Estudio de la distribución docente de pruebas del software y refactoring para la incorporación de metodologías ágiles

Raúl Marticorena, Carlos López
Área de Lenguajes y Sistemas Informáticos
Universidad de Burgos
e-mail: {rmartico, clopezno}@ubu.es

Yania Crespo
Dpto. de Informática
Universidad de Valladolid
e-mail: yania@infor.uva.es

Resumen

En este trabajo se presenta una propuesta para incorporar en las titulaciones de informática la docencia relativa a pruebas del software y refactoring.

Se busca introducir de una manera gradual y acorde a las etapas de desarrollo las distintas técnicas y herramientas en dichos campos, frente a la problemática actual de intentar llevar a cabo metodologías ágiles en experiencias de laboratorio y trabajos final de carrera, debido a la falta de conocimientos en estos temas por parte de los alumnos.

El objetivo es estudiar una correcta distribución, exposición y experiencia práctica de ambos flujos de trabajo, para que el alumnado pueda aplicarlas a metodologías ágiles.

1. Introducción

En los actuales planes de estudio de las ingenierías técnicas en informática e ingenierías informáticas, la docencia respecto a las pruebas y refactoring se aborda desde un mero marco teórico.

Aunque este hecho se va reduciendo gracias a la inclusión de herramientas en laboratorio que asistan a dichos flujos de trabajo, por regla general, se suelen abandonar rápidamente para volver a metodologías más tradicionales en las que toman un rol menor.

Ésto se confirma en la escasa aplicación de dichos conceptos a los trabajos de fin de carrera, quedando como requisito a solicitar de manera explícita por parte del tutor. Normalmente las pruebas se incluyen finalmente de manera descriptiva sin realizar componentes de pruebas auto comprobables (self-testing) [3], y con la sensación por parte del alumno de cumplir un mero trámite.

Si además se intenta sumergir a los alumnos en metodologías ágiles, en últimos cursos de ingeniería informática, se observa que las dificultades se encuentran concentradas en estos temas, que a nuestro juicio, deberían haber sido cubiertos en cursos previos. Principalmente las dificultades surgen en el desarrollo dirigido por las pruebas y la aplicación de refactorización continua del código generado. Esta conclusión se ve confirmada por experiencias en laboratorios como se señala en [7].

Se propone un estudio para la inclusión de dichos temas de una manera progresiva a lo largo de los cursos teniendo en cuenta las troncalidades: *Ingeniería del Software (ISW)* y *Metodología y Tecnología de la Programación (MTP)*. El objetivo final es que los alumnos salgan preparados a nivel teórico y práctico en ambos campos. Ésto posibilitaría poder realizar prácticas y proyectos con metodologías ágiles en los que el alumno sólo se tiene que enfrentar con un nuevo proceso, pero no con nuevas actividades a aplicar en sus flujos de trabajo.

En lo que sigue el artículo se estructura así: en la Sección 2 se presenta el contexto actual de partida para incorporar dicho estudio, en la Sección 3 se presentan brevemente las pruebas del software seleccionadas, en la Sección 4 la distribución de las pruebas dentro de las troncalidades, en la Sección 5 se introduce el concepto de refactoring, en la Sección 6 su análisis y distribución, finalmente en la Sección 7, se concluye y se muestran las líneas de trabajo futuro.

2. Contexto inicial

Antes de arrancar la propuesta es conveniente indicar la situación inicial. En nuestro caso se parte de que los alumnos de la titulación técnica, en asignaturas de programación de nivel I, estudian un lenguaje imperativo como C de carácter estructurado. Es en cursos posteriores donde se empieza a hacer hincapié en el paradigma de orientación a objetos (OO), utilizando Java como

lenguaje de programación en asignaturas de programación de nivel II.

Dicha elección se mantiene en cursos posteriores, dadas las ventajas aportadas por el lenguaje así como la gran cantidad de herramientas de libre distribución que asisten al proceso de desarrollo.

Debido a ésto, nuestra propuesta inicial se centra en herramientas que trabajan sobre dicho lenguaje, garantizando que la práctica totalidad de alumnos dominan el lenguaje, pudiéndonos centrar en los aspectos concretos que aquí trataremos.

La propuesta no está ligada a un único lenguaje. Lo que se plantea es un guión de análisis e integración de técnicas que normalmente deben acompañarse de una realización concreta en herramientas. Ante un cambio de lenguaje a utilizar, simplemente quedaría pendiente una revisión de las mismas.

La importancia que han cobrado ambos aspectos, pruebas y refactoring, garantizan la existencia de herramientas para multitud de lenguajes. Aunque no todas las herramientas aquí señaladas se han integrado plenamente en laboratorios, la práctica totalidad ha sido probada e integrada en mayor o menor medida a proyectos fin de carrera o asignaturas de la troncalidad de *Sistemas Informáticos (SI)* del ciclo superior.

3. Pruebas del software

La importancia de las pruebas en cualquier proceso de desarrollo es señalado claramente tanto en procesos clásicos, como el modelo en cascada, así como en otros más actuales, como el proceso unificado de desarrollo [4] o metodologías ágiles.

El planteamiento actual es que las pruebas son parte integral e incluso dirigen el desarrollo [7]. Sin embargo el alumnado, dentro de los planes de estudio actuales, normalmente aborda este aspecto desde un punto de vista teórico, en asignaturas de *ISW* y más práctico, pero con escasa profundidad, en asignaturas de *MTP*. En ciclos superiores se vuelven a revisar estos conceptos pero con escaso bagaje práctico.

A continuación se hará una breve clasificación de los tipos de pruebas para abordar a posteriori las troncalidades adecuadas para su impartición teniendo en cuenta requisitos previos.

La clasificación básica de los distintos tipos de pruebas a abordar son: pruebas unitarias, pruebas de integración, pruebas de sistema y pruebas de aceptación (siendo conscientes de la existencia de otras posibles ampliaciones y clasificaciones).

4. Distribución de los tipos de pruebas en las troncalidades

Seleccionado este conjunto de pruebas se señalan a continuación las troncalidades donde se encuadran, dejando a consideración del propio plan de estudios las asignaturas implicadas. También se indican un conjunto de herramientas que pueden asistir a dichas actividades en los laboratorios para la obtención de componentes de prueba auto comprobables. La distribución final se resume en la Tabla 1.

Tipo de pruebas	Troncalidad	Ciclo	Ejemplo de herramienta / bibliotecas
Unitarias	*MTP*	1º	JUnit jcoverage
Integración	*ISW* *MTP*	1º	JUnit stubs Mock Objects
Sistema	*ISW* *MTP*	1º- 2º	Jacareto Cactus HttpUnit
Aceptación	*ISW*	1º- 2º	-

Tabla 1. Propuesta de inclusión de pruebas

4.1. Pruebas unitarias

Se centran en los aspectos de modularidad. Dentro de los actuales planes de estudio el concepto de modularidad y pruebas se enmarca en el descriptor de las asignaturas de la troncalidad *MTP*. Habitualmente, se desarrollan en asignaturas de programación de nivel II o III, en las que actualmente el concepto de módulo es la clase [10], debido a la tendencia actual del paradigma de programación OO.

Bajo esta premisa, las pruebas unitarias se abordan sobre clases principalmente con criterios de caja negra y caja blanca. En el primer caso, la idea de probar una interfaz sin conocer detalles de su implementación concreta coincide plenamente con uno de los principios básicos de la OO, la encapsulación. Por otro lado disponiendo del código fuente de los módulos, existe la posibilidad de aplicar técnicas de caja blanca midiendo los grados de cobertura de sentencia, de decisión, de

condición, etc. respecto a las pruebas de caja negra anteriormente diseñadas e implementadas.

Mientras que en clases de teoría se plantean los conceptos básicos, éstos se ven reafirmados por la experiencia obtenida en laboratorio. Las prácticas se realizan en un lenguaje OO ampliamente extendido como Java, junto con la utilización de un framework de pruebas unitarias como JUnit [6].

Al alumno se le plantea la construcción de módulos (clases) a partir de una especificación (`interface` Java). El alumno debe iniciar la programación en paralelo de la prueba unitaria y una implementación de dicha interfaz. Se solicita realizar pruebas completas de todos los métodos llegando a cubrir completamente la funcionalidad esperada del módulo, incidiendo sobre la ventaja de programar las pruebas de cara a su automatización.

Se ha observado que el hecho de iniciar la programación de la prueba en paralelo a la implementación, por parte de los mismos programadores, lleva a programar pruebas dependientes de la implementación. Esto puede ser solventado enfrentando sus pruebas a otras implementaciones para detectar este tipo de errores.

Desde el punto de vista de pruebas de caja blanca, se puede reafirmar la utilidad de automatizar las pruebas, en nuestra experiencia con JUnit, utilizando herramientas que permitan medir la cobertura de las pruebas diseñadas. Para ello se pueden utilizar herramientas como jcoverage [12], aunque desgraciadamente dado su carácter gratuito sólo mide la cobertura de sentencia y de decisión. Ahora sí, el trabajo se centra sobre la implementación.

Como se señala en [8] las pruebas de caja blanca nos pueden señalar código sin probar, lo que obliga a una revisión de la batería de pruebas seleccionada y eliminar código no necesario.

Mediante el uso de herramientas que permitan automatizar principalmente las pruebas de caja negra y caja blanca, se fija en el alumno la idea de probar de forma exhaustiva todos los módulos.

4.2. Pruebas de integración

Las pruebas de integración se llevan a cabo durante la construcción del sistema. Involucran a un número creciente de módulos, en contraste con la fase anterior de pruebas, probando subsistemas como conjuntos de módulos.

Según nos vamos acercando al sistema total, estas pruebas se van basando más y más en la especificación de requisitos del usuario. Las pruebas finales de integración cubren todo el sistema. En todas estas pruebas funcionales se siguen utilizando las técnicas anteriormente vistas.

Se pueden utilizar dos enfoques: ascendente o descendente. En el enfoque descendente se plantea ir de los niveles superiores (módulo raíz [10]) a los inferiores mientras que en el ascendente se prueban primero los módulos con menores dependencias hacia los que dependen de ellos.

De cara a la realización de las prácticas en laboratorio junto con el diseño del plan de pruebas y diseño de las mismas, bien en un plan ascendente o descendente, se puede utilizar de nuevo JUnit u otros frameworks de pruebas unitarias. En este caso la frontera de prueba unitaria se rompe puesto que, como se ha demostrado en la práctica, es difícil probar un módulo de una manera completamente aislada.

En el enfoque ascendente se puede trabajar en el laboratorio con los módulos de manera independiente. Cuando se va a probar uno de los módulos de mayor nivel se han probado ya los módulos de los que depende. Además el hecho de haber automatizado el proceso facilita las pruebas de regresión en casos de prueba con éxito (detección del bug).

En el enfoque descendente se trabaja con la problemática de que para probar uno de los módulos de alto nivel es necesaria la construcción de stub en un primer nivel o bien la utilización de Mock Objects [11] [8]. Obviamente esta segunda solución, aunque más compleja, puede ser necesaria y por lo tanto debe quedar supeditado a la disponibilidad de sesiones de prácticas en las asignaturas por parte del profesor responsable.

Dentro de los actuales planes de estudio un marco adecuado serían asignaturas en las troncalidades de *MTP* e *ISW*. En dichas troncalidades se aborda la integración de módulos y la programación a un alto nivel, con sistemas más complejos en los que el número de módulos y la interacción entre ellos sigue siendo fundamental.

4.3. Pruebas de sistema

Las pruebas de sistema coincidirían con las fases finales de las pruebas de integración. Se trata de probar el sistema en su totalidad. El método tradicional es pasar una serie de baterías de pruebas de manera manual por parte del participante con rol de probador. Se describen en una serie de plantillas de manera textual las entradas o pasos a realizar frente a las salidas esperadas.

La troncalidad más adecuada a nuestro juicio para realizar este tipo de pruebas es *ISW* en cuanto al planteamiento teórico de dichas pruebas, basado normalmente en las especificaciones.

Para el planteamiento de pruebas de sistema sobre producto final, en las asignaturas de programación de nivel III en la troncalidad de *MTP* se sugiere el desarrollo de interfaces gráficas de usuario junto con la programación de entornos distribuidos en web.

La realización automática en laboratorios se enfrenta a que normalmente este tipo de pruebas incluye la interacción con algún tipo de interfaz final hombre-máquina normalmente dirigido por eventos, bien aplicaciones autónomas con una interfaz compleja o bien con entornos distribuidos a través de la web.

La propuesta a realizar en laboratorios consiste en la utilización de herramientas CASE para las prácticas de *ISW* que permitan la generación de informes de pruebas. En asignaturas de *MTP* se propone utilizar entornos de pruebas que permitan la grabación y reproducción de eventos de usuario junto con la inserción de pruebas asociadas. Ejemplos de este tipo de entornos en Java los tenemos en Jacareto [14].

Por otro lado en la actualidad existen extensiones a frameworks como JUnit que permiten o facilitan las pruebas en entornos distribuidos como Cactus [1] o HttpUnit [4] .

4.4. Pruebas de aceptación

Estas pruebas son dirigidas por el cliente. En el caso que nos ocupa, este rol es asumido en su práctica totalidad por el profesor de acuerdo a las especificaciones (enunciado) de las prácticas solicitadas.

En este punto es conveniente señalar que aunque dichas pruebas se realicen de manera manual siguiendo algún tipo de plan, en el que normalmente se aplican criterios de caja negra o conjetura de pruebas para la elección de las mismas, es conveniente que el alumno vea que si ha seguido el plan propuesto de pruebas, la confiabilidad en la calidad del producto es evidente. La troncalidad en la que se encuadran este tipo de pruebas es *ISW*.

5. Refactoring

En la actualidad, el término refactoring está cobrando gran interés. Los motivos principales son: la incorporación como una de las prácticas en las metodologías de desarrollo ágiles como eXtreme Programming (XP) [1], y la incorporación en herramientas de desarrollo integradas.

Desde un punto de vista docente, es interesante analizar la diferencia de significados que toma la palabra refactorización cuando su usa como verbo y cuando se usa como nombre. Además es interesante conocer las dependencias con otras áreas de conocimiento relacionadas.

Como verbo, refactorizar es la actividad de reestructurar el software aplicando una serie de transformaciones sin cambiar su comportamiento externo. Esta perspectiva de la refactorización es más adecuada en asignaturas de segundo ciclo de la troncalidad de *ISW* dedicadas a explicar y aplicar procesos de desarrollo del software. Para poder ser aplicada es necesario tener un conocimiento de un catálogo de refactorizaciones y disponer de herramientas.

Como nombre, una refactorización es un cambio en la estructura interna de un programa para facilitar su comprensión y abaratar el esfuerzo de cambios futuros, sin modificar el comportamiento externo del sistema. Esta perspectiva es más adecuada en asignaturas de la troncalidad *MTP* que se enfrenten a problemas de código legado con la incorporación de nuevos requisitos, implicando modificaciones de código. En el caso de no disponer de una dedicación temporal plena para explicar un catálogo completo de refactorizaciones, el problema del docente es la selección de un conjunto de refactorizaciones de trabajo. Una vez seleccionadas, las transformaciones de código pueden ser guiadas a través de clases de teóricas donde se indica la refactorización a aplicar. Después las refactorizaciones se pueden realizar manualmente para ser comprobadas por una herramienta que asista el proceso. Uno de los

conocimientos previos deseables que se debería poseer para aplicar una refactorización es el conocimiento de frameworks que permitan definir y ejecutar casos de pruebas unitarias y de integración.

En [3] se mantiene una lista actualizada de las herramientas para realizar refactorizaciones. Su forma de distribución varía: entornos de desarrollo integrados como IntelliJ, o plug-in a conectar como JRefactory en un entorno comercial. Existen herramientas de refactoring para una gran variedad de lenguajes de programación: Java, Delphi, C#, VisualBasic, etc. Las herramientas evaluadas mantienen un catálogo reducido de refactorizaciones, proporcionando todas funcionalidades similares. En nuestro contexto particular se ha seleccionado la versión de evaluación de IntelliJ.

6. Distribución de las refactorizaciones en las troncalidades

En los catálogos existen gran cantidad de refactorizaciones, cuya comprensión exige una amplia variedad de requisitos de conocimientos. Este hecho hace necesario que la presentación de las refactorizaciones a los alumnos pueda ser distribuidas en distintas asignaturas dentro de las troncalidades, dependiendo del plan de estudios. Bajo esta situación, la elección de una refactorización a presentar a los alumnos requiere de un estudio de sus mecanismos[1] para extraer sus dependencias.

Dado que existen muchísimos tipos de dependencias de conocimientos previos para presentar todas las refactorizaciones, en este apartado se presenta un estudio del catálogo actualizado de Fowler [3], cuya intención es dar una visión global al docente a nivel de dependencias entre refactorizaciones y requisitos de conocimientos previos. Se pretende facilitar la elección de las refactorizaciones conforme su contexto particular (tiempo de dedicación, plan de estudios, asignatura, etc.) Los resultados de dicho estudio se muestran en la Tabla 3. A continuación se presentan los conocimientos necesarios para comprender dicha tabla.

Se diferencian dos tipos de dependencias entre las refactorizaciones:

- Dependencias de similitud: las refactorizaciones tienen motivaciones similares.
- Dependencias de uso: para realizar una refactorización puede ser necesario la aplicación de otras refactorizaciones.

Esta clasificación permite establecer un orden de relevancia de las refactorizaciones en función del número de dependencias. También puede servir para discriminar las refactorizaciones a nivel de esfuerzo, ya que la explicación de una refactorización puede llevar implícito la presentación de las refactorizaciones dependientes a nivel de uso. Para capturar estas relaciones en la Tabla 3 se han incluido dos columnas: similitud y uso. En las celdas correspondientes a cada refactorización se incluyen los identificadores de las refactorizaciones dependientes.

Los requisitos de conocimientos necesarios para presentar una refactorización hacen referencia a los conceptos básicos y avanzados asociados a la orientación objetos. Se consideran conocimientos básicos: clases, métodos, atributos, modificadores de acceso, herencia, polimorfismo, instrucciones, expresiones y visibilidad. Se consideran conocimientos avanzados: reflectividad, patrones de diseño, aserciones y excepciones. En la Tabla 3 sólo se representan los requisitos de conocimientos avanzados.

Esta distribución del conocimiento se utiliza para obtener una clasificación de las refactorizaciones por niveles de conocimiento. Se han diferenciado tres niveles: no aplicable (-), conceptos básicos (1) y conceptos avanzados (2). La interpretación de los valores de las columnas de diseño e implementación se corresponden con los distintos niveles.

Aunque las refactorizaciones tienen siempre como finalidad modificar código existente, se ha considerado interesante desde una perspectiva docente intentar diferenciar aquellas que podrían ser más propias en asignaturas de *ISW* y en asignaturas de *MTP*. Esta diferenciación nos lleva a la siguiente clasificación de refactorizaciones:

- Refactorizaciones de diseño que consultan y transforman abstracciones exclusivamente de diseño.
- Refactorizaciones de implementación que consultan y transforman abstracciones propias de los lenguajes de programación como son las instrucciones.

[1] Elemento de la plantilla de definición de refactorizaciones de Fowler

El conjunto de refactorizaciones de implementación es un conjunto completo respecto a las refactorizaciones del catálogo. El conjunto de refactorizaciones de diseño no es completo. Esta característica se representa en la tabla por la presencia del guión (-) en las celdas de la columna de diseño. En la Tabla 2, se muestra la distribución de refactorizaciones en las troncalidades de *ISW* y *MTP* del primer y segundo ciclo. Este estudio permite clasificar las refactorizaciones por tipos conforme a los valores de las dos últimas columnas de la Tabla 3.

Nivel Diseño	Nivel Impl.	Troncalidad	Ciclo	Ejemplo de herramienta / bibliotecas
-	1	*MTP*	1º	IntelliJ
1	1	*ISW* *MTP*	1º	IntelliJ
-	2	*ISW* *MTP*	1º-2º	IntelliJ
2	2	*ISW*	1º-2º	IntelliJ

Tabla 2. Propuesta de inclusión de refactorizaciones

Identificador	Nombre[2] de la refactorización	Similitud	Dependencias de uso	Requisitos de conocimientos avanzados	Diseño	Implementación
1	Add Parameter	51			1	1
2	Change Bidirectional Association to Unidirectional		81,77		1	1
3	Change Reference to Value		50	Factory Method	2	2
4	Change Unidirectional Association to Bidirectional				1	1
5	Change Value to Reference		56,51	Factory Method	2	2
6	Collapse Hierarchy		40,41,43, 42		1	1
7	Consolidate Conditional Expression		18		-	1
8	Consolidate Duplicate Conditional Fragments				-	1
9	Convert Dynamic to Static Construction			Reflectividad	-	2
10	Convert Static to Dynamic Construction	17,21		Reflectividad	-	2
11	Decompose Conditional		65		-	1
12	Duplicate Observed Data		77	Observer	2	2
13	Encapsulate Collection		18,36,51		1	1
14	Encapsulate Downcast		13		-	1
15	Encapsulate Field				1	1
16	Extract Class		35,36		1	1
17	Extract Interface				1	1
18	Extract Method		80,72		-	1
19	Extract Package		21,34,10		1	1
20	Extract Subclass		56,51,43, 42,77,54, 36	Factory Method	2	2
21	Extract Superclass		40,41,39,51, 81,18,22	Template Method	2	2
22	Form Template Method		41,51	Template Method	2	2
23	Hide Delegate			Delegate	2	2
24	Hide Method				1	1
25	Inline Class		17,35,36		1	1
26	Inline Method				-	1
27	Inline Temp				-	1
28	Introduce Assertion		18	Aserciones	2	2
29	Introduce Explaining Variable				-	1
30	Introduce Foreign Method				-	1

[2] La decisión de no traducir los nombres de las refactorizaciones y patrones, es debido a que los autores consideran que forman parte del vocabulario asociado al proceso de refactorización, considerándolas palabras clave.

31	Introduce Local Extension				1	1
32	Introduce Null Object			Null Object	1	1
33	Introduce Parameter Object		1,36,18	Inmutable	1	1
34	Move Class		19		1	1
35	Move Field		15,77		1	1
36	Move Method				1	1
37	Parameterize Method				1	1
38	Preserve Whole Object				1	1
39	Pull Up Constructor Body		18		1	1
40	Pull Up Field		77		1	1
41	Pull Up Method		40,77		1	1
42	Push Down Field				1	1
43	Push Down Method				1	1
44	Reduce Scope of Variable				-	1
45	Remove Assignments to Parameters				-	1
46	Remove Control Flag				-	1
47	Remove Double Negative	26			-	1
48	Remove Middle Man			Delegate	2	2
49	Remove Parameter	1,51			1	1
50	Remove Setting Method			Inmutable	1	1
51	Rename Method				1	1
52	Replace Array with Object				-	1
53	Replace Assignment with Initialization				-	1
54	Replace Conditional with Polymorphism		18,36		-	1
55	Replace Conditional with Visitor	54	73,74,18	Visitor	-	2
56	Replace Constructor with Factory Method			FactoryMethod Reflectividad	-	2
57	Replace Data Value with Object		5		1	1
58	Replace Delegation with Inheritance		51	Delegate	1	1
59	Replace Error Code with Exception			Excepciones	-	2
60	Replace Exception with Test			Aserciones	-	2
61	Replace Inheritance with Delegation			Delegate	1	1
62	Replace Iteration with Recursion				-	1
63	Replace Magic Number with Symbolic Constant				-	1
64	Replace Method with Method Object				-	1
65	Replace Nested Conditional with Guard Clauses		7		-	1
66	Replace Parameter with Explicit Methods				1	1
67	Replace Parameter with Method		49		1	1
68	Replace Record with Data Class				1	1
69	Replace Recursion with Iteration		81		-	1
70	Replace Static Variable with Parameter		1		-	1
71	Replace Subclass with Fields		56,26		1	1
72	Replace Temp with Query	80	78		-	1
73	Replace Type Code with Class		51		1	1
74	Replace Type Code with State/Strategy			State	2	2
75	Replace Type Code with Subclasses				1	1
76	Reverse Conditional				-	1
77	Self Encapsulate Field				-	1
78	Separate Query from Modifier				-	1
79	Split Loop		18,72		-	1
80	Split Temporary Variable				-	1
81	Substitute Algorithm				-	1

Tabla 3. Estudio del catálogo de refactorizaciones

A la hora de interpretar la tabla, tomando como ejemplo la refactorización (18) Extract Method se deduce que: no existen similitudes con otras refactorizaciones; está relacionada con (80) Split Temporary Variable y (72) Replace Temp with Query; no tiene requisitos de conocimientos avanzados; en diseño no es aplicable (requiere información a nivel de instrucción) y en implementación es aplicable requiriendo un nivel de conocimiento básico.

7. Conclusiones y líneas de trabajo futuro

El objetivo buscado con esta propuesta es que en últimos cursos de ingeniería superior en asignaturas vinculadas a las troncalidades de *ISW* y *SI*, se pueda afrontar de manera satisfactoria la incorporación de metodologías ágiles sin encontrarse problemas de base, señalando las troncalidades implicadas.

A la hora de llevar a cabo un método ágil, como por ejemplo XP, ni el profesor ni los alumnos deben verse entorpecidos por un problema añadido y que ha podido ser resuelto con una correcta planificación de contenidos a lo largo de la carrera.

Se puede abordar directamente la realización de prácticas en desarrollo de proyectos en los que temas como la programación (habitualmente en orientación a objetos), pruebas y refactorización continuada de código, puedan ser realizados de una manera natural. No siendo ésto finalmente el objetivo, ni tampoco un obstáculo, sino simplemente un instrumento más del proceso.

La introducción de temas sobre pruebas y refactoring, y su aplicación a troncalidades, se ha desglosado en función de su complejidad de forma genérica. Ésto favorece la presentación gradual de los mismos a lo largo de los distintos cursos de un plan de estudios, quedando pendiente la asignación concreta a las asignaturas.

En nuestro conocimiento, la carencia de trabajos en esta línea nos deja abierta el estudio de las relaciones entre refactoring y pruebas, y evaluación del éxito o fracaso de una adaptación completa de la propuesta.

Este planteamiento junto con otros ya realizados en otros trabajos previos [9], respecto a la inclusión de patrones de diseño, nos llevan a completar y establecer como línea de trabajo futuro la integración y realización completa de todos estos elementos, acorde a las necesidades actuales en temas de ingeniería del software.

Referencias

[1] Apache Software Foundation. Jakarta Cactus. http://jakarta.apache.org/cactus/. Última visita 26 de enero del 2004.

[2] Beck, K. *Una Explicación de la programación Extrema. Aceptar el cambio.* Addison Wesley, 2002.

[3] Fowler, M. *Refactoring. Improving the Design of Existing Code.* Addison Wesley, 2000. http://www.refactoring.com

[4] HttUnit. http://httpunit.sourceforge.net/ Última visita 26 de enero de 2004.

[5] Jacobson, I. et al., *El Proceso Unificado de Desarrollo de Software.* Addison Wesley, 2000.

[6] JUnit. *Testing Resources for Extreme Programming*. http://www.junit.org. Última visita 26 de enero de 2004.

[7] Letelier, P. & Canós, J.H. *Incorporando Extreme Programming como Metodología de Desarrollo en un Laboratorio de Sistemas de Información.* JENUI 2003, pp 257- 264.

[8] Link, J. et al. *Unit Testing in Java. How Tests Drive the Code.* Morgan Kaufmann, 2003.

[9] López, C. et al. *Inclusión de patrones de diseño en un plan de estudios de Ingeniería Técnica en Informática de Gestión.* JENUI 2003, pp 265-272.

[10] Meyer, B. *Construcción de Software Orientado a Objetos 2ª Edición.* Prentice Hall, 1998.

[11] Mock Objects. http://www.mockobject.com. Última visita 26 de enero de 2004.

[12] Morgan, P. et al., jcoverage http://www.jcoverage.com/. Última visita 26 de enero de 2004.

[13] Sommerville, I. et al. *Ingeniería del Sofware. 6ª Edición*. Addison Wesley, 2002.

[14] Spannagel, C. *Jacareto A Capture&Replay Framework*. http://www.ph-ludwigsburg.de/mathematik/personal/spannagel/jacareto/. Última visita 26 de enero del 2004.