*Revista*
*Española de*

# Innovación, Calidad e Ingeniería del Software

REICIS

Volumen 3, No. 3, diciembre, 2007

**Web de la editorial: www.ati.es**
**E-mail: reicis@ati.es**
**ISSN: 1885-4486**



Publicado por la Asociación de Técnicos en Informática

# Revista Española de Innovación, Calidad e Ingeniería del Software (REICIS)

# Contenidos

REICIS

# Generación e implementación de pruebas del sistema a partir de casos de uso

Javier J. Gutiérrez, María J. Escalona, Manuel Mejías, Arturo H. Torres, Jesús Torres
Departamento de Lenguajes y Sistemas Informáticos. Universidad de Sevilla
{javier, escalona,risoto, jtorres}@lsi.us.es

**Abstract**

Nowadays, there are several papers and book chapters which describe how to generate test cases from use cases. However, there is a lack of approaches showing how to implement those test cases into automate executable test cases. This paper introduces an architecture based on the UML testing profile and a set of steps to implement test objectives, which are defined as use case scenarios and operational variables.

**Keywords:** Functional testing, test automation, use case testing

**Resumen**

En la actualidad, existe una amplia cantidad de trabajos y capítulos de libros que proponen cómo obtener pruebas a partir de requisitos funcionales definidos como casos de uso. Sin embargo, existe una carencia de referencias que muestren cómo implementar dichas pruebas en código de pruebas automáticas. Este trabajo presenta una arquitectura, basada en el perfil de pruebas de UML, y un conjunto de pasos para la implementación en código ejecutable de pruebas de casos de uso definidas mediante escenarios y variables operacionales.

**Palabras clave:** Pruebas funcionales, automatización de las pruebas, pruebas de casos de uso.

## 1. Introducción

Un código sin fallos no tiene por qué resultar en un sistema sin fallos. Por ello, las pruebas del sistema cobran una gran importancia dentro de la etapa de pruebas. Las pruebas del sistema engloban tantos tipos de prueba como tipos de requisitos se puedan definir y probar con la ejecución del sistema o con la verificación de sus elementos. Habitualmente, incluye requisitos funcionales, de seguridad, de rendimiento, de fiabilidad, de accesibilidad, etc.

Este trabajo se centra en la prueba de requisitos funcionales definidos como casos de uso, es decir, en desarrollar un conjunto de casos de prueba que verifiquen si el comportamiento definido en dichos casos de uso ha sido correctamente implementado en el sistema. Al abordar la automatización de las pruebas de sistemas (y de otros tipos de pruebas), se pueden identificar tres niveles de automatización [1]. El primero es la

automatización de la generación de casos de prueba a partir de los requisitos, el segundo es la automatización de la ejecución de los casos de prueba y el tercero es la automatización de la comprobación de sus resultados. Existe un amplio número de artículos y capítulos de libros que describen cómo generar pruebas a partir de casos de uso (primer nivel de automatización). Sin embargo, varios trabajos comparativos y casos prácticos [2] [3] [4], exponen que la práctica totalidad de estas propuestas sólo generan descripciones de los escenarios y valores de prueba de una manera narrativa. La aportación original de este trabajo es un conjunto de pasos para la generación de código de prueba que permita llevar esos resultados al segundo y tercer nivel de automatización. Esto significa comprobar, de manera automática, si el sistema implementa el comportamiento definido en sus casos de uso. El resultado final será una prueba del sistema definida como código ejecutable con un conjunto de asertos para evaluar el resultado obtenido.

En este trabajo se van a utilizar dos artefactos diferentes, aunque complementarios, referentes a las pruebas del sistema. El primer artefacto se obtiene a partir de los casos de uso y contiene descripciones textuales de los escenarios a probar y sus valores de prueba (un ejemplo se muestra en la sección 2). El segundo artefacto será la implementación de dichos escenarios y valores de prueba en un fragmento de código ejecutable (como se muestra en la sección 4). Para evitar confusiones entre ambos artefactos, al primero de ellos se le llamará objetivo de prueba y, al segundo, caso de prueba.

La organización de este trabajo se describe a continuación. La sección 2 resume una serie de trabajos previos de los autores para la automatización de la generación de pruebas automáticas a partir de casos de uso. Los resultados obtenidos serán el punto de partida para la codificación de pruebas automáticas. Después, la sección 3 expone una arquitectura para la implementación de pruebas del sistema (a partir de los elementos definidos en el perfil de pruebas de UML [5]) y un conjunto de pasos para la redacción de pruebas como código ejecutable. La sección 4 muestra un caso práctico. Finalmente, la sección 5 describe otros trabajos relacionados, las conclusiones y los trabajos futuros.

## 2. Generación de objetivos de prueba a partir de casos de uso

A partir de los resultados de los estudios comparativos citados en la sección 1, se ha identificado que las dos técnicas más utilizadas para generar objetivos de prueba son la

definición de escenarios y la aplicación de técnicas de partición de dominios de variables (ambas documentadas en [6]). Estas dos técnicas se resumen en los siguientes párrafos.

La primera técnica, basada en los escenarios de un caso de uso, consiste en transformar el comportamiento de un caso de uso a un grafo y aplicar un criterio de recorrido para identificar distintos escenarios y utilizarlos como casos de prueba. Algunas propuestas que usan esta técnica son [7] [8] y [9].

La segunda técnica, basadas en técnicas de partición de dominios, consiste en identificar los puntos que pueden variar entre dos escenarios o realizaciones de un caso de uso (en [10] a estos puntos se les llaman variables operacionales), definir sus dominios, dividirlos en particiones, agrupando en cada partición aquellos valores para los que el caso de uso defina el mismo comportamiento, y, por último, construir una tabla con todas las combinaciones posibles de las particiones de cada variable operacional. Cada fila de la tabla, es decir, cada combinación de particiones, es un caso de prueba. Algunas propuestas que usan esta técnica son [10] o [11] para los puntos de variabilidad en casos de uso de familias de productos.

Existen también algunas técnicas complementarias a las dos técnicas anteriores. Por ejemplo, en la referencia [12] se proponen técnicas de análisis del lenguaje natural para la transformación de un caso de uso a grafo.

En este trabajo se definen los objetivos de prueba con la información de las dos técnicas anteriores. Un objetivo de prueba está compuesto de una secuencia de pasos, sin alternativa posible, de un conjunto de variables operacionales (cada una tomando valores de una partición concreta) y las pre y poscondiciones relevantes para dicho escenario. Un ejemplo de objetivo de prueba se muestra en la tabla 1. A continuación se resumen los pasos para obtener escenarios y combinaciones de particiones de un caso de uso automáticamente.

Para la generación de los escenarios de prueba, en primer lugar, se construye un diagrama de actividades a partir de los pasos de la secuencia principal y alternativas del caso de uso. En el diagrama de actividades, se identifican las acciones realizadas por el sistema y las acciones realizadas por los actores. Después, se realiza un recorrido mediante un criterio de suficiencia y cada camino del diagrama de actividades, será un escenario del caso de uso y, por tanto, una prueba potencial.

Se ha desarrollado una herramienta de código libre llamada ObjectGen, aún en fase experimental (www.lsi.us.es/~javierj/objectgen/), la cual permite obtener de manera

automática el diagrama de actividades y la lista de caminos a partir de un conjunto de casos de uso. Los algoritmos utilizados se describen con más detalle en [13]. Un ejemplo se muestra en la sección 4.1.

Para la generación de combinaciones de particiones se toma como punto de partida el diagrama de actividades obtenido en el párrafo anterior. A partir de él se identifican las variables operacionales presentes en el caso de uso. A partir de las variables operacionales, se aplica el método de Categoría-Partición (llamado por sus siglas: CPM) [14], considerando cada variable operacional como una categoría, para definir distintas particiones en los dominios de las variables operacionales tomando como base los nodos decisión del diagrama de actividades.

<table>
<tr><td>Name</td><td colspan="3">UC-01. 09</td></tr>
<tr><td>Test objective</td><td colspan="3">Main scenario</td></tr>
<tr><td>Priority</td><td colspan="3">No.</td></tr>
<tr><td>Initial state</td><td colspan="3">No.</td></tr>
<tr><td>Action sequence</td><td colspan="3"><table>
<tr><th>Id</th><th>Action description</th><th>Variables</th></tr>
<tr><td>1</td><td>The visitor selects the option for introduce a new link.</td><td>No.</td></tr>
<tr><td>2</td><td>The system recovers all the stored categories and it asks for the information of a link.</td><td>No.</td></tr>
<tr><td>D01</td><td>Not(there was an error recovering the categories) AND Not(there were not categories found)</td><td>V01 = P02<br>V02 = P02</td></tr>
<tr><td>D03</td><td>Not(cancel this operation)</td><td>V03 = P02</td></tr>
<tr><td>3</td><td>The visitor introduces the information of the new link.</td><td>V04 = P02</td></tr>
<tr><td>D04</td><td>Not(the link name, category or URL are empty)</td><td>No.</td></tr>
<tr><td>4</td><td>The system stores the new link.</td><td>V05 = P02</td></tr>
<tr><td>D05</td><td>Not(there is an error storing the link)</td><td></td></tr>
<tr><td></td><td>End.</td><td></td></tr>
</table></td></tr>
<tr><td>Final state</td><td colspan="3">A new links has been stored into the system.</td></tr>
<tr><td>Test values</td><td colspan="3"><table>
<tr><th>Instance</th><th>Partition</th><th>Sample value</th></tr>
<tr><td>V4_I1</td><td>P02</td><td>Nme: Test link<br>Category: default<br>URL: www.test.com</td></tr>
</table></td></tr>
<tr><td>Notes</td><td colspan="3">No.</td></tr>
</table>

Tabla 1. Ejemplo de objetivo de prueba

Para reducir el número de combinaciones y evitar combinaciones imposibles de realizar en la práctica, se identifican, a continuación, un conjunto de restricciones. Dichas

restricciones se expresan de la siguiente manera: *si una variable operacional V01 toma valor en su partición P01, entonces las variables V02…Vn no necesitan tomar valores.* Por último se calculan todas las posibles combinaciones teniendo en cuenta las restricciones identificadas.

También se ha desarrollado otra herramienta de código libre llamada ValueGen, también en fase experimental, la cuál permite obtener de manera automática un primer conjunto de variables operacionales, particiones, restricciones y combinaciones a partir de los casos de uso. Los algoritmos propuestos se describen en más detalle en [15]. Un ejemplo también se muestra en la sección 4.1.

La combinación de un camino en el diagrama de actividades con el conjunto de particiones de las cuáles las variables operacionales presentes en el camino deben tomar sus valores será un objetivo de prueba. El mismo objetivo de prueba que se utilizará en el caso práctico de la sección 4 se muestra en la tabla 1. En dicho objetivo, se puede apreciar cómo se incluye un escenario del caso de uso de la tabla 3 (sección 4), las variables operacionales y particiones necesarias, un ejemplo de valores de prueba, así como las precondiciones y poscondiciones presentes en el caso de uso. Este objetivo de prueba se muestra en inglés, dado que las dos herramientas utilizadas sólo soportan casos de uso en este idioma.

## 3. Implementación de pruebas del sistema

A continuación se describe cómo implementar los resultados obtenidos en la sección anterior. En la sección 3.1, se describe una arquitectura genérica para pruebas del sistema a partir de los elementos definidos en el perfil de pruebas de UML (llamado a partir de aquí por sus siglas en inglés: UMLTP). En la sección 3.2, se describe un conjunto de pasos para la implementación de pruebas funcionales del sistema, tomando como base la arquitectura definida en la sección 3.1.

### 3.1. Una arquitectura de pruebas del sistema

La arquitectura para la ejecución y comprobación automática de pruebas del sistema se muestran en la figura 1. Esta arquitectura está basada en la arquitectura xUnit [15], la cual es una de las más populares e implementada en la mayoría de las herramientas. A continuación, se describen brevemente los elementos de la arquitectura de prueba y su relación con los elementos definidos en el UMLTP cuando existe.

La clase *UserEmulator*, define el elemento que interactúa con el sistema bajo prueba utilizando las mismas interfaces que un actor. Si, por ejemplo, el sistema a prueba es una aplicación *web* (como en el caso práctico) la clase *UserEmulator* será capaz de interactuar con el navegador *web* para indicarle la URL qué tiene que visitar, rellenar formularios, pulsar enlaces, etc. Este elemento está estereotipado como un *Test Component*, ya que el UMLTP define de esta manera a los elementos auxiliares de los casos de prueba.

A partir de la interacción de dicha clase con el sistema, se obtendrán uno o varios resultados (clase *SystemOutput*); por ejemplo, en el caso del sistema web, se obtendrá código HTML. El perfil de pruebas de UML no define ningún elemento para representar los resultados obtenidos del sistema a prueba, por lo que se ha modelado con una clase sin estereotipar.

La clase *UserInterface* representa las interfaces externas del sistema bajo prueba. A partir de este tipo de elementos pueden definirse propiedades relevantes para las pruebas de cada interfaz mediante el estereotipo *sut* definido en el UMLTP.

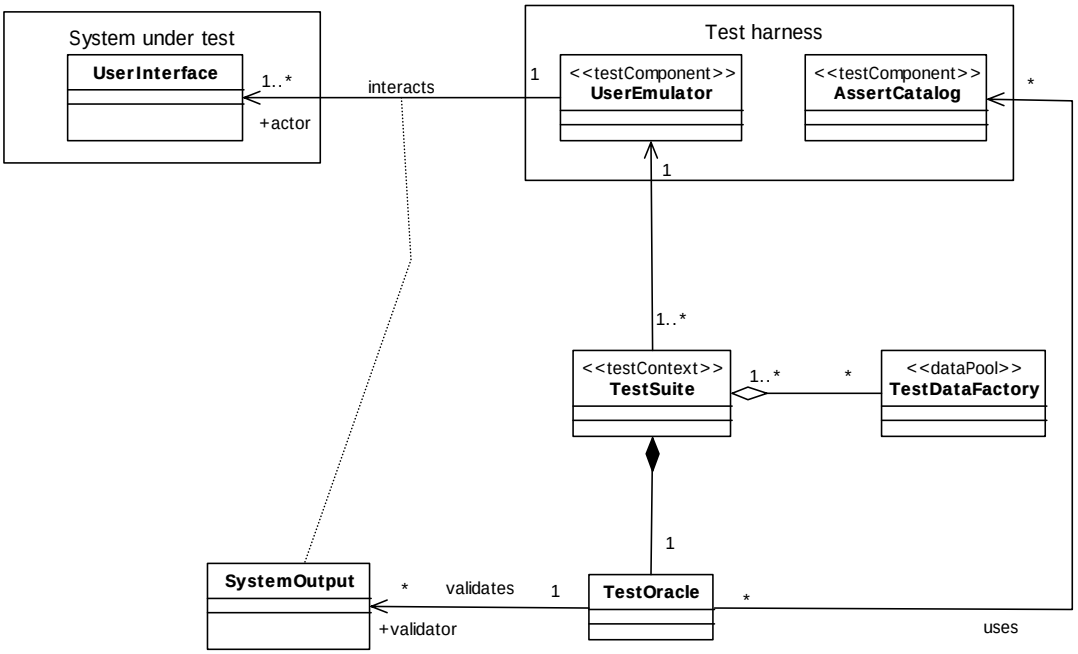


Figura 1. Arquitectura de pruebas

La clase *TestSuite*, estereotipada como un *Test Context* del perfil de pruebas de UML, representa un conjunto de casos de prueba (definidos en la siguiente sección). En el perfil de pruebas, todo *Test Context* tiene un elemento *Arbiter* y un elemento *Scheduler*, sin embargo, ambos elementos no se van a utilizar en esta propuesta (los proporcionará el *test harness*, sección 4.2), por lo que no aparecen en la figura 1.

La clase *AssertCatalog*, también estereotipada como *Test Component*, define la colección de asertos a disposición de los casos de prueba para determinar si el resultado obtenido del sistema a prueba (clase *SystemOutput*) es correcto o no. Tanto el *UserEmulator* como el *AssertCatalog* se han agrupado en un clasificador que representa el *Test Harness*, dado que estos dos elementos, suelen ser comunes para todas las prueba de diversos sistemas y existe gran variedad de ofertas en el mercado tanto de pago como libres y gratuitas.

El perfil de pruebas define el elemento *ValidationAction* (acciones de validación) que, como su nombre denota, permite indicar una operación concreta para determinar el resultado de un caso de prueba. Sin embargo, el perfil de prueba no define ningún elemento genérico para denotar todas las acciones de validación de un caso de prueba. Por este motivo se ha introducido dicho elemento en el marco de trabajo mediante la clase no estereotipada *TestOracle*. Esta clase sirve de contenedor de todas las acciones de validación (expresadas mediante los asertos del elemento *AssertCatalog* sobre el resultado obtenido y almacenado en el elemento *SystemOutput*) que determinarán el veredicto de los casos de prueba del contexto de prueba.

Por último, la clase *TestDataFactory* (estereotipada como *Data Pool* según el UMLTP) contendrá un conjunto de métodos, con el estereotipo *Data Selector*, para crear y/o seleccionar los distintos valores de prueba según las distintas particiones identificadas en las variables de los casos de uso.

Como se ha mencionado al principio, esta arquitectura es similar a la arquitectura xUnit, utilizada principalmente en pruebas unitarias. La principal diferencia estriba en que, en una prueba unitaria, la propia prueba invoca al código en ejecución, mientras que una prueba funcional del sistema necesita un mediador (el elemento *UserEmulator*) que sepa cómo manipular su interfaz externa. En la siguiente sección, se describen un conjunto de pasos para implementar los elementos de la figura 1 a partir de los objetivos de prueba.

### 3.2. Implementación de los casos de prueba

Para la implementación de los casos de prueba se van a aplicar los patrones *isolated test* y *test method* documentados en [15]. El primer patrón indica que cada caso de prueba debe ser independiente de los demás. Por ello, como se ha visto anteriormente, cada objetivo de prueba se codificará como un caso de prueba, ya que los objetivos de prueba no tiene dependencias entre ellos y cualquier objetivo de prueba puede verificarse independientemente de los demás. El segundo patrón indica que cada caso de prueba debe ser implementado como un método. Esto concuerda con la definición del UMLTP dónde un caso de prueba no es un elemento arquitectónico, por tanto no se ha definido en la sección anterior, sino la definición de un comportamiento como un método (estereotipado como *Test Case*) dentro de un elemento *Test Suite* (estereotipado como *Test Context*).

El comportamiento genérico de un caso de prueba que se adopta con mayor frecuencia se describe, entre otros trabajos, también [15] y se lista en la tabla 2. El segundo paso de la tabla 2 ha sido refinado en este trabajo con los pasos 2.1 y 2.2.

1. Invocación del *set up* del caso de pruebas.
2. Invocación del método de prueba
   2.1. Ejecución de una acción sobre el sistema.
   2.2. Comprobación del resultado de la acción (*opcional*).
3. Invocación del *tear down* del caso de pruebas.

Tabla 2. Comportamiento genérico de un caso de prueba

Como se puede ver en la tabla 2, cada método de prueba tiene asociados otros dos métodos. El primero, método *set-up*, establece el estado adecuado del sistema para la ejecución de la prueba. El segundo, *tear down*, restaura el estado original del sistema. Estos métodos también estarán definidos en el *Test Suite*.

Cada caso de uso tendrá asociado un elemento *TestSuite* (figura 1). Dicha *suite* contendrá los métodos de pruebas y métodos asociados de todos los escenarios de dicho caso de uso. A continuación se describe cómo implementar estos métodos y los demás elementos del modelo de la figura 1.

Como se ha visto, en cada uno de los pasos del objetivo de prueba se indica si es realizado por un actor o por el sistema a prueba. Esta información es relevante a la hora de la codificación de los métodos de prueba de la *suite*. Todos los pasos realizados por un actor se traducirán, en el código del caso de prueba, a una interacción entre el caso de prueba y el sistema a través del elemento *UserEmulator*.

El *test oracle* de un caso de prueba será el conjunto de acciones de validación obtenidas, principalmente, a partir de los pasos realizados por el sistema. Por ejemplo, en aquellos pasos en los que el sistema realice una petición de información u órdenes a los actores, se deberán definir asertos que permitan evaluar que todos los elementos de la interfaz son los correctos. En función de las poscondiciones del objetivo de prueba pueden añadirse asertos adicionales para verificar que el estado en que queda el sistema es el indicado en la poscondición. Las acciones de validación se implementan utilizando el catálogo de asertos proporcionado por el *test harness* sobre el resultado devuelto por el *UserEmulator*.

Además de los escenarios, se han utilizado variables operacionales para la definición de los valores de prueba necesarios (tabla 1). Antes de su implementación, estas variables se van a clasificar en tres tipos, cada uno de los cuáles se implementará de manera distinta.

El primer tipo lo componen aquellas variables operacionales que indican un suministro de información al sistema por parte de un actor. Para cada variable de este tipo se definirá una nueva clase cuyos objetos contendrán los distintos valores de prueba para dicha variable. Estas clases se codificarán aplicando el patrón *Value Object* [15], por lo que tendrán un método constructor parametrizado para establecer el valor de sus atributos y un conjunto de métodos *set* para acceder al valor de dichos atributos. Además, para cada partición del dominio identificada, el elemento *TestDataFactory* tendrá, al menos, un método estereotipado como *Data Selector* que devolverá un valor de prueba perteneciente a dicha partición. Un ejemplo de una variable operacional de este tipo se muestra en el caso práctico, en la tabla 6 (a).

El segundo tipo lo componen aquellas variables operacionales que indican una selección entre varias opciones que un actor tiene disponible. En este caso, no tiene sentido implementar estas variables como métodos del *TestDataFactory*. En su lugar, dicha selección se implementará directamente como parte del código que implementa la

interacción entre el actor y el sistema. Un ejemplo de una variable operacional de este tipo se muestra en el caso práctico en la tabla 6 (a).

El tercer tipo lo componen aquellas variables operacionales que indican un estado del sistema, bien una información que el sistema almacena o una configuración concreta. Para implementar el método de *set up* del caso de prueba, se debe escribir el código necesario para establecer adecuadamente el valor de las variables operacionales que describen los estados del sistema, o bien comprobar que dichos valores son los adecuados. De manera análoga, el método *tear down* debe restaurar dichos valores a sus estados originales. Además, dicho método debe eliminar, si es procedente, la información introducida por el caso de prueba en el sistema durante la ejecución del caso de prueba. Varios ejemplos de variables operacionales de este tipo se muestran en el caso práctico también en la tabla 6 (a).

A continuación se muestra un caso práctico de todo lo expuesto en las secciones anteriores.

## 4. Un caso práctico

Como caso práctico, para ilustrar todo lo visto, se ha tomado una aplicación web para la gestión de un catálogo de enlaces on-line. Esta aplicación permite realizar consultas a partir de los enlaces almacenados, o bien añadir nuevos enlaces al catálogo. En primer lugar se aplicará lo visto en la sección 2 para obtener un conjunto de objetivos de prueba a partir de un caso de uso (sección 4.1). Después, se definen las características del *Test harness* utilizado (sección 4.2). Finalmente, se aplica lo visto en la sección 3 para implementar un caso de prueba a partir de un objetivo de prueba (sección 4.3). Los artefactos del sistema bajo prueba se han definido en inglés, ya que el español no está soportado por las herramientas ObjectGen y ValugeGen.

### 4.1. Objetivos de prueba

El caso de uso de la tabla 3, describe la introducción de un nuevo enlace en el sistema. Como complemento, se muestra también el requisito de almacenamiento de información que describe la información manejada por cada enlace (tabla 4). Los patrones usados se han tomado de la metodología de desarrollo NDT [16] [17].

A partir del caso de uso, y de manera automática, se han generado un diagrama de actividades (figura 2).

| | | | |
|---|---|---|---|
| **Name** | UC-01. Add new link | | |
| **Precondition** | No | | |
| **1Main sequence** | 1 | The user selects the option for introducing a new link. | |
| | 2 | The system recovers all the stored categories and it asks for the information of a link. | |
| | 3 | The user introduces the information of the new link. | |
| | 4 | The system stores the new link. | |
| **Alternatives** | 3.1 | At any time, the user can cancel this operation, then this use case ends. | |
| **Errors** | 2.1 | If there was an error recovering the categories, then the system shows an error message and this use case ends. | |
| | 2.2 | If there were not categories found, then the system shows and error message and this use case ends. | |
| | 3.2 | If the link name, category or link URL is empty, then the system shows an error message with the result of repeat step 2. | |
| | 4.1 | If there is an error storing the link, then the system shows an error message and this use case ends. | |
| **Post condition** | A new link is stored. | | |

Tabla 3. Caso de uso a prueba

| | | |
|---|---|---|
| **Name** | SR-01. Link. | |
| **Specific data** | *Name* | *Nature* |
| | Identifier | Integer |
| | Name | String |
| | Category | Integer |
| | URL | String |
| | Description | String |
| | Approved | Boolean |
| | Adding date | Date |
| **Restrictions** | The identifier must be unique.<br>Name, category, URL and approved are mandatory<br>Default value for approved is false (0) and for date is the actual date. | |

Tabla 4. Requisito de almacenamiento complementario al caso de uso

Dado que el caso de uso presenta bucles no acotados, con un número potencialmente infinito de repeticiones (el número de veces que el actor introduce un enlace erróneo), el criterio de cobertura elegido consiste en obtener todos los caminos posibles para una repetición de ninguna o una vez de cada uno de los bucles.

Todos los escenarios obtenidos con este criterio, y traducidos al español, se listan en la tabla 5 (a). Para este caso práctico, seleccionamos el escenario 09 para su implementación. Este escenario se describe en detalle en la tabla 5 (b) (dado que el caso de uso se ha redactado en inglés, su escenario principal también se muestra en inglés).

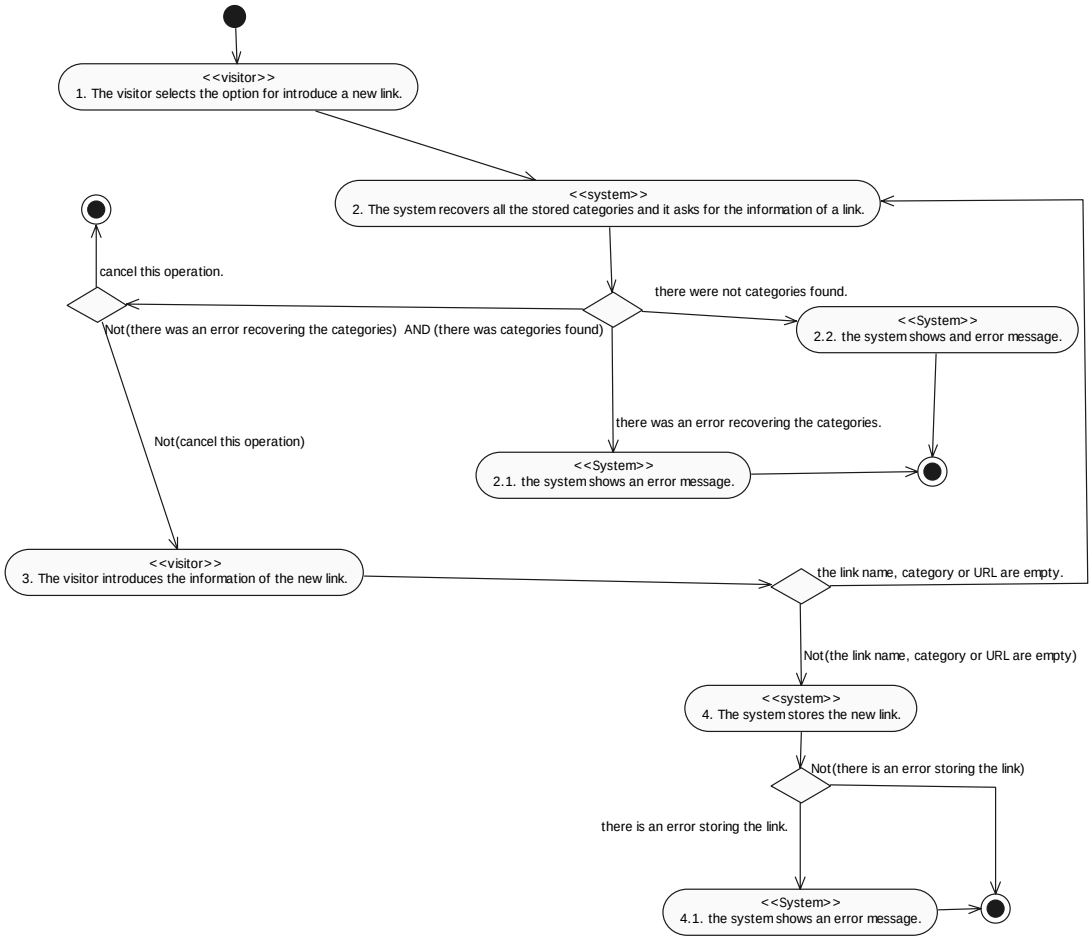


Figura 2. Diagrama de actividades del caso de uso.

A continuación, se ha aplicado CPM tal y como se ha descrito en la sección 2. Todas las variables operacionales encontradas automáticamente se enumeran en la tabla 6 (a) y las sus particiones se enumeran en la tabla 6 (b), ambas traducidas al español. En este caso, no se ha continuado refinando el conjunto de particiones aunque para algunas variables, como V04, sí podrían identificarse particiones adicionales para, por ejemplo, distinguir entre los distintos motivos por los que un enlace puede ser erróneo.

Todas las combinaciones válidas calculadas mediante la herramienta ValueGen se muestran en la tabla 7. Un "*" significa que dicha variable no toma valor en esa combinación (a causa de las restricciones identificadas como se ha mostrado en la sección

2). Como algunas variables y particiones (en concreto cuando V04 = P01) provocan un bucle y que se vuelvan a evaluar algunas variables por segunda vez, se ha identificado esta segunda instancia, añadiendo "_2" al final del nombre de la variable en la tabla 7.

| Escenario | Descripción |
|---|---|
| 01 | Aparece un error recuperando las categorías. |
| 02 | El usuario cancela la operación. |
| 03 | El usuario introduce un enlace incorrecto y, después, aparece un error recuperando las categorías. |
| 04 | El usuario introduce un enlace incorrecto y, después, el usuario cancela la operación. |
| 05 | El usuario introduce un enlace incorrecto y, después, aparece un error al almacenar el enlace. |
| 06 | El usuario introduce un enlace incorrecto y, después, el usuario introduce un enlace correcto. |
| 07 | El usuario introduce un enlace incorrecto y, después, el sistema no encuentra ninguna categoría. |
| 08 | Aparece un error al almacenar el enlace. |
| *09* | *El usuario introduce un enlace correcto (camino principal).* |
| 10 | El sistema no encuentra ninguna categoría. |

(a)

| Paso | Descripción |
|---|---|
| 01 | The user selects the option for introduce a new link. |
| 02 | The system recovers all the stored categories and it asks for the information of a link. |
| 03 | Not(there was an error recovering the categories) AND Not(there were not categories found) |
| 04 | Not(cancel this operation) |
| 05 | The user introduces the information of the new link. |
| 06 | Not(the link name, category or URL are empty) |
| 07 | The system stores the new link. |
| 08 | Not(there is an error storing the link) |

(b)

Tabla 5. (a) Escenarios del caso de uso y (b) escenario principal

| Variable | Descripción | Tipo |
|---|---|---|
| V01 | Error al recuperar categorías. | Sistema. |
| V02 | Categorías encontradas. | Sistema. |
| V03 | Opción del usuario | Actor. |
| V04 | Datos del enlace | Información. |
| V05 | Error al almacenar el enlace, | Sistema.. |

(a)

| Variable | Particiones |
|---|---|
| V01 | P01: Ocurre un error.<br>P02: No ocurre un error. |
| V02 | P01: No se encontraron categorías.<br>P02: Sí se encontraron categorías. |
| V03 | P01: Cancela la operación.<br>P02: No cancela la operación. |
| V04 | P01: El nombre, categoría o URL están vacías.<br>P02: El enlace es correcto. |
| V05 | P01: Error almacenando el enlace.<br>P02: No ocurre un error. |

(b)

Tabla 6. (a) Variables y (b) particiones identificadas para el caso de uso

| Id | Combinación |
|---|---|
| 1 | V01:P01 V02:* V03:* V04:* V05:* |
| 2 | V01:P02 V02:P01 V03:* V04:* V05:* |
| 3 | V01:P02 V02:P02 V03:P01 V04:* V05:* |
| 4 | V01:P02 V02:P02 V03:P02 V04:P01 V01_2:P01 V02_2:* V03_2:* V04_2:* V05:* |
| 5 | V01:P02 V02:P02 V03:P02 V04:P01 V01_2:P02 V02_2:P01 V03_2:* V04_2:* V05:* |
| 6 | V01:P02 V02:P02 V03:P02 V04:P01 V01_2:P02 V02_2:P02 V03_2:P01 V04_2:* V05:* |
| 7 | V01:P02 V02:P02 V03:P02 V04:P01 V01_2:P02 V02_2:P02 V03_2:P02 V04_2:P02 V05:P01 |
| 8 | V01:P02 V02:P02 V03:P02 V04:P01 V01_2:P02 V02_2:P02 V03_2:P02 V04_2:P02 V05:P02 |
| 9 | V01:P02 V02:P02 V03:P02 V04:P02 V05:P01 |
| 10 | V01:P02 V02:P02 V03:P02 V04:P02 V05:P02 |

Tabla 7. Combinaciones válidas para las variables y particiones identificadas

Para la implementación del escenario de éxito (escenario 09), todas las variables operacionales deben tener un valor perteneciente a las particiones P02, lo cuál coincide con la combinación 10.

La combinación de los pasos del escenario 09 con sus variables operacionales y particiones, así como la precondición y poscondición del caso de uso, dan lugar al objetivo de prueba mostrado en la tabla 1. En la siguiente sección, se definen los elementos pertenecientes al *test harness* usados en este caso práctico.

## 4.2. *Test harness*

Tal y como se describió en la figura 1, el *test harness* permite simular el comportamiento del usuario y ofrecer un conjunto de asertos para evaluar el resultado obtenido.

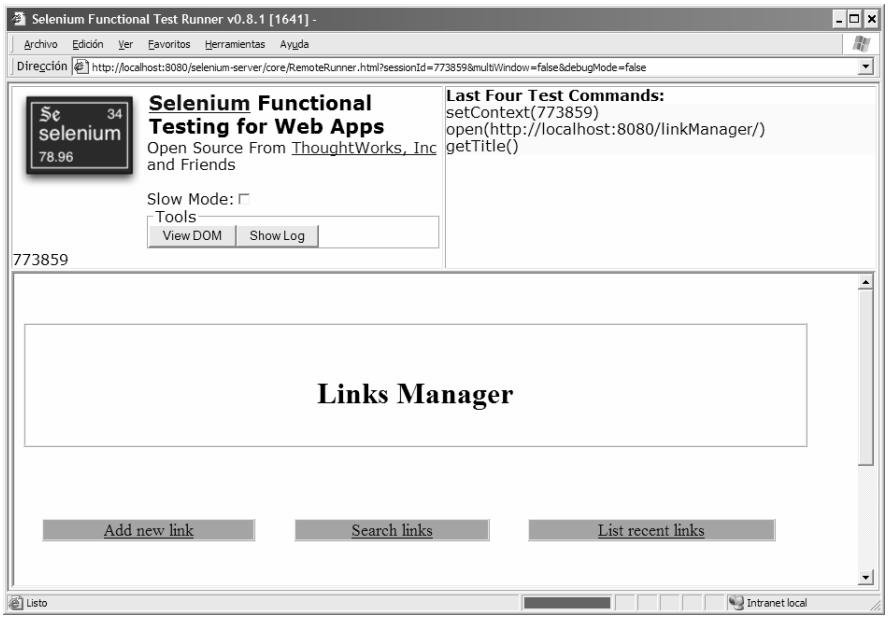


Figura 3. Ejecución del caso de prueba con la herramienta Selenium.

En este caso práctico, al ser el sistema bajo prueba una aplicación *web*, es necesario que el *test harness* sea capa de comunicarse con el navegador *web* y pueda ejecutar asertos sobre el código HTML recibido como respuesta. La herramienta elegida ha sido Selenium (www.openqa.org/selenium), gratuita y de código abierto. Como se puede ver el la figura 3, Selenium permite abrir un navegador *web* e interactuar con él de la misma manera que lo haría un actor humano. Esta herramienta está basada en la popular herramienta JUnit, por lo que Selenium trabaja con la misma arquitectura, incorpora el mismo conjunto de asertos que JUnit y, además, funciones adicionales para acceder a los resultados visualizados en el navegador *web*. Además, Selenium permite escribir las pruebas en varios lenguajes, aunque en este caso práctico se ha utilizado el lenguaje Java.

### 4.3. Implementación de un caso de prueba

En primer lugar se ha implementado el elemento *TestDataFactory* y los valores de prueba. De la tabla 6 (a), sólo la variable V04: Datos del enlace, hace referencia a una información suministrada desde el exterior al sistema durante la ejecución de la prueba (como se ha visto en la sección 3.2). Se ha desarrollado una clase (clase *Link* en la figura 4), aplicando el patrón *value object* resumido con anterioridad, para representar los diferentes enlaces. Los atributos han sido identificados a partir del requisito de almacenamiento de la tabla 4. Por cada partición posible, se ha añadido un método estático al elemento *TestDataFactory* para obtener un objeto enlace con valores adecuados a su categoría. Como se mencionó en la sección 4.2, sólo se han tenido en cuenta las dos particiones principales: enlaces correctos e incorrectos (tabla 6 (b)), sin entrar en más subdivisiones, por lo que sólo hay dos métodos *Data Selector*.

Después, se ha definido el *test suite* (clase *TestUC01* en la figura 4) con un método de pruebas, un método *set-up* y un método *tear-down* por cada escenario. En la figura 4, se muestra sólo los métodos correspondientes al escenario de la secuencia principal del caso de uso (escenario 09).

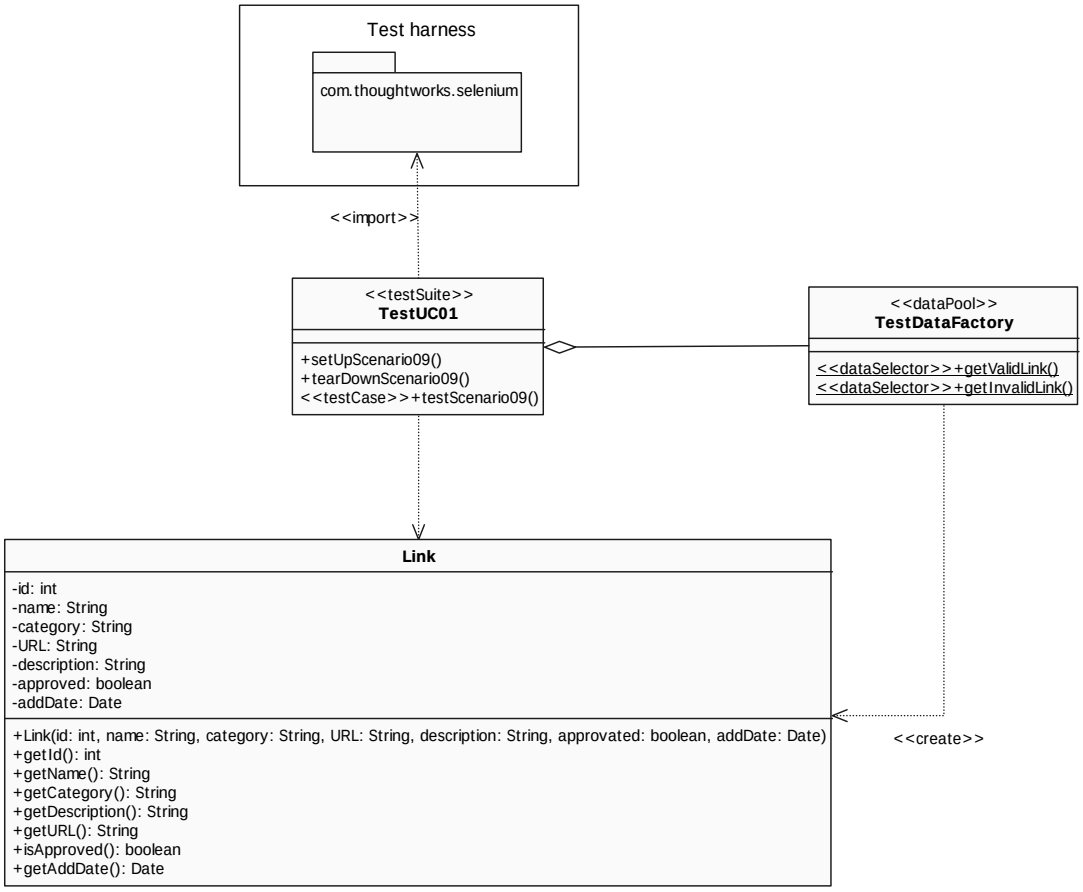


Figura 4. Implementación del caso de pruebas

A continuación, se toman todos los pasos ejecutados por el actor humano y se traducen a código Java para la herramienta Selenium. Dicha traducción se ha realizado a mano y el código resultante se muestra en la tabla 8. La referencia *s* apunta a una instancia del *test harness* que, para este caso concreto, es una instancia de una clase llamada *Selenium*. El método *click* realiza una pulsación sobre el elemento indicado (bien en el enlace *AddNewLink*, bien en el botón *submit* del formulario), y el método *type*, rellena un control de un formulario.

Como se mencionó en la sección 4.2, la variable V03: Opción del usuario, se implementa como parte del código del caso de prueba (última línea en cursiva que aparece en la tabla 8).

Por las características específicas de Selenium, es necesario añadir esperas a que la página se cargue antes de continuar mediante el método *waitForPageLoad(tiempo)*. Esto

significa que, en la implementación del paso 1, la prueba esperará como máximo 5 segundos a que la página se cargue, si no, dará la prueba como no superada.

| **Paso:** | **Código:** |
|---|---|
| 01: The user selects the option for introduce a new link. | `s.click("AddNewLink");`<br>`s.waitForPageToLoad("5000");` |
| 05: The user introduces the information of the new link. | `Link l = TestDataPool.getValidLink();`<br>`s.type("name", l.getName());`<br>`s.type("URL", l.getURL());`<br>`s.type("description", l.getDescription());`<br>*`s.click("addEvent_0");`* |

Tabla 8. Traducción a código ejecutable de los pasos realizados por el usuario en el escenario principal

Después, se escriben los asertos a partir de los pasos que realiza el sistema. El código resultante para la herramienta Selenium se muestra en la tabla 9. En el primer conjunto de asertos (paso 02) se verifica el título de la página obtenida como respuesta y que estén presentes todos los controles necesarios del formulario. En el segundo conjunto de asertos (paso 07), se verifica el título de la página obtenida y que no esté presente el mensaje de error. Además, en el paso 07, se ha incluido un aserto adicional que verifique la poscondición, es decir, que compruebe que el enlace está correctamente almacenado en el sistema tal. Para ello se ha añadido un método auxiliar *isLinkStored* (última línea del paso 07, tabla 9), el cuál indica si el enlace suministrado como parámetro ha sido almacenado por el sistema.

| **Paso:** | **Código:** |
|---|---|
| 02: The system recovers all the stored categories and it asks for the information of a link. | `assertEquals("Add new link form", sel.getTitle());`<br>`assertTrue(s.isTextPresent("Name*:"));`<br>`assertTrue(s.isElementPresent("addEvent_name"));`<br>`assertTrue(s.isTextPresent("Category*:"));`<br>`assertTrue(s.isElementPresent("addEvent_category"));`<br>`assertTrue(s.isTextPresent("URL*:"));`<br>`assertTrue(s.isElementPresent("addEvent_URL"));`<br>`assertTrue(s.isTextPresent("Description:"));`<br>`assertTrue(s.isElementPresent("addEvent_description"));`<br>`assertTrue(s.isTextPresent("Date:"));`<br>`assertTrue(s.isElementPresent("addEvent_date"));`<br>`assertTrue(s.isElementPresent("addEvent_0"));` |
| 07: The system stores the new link. | `assertEquals("Links manager", s.getTitle());`<br>`assertFalse(s.isTextPresent("Error storing new link"));`<br>`assertTrue(isLinkStored(TestDataPool.getValidLink()));` |

Tabla 9. Traducción a código ejecutable del *test oracle* para el escenario principal

Después, el código de las tablas 8 y 9 se reordena siguiendo la secuencia de pasos del escenario de la tabla 5 (b). Los pasos 03, 04 y 06 no aparecen en ninguna de las dos tablas

porque no son acciones de la prueba sino definiciones de variables y particiones y, por tanto, ya han sido tenidas en cuenta al elegir las particiones adecuadas.

Finalmente, la implementación del método de *set-up* consiste en comprobar que todas las variables operacionales del sistema, mostradas en la tabla 6 (a), tengan un valor de la partición P02. Es decir, comprobar que hay categorías almacenadas en el sistema y que no hay ninguna circunstancia que ocasione un error al recuperar las categorías o insertar el nuevo enlace. La implementación del método de *tear down*, consiste en la restauración del conjunto original de enlaces almacenado en el sistema.

El código resultado puede descargarse de la misma página que hospeda las herramientas ObjectGen y ValueGen.

## 5. Conclusiones

En este trabajo se ha mostrado un proceso para implementar casos de prueba a partir de objetivos de prueba para casos de uso. Existe un número muy limitado de trabajos que aborden este proceso desde una perspectiva similar. Por ello, se han adaptado varias ideas (arquitectura xUnit, patrones, etc.) de la codificación de pruebas unitarias. Otros trabajos relacionados con la generación de pruebas ejecutables se citan en los siguientes párrafos.

En [15] se pueden encontrar distintos patrones para la implementación de casos de prueba unitaria. Varias de sus ideas se han utilizado en este trabajo. En [18] se muestra un ejemplo de generación automática de código de pruebas para pruebas unitarias, basado en técnicas de reflexión aplicadas sobre el código original. En este caso, los objetivos de prueba se definen cómo combinaciones de valores de prueba a verificar por las pruebas generadas. En [19] se describe un caso práctico sobre la prueba de sistemas móviles a través de GUI utilizando como punto de partida modelos y lenguajes específicos de dominio. En concreto, para dicho caso práctico se describió un lenguaje de modelado específico, el cuál se implementó con posterioridad mediante un conjunto de eventos de la interfaz gráfica. Siguiendo esta línea, una posible extensión del trabajo presentado en este artículo consistiría en definir un *script* de prueba en un lenguaje independiente que después pueda ser implementado en distintas herramientas. Un ejemplo preliminar de esto se puede encontrar en [20].

Como se ha mencionado a lo largo de este trabajo, se ha conseguido automatizar la generación de pruebas mediante dos herramientas (el primer nivel de automatización mencionado en la introducción). Para la generación automática de código ejecutable, como se ha visto en el caso práctico, necesita tener muchos datos específicos de la aplicación y de la interfaz, sino definidos de una manera procesable automáticamente y enriquecerlos con una semántica para que el sistema sepa lo que son. Nuestros próximos trabajos apuestan por el uso de modelos de diseño, asociados a los casos de uso y con la convención de nombres, desde los requisitos hasta la implementación, para poder aplicar generación automática.

## Agradecimientos

Este trabajo ha sido realizado en el marco del Ministerio de Ciencia y Educación de España, bajo el Programa de Investigación, Desarrollo e Innovación, proyecto QSimTec (TIN2007-67843-C06-03) y REPRIS (TIN2005-24792-E).

## Referencias

[1] Meudec C. "ATGen: Automatic Test Data Generation Using Constraint Logic Programming and Symbolic Execution". *ACM SIGSOFT Software Engineering Notes.* Vol. 23, nº 2, pp. 53-62, 1998.

[2] Denger, C. Medina M. *Test Case Derived from Requirement Specifications*. Fraunhofer IESE Report, 2003.

[3] Gutiérrez, J.J., Escalona M.J., Mejías M., Torres, J. "Generation of test cases from functional requirements. A survey". *4º Workshop on System Testing and Validation*, Alemania, 2006.

[4] Roubtsov S. y Heck P. "Use Case-Based Acceptance Testing of a Large Industrial System: Approach and Experience Report". *Testing: Academic & Industrial Conference (TAIC PART*). Windsor, Reino Unido, 2006.

[5] Object Management Group. *The UML Testing Profile*. Disponible en: www.omg.org . 2003.

[6] Díaz E. Blanco R. Tuya J. "Los Métodos de Generación de Casos de Prueba y su Automatización". En: Tuya J. Ramos I. Dolado J. (editores). *Técnicas cuantitativas para la Gestión en la Ingeniería del Software*. Editorial Netbiblo, 2007.

[7] Fernández L. Lara P. Escribano J.J. "Gestión Cuantitativa del Diseño de Casos de Prueba Mediante Casos de Uso". En: Tuya J. Ramos I. Dolado J. (editores). *Técnicas cuantitativas para la Gestión en la Ingeniería del Software*.. Editorial Netbiblo. 2007.

[8] Naresh, A., "Testing From Use Cases Using Path Analysis Technique", *International Conference On Software Testing Analysis & Review*, EE.UU, 2002.

[9] Ruder A., "UML-based Test Generation and Execution". *Rückblick Meeting*, Berlin. 2004.

[10] Binder, R.V. *Testing Object-Oriented Systems*. Addison Wesley. 1999.

[11] Bertolino, A., Gnesi, S. "PLUTO: A Test Methodology for Product Families2. *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg. 3014 / 2004. pp 181-197. 2004.

[12] Boddu R., Guo L. y Mukhopadhyay S., "RETNA: From Requirements to Testing in Natural Way", *12th IEEE International Requirements Engineering RE'04*. 2004.

[13] Gutiérrez J.J. Escalona M.J. Mejías M. Torres J., "Modelos y algoritmos para la generación de objetivos de prueba". *XIII Jornadas sobre Ingeniería del Software y Bases de Datos JISBD*. Sitges, España, 2006.

[14] Ostrand T. J. y Balcer M. J., "The Category-Partition Method". *Communications of the ACM*, pp 676-686, 1988.

[15] Beck K., *Test-Driven Development: By Example*, Addison-Wesley, 2002.

[16] Escalona M.J., *Models and Techniques for the Specification and Analysis of Navigation in Software Systems*. Ph. European Thesis. University of Seville. Sevilla, España, 2004.

[17] Escalona M.J., Gutiérrez J.J., Villadiego D., León A. y Torres A.H., "Practical Experiences in Web Engineering". 1*5th International Conference On Information Systems Development*. Budapest, Hungria, 2006.

[18] Polo M., Tendero S.y Piattini M, "Integrating Techniques and Tools for Testing Automation", *Journal of Software Testing, Verification and Reliability*, vol.17, pp. 3-39, 2006.

[19] Katare M. et-al. "Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach". *Testing: Academic & Industrial Conference (TAIC PART*). Windsor, Reino Unido, 2006.

[20] Gutiérrez J.J., Escalona M.J., Mejías M. y Reina A.M., "Modelos de pruebas para pruebas del sistema". *Taller de Desarrollo de Software Dirigido por Modelos. XIII Jornadas sobre Ingeniería del Software y Bases de Datos JISBD*, Sitges, España, 2006.