Revista de Procesos y Métricas de las Tecnologías de la Información (RPM) ISSN: 1698-2029
VOL. 1, Nº 2, Agosto 2004, 31-41

PROCESO Y HERRAMIENTAS PARA LA PRODUCTIVIDAD EN EL ASEGURAMIENTO Y MEDICIÓN DE CALIDAD EN DESARROLLOS JAVA

Luis Fernández Sanz
Dpto. Programación e Ingeniería del Software. Universidad Europea de Madrid.
c/Tajo s/n. Villaviciosa de Odón, Madrid, Spain.
E-Mail : lufern@uem.es

Pedro J. Lara Bercial
Dpto. Programación e Ingeniería del Software. Universidad Europea de Madrid.
c/Tajo s/n. Villaviciosa de Odón, Madrid, Spain.
E-Mail : pedro@uem.es

Abstract: Software Quality Assurance is a key factor for success in software development projects. Although best practices are well-known, it is usual that its activity and implementation costs are perceived as a handicap by developers and managers. In this paper, we present several ideas to achieve an adequate combination of techniques like inspections and testing with the development environment and process, emphasizing interesting strategies of automation through the use of tools and guides for customizing their application to improve efficiency and staff productivity.

Resumen: El aseguramiento de la calidad del software es un elemento esencial para el éxito de un proyecto de desarrollo. Aunque las mejores prácticas para su eficacia son bien conocidas, su coste de actividad e implantación resulta muchas veces un freno tanto para gestores como para desarrolladores. En este artículo, proponemos ideas para la adecuada combinación de técnicas como inspecciones y pruebas en el entorno y el proceso de desarrollo, haciendo énfasis en las estrategias de automatización con herramientas y personalización de las mismas que mejoren su eficiencia y la productividad del personal.

Palabras Clave: Aseguramiento de calidad, desarrollo de software, herramientas, pruebas , inspecciones, estilo de programación

1. INTRODUCCIÓN

El éxito de un proyecto de desarrollo se basa en la obtención de un nivel adecuado de calidad en el software desarrollado para satisfacer al usuario a la vez que también supone el cumplimiento de plazos y presupuestos para el cliente. Por supuesto, para que el proyecto sea rentable e interesante para la organización de desarrollo, debe realizarse con la máxima eficiencia y productividad Para poder combinar ambas facetas de manera eficaz, no sólo debemos establecer un proceso de aseguramiento de calidad del software adaptado a las necesidades técnicas del proyecto sino también a la organización de desarrollo logrando minimizar los recursos para la consecución y el control de calidad.

Las técnicas y procesos para el aseguramiento de la calidad son bien conocidos y, además, son parte de las buenas prácticas recomendadas por los modelos de evaluación y mejora de procesos en sus primeros niveles: [CMM Product Team, 2002a] [CMM Product Team, 2002b], proceso básico en ISO 15504 SPICE [ISO, 1999].En general, las técnicas que más frecuentemente, con buenos resultados, se utilizan en el aseguramiento de calidad de software se corresponden con la medición de software, los procesos de revisión y auditoría y las pruebas de software (estas dos últimas encuadradas como principales técnicas en lo que se suele conocer

como verificación y validación). Como indica el estándar IEEE 730 de aseguramiento de calidad [IEEE, 1998], estas técnicas componen el núcleo de actuación en el proyecto dentro de un buen entorno con adecuada gestión de configuración.

Este enfoque de aseguramiento de calidad no sólo es parte de las recomendaciones genéricas de normas y autores sino una realidad ya experimentada en multitud de experiencias en organizaciones de desarrollo de software. Desde las experiencias exitosas de Fagan en IBM introduciendo las inspecciones de diseño y código [Fagan, 1976], son numerosas las organizaciones que han aplicado, por ejemplo, estrategias de procesos de revisión e inspección con buenos resultados. AETNA, IBM Respond, Ameican Express son ya mencionadas en [Fagan, 1986] a las que se unen las diferentes experiencias recopiladas en [Wheeler et al., 1996] referidas a ATT, Hewlett-Packard, Bell Northern Research, Bull, ICL, JPL, Shell Research, etc. Estas experiencias junto con los programas de aseguramiento de calidad con extensos planes de medición como los de Hewlett-Packard [Grady y Caswell, 1987] demuestran que los beneficios logrados son interesantes y rentables con disminuciones en esfuerzos de desarrollo (cifras de 25% o 14%, por ejemplo), interesantes ROI (8:1 o ¡30:1!), disminución de defectos (38%, por ejemplo). También se sabe que los beneficios de detección de defectos no dependen sólo de la aplicación de controles puntuales sino del diseño de una estrategia que combina las fortalezas de las distintas técnicas: por ejemplo, los procesos de revisión detectan defectos que las pruebas difícilmente revelan y al revés. De hecho, la combinación de inspecciones y revisiones y posteriores pruebas adecuada es la estrategia que consigue mejores resultados en cuanto a detección de defectos.

En algunas ocasiones, las referencias a organizaciones extranjeras pueden sugerir la idea de que en España no se aplican estas técnicas o que no se cuenta con suficientes datos sobre su implementación. Sin embargo, no es éste el panorama. Además de los sectores aeroespaciales y defensa, frecuentemente obligados por normas internacionales, distintas organizaciones han trabajado en mejorar su proceso de aseguramiento de calidad incluyendo inspecciones, mediciones y pruebas adecuadas. Telefónica I+D ya estableció un programa de inspecciones de código y de mejora del estilo de programación con interesantes beneficios a principios de los noventa [Borrajo, 1994] y en CRISA, una empresa aeroespacial, se implantaron revisiones de código y de estilo para la mejora de la calidad en la misma época [Aldea y Gallego, 1994]. Desde entonces han sido muchas las iniciativas realizadas; por citar alguna, podemos mencionar la mejora en los procesos de Fagor Automation que incluía inspecciones y revisiones de estilo presentada [Sagardui y Onandia, 2002] y que fue presentada dentro de las VII Jornadas de Innovación y Calidad del Software (www.jics.net).

En general, según se deduce de estas experiencias, una buena estrategia de aseguramiento de calidad de software debe suponer la realización de controles precoces por varias razones:

- Los costes de corrección de defectos crecen a medida que avanza el proyecto por lo que conviene incluir controles específicos de los productos desarrollados lo más cerca posible del momento de su realización. Los procesos de revisión son esenciales para este propósito ya que permiten controlar productos no ejecutables (diseños, etc.).
- Las pruebas del producto no permitirán nunca asegurar la ausencia de defectos en el producto ya que la complejidad de software hace inviable la realización de un control definitivo, exhaustivo y completo. Además, suelen suponer un esfuerzo muy grande para su efectividad en detección de defectos. Por ello, deberían combinarse con las revisiones además de concentrarse en los aspectos y partes del software que requieren mayor control aparte de incorporar elementos de verificación de su efectividad a través de medidas de cobertura de ejecución.
- Dado que los costes de aplicación de revisiones, inspecciones y pruebas pueden ser muy elevados es necesario incrementar la eficiencias de dichas tareas. Las dos posibilidades más comunes son la siguientes:
  - Puesto que no todos los proyectos deben afrontar mayor intensidad de control a las partes del software más propensas a problemas o a aquellas más críticas, es posible adaptar los criterios de calidad. En definitiva, se trata de aplicar el principio de Pareto: localizar el 20% del producto que produce el 80% de los problemas. En el resto del código, se realizan controles con una intensidad convencional. En

general, es necesario contar con medidas o indicadores que permitan detectar o elegir a priori las partes destinadas a un mayor intensidad de control (tanto mediante revisiones como con pruebas).

- Dado que el esfuerzo del personal cualificado es el principal coste de estas tareas, es interesante tratar de ahorrar haciendo que ciertos controles se pueden efectuar de forma automática o con el apoyo de herramientas. Para una mayor eficacia, no resulta necesario que los controles de dichas herramientas queden exclusivamente a disposición del personal de aseguramiento de calidad, los revisores o el jefe de proyecto. Si se integran en el entorno del desarrollador, éste puede controlar su trabajo frente a los estándares y la gestión del proyecto simplemente solicitará que cada entrega incluya el informe de la herramienta que confirma que se cumplen los normas fijadas (independientemente de que se hagan repeticiones aleatorias de controles para disipar sospechas de manipulaciones).

En este artículo, veremos cómo se puede establecer un proceso que combine los controles del aseguramiento de calidad y las estrategias para minimizar los costes de su aplicación, incrementando la facilidad para ser incorporado en las organizaciones de desarrollo.

2. PROPUESTA DE PROCESO

Nuestra propuesta, en sentido amplio, pasaría por establecer un proceso que desde el diseño del software permita incorporar la filosofía de aseguramiento y medición de calidad productivos en desarrollos en Java. En este proceso integraremos tanto las recomendaciones de diseño e implementación como las técnicas para su aplicación y control con soporte de herramientas. Aunque presentaremos una implementación que utiliza un entorno de desarrollo gratuito (Eclipse) para diseñar y codificar aplicaciones en Java, que permite la utilización de herramientas, así mismo gratuitas, para la detección de no adecuaciones al convenio de estilo y para la evaluación de métricas de diseño y codificación. Realmente mencionaremos también herramientas comerciales que pueden realizar estas operaciones aportando otras ventajas y que también intervienen en la evaluación de cobertura de pruebas. De hecho en nuestra propuesta de proceso, incluiremos también la mejora de las pruebas de software desde la actividad de análisis de requisitos según lo presentado en [Fernández et al, 2003].

La implementación del proceso que se propone en este artículo, se corresponde con parte de la figura 1 y consiste en la inclusión en las fases de diseño e implementación de una serie de controles o actividades de protección encargadas de asegurar en cierta medida la calidad de los resultados obtenidos únicamente en la fase de pruebas, pero es susceptible de ser extendido a otras fases del desarrollo.

De esta manera el proceso se compondría de:

- Una fase de análisis que entre otras cosas obtiene los diferentes casos de uso analizando la perspectiva que el usuario tiene de la aplicación siguiendo la propuesta de[Fernández et al, 2003].
- Una fase de diseño, en la que además de producir resultados clásicos en la orientación a objetos como son los diagramas de clases, de colaboración y el diseño de los casos de prueba, se produce también una matriz o cubo tridimensional de trazabilidad entre Casos de Uso, Clases y Casos de Prueba a partir de la cual:
  - Utilizando herramientas de asignación de riesgos como las presentadas en [Wang et Al., 2003] basadas en la metodología de asignación de riesgos en función de diferentes artefactos UML descrita en [Goseva-Popstojanova et al, 2003] establecer un ranking de Casos de uso en función del riesgo.
  - Utilizando como entradas algunos de estos resultados, aplicaríamos herramientas de medida que evaluasen, a través de ciertas métricas la propensión a tener defectos de cada clase [Abreu y Melo, 1996].

- Establecer un ranking de casos de prueba ordenándolos en función del nivel de riesgo de los casos de uso asociados a ellos y de la propensión al defecto de las clases relacionadas con ellos en nuestro cubo de trazabilidad.

- Ciclar en las fases de Ejecución de Pruebas, Rediseño y Remodificación tantas veces como sea necesario para conseguir el nivel deseado de defectos detectados. Se combinaría esta evaluación de pruebas con la aplicación de las más tradicionales medidas de cobertura.

A continuación indicaremos el soporte que las herramientas pueden jugar en la automatización y eficiencia de estas actividades a la vez que presentamos cómo las métricas y las técnicas de inspecciones y pruebas pueden aportar buenos resultados para la mejora de la calidad del software. En el resto del artículo presentaremos dos herramientas integradas en un mismo entorno de desarrollo, con un espacio de trabajo común para diseñadores, programadores, y encargados de aseguramiento de calidad. Por otra parte, comentaremos cómo herramientas comerciales también aportan su apoyo a evaluar la cobertura de pruebas.

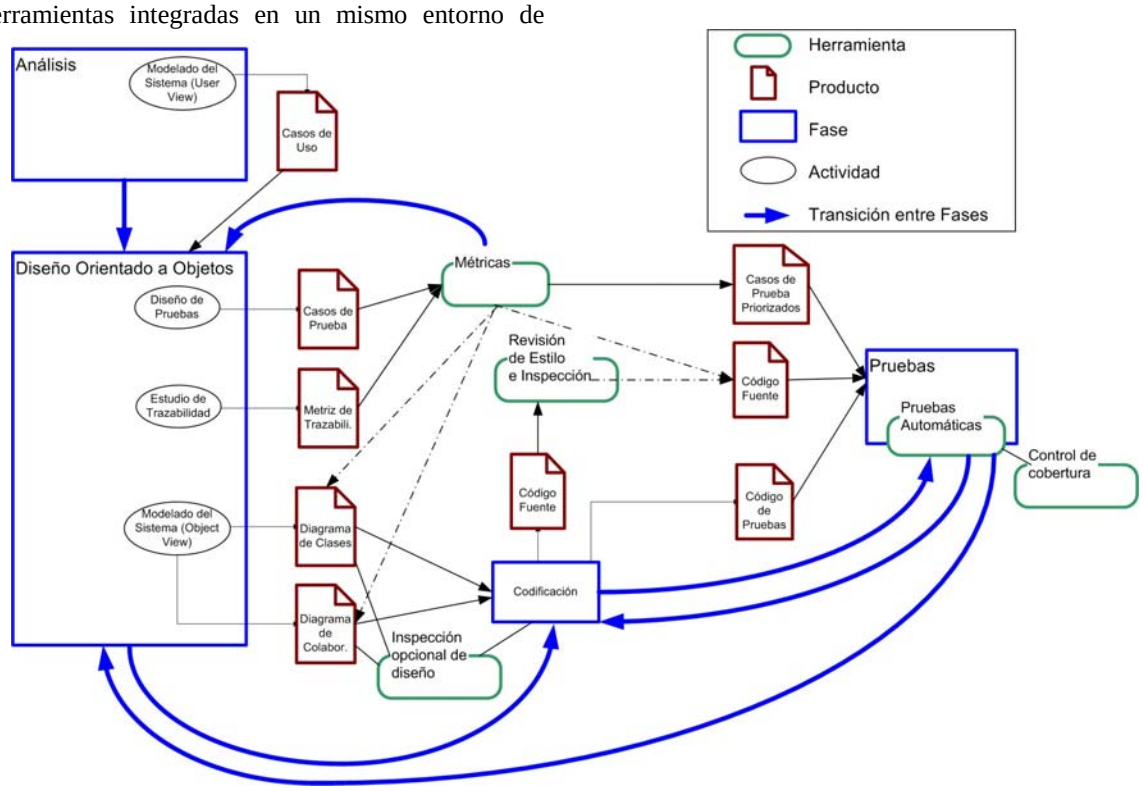

Figura 1. Proceso propuesto

2.1 Entornos de desarrollo

Todo proceso eficiente debe contar con la ayuda de herramientas de apoyo al desarrollo que puedan integrarse con los entornos de desarrollo más conocidos: JBuilder, JDeveloper, JEdit, NetBeans, JCreator, etc. En [Lara y Fernández, 2004] ya se explicaba como integrar con JCreator, uno de los IDE gratuitos más utilizados, dos herramientas llamadas JCSC y AStyle. JCSC es una herramienta escrita en Java que chequea el estándar y las convenciones de SUN contra el códigos escrito, cubriendo prácticamente todas las recomendaciones de estilo, así como la correcta disposición de clases, atributos, métodos y, en general, cualquier entidad que pueda aparecer en un fichero Java. AStyle, por su parte, revisa y modifica, automáticamente, el código utilizando una serie de filtros que alinean y reformatean según unas reglas fijas.

En ese artículo quedaba patente la necesidad de un sistema más cómodo y sencillo de integración de dichas herramientas, que al menos supusiese un procedimiento común para todas ellas. Eclipse es un IDE de código abierto (open source) que soporta muchos lenguajes de programación. Este IDE viene con un enotorno de Desarrollo de Java (Java Development Environment o JDE) que incluye un editor de resalte de sintaxis, un compilador, un depurador, un navegador de clases y un administrador de archivo/proyecto. Una de las características fundamentales de Eclipse es que esta desarrollado enteramente con una arquitectura de plug-ins de forma que puede extenderse con todo tipo de herramientas de ayuda [Bolour, 2003].

Un plug-in en Eclipse es un componente que proporciona un servicio dentro del entorno de trabajo del IDE,. Un componente es un objeto que puede ser integrado en un sistema una vez que dicho sistema ya se ha liberado para explotación. El entorno de ejecución de Eclipse permitee activar y ejecutar varios plug-ins simultáneamente para diferentes actividades del desarrollo. Éstos deben estar desarrollados heredando de la clase abstracta org.eclipse.core.runtime.Plugin, con facilidades genéricas para manejo y gestión de plug-ins.

Cualquier instalación de Eclipse tiene una carpeta llamada “plugins” donde cada uno tiene su propio directorio para almacenar las clases que necesita, así como un fichero de manifiesto en XML, llamado plugin.xml, del que Eclipse lee la información necesaria para activarlo (Figura 2.)

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Figura2. Fichero de manifiesto de plug-in para JUnit

Como plug-in de UML, se ha utilizado una versión gratuita desarrollada por Omondo, llamada EclipseUML, que permite dibujar diagramas de de clases y de colaboración UML 1.4 [OMG, 2001].

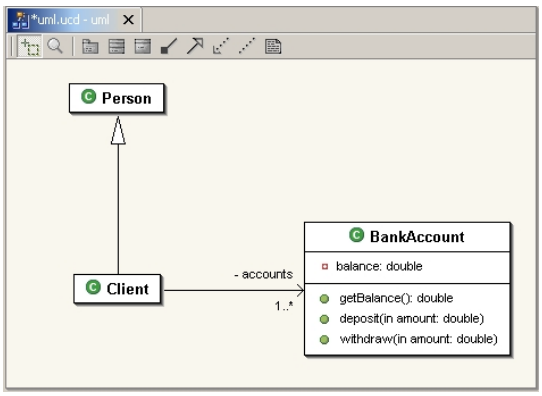

Figura 3. Editor de EclipseUML además de una vista gráfica para definir clases y relaciones (Figura 3).

Esta versión permite mantener sincronizado el código fuente con el diagrama de clases de forma que los cambios realizados en la vista gráfica son automáticamente realizados en el fuente de la clase o clases correspondientes. Si queremos replicar esta funcionalidad en diagramas de secuencia,se requiere la versión *Studio* de EclipseUML de pago.

2.2 Análisis de Pareto del diseño

Basándose en lo definido por [Archer y Stinson, 95], [García y Harrison, 2000] describen un conjunto representativo de métricas orientadas a objetos, cuyo correlación con número de defectos o esfuerzos de mantenimiento ya ha sido validada [Abreu y Melo, 1996] y [Basili et Al., 1995]. De hecho se pueden establecer determinados rangos de valores fuera de los cuales una clase o pieza del código es más propensa a errores. Podemos contar así con:

- Métricas a Nivel de Sistema: definidas por [Abreu y Melo, 1996] y conocidas como MOOD incluyen las conocidas medidas de proporción de métodos ocultos (MHF) y de métodos heredados (MIF) (sobre la base del total de métodos del sistema), proporción de atributos ocultos (AHF) y de atributos heredados (AIF) (sobre la base del total de atributos del sistema), proporción de polimorfismo (PF) (relación del numero de métodos de una clase redefinidos en sus clases derivadas y el número total de métodos de las clase padre por el número de descendientes) y Proporción de Acoplamiento (CF) (relación entre número de relaciones entre clases excepto las de herencia y el numero total de pares de clases).
- Métricas de Acoplamiento: con la métrica de Acoplamientro entre clases de objetos (CBO) definida por [Chidamber y Kemerer, 1994] que mide el nivel de dependencia de unas clases con otras.
- Métricas de Herencia: un uso excesivo de la herencia puede ser peligroso para la calidad [Cartwright y Shepperd, 1996]. Algunas de las métricas mas representativas son la de Profundidad del Árbol de Herencia (DIT) y el número de hijos (NOC) definidas por [Chidamber y Kemerer, 1994] y orientadas a la cuantificación del árbol de herencia. También interesa el índice de Especialización por clase (SIX), que muestra en qué medida las subclases

redefinen el comportamiento de sus superclases [Lorenz y Kidd, 1994].

- Métricas de Clases: estas métricas permiten detectar clases que necesiten ser redefinidas destacando ciertos aspectos de su nivel de abstracción. Por ejemplo, se pueden incluir la Respuesta de una Clase (RFC), definida por [Chidamber y Kemerer, 1994] (cuenta las ocurrencias de llamadas a otras clases desde una clase particular), los Métodos Ponderados por Clase (WMC) (complejidad de clase en función de la complejidad de sus métodos [Chidamber y Kemerer, 1994]) y la Falta de Cohesión en los Métodos (LCOM) (número de atributos comunes usados por diferentes métodos) de la que existen dos medidas propuestas [Chidamber y Kemerer, 1994] [Henderson-Sellers, 1996].

Lógicamente la aplicación de métricas debe estar apoyada por herramientas que agilicen la recogida y el cálculo de valores. Para ello, aprovechando el entorno propuesto, se han utilizado para nuestra propuesta dos plug-ins. Uno desarrollado por TeamInABox (TIB), que tiene como objetivo permitir al usuario estudiar los valores de diferentes métricas de software, tanto desde el punto de vista del diseño como desde el punto de vista de la codificación. El otro, Metrics 1.3.5, parecido y absolutamente open-source que complementa al primero con alguna de las métricas que éste no cubre pero que solo funciona con versiones de Eclipse superiores a la 3.0M8.

Los valores de las métricas son medidos en cada ciclo de compilación del software y el programador es avisado cada vez que alguno de los valores se sale de los rangos establecidos como válidos. Estos rangos son configurables desde el propio entorno y pueden adaptarse en función de la experiencia personal, del tipo de equipo de desarrollo, del tipo de proyecto, etc. Además existe la posibilidad de obtener un informe resumen de carácter visual, exportable en diferentes formatos (HTML, CSV, etc.) como el que muestra la figura 4.

Las métricas validadas que cubren y cuya influencia por tanto está demostrada empíricamente son las siguientes en el Plug-in de TIB:

- Métricas de Acoplamiento: Acoplamientro entre clases de objetos (CBO).
- Métricas de Clase: Métodos Ponderados por Clase (WMC) y Falta de Cohesión en los Métodos (LCOM), tanto la versión de [Chidamber y Kemerer, 1994] como la de [Henderson-Sellers, 1996].
- Métricas de Métodos: Líneas de Código por Método (LOC) y Número de Mensajes Enviados (NOM).

En el caso del Plug-in Metrics 1.3.5.:

- Métricas de Herencia: Profundidad en el árbol de herencia (DIT) y Indice de Especialización.

Además de estas métricas el plug-in de TIB ofrece otras métricas a nivel de Métodos como la de McCabe [McCabe, 1976], número de campos, nivel de anidamiento en un método, número de Parámetros y número de Instrucciones (incluyendo las llamadas al constructor super, for, if, return, switch, throw, try, match, finally, while, asignaciones, llamadas a métodos y pre y post incrementos y decrementos)

Aunque para una evaluación ideal de los resultados es conveniente realizar esta labor de ajuste en función de los experimentos existentes al respecto en cuanto a validación de las métricas correspondientes [Abreu y Melo, 1996][Basili et Al, 1995] o, mejor aún, recopilando datos en proyectos piloto y determinando los valores umbrales de control que determinan cuándo un trozo de código puede ser más propenso a problemas (*defect-prone*). Aprovechando los resultados de las métricas, se pueden conseguir diversos resultados positivos:

- Si no hay posibilidad de aplicar un nivel intensivo de inspecciones y pruebas a todo el software, se pueden seleccionar los componentes de código más propensos a problemas para concentrar en ellos un mayor esfuerzo de detección y evaluación.
- Los diseñadores pueden obtener un feedback que les permita reconsiderar su diseño e implementación para obtener mejores resultados cuantitativos. Tras acumular suficientes datos de proyectos, los responsables de calidad pueden establecer límites/umbrales para cada métrica que deberán cumplir (salvo excepciones autorizadas) las entregas de los desarrolladores.
- Los inspectores cuenta con importante información cuantitativa y cualitativa que permita realizar una mejor detección de defectos. Así, los inspectores recibirán no sólo el material a revisar sino también el informe de métricas correspondiente al mismo.

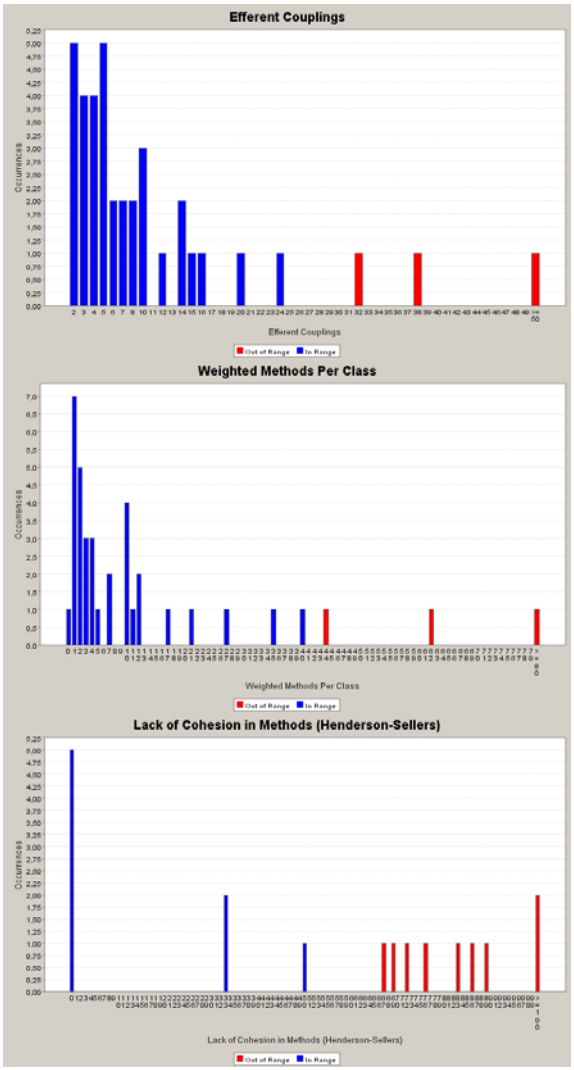

Figura 3. Resúmenes gráficos de valores de métricas

2.3 Apoyo a las inspecciones de código

Los procesos de revisión como las inspecciones suponen gran esfuerzo en horas de personal cualificado aunque sus beneficios son claros [Wheeler et al., 1996]. Lógicamente, las inspecciones de código suelen trabajar sobre entregas que, al menos, han pasado la primera "compilación limpia", sin preocuparse por errores de sintaxis capturados por el compilador. El siguiente paso evidente es poder incorporar algún control adicional que permita concentrarse a los inspectores en aspectos más complejos de su labor. Lo más sencillo y habitual son las normas de estilo y de documentación.

No se puede negar que la calidad del software es una cualidad muy compleja. Los modelos que pretenden aportar un marco para su evaluación (como el estándar ISO 9126 [ISO]) suelen estar basados en complejas estructuras de subcaracterísticas a evaluar: facilidad de uso (*usability*), fiabilidad, funciones precisas y seguras, eficiencia, portabilidad, etc. Algunas de ellas son claramente contrapuestas por lo que resulta esencial definir el perfil de calidad requerido por el usuario lo antes posible en el proyecto.

En este ámbito, el cumplimiento de reglas de estilo de programación redunda en mejoras, por ejemplo, en la facilidad de mantenimiento, la portabilidad o la conformidad a estándares. A menudo se tiende a pensar que un buen estilo de programación es algo que te enseñan (en algunos casos ni eso), mientras estas estudiando y que luego nunca utilizas. En el mundo de la informática la presión en los momentos en que se acercan las entregas de software hace que el estilo de programación pase a ser lo primero de lo que se prescinde cuando el tiempo apremia ya que ningún programa deja de funcionar, al menos aparentemente, debido a un mal formateo del código o a la falta de documentación. Por lo tanto, ¿por qué perder tiempo en lavar la cara al código cuando lo importante es que haga lo que tiene que hacer? La respuesta es sencilla con los siguientes datos:

- Entre un 50% y un 80% del coste del software corresponde al mantenimiento. Las mejoras de eficiencia en esta actividad permiten un gran ahorro de coste y esfuerzo.
- Difícilmente es el autor del software original el encargado de mantenerlo.

En nuestra propuesta para Java, utilizaremos el convenio de estilo de codificación propuesto por [SUN, 1996] para el lenguaje de programación Java. En él podemos encontrar una serie de recomendaciones acerca de diferentes aspectos que afectan a la calidad del código y, por lo tanto, a características deseables en todo software; directamente, por ejemplo, a la facilidad de mantenimiento, flexibilidad y la capacidad de reutilización; indirectamente, como resultado de un entorno de desarrollo mas cómodo para los programadores, a otras como la fiabilidad y la

integridad del software. Resumiendo dicho documento, las recomendaciones son:

Archivos y Organización Interna: Se especifica cuales deben ser los nombres y extensiones de los diferentes tipos de ficheros involucrados en cualquier desarrollo java. Así como cuál debe ser su organización interna en cuanto al orden y formato de las tres partes en que se divide todo fichero java: Comentarios Iniciales, Sentencias Package e Import, Declaraciones de Clases e Interfaces.

Formato: Da unas reglas estrictas acerca de cómo realizar el sangrado y alineación, longitudes de línea, cortes en la líneas largas, uso de llaves y cuándo utilizar espacios y líneas en blanco.

Estructuras de Control: Especifica formato y uso de las principales estructuras de control como sentencias simples, compuestas, Return, del tipo If (if, if-else, etc.), For, While/Do-while, Switch y Try-Catch.

Guía rápida de convenciones de código Java

Formato:

- Sangría:
  - Use 4 espacios como unidad de sangrado.
  - Use tabuladores o espacios para sangrar y no ambos.
  - No sangre las clases o interfaces más externas de un fichero
  - Sangre los atributos, métodos y clases internas un nivel.
- Uso de llaves:
  - Ponga la llave abierta en la misma línea de la declaración
  - Ponga la llave cerrada en una nueva línea, alineada con la declaración
- Formato de líneas:
  - No escriba líneas de más de 80 caracteres
  - Parta las líneas detrás de una coma o delante de un operador binario, preferiblemente al mayor nivel posible si hay más de una operación anidada.
  - Alinee la línea cortada con el comienzo de la expresión a la que pertenece.
- Líneas y Espacios en Blanco:
  - Use una línea en blanco:
  - antes de un comentario,
  - entre dos métodos,
  - después de la cabecera de un método,
  - después de un bloque de declaraciones locales.
  - entre dos secciones de semántica diferente dentro del mismo bloque
- Use un espacio:
  - entre una palabra reservada y un paréntesis abierto
  - después de una coma
  - alrededor de un operador binario (excepto del ".") o ternario
  - entre las tres partes de un for
  - después de un casting

Prácticas de Programación:

- Una instrucción y una declaración por línea
- Inicializar variables en la declaración
- Poner bloques entre llaves, incluso cuando tienen una sola línea
- Usar el nombre de la clase para referirse a atributos o métodos estáticos (de clase)
- Usar siempre paréntesis, aunque no hagan falta
- Evitar el uso de literales (Excep.: 0, 1, y –1)
- Ajustarse a los convenios de nombrado:
  - Cualquier nombre de algo debe ser una palabar o frase corta pero descriptiva y sin abreviaturas
  - Clases e Interfaces: Siempre sustantivos. Ejemplos: TextField y MouseListener
  - Métodos: Siempre verbos. Ejemplo: setBackground
  - Variables: Siempre nombres. Ejemplo: fontSize
  - Constantes: Siempre en mayusculas. Ejemplo: EXIT_ON_CLOSE

Documentación:

- Comentarios de Implementación:
  - No añada comentarios obvios.
  - Todo comentario debe ser precedido por línea en blanco.
  - Minimice la necesidad de comentarios eligiendo bien los nombres de atributos, variables y métodos
  - Suelen ser de utilidad comentarios que describen el funcionamiento de bloques completes de código

```
// single-line comment
/* single-line comment */
/*
 * block comment
 */
statement;    // trailing comment
```

- Comentarios Javadoc:
  - Debe documentar clases, interfaces, metodos, and atributos.
  - Deben describir cada entidad de manera independiente de la implementación.

```
/**
 * Javadoc comment
 */
/** Javadoc comment */
```

Tabla 1. Convenio de Codificación Java I

Estructuras de Control	

Sentencias simples:
Cada línea debe tener una sola sentencia
Sentencias return:
No deben usarse parentesis para encerrar lo que se devuelve.
Sustituya algo como:
if (booleanExpression) {
return true;
} else {
return false;
}
por:
return booleanExpression;
Sustituya algo como:
if (condition) {
return x;
}
return y;
por:
return (condition ? x : y);
Sentencias while:
Use:
while (condition) {
statements;
}
Sentencias for:
Use:
for (initialization; condition; update) {
statements;
}
Declare la variable de control dentro del bucle:
for (int i = 0; i < size; ++i) {
statements;
}
Sentencias do-while:
Use:
do {
statements;
} while (condition);
	Sentencias if-else:
	Use:
	if (condition) {
	statements;
	}
	if (condition) {
	statements;
	} else {
	statements;
	}
	if (condition) {
	statements;
	} else if (condition) {
	statements;
	} else {
	statements;
	}
	Sentencias switch:
	Siempre debe haber clausula *defalult*.
	Utilice */* falls through */* cuandop uno de los *case* no terine en *break*
	Use:
	switch (condition) {
	case ABC:
	statements;
	/* falls through */
	case DEF:
	statements;
	break;
	default:
	statements;
	break;
	}
	Bloques try-catch
	Use:
	try {
	statements;
	} catch (ExceptionClass e) {
	statements;
	}

Tabla 2. Convenio de Codificación Java (cont.)

Prácticas de Programación: Directrices sobre cómo nombrar clases, atributos, métodos, variables y constantes, sobre cómo y dónde declarar cualquier entidad y, en general, todo un conjunto de buenas prácticas de programación en Java.

Documentación: Reglas sobre cómo documentar una clase Java, distinguiendo entre los comentarios Javadoc y aquellos de implementación necesarios para el entendimiento de la información almacenada en una clase y su comportamiento interno.

A modo de guía rápida de referencia, todos estos convenios a excepción de los referentes a archivos, pueden resumirse en las tablas 1 y 2.

El Plug-in Checkstyle es una herramienta que inspecciona el código fuente para asegurar que cumple con un conjunto de convenciones de código que incluyen las descritas en el apartado 1.1. Es importante destacar que no corrige automáticamente aquellos trozos de código que no cumplen los estándares, sino que informa de ello de la misma manera que el compilador informa de los errores de compilación. De esta manera es posible utilizar también la medida del número de veces que incumple el estándar como indicador de cómo de bien o mal está programada una clase.

Independientemente de lo anterior el funcionamiento del plug-in permite la continua comprobación de las normas a medida que se va escribiendo código y alerta inmediatamente cada

vez que se produce un incumplimiento de las normas. Por supuesto cada norma es individualmente configurable o incluso desactivable a petición del desarrollador.

Evidentemente existen diversas herramientas comerciales que son capaces de realizar también chequeo de estilo también configurable. En nuestro caso, hemos trabajado con Telelogic Logiscope que cuenta con la ventaja de dar soporte a múltiples lenguajes (Ada, C/C++, Java) aunque no se integra directamente en el entorno. El control de reglas de estilo es sólo un módulo (RuleCHecker) de esta herramienta que incluye también una suite de métricas (Audit) y el control de cobertura y al ayuda al diseño de pruebas (TestChecker) para los mismos lenguajes. Sobre este módulo comentamos en el siguiente apartado la actividad de pruebas necesaria para completar el proceso propuesto.

3. PRUEBAS DE SOFTWARE

En nuestra propuesta de proceso, debemos complementar la actividad de aseguramiento de calidad con una adecuada estrategia de pruebas. Es conocido que las inspecciones capturan un alto porcentaje de defectos y que no resultan detectables fácilmente con pruebas pero también ocurre al contrario. Los mejores resultados globales de detección de defectos se obtienen con una adecuada combinación de ambas técnicas.

La estrategia de pruebas incluida está inspirada también en la mejora de la eficiencia simultánea a la priorización y selección de intensidad de control. Incluye los siguientes puntos:

- Creación simultánea de casos de prueba de aceptación a partir de la definición de requisitos e interacción funcional descrita en UML como casos de uso complementados con diagramas de actividad [Fernández et al, 2003]. Esta posibilidad incentiva el cuidado de la especificación UML ya que, por el procedimiento descrito, se genera automáticamente un diseño bastante detallado de las pruebas de aceptación necesarias para cubrir la funcionalidad y la interacción solicitada.
- Diseño detallado de pruebas a partir de la información de diseño (como ya se ha descrito en el apartado 2 de este artículo) donde el análisis de Pareto del diseño (apartado 2.2) colabora en el establecimiento de un clasificación de casos en función del riesgo para ayudar a decidir la posible intensificación selectiva de controles incluida en nuestra filosofía de proceso.
- Ejecución de pruebas priorizada en función del riesgo final aceptable para el proyecto y el tiempo y los recursos disponibles. Aparte de la información de priorización obtenida de las anteriores etapas las buenas prácticas de pruebas recomiendan el control de la eficacia de las mismas. La manera más habitual y sencilla de lograrlo es el control de cobertura de la ejecución de pruebas. En nuestro caso, hemos contado con la ayuda del módulo TestChecker de Logiscope que permite controlar si las pruebas han pasado por todas las sentencias, todas las opciones de decisiones (caminos de decisión a decisión: DDP), etc. Esta información se puede obtener tanto de forma gráfica como valores estadísticos. Así, la gestión del proyecto puede decidir si completar el 100% de la cobertura de sentencias (como es habitual en proyectos para la Agencia Espacial Europea por normativa) o, dadas las circunstancias, llegar al 80% (asumiendo ciertos riesgos) o completar el 100% en componentes propensos a defectos y reducir el porcentaje al 70%, por ejemplo, en el resto. El proceso se resume en un ciclo que consiste en ejecutar casos seleccionados (por ejemplo, los generados desde el análisis y el diseño), comprobar la cobertura y añadir nuevos casos según se requiera para completar el criterio de pruebas.

4. CONCLUSIONES

Hemos presentado un proceso que combina la eficacia en la detección de defectos y el aseguramiento de calidad en el desarrollo de software con la eficiencia y la productividad en la aplicación del mismo. Este equilibrio permite optimizar los beneficios de la mejora de calidad conseguida a medio plazo con la minimización de costes necesarios para ello. Esta combinación permite reducir la resistencia del *management* en la promoción de esta filosofía a la vez que facilita que los desarrolladores se integren con facilidad en un proceso que evalúa y controla la calidad. En todo el proceso, la automatización y la adaptación a las circunstancias del proyecto y de la organización.

5. REFERENCIAS

Aldea, F., Gallego, I,1994 “Inspecciones de código en C”, I Jornadas Ibéricas de Calidad del Software, Lisboa, pp. 8.1-8-11

Archer C. y Stinson, 1995, M. Object Oriented software measures, Technical Report CMU/SEI-95-TR-002, abril 1995.

Brito e Abreu F. y Melo W., 1996, Evaluating the impact of object oriented design on software quality, Proceedings of 3rd international software metrics symp.

Basili V.R., Briy L. y Melo W., 1995, A Validation of OO design metrics as quality indicators, Technical report CS-TR-3443, 1995.

Borrajo Iniesta J., 1994, "A Toolkit and a Set of Metrics to Support Technical Reviews," en Software Quality Management II, Volume II: Building Quality into Software, M. Ross et al., eds., Computational Mechanics, Southampton, U.K. 1994, pp. 579-594.

Bolour A, 2003, Notes on the Eclipse Plug-in Architecture, Bolour Computing. (http://www.eclipse.org/articles/index.html)

Cartwright M. y Shepperd M, 1996, An empirical investigation of object oriented software in industry, Department of Computing, Talbot Campus, Bournemouth Uiversity, Technical Report TR 96/01.

Chidamber S.R. y Kemerer C.F., 1994, A metricsuite for object oriented design, IEEE transactions on Software Engineering, vol. 20, nº 6, junio, pp 467-493.

CMMi Product Team, 2002, CMMISM for Software Engineering, Version 1.1, Continuous Representation (CMMI-SW, V1.1, Continuous) CMU/SEI-2002-TR-028, Software Engineering Institute.

CMMi Product Team, 2002, CMMISM for Software Engineering, Version 1.1, Staged Representation (CMMI-SW, V1.1, Staged) CMU/SEI-2002-TR-029, Software Engineering Institute.

Fagan, M. E., 1976, Design and code inspections to reduce errors in program development, IBM Systems Journal, vo. 15, nº 3, pp. 182-211.

Fagan, M. E.,1986, Advances in software inspections, IEEE Transactions on Software Engineering, vol. 12, nº 7, pp. 744-751.

Fernández, L., Lara, P., Gutiérrez, C., 2003, , Actas de las VIII Jornadas de Innovación y Calidad del Software, ATI, pp. 26-32.

Garcia D, Harrison R., 2000, “Medicion en la Orientación a Objetos”en L. Fernadez y J. Dolado, Medición para la gestión en la Ingeniería del Software, Ra-Ma, pp. 75-92.

Goseva-Popstojanova K., 2003, Architectural-Level Risk Analysis Using UML, IEEE transactions on software engineering, vol. 29, nº 10, octubre 2003, pp. 946-960.

Grady, R. y Caswell, D., 1987, Software Metrics: Establishing a Company-Wide Program, Prentice-Hall.

Henderson-Sellers B., 1996, Object Oriented metrics measurements of complexity, Prentice-Hall.

IEEE, 1998, IEEE Std. 730-1998. Standard for Software Quality Assurance. IEEE Computer Society.

ISO, 1999, Information Technology. Software process assessment. Part 2. A reference model for processes and process capability, ISOIEC Tr 15504-2, ISO.

Lara P.J., Fernández L., 2004, ¿Cómo mejorar el estilo en desarrollos Java?, Dr Dobbs Journal España, abril 2004, pp 54-59

Lorenz M., Kidd J., 1994, Object Oriented Metrics, Prentice-Hall.

McCabe T., 1976, A complexity Metrics, IEEE transactions on software engineering, vol. 2, nº 4, diciembre, pp. 308-320

OMG, 2001, Unified Modelling Language Specification, version 1.4.

Sagardui, G. y Onandia, J., 2002, Plan de calidad del software en Fagor Automation, Actas de las VII Jornadas de Innovación y Calidad del Software, ATI, pp. 63-70 .

SUN, 1996, Code Conventions for the Java Programming Language. http://java.sun.com/docs/codeconv/.

Wang T., Hassan A., Guedem A. Abdelmoez W., Goseva-Popstojanova K, Ammar, H., 2003, Architectural level risk assessment tool based on UML Specification, Proceedings of the 25th International Conference on Software Engineering (ICSE’03)

Wheeler, D.A., Brykczynski, B., Meeson, R. A., 1996, Software Inspection. An industry best practice, IEEE Computer Society.