# Implementación de pruebas del sistema. Un caso práctico

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

**Resumen**

Las pruebas funcionales del sistema permiten verificar que el sistema en desarrollo satisface sus requisitos funcionales. Existen una amplia cantidad de trabajos y capítulos de libros que proponen cómo obtener objetivos de prueba a partir de requisitos funcionales expresados como casos de uso. Sin embargo, existe una carencia de trabajos que muestren cómo implementar dichos objetivos en pruebas automáticas. Este trabajo presenta un ejemplo, basado en el perfil de pruebas de UML, para la implementación en código ejecutable de objetivos de prueba definidos mediante escenarios y variables operacionales.

## 1. Introducción

Un código sin fallos no tiene por qué derivar en un sistema sin fallos. Por ello, la fase de prueba de sistemas cobra una gran importancia en el desarrollo de sistemas software. El proceso de prueba a nivel de sistema engloba tantos tipos de prueba como tipos de requisitos se puedan definir y probar con la ejecución del sistema o mediante la verificación de sus distintos elementos. Habitualmente, esto engloba requisitos funcionales, de seguridad, de rendimiento, de fiabilidad, de accesibilidad, etc.

Al abordar la automatización de las pruebas de sistemas, se pueden identificar, a grandes rasgos, tres niveles claramente separados [10]. El primero es la automatización del proceso de 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 comprobación de sus resultados. Este trabajo se centra en el segundo nivel.

Existen un amplio número de trabajos y artículos que describen cómo generar objetivos de prueba a partir de casos de uso. Sin embargo, varios trabajos comparativos y casos prácticos, como [3] [6] y el [11], exponen que la mayoría de estas propuestas no son capaces de generar pruebas directamente ejecutables sobre el sistema. La aportación original de este trabajo es un proceso para la generación de código de prueba que permita, de manera automática, comprobar si el sistema implementa el comportamiento definido en sus casos de uso. En concreto, este trabajo se centra en la implementación de pruebas que simulan el comportamiento de actores humanos sobre un sistema dotado de interfaces gráficas.

La generación automática de objetivos de prueba, ya ha sido tratada en trabajos anteriores, cómo [7]. Un resumen de este proceso se incluye en la sección 2, dado que dichos objetivos serán el punto de partida para la implementación de las pruebas del sistema. Después, la sección 3 expone una arquitectura para la implementación de pruebas del sistema y un conjunto de buenas prácticas 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.

En esta sección se resumen trabajos anteriores de los autores para obtener objetivos de prueba, los cuáles son el punto de partida para desarrollar pruebas automáticas según se describe en las secciones 3 y 4.

El perfil de pruebas de UML [14] define un objetivo de pruebas como un elemento con un nombre concreto que define qué debe ser probado. En el contexto de pruebas del sistema a partir de los casos de uso, un objetivo de prueba puede

expresarse como un escenario del caso de uso. Dicho escenario estará compuesto de una secuencia de pasos, sin alternativa posible, y de un conjunto de valores de prueba, así como las precondiciones y poscondiciones relevantes para dicho escenario.

Para la generación de los escenarios de prueba, en primer lugar, se construye un diagrama de actividades a partir de la secuencia principal y secuencias erróneas y alternativas del caso de uso. En el diagrama de actividades, se estereotipa las acciones realizadas por el sistema y las acciones realizadas por los actores. Después, se realiza un análisis de caminos y, cada camino del diagrama de actividades, será un escenario del caso de uso y, por tanto, un potencial objetivo de prueba. Se ha desarrollado una herramienta de código libre llamada ObjectGen, aún en fase experimental (www.lsi.us.es/~javierj/objectgen/) la cuál permite obtener de manera automática el diagrama de actividades y la lista de caminos a partir de un conjunto de casos de uso. Este proceso se describe en más detalle en [7].

Otra alternativa, o técnica complementaria, es la definición de conjuntos de valores de prueba a partir de las variables operacionales de un caso de uso. El término variable operacional se define en [2] como cualquier elemento de un caso de uso que puede variar entre dos instancias de dicho caso de uso. A partir de las variables operacionales, se aplica el proceso de Categoría-Partición [12] (considerando cada variable operacional como una categoría) para definir distintas particiones en los dominios de las variables operacionales, valores de prueba para cada partición y restricciones o dependencias entre particiones. Los autores de este trabajo también han desarrollado una herramienta de código libre llamada ValueGen, aún en fase experimental, la cuál permite obtener de manera automática un primer conjunto de variables operacionales y particiones para los casos de uso, los cuáles pueden refinarse manualmente con posterioridad. Este proceso se describe en más detalle en [15].

La combinación de un camino en el diagrama de actividades con el conjunto de particiones de las cuáles las variables operacionales presentes deben tomar sus valores será un objetivo de prueba.

## 3. Implementación de pruebas del sistema

Una vez obtenidos los objetivos de prueba de manera automática con las técnicas y herramientas resumidas en la sección anterior, es posible implementar casos de prueba que cubran dichos objetivos. A continuación, 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. En la sección 3.2, se describe un conjunto de buenas prácticas para la implementación de pruebas funcionales del sistema.

### 3.1. Una arquitectura de prueba del sistema.

La arquitectura para la ejecución y comprobación automática de pruebas del sistema se muestran en la figura 1. A continuación, se describen brevemente los elementos de la arquitectura de prueba.

Esta arquitectura es similar a la arquitectura necesaria para la automatización de otros tipos de pruebas, como las 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.

La clase *UserInterface* representa la interfaz externa del sistema bajo prueba y ha sido estereotipada de acuerdo con la definición del perfil de prueba de UML.

La clase *UserEmulator*, define el elemento que podrá interactuar con el sistema utilizando las mismas interfaces que una persona real. 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. A partir de la interacción de dicha clase con el sistema, se obtendrán uno o varios resultados (clase *TestResult*), 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.

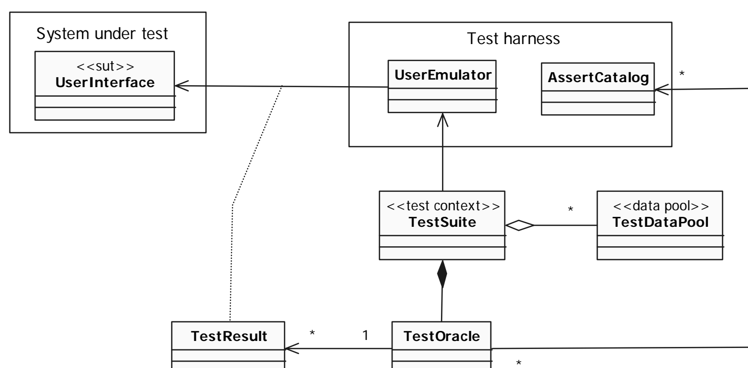


Figura 1. Arquitectura de prueba.

La clase *AssertCatalog* define la colección de asertos a disposición de los casos de prueba para determinar si el resultado obtenido del sistema a prueba (clase *TestResult*) 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.

La clase *TestSuite*, estereotipada como un *Test Context* del perfil de pruebas de UML, representa un conjunto de casos de prueba (*Test Case* en el perfil de UML). 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, por lo que han sido omitidos de la figura 1.

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 la ejecución de asertos sobre el resultado obtenido) que determinarán el veredicto de los casos de prueba del contexto de prueba. El perfil de prueba de UML define el elemento *Arbiter*, el cuál evalúa los veredictos de todos los casos de prueba de un contexto de prueba y determina el resultado final. El árbitro también suele ser incorporado en el propio *test harness* (como por ejemplo en JUnit), como un elemento que ejecuta el *Test Suite*. Este elemento se encarga de recordar el valor de todos los casos de prueba ejecutados y determinar el resultado final del *Test Suite*.

La clase *TestDataPool* (estereotipada como Data Pool según el perfil de UML) contendrá un conjunto de métodos *Data Selector* para seleccionar los distintos valores de prueba según las distintas particiones identificadas en las variables de los casos de uso (como se ha comentado en la sección 2).

A continuación, se describen un conjunto de buenas prácticas para implementar los elementos de la figura 1 a partir de la información de los escenarios de casos de uso.

### 3.2. Implementación de los casos de prueba

A partir del marco de trabajo definido en la sección anterior, se define a continuación cómo implementar las pruebas del sistema para verificar la implementación de los casos de uso.

Un caso de prueba es una implementación de un objetivo de prueba. Según el perfil de pruebas de UML, un caso de prueba no es un elemento arquitectónico sino la definición de un comportamiento dentro de un elemento estereotipado como *Test Context*. El comportamiento que se adopta con mayor

frecuencia se describe, entre otros trabajos, en [1] y se lista en la tabla 1. El segundo paso de la tabla 1 ha sido refinado por los autores de este trabajo con los pasos 2.1 y 2.2. Este comportamiento ha sido implementado, por ejemplo, en las herramientas tipo XUnit.

Cada caso de uso tendrá asociado un *test suite* (figura 1). Dicha *suite* contendrá las pruebas de todos los escenarios de dicho caso de uso. Cada uno de los casos de prueba se codificará como tres métodos, al menos dentro del *test suite* correspondiente. Dos métodos para el *set up* y el *tear down*, y un método para el caso de prueba en sí. A continuación se describe cómo implementar estos métodos y los demás elementos del modelo de la figura 1.

Como se ha visto en los objetivos de prueba, en cada uno de los pasos del escenario debe indicarse si el realizado por un actor o por el sistema a prueba. Esta información es muy 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. El *test oracle* de un caso de prueba será el conjunto de *ValidationActions*, o acciones de validación, obtenidas principalmente, a partir de los pasos realizados por el sistema. En función de las poscondiciones pueden añadirse asertos adicionales. 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 interfaces son los correctos y, en la medida de lo posible, que no hay elementos adicionales. Las acciones de validación se implementan utilizando el catálogo de asertos proporcionado por el *test harness* sobre el resultado devuelto por el *UserEmulator*.

Como se ha visto en la sección 2, se va a aplicar la técnica de variables operacionales y de categoría-partición para la definición de los valores de prueba necesarios. A partir de la experiencia de los autores de este trabajo, se han identificado tres tipos distintos de variables operacionales. Cada uno de los tipos se implementará de manera distinta en los casos de prueba.

El primer tipo lo componen aquellas variables operacionales que indican un suministro de información al sistema por parte de un actor externo.

Tabla 1. Comportamiento genérico de un caso de prueba.

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.
3. Invocación del *tear down* del caso de pruebas.

Para cada variable de este tipo se definirá una nueva clase cuyos objetos contendrán los distintos valores de prueba para dicha variable. Para cada partición del dominio identificada, el elemento *TestDataPool* tendrá, al menos, una operación que devuelva un valor de prueba perteneciente a dicha partición. Un ejemplo de una variable operacional de este tipo se muestra en el caso práctico.

El segundo tipo lo componen aquellas variables operacionales que indican una selección entre varias opciones que un actor externo tiene disponible. En este caso, no tiene sentido implementar estas variables como métodos del *TestDataPool*. 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.

El tercer tipo lo componen aquellas variables operacionales que indican un estado del sistema. 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 up* debe restaurar dichos valores a sus estados originales. Además, el método *tear down* 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.

En la sección 4 se muestra un caso práctico de todo lo expuesto en la sección 3.1 y en los párrafos anteriores.

Tabla 2. Caso de uso bajo prueba.

| **Name** | UC-01. Add new link |
|---|---|
| **Precondition** | No |
| **Main sequence** | 1 The user selects the option for introduce a new link.<br>2 The system recovers all the stored categories and it asks for the information of a link.<br>3 The user introduces the information of the new link.<br>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.<br>2.2 If there were not categories found, then the system shows and error message and this use case ends.<br>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.<br>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. |
| **Notes** | The categories that a user may choose, are all the categories registered by an administrator into the system. |

## 4. Un caso práctico

En este caso práctico, 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 las secciones anteriores 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 utilizadas.

### 4.1. Objetivos de prueba.

El caso de uso de la tabla 2, 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 3). Los patrones usados se describen en la metodología de elicitación NDT [4] [5].

A partir del caso de uso, y de manera automática, se han generado un conjunto de escenarios los cuáles serán los objetivos de prueba de dicho caso de uso. Dado que el caso de uso presenta bucles no acotados, con un número infinito de potenciales repeticiones, criterio de cobertura elegido para obtener los caminos es el criterio 01, el cuál consiste en obtener todos los caminos posibles para una repetición de ninguna o una vez de cada uno de los bucles.

Tabla 3. Requisito de información de los enlaces.

<table>
<tr><td>Name</td><td>SR-01. Link.</td></tr>
<tr><td>Specific data</td><td><table>
<tr><td>Name</td><td>Domain</td></tr>
<tr><td>Identifier</td><td>Integer</td></tr>
<tr><td>Name</td><td>String</td></tr>
<tr><td>Category</td><td>Integer</td></tr>
<tr><td>URL</td><td>String</td></tr>
<tr><td>Description</td><td>String</td></tr>
<tr><td>Approved</td><td>Boolean</td></tr>
<tr><td>Date</td><td>Date and time</td></tr>
</table></td></tr>
<tr><td>Restrictions</td><td>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.</td></tr>
</table>

Todos los escenarios obtenidos con este criterio y traducidos al español se listan en la tabla 4. Para este caso práctico, seleccionamos el escenario 09, el cuál se describe en detalle en la tabla 5 (dado que el caso de uso se ha redactado en inglés, su escenario principal también se muestra en inglés), para su implementación. El proceso utilizado se describe en detalle en [7].

Tabla 4. Escenarios del caso de uso.

| 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. |

También ha sido posible aplicar el método de categoría-partición. Todas las variables operacionales encontradas automáticamente por la herramienta ValueGen se enumeran en la tabla 6. Las particiones para cada una de dichas variables (también encontradas por la herramienta ValueGen) se enumeran en la tabla 7.

En este caso, no se ha continuado refinando el conjunto de particiones aunque para algunas variables, como V04, sí podrían identificarse particiones adicionales.

Para la implementación del escenario de éxito, todas las variables operacionales deben tener un valor perteneciente a las particiones C02.

Tabla 5. Escenario principal.

| 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) |

Tabla 6. Variables identificadas para el caso de uso.

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

Tabla 7. Categorías par alas variables identificadas.

| Variable | Particiones |
|---|---|
| V01 | C01: Ocurre un error.<br>C02: No ocurre un error. |
| V02 | C01: No se encontraron categorías.<br>C02: Sí se encontraron categorías. |
| V03 | C01: Cancela la operación.<br>C02: No cancela la operación. |
| V04 | C01: El nombre, categoría o URL están vacías.<br>C02: El enlace es correcto. |
| V05 | C01: Error almacenando el enlace.<br>C02: No ocurre un error. |

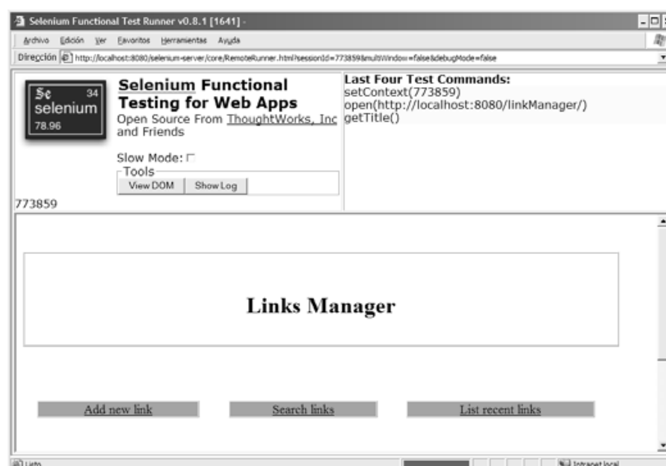


Figura 2. Ejecución del caso de prueba del escenario principal con la herramienta Selenium.

En la siguiente sección, se definen los elementos pertenecientes al *test harness* usados en este caso práctico.

## 4.2. Test harness.

Tal y cómo se describió en la figura 1, el *test harness* tiene la misión de simular el comportamiento del usuario y ofrecer un conjunto de asertos para evaluar el resultado obtenido.
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 sea capaz también de realizar comprobaciones en el código HTML recibido como respuesta. Por ellos, hemos elegido la herramienta de código abierto Selenium (www.openqa.org/selenium) la cuál cumple estas características.

Como se puede ver el la figura 2, Selenium ofrece un interfaz que permite abrir un navegador web e interactuar con él de la misma manera que un actor humano.

Respecto al catálogo de asertos, al estar basado en la popular herramienta JUnit, Selenium incorpora el mismo conjunto de asertos que JUnit y, además, funciones para acceder a los resultados visualizados en el navegador web.

## 4.3. Implementación de un caso de prueba

En primer lugar se ha implementado el *data pool* y los valores de prueba tal y como se muestra en la figura 3.

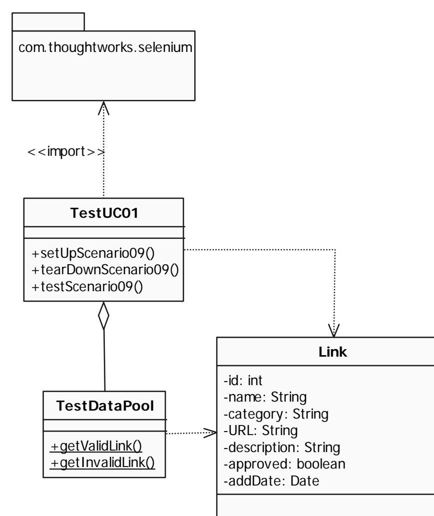


Figura 3. Implementación del caso de prueba.

De la tabla 6, sólo la variable V04: Datos del enlace, hace referencia a una información suministrada desde el exterior al sistema durante la ejecución de la prueba. Se ha desarrollado una clase *bean* para representar los diferentes enlaces. Por cada categoría posible, se ha añadido un método estático al *pool* 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 válidos e inválidos, sin entrar en más subdivisiones. El código resultado puede descargarse de la misma página que hospeda las herramientas ObjectGen y ValueGen.

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 realiza actualmente a mano y se muestra en la tabla 8.

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 del segundo paso). En la tabla 8, se puede observar como el caso de prueba no pulsa en la opción de cancelar.

Por las características específicas de Selenium, es necesario añadir esperas a que la página se cargue antes de continuar. 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.

A continuación, en la tabla 9, se escribir los asertos a partir de los pasos que realiza el sistema.

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

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

| **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()));` |

Como se puede observar en las tablas 8 y 9, la variable *s* es la referencia al objeto que interactúa con el navegador, la cuál ofrece métodos para realizar acciones con el navegador y comprobar la página visualizada.

En este caso sería necesario un aserto adicional que comprobara que el enlace está correctamente almacenado en el sistema tal y como dicta la poscondición del caso de uso. Para ello se ha añadido nuevo método auxiliar `isLinkStored` (usado en la última línea del paso 07, tabla 9).

La implementación del método de *set-up* consistió en comprobar que todas las variables operacionales de la tabla 2 tuvieran un valor de la partición C02. Es decir, comprobar que hay categorías 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*, consistió en la restauración del conjunto original de enlaces almacenado en el sistema.

## 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. Otros trabajos relacionados con la generación de pruebas ejecutables se citan en los siguientes párrafos.

En [1] se pueden encontrar distintos patrones para la implementación de casos de prueba aunque dichos patrones están muy orientados a la prueba unitaria de código. En [13] 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 [9] 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. 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 [8].

Como se ha mencionado a lo largo de este trabajo, se ha conseguido automatizar la generación de objetivos de prueba mediante dos herramientas. El resto, aún, debe ser realizado manualmente. El camino hacia la automatización total aún es largo. Como se puede ver en el caso práctico, para la generación automática es necesario tener muchos datos específicos de la interfaz. Será necesario no solo tener esos datos, sino definirlos de una manera procesable automáticamente y enriquecerlos con una semántica para que el sistema sepa lo que son. Nuestros futuros trabajos se basan en la convención de nombres, desde los requisitos de almacenamiento hasta la implementación, para poder aplicar generación automática.

## Referencias


[1] Beck K. 2002. Test-Driven Development: By Example. Addison-Wesley ed.

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

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

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

[5] Escalona M.J. Gutiérrez J.J. Villadiego D. León A. Torres A.H. 2006. Practical Experiences in Web Engineering. 15th International Conference On Information Systems Development. Budapest, Hungary, 31 August – 2 September.

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

[7] Gutiérrez J.J. Escalona M.J. Mejías M. Torres J. 2006. Modelos Y Algoritmos Para La Generación De Objetivos De Prueba. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD 06. Sitges. Spain.

[8] Gutiérrez J.J. Escalona M.J. Mejías M. Reina A.M. 2006. Modelos de pruebas para pruebas del sistema. Taller de Desarrollo de Software Dirigido por Modelos. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD. Sitges. Spain.

[9] Katare M. et-al. 2006. Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach. TAIC PART 06. Windsor. UK.

[10] Meudec C. ATGen: Automatic Test Data Generation Using Constraint Logic Programming and Symbolic Execution

[11] Roubtsov S. Heck P. 2006. Use Case-Based Acceptance Testing of a Large Industrial System: Approach and Experience Report. TAIC-PART 06. Windsor, UK.

[12] Ostrand T. J., Balcer M. J. 1988. Category-Partition Method. Communications of the ACM. 676-686.

[13] Polo M. Tendero S. Piattini M. 2006. Integrating Techniques and Tools for Testing Automation. Software Testing, Verification and Reliability 17: 3-39

[14] Object Management Group. 2003. The UML Testing Profile. www.omg.org.

[15] Gutiérrez J.J. Escalona M.J. Mejías M. Torres J. Torres A. 2007. Generación automática de objetivos de prueba a partir de casos de uso mediante partición de categorías y variables operacionales. Jornadas sobre Ingeniería del Software y Bases de Datos JISBD 07. Zaragoza. Spain.