnt) {
        //operaciones para el tipo TypeN
    }
}
```

[7] las operaciones a realizar eran diferentes en los diferentes bucles repartidos a lo largo del código de la aplicación.

Además, se propone crear una factoría para dichas estrategias:

```
private static class StrategyFactory {
    private static Properties strategies;

    static {
        //HACK: cargar desde fichero!!
        strategies = new Properties();

        strategies.setProperty(
            "my.sample.Type1",
            "my.strategy.Type1Strategy"
        );
        ...
        strategies.setProperty(
            "my.sample.TypeN",
            "my.strategy.TypeNStrategy"
        );
    }

    public static Strategy getStrategy(
            Element elmnt) {
        Strategy strategy =
                new NullStrategy();
        String type = strategies.
                getProperty(
                elmnt.getClass()
                        .getName());
        if (type != null) {
          strategy = (Strategy)
                  Class.forName(type)
                  .getInstance();
          ...
        }
        return strategy;
    }
}
```

De modo que, usando este código refactorizado, los bucles del código inicial[8] quedarían resueltos como se muestra a continuación:

```
for (Element elmnt:elementList) {
     StrategyFactory.getStrategy(elmnt)
            .operation(element);
}
```

Resumen

- La refactorización permite reestructurar el código, con el objetivo de mejorar su comprensión, sin modificar el interfaz ni el comportamiento, a partir de un conjunto de pequeñas transformaciones.
- Para asegurar el éxito el código que va a ser objeto de la refactorización debe tener asociadas un conjunto de pruebas automatizadas.
- El diario de deuda técnica es una herramienta que facilita determinar cuando se deben aplicar las refactorizaciones bajo demanda.
- Los IDEs actuales proporcionan un conjunto de refactorizaciones básicas.
- El libro Refactoring de Martin Fowler incluye un completo catálogo de refactorizaciones avanzadas.
- Los desarrolladores deben diseñar o adaptar las refactorizaciones de los catálogos en función de las necesidades concretas de su proyecto.

[8] Como las operaciones en cada bucle realizaban operaciones diferentes, bastó con añadir estas operaciones como métodos en el interface Strategy (y en las clases que lo implementan) y realizar en cada bucle la llamada correspondiente.

Anexo – Recordando Orientación a Objetos

Introducción

En varios puntos del libro se hace referencia a los Principios de la Orientación a Objetos. Por ello, y a modo de repaso, a continuación se explican *grosso modo* los conceptos básicos del paradigma de la Orientación a Objetos:

- Abstracción,
- Encapsulamiento,
- Herencia, y
- Polimorfismo.

Además, se presentan de manera breve algunos de los principios básicos del paradigma, los cuales es conveniente tenerlos en mente para alcanzar con éxito la excelencia del código:

- **Principios generales**
  - Principio DRY (*Don't Repeat Yourself*)
  - Principio KISS (*Keep It Simple and Stupid*)
- **Principios SOLID**
  - Principio de Responsabilidad Única (SRP – *Single Responsibility Principle*)
  - Principio Abierto-Cerrado (OCP – *Open Closed Principle*)
  - Principio de Sustitución de Liskov (LSP - *Liskov Substitution Principle*)
  - Principio de Segregación de Interfaces (ISP - *Interface Segregation Principle*)
  - Principio de Inversión de Dependencia (DIP – *Dependency Inversion Principle*)
- **Patrones GRASP**
  - Experto en información
  - Creador
  - Controlador
  - Alta cohesión
  - Bajo acoplamiento
  - Polimorfismo
  - Fabricación Pura
  - Indirección
  - Variaciones Protegidas

Además, es altamente recomendable tener un conocimiento, al menos básico, de diferentes catálogos de patrones como los patrones “GoF” (extraídos del libro “Design Patterns” escrito por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides – grupo conocido como “Gang of Four”) o los patrones “Core J2EE” (del libro “Core J2EE Patterns: Best Practices and Design Strategies” escrito por Deepak Alur, John Crupi y Dan Malks, arquitectos del Sun Java Center).

Conceptos Básicos

- **Abstracción**

  La abstracción es el mecanismo por el cual una clase se aísla de su contexto mostrando tan solo una interfaz que representa su comportamiento.

  Esta interfaz está compuesta por el conjunto de propiedades y métodos públicos de la clase, y permite mantener oculta su implementación al resto de clases.

- **Encapsulamiento**

  El encapsulamiento es el mecanismo por el cual se agrupan todos los elementos pertenecientes a una misma entidad a un mismo nivel de abstracción, o lo que es lo mismo, a la agrupación de dichos elementos en una misma clase.

- **Herencia**

  La herencia es un mecanismo de reutilización usado en la Orientación a Objetos por el cual una clase deriva de otra para extender la funcionalidad proporcionada por la clase original. La clase inicial se denomina clase padre, clase base o superclase, mientras que la clase resultado de aplicar la herencia se denomina clase hija, clase derivada o subclase.

- **Polimorfismo**

  El polimorfismo permite definir múltiples clases (derivadas de una misma clase base) con métodos o propiedades denominados de forma idéntica, pero con funcionalidad diferente, gracias a lo cual dichas clases pueden ser intercambiadas en tiempo de ejecución en función de las necesidades del cliente.

Principios Generales

- **DRY (No te repitas)**

> ***En un mismo sistema, cada porción de conocimiento debería tener una única representación, no ambigua y autoritaria.***

Cuando se aplica satisfactoriamente este principio, una modificación de un elemento no

provocará cambios en aquellos elementos con los que no está relacionado de manera lógica, mientras que aquellos elementos con los que sí está relacionado lógicamente cambiarán de manera predictiva y uniforme, manteniendo así su sincronización.

Este principio es especialmente importante en arquitecturas multicapa.

- **KISS**

La simplicidad debe ser un objetivo vital del diseño, evitando a toda costa toda complejidad innecesaria.

Este principio establece que los diseños deben ser lo más sencillos posible. Se debe evitar el sobre-diseño, o lo que es lo mismo, se debe evitar el 'antipatrón' Gas Factory (que recibe su nombre por la evocación de la complejidad de una refinería, donde un número incontable de tuberías corre en todas direcciones).

Está alineado con "*La Navaja de Occam*" y las máximas de Einstein: "*Todo debería hacerse tan simple como sea posible, pero no más simple*"; de Leonardo Da Vinci: "*La simplifidad es la última sofisticación*"; y la de Antoine de Saint Exupery "*La perfección se alcanza, no cuando no hay nada más que añadir, si no cuando no hay nada más que quitar*".

Principios SOLID

- **Principio de Responsabilidad Única (SRP – Single Responsibility Principle)**

Una clase debería tener solo una razón para cambiar.

Esto equivale a decir que cada clase debería tener una única responsabilidad, y todos los servicios que proporcione deberían estar orientados a realizar esa responsabilidad.

Aunque es un principio fácil de comprender, salvo en los casos más triviales suele resultar complejo llevarlo a la práctica por la dificultad que entraña descubrir cuando una clase tiene más de un motivo para cambiar.

- **Principio Abierto-Cerrado (OCP – Open Closed Principle)**

Una clase debe estar abierta a la extensión pero cerrada a la modificación.

O lo que es lo mismo, debe ser posible extender la funcionalidad de una clase sin necesidad de modificar su código fuente.

Aunque en una primera lectura este principio puede parecer contradictorio o incluso imposible, existen diferentes técnicas y/o diseños de la arquitectura que facilitan su implementación.

- **Principio de Sustitución de Liskov (LSP - Liskov Substitution Principle)**

Toda subclase debe soportar ser sustituida por su superbase.

Este principio establece que un cliente de una superclase continuará funcionando si se cambia dicha clase base por una de sus subclases.

O hablando en términos de contratos: las precondiciones de los métodos de la subclase no podrán ser más fuertes que las de la clase base, y sus postcondiciones no podrán ser más débiles.

- **Principio de Segregación de Interfaces (ISP - Interface Segregation Principle)**

Es preferible tener interfaces específicos para cada cliente que uno de propósito general.

Este principio indica que no debe obligarse a los clientes a depender de métodos que no utilizan.

Cuando se habla de interfaces específicos para cada cliente en realidad debe entenderse por cada tipo de cliente, es decir, los clientes deben categorizarse por tipos, creando entonces interfaces para cada uno de estos tipos.

- **Principio de Inversión de Dependencia (DIP – Dependency Inversion Principle)**

> ***Depender de abstracciones y no de concreciones.***

Tradicionalmente, las dependencias entre clases se han mantenido de tal modo que las clases de más alto nivel dependen de las clases de detalle. Sin embargo este principio establece que las dependencias de las clases que contienen los detalles de la implementación deben mantener una relación de dependencia hacia las clases abstractas, es decir, se debe invertir las dependencias.

Patrones GRASP

- **Experto en información**

Este patrón establece que la responsabilidad de llevar a cabo una acción debe recaer sobre la clase que conozca toda la información necesaria para realizarla.

- **Creador**

La responsabilidad de crear o instanciar un objeto la debe tener la clase que:
  - tiene la información necesaria para crearlo, o
  - usa directamente sus instancias, o
  - almacena o maneja varias instancias, o
  - contiene o agrega la clase del objeto.

- **Controlador**

La responsabilidad de controlar el flujo de eventos del sistema debe recaer en clases específicas que centralizan las actividades de validación, seguridad… Sin embargo, dicho controlador no debe realizar estas actividades, si no que las delegará en otras clases con las que mantiene un modelo de alta cohesión.

- **Alta cohesión**

Cada elemento del diseño debe realizar una única labor auto-identificable del sistema, que no debe ser desempeñada por el resto de elementos.

- **Bajo acoplamiento**

El número de dependencias entre clases se debe mantener en el mínimo posible.

- **Polimorfismo**

La responsabilidad de definir la variación de comportamiento basado en tipos debe corresponder a los tipos para los cuales ocurre dicha variación.

Para llevar a cabo este principio se debe hacer uso del concepto de polimorfismo.

- **Fabricación Pura**

Se denomina fabricación pura a una clase que se crea de manera intencionada para disminuir el acoplamiento, aumentar la cohesión y/o potenciar la reutilización del código, sin que dicha clase represente una entidad del dominio del problema.

- **Indirección**

Para reducir el acoplamiento entre dos clases se debe asignar la responsabilidad de mediación a una clase intermedia.

- **Variaciones Protegidas**

Las clases se deben proteger de las variaciones de otras clases envolviendo la clase inestable con un interface, y usando el polimorfismo para crear las diferentes implementaciones de dicho interface.

Lista de ilustraciones

Lista de ilustraciones

Trabajos citados

Trabajos citados

Chrissis, M. B., Konrad, M., & Shrum, S. (2007). *CMMI: Guidelines for Process Integration and Product Improvement.* Addison Wesley.

Cockburn, A., & Williams, A. (2001). The costs and benefits of pair programming. En *Extreme programming examined .* Boston: Addison-Wesley Longman Publishing Co.

Cunningham, W. (1992). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*.
Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (2000). *Refactoring: Improving the Design of Existing Code.* Addison Wesley.

ISO. (2001). *ISO/IEC 9126 Software engineering — Product quality.* International Organization for Standarization.

ISO. (2005). *ISO/IEC 2510:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation.* International Organization for Standarization.

Man Lui, K., & Chan, K. C. (2006). Pair programming productivity: Novice–novice vs. expert–expert. *International Journal of Human-Computer Studies*.

Padberg, F., & Muller, M. (2003). Analyzing the cost and benefit of pair programming . *Software Metrics Symposium*.

Raymond, E. S. (1997). *La catedral y el bazar.* O'Reilly.

Sun Microsystems. (12 de September de 1997). *Oracle.* Recuperado el Enero de 2012, de http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Índice

Índice

O

P

R

S

T

X