Actas de Talleres de Ingeniería del Software y Bases de Datos

Volumen 1, Número 4, Septiembre de 2007

ISSN 1988-3455

PRIS 2007:
Taller sobre Pruebas en Ingeniería del Software

Editor: Javier Tuya

Presentación

En este número de las Actas de Talleres de Ingeniería del Software y Bases de Datos se publican los trabajos presentados en la segunda edición del Taller sobre Pruebas en Ingeniería del Software (PRIS 2007: http://in2test.lsi.uniovi.es/pris2007/) celebrado en Zaragoza, el 11 de Septiembre de 2007 en el marco de las Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2007) y el Congreso Nacional de Informática (CEDI 2007).

Al igual que la primera edición del taller, esta segunda ha sido organizada como parte de las actividades de la Red para la promoción y mejora de las Pruebas en Ingeniería del Software (RePRIS: http://in2test.lsi.uniovi.es/repris/).

El objetivo general del taller fue establecer un foro de discusión sobre las actividades de I+D+i que se realizan relacionadas con la prueba del software. En esta segunda edición se seleccionaron 8 contribuciones diferentes tras haber sido seleccionadas mediante un proceso de revisión por pares.

Las contribuciones se agrupan en dos bloques principales. En el primero se presentan dos artículos con resultados de investigación relativos a métodos de diseño de pruebas y priorización de casos, seguidos de dos artículos que muestran resultados experimentales sobre pruebas de consultas SQL y prácticas de prueba utilizadas por los desarrolladores. El segundo bloque contiene artículos que abordan aspectos de capacitación profesional, gestión de pruebas y experiencias en la implantación de métodos de prueba.

Javier Tuya

Agradecimientos

La Red para la promoción y mejora de las Pruebas en Ingeniería del Software (RePRIS) ha sido financiada por el Plan Nacional de I+D+I del Ministerio de Educación y Ciencia y fondos FEDER (acción especial TIN2005-24792-E).

ISSN 1988-3455

Revisores y Organizadores de PRIS 2007

Javier Tuya (Universidad de Oviedo)
Raynald Korchia (SOGETI)
Juan Garbajosa (Universidad Politécnica de Madrid)
Luis Fernández Sanz (Universidad Europea de Madrid)
Macario Polo (Universidad de Castilla la Mancha)

Contenidos

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

ISSN 1988-3455

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.

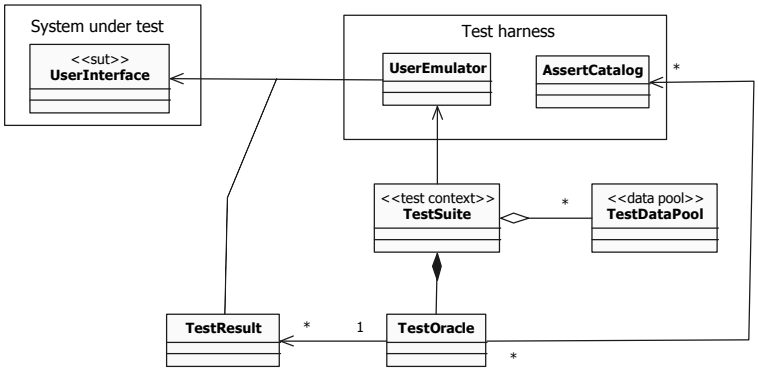

Figura 1. Arquitectura de prueba.

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. 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.
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.

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

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].

ISSN 1988-3455

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. C02: No ocurre un error.
V02	C01: No se encontraron categorías. C02: Sí se encontraron categorías.
V03	C01: Cancela la operación. C02: No cancela la operación.
V04	C01: El nombre, categoría o URL están vacías. C02: El enlace es correcto.
V05	C01: Error almacenando el enlace. C02: No ocurre un error.

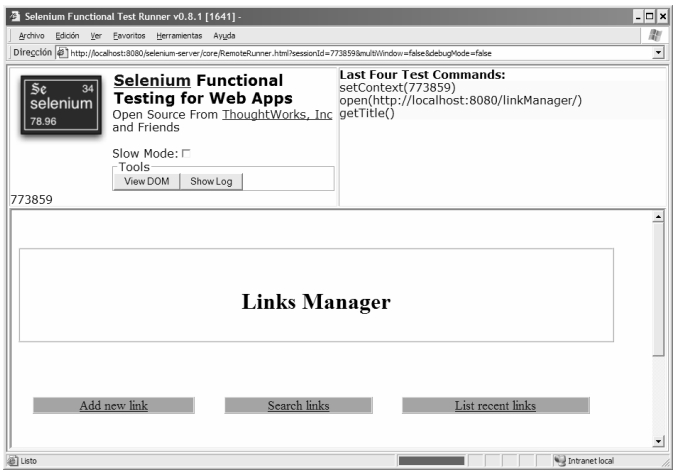

Figura 2. Ejecución del caso de prueba del escenario principal con la herramienta Selenium.

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.

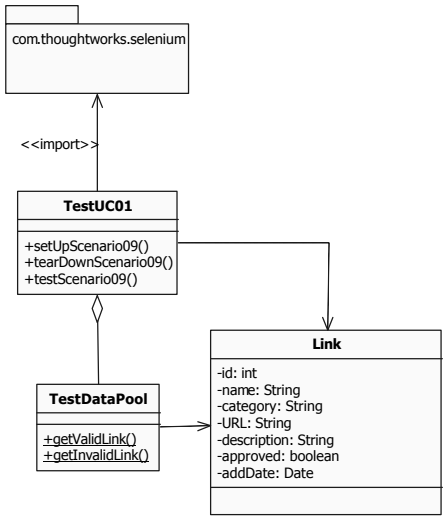

Figura 3. Implementación del caso de prueba.

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");` `s.waitForPageToLoad("5000");`

05: The user introduces the information of the new link.	`Link l = TestDataPool.getValidLink();`
	`s.type("name", l.getName());`
	`s.type("URL", l.getURL());`
	`s.type("description", l.getDescription());`
	`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());`
	`assertTrue(s.isTextPresent("Name*:"));`
	`assertTrue(s.isElementPresent("addEvent_name"));`
	`assertTrue(s.isTextPresent("Category*:"));`
	`assertTrue(s.isElementPresent("addEvent_category"));`
	`assertTrue(s.isTextPresent("URL*:"));`
	`assertTrue(s.isElementPresent("addEvent_URL"));`
	`assertTrue(s.isTextPresent("Description:"));`
	`assertTrue(s.isElementPresent("addEvent_description"));`
	`assertTrue(s.isTextPresent("Date:"));`
	`assertTrue(s.isElementPresent("addEvent_date"));`
	`assertTrue(s.isElementPresent("addEvent_0"));`

07: The system stores the new link.	`assertEquals("Links manager", s.getTitle());`
	`assertFalse(s.isTextPresent("Error storing new link"));`
	`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.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Programa Nacional de I+D+i del Ministerio de Educación y Ciencia y fondos FEDER, con la Red para la Promoción y Mejora de las Pruebas en Ingeniería del Software: RePRIS (TIN2005-24792-E).

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.

Priorización de casos de prueba mediante mutación

Macario Polo, Ignacio García-Rodríguez y Mario Piattini
Escuela Superior de Informática – Universidad de Castilla-La Mancha – España
{Macario.Polo, Ignacio.GRodriguez, Mario.Piattini}@uclm.es

Resumen

Se presenta un algoritmo para priorización de casos de prueba y una herramienta que lo implementa. El algoritmo obtiene altas reducciones en el tamaño del suite original sin pérdida de calidad. El criterio de selección de casos es el porcentaje de mutantes que mata cada caso de prueba.

1. Introducción

La priorización de casos de prueba es una práctica utilizada para disminuir los costes de las pruebas de regresión: básicamente, consiste en reejecutar aquellos casos que, de acuerdo con algún criterio de calidad, resultan más importante. En los últimos años, las organizaciones han adoptado los entornos X-Unit como herramientas para automatizar sus procesos de pruebas, lo que está permitiendo aplicar a nivel industrial los resultados obtenidos durante años de investigación en el plano académico.

A partir de diferentes *surveys*, en un artículo reciente hemos discutido el grado actual de automatización en compañías de desarrollo de software [1]: el estudio que muestra mejores resultados es el de Ng et al. en Australia [2], en donde el 79.5% de las compañías encuestadas automatizan la ejecución de pruebas y el 75% las de regresión; además, 38 de las organizaciones consultadas (58.5%) utilizan métricas para pruebas, siendo el número de defectos la más utilizada (31 organizaciones). Como es bien sabido, los entornos X-Unit permiten automatizar la ejecución de pruebas y las de regresión, y el principal resultado que muestran es el número de casos de prueba que encuentran fallo. Por tanto, y aunque el nivel de detalle de los trabajos consultados no es más fino, parece muy probable que las herramientas utilizadas por las compañías mencionadas sean del tipo X-Unit. De hecho, Do et al. afirman que JUnit está siendo cada vez más utilizada por las compañías de desarrollo de software [3].

Este artículo describe una herramienta que prioriza casos de prueba de JUnit en función del número de mutantes muertos por los mismos casos, en formato MuJava [4]. La herramienta puede ejecutarse en modo *standalone* o desde *testooj*, una herramienta de automatización de las pruebas de programas Java [1, 5].

2. Trabajos relacionados

En esta sección se resumen algunos trabajos relacionados con este artículo. Dependiendo de la estrategia de generación de casos utilizada, *testooj* puede generar un número de casos demasiado grande, por lo que se desarrolló e implementó el algoritmo que se presenta como parte principal de este artículo. Previamente, en la siguiente subsección se describen algunos algoritmos relacionados con la reducción del conjunto de casos de prueba.

2.1. Algoritmos para la reducción del conjunto de casos de prueba

El problema de la "reducción optima del *test-suite*", tal y como lo enuncian Jones y Harrold [6], es el siguiente:

Dado: Un *Test Suite T*, un conjunto de requisitos r_1, r_2, ..., r_n, que deben ser satisfechos por los casos de prueba en relación a la cobertura de un programa, *Problema: Encontrar* $T' \subseteq T$ tal que T' satisface todos los requisitos r_i y $(\forall T'' \subseteq T, T''$ satisface $r \Rightarrow |T'| \leq |T''|)$

En otras palabras, el problema consiste en encontrar, a partir de un conjunto T de casos de prueba, un subconjunto T'' de casos de prueba de cardinal mínimo que consiga la misma cobertura que T. A continuación revisamos algunos algoritmos.

ISSN 1988-3455

2.1.1. Algoritmo HGS

Harrold, Gupta y Sofa [7] presentan un algoritmo voraz (referido en la literatura como algoritmo *HGS*) para reducir el tamaño del *test-suite,* mientras que se preservan los requisitos de prueba del original. El algoritmo es aplicable para varios requisitos de prueba (p.ej., obtener los mejores casos en cuanto a cobertura de todos los usos y de condición-decisión).

2.1.2. Algoritmo de Heimdahl y George

Heimdahl y George [8] proponen también un algoritmo voraz para reducir el *test-suite:* básicamente, toman un caso de prueba al azar, lo ejecutan y miden la cobertura alcanzada. Si ésta es mayor que la del caso que alcanzaba más cobertura, lo añaden al conjunto reducido. El algoritmo lo ejecutan cinco veces para obtener cinco conjuntos reducidos distintos. Ya que se confía completamente en el azar, la calidad de los resultados no está garantizada.

2.1.3. Algoritmo de McMaster y Memon

McMaster y Memon [9] presentan otro algoritmo voraz. El criterio para seleccionar casos de prueba es el número de llamadas a la pila que realice el programa bajo prueba ante el caso de prueba. Como se observa, éste no es un requisito de pruebas demasiado común.

2.1.4. Resumen

El problema descrito es NP-completo y su solución, por tanto, no puede ser obtenida en tiempo polinomial. Así, todos los algoritmos propuestos obtienen soluciones próximas a la optima: el tamaño del *test-suite* se reduce manteniendo la calidad, pero no hay garantía de que el tamaño conseguido sea el mínimo posible.

El criterio de selección de casos puede ser cualquiera (cobertura de bloques, de sentencias, de condiciones…) o, por ejemplo, y como se presenta en este artículo, el número de mutantes que mata cada caso de prueba.

El algoritmo que se presenta en este artículo también es voraz y, así, garantiza que la cobertura del conjunto reducido es la misma que la del original. La principal diferencia con respecto a otros trabajos es el criterio de selección de casos (número de mutantes muertos) y el formato de los casos, que es compatible con los entornos JUnit.

2.2. Estrategias de combinación para generación de casos de prueba

En un artículo reciente, Grindal et al. describen 16 estrategias para generación de casos de prueba que clasifican en diversas categorías [10]: partiendo del conjunto de valores de prueba, el objetivo de estas estrategias es obtener Buenos conjuntos de casos de prueba que logren alta cobertura en el programa que se está probando.

De la estrategia seleccionada depende el tamaño del *test-suite*, como también la cobertura alcanzada en la clase bajo prueba. A modo de ejemplo, la Tabla 1 muestra el número de casos de prueba generados con tres de las estrategias revisadas, así como el porcentaje de mutantes muertos al ejecutarlos sobre una clase *Triangle*, que representa el clásico problema de determinación del tipo de un triángulo. Como se ve, *All combinations* construye el *test-suite* de mayor tamaño (216 casos), aunque también es la estrategia que obtiene mayor cobertura (medida, en este caso, como el porcentaje de mutantes muertos). En efecto, *All combinations* se utiliza habitualmente como estrategia de base para comparar el número y calidad de los casos generados por las técnicas que los investigadores van proponiendo.

Estrategia	Nº de casos	% muertos
Each choice	7	51%
Anti random	8	65%
All combinations	216	100%

Tabla 1. Resultados obtenidos con varias estrategias en el problema del triángulo

3. Reducción del conjunto de casos de prueba basado en mutación

En esta sección se describe un algoritmo voraz que utiliza el porcentaje de mutantes muertos como criterio para incluir casos de prueba en el conjunto reducido. La implementación dada al algoritmo permite seleccionar casos de prueba JUnit, si bien la selección se realice en función del porcentaje de mutantes que matan los mismos casos, en formato MuJava.

La equivalencia entre los casos de prueba JUnit y MuJava de uno a otro formato se discute y describe en la comunicación [5], presentada el año pasado en este mismo foro y en [1]; la Figura 1 ilustra la correspondencia entre uno y otro formato

con un sencillo ejemplo. Siendo *tj* y *tm* dos casos de prueba equivalentes en formatos JUnit y MuJava, el algoritmo selecciona *tj* basándose en los mutantes muertos por *tm*.

```
public void test1() {
    Account o=new Account();
    o.deposit(1000);
    assertTrue(o.getBalance()==1000);
}
```

```
public String test1() {
    Account o=new Account();
    o.deposit(1000);
    return o.toString();
}
```

Figura 1. Dos casos de prueba equivalentes en formatos JUnit y MuJava

La Figura 2 muestra la función principal del algoritmo. Como entradas, recibe el conjunto completo de casos de prueba, la clase bajo prueba y el conjunto completo de mutantes. En la línea 2, ejecuta todos los casos contra la clase y contra los mutantes, guardando los resultados en la variable *testCaseResults*.

```
1. reduceTestSuite(completeTC : SetOfTestCases,
   cut : CUT, mutants : SetOfMutants)
      : SetOfTestCases
2. testCaseResults = execute(completeTC, cut,
      mutants)
3. requiredTC = ∅
4. n=|mutants|
5. while (n>0)
6.    mutantsNowKilled = ∅
7.    testCasesThatKillN =
      getTestCasesThatKillN(completeTC, n, mutants,
        mutantsNowKilled, testCaseResults)
8.    if |testCasesThatKillN>0| then
9.      requiredTC = requiredTC ∪ testCasesThatKillN
10.     for i=1 to |testCasesThatKillN|
11.       testCase = testCasesThatKillN[i]
12.       testCase.removeAllTheMutantsItKills()
13.     next
14.     n = |mutants|-|mutantsNowKilled|
15.   else
16.     n = n – 1
17.   end if
18.end_while
19.return requiredTC
20.end
```

Figura 2. Función principal del algoritmo, que devuelve el *test-suite* reducido

Entonces, el algoritmo está preparado para seleccionar, en varias iteraciones, los casos de prueba que matan más mutantes (líneas 5-18).

La primera vez que el algoritmo entra en este bucle y llega a la línea 7, el valor de *n* (utilizado para dejar de iterar) es |*mutants*| (cardinal del conjunto de mutantes): en este caso especial, el algoritmo busca algún caso de prueba que mate a todos; si lo encuentra, el algoritmo añade el caso a *requiredTC*, actualice el valor de *n* y para; en otro caso, decrementa *n* (línea 16) y se introduce nuevamente en el bucle.

Supongamos que *n* es inicialmente 100 (es decir, hay 100 mutantes de la clase bajo prueba), y supongamos que el algoritmo no encuentra casos de prueba que maten a *n* mutantes hasta que *n=30*. Con este valor, la función *getTestCasesThatKillN* (llamada en la línea 7) devuelve todos los casos de prueba que maten a *n* mutantes diferentes: así, si dos casos (tc_1 y tc_2) matan los mismos 30 mutantes, *getTestCasesThatKillN* devuelve solo un caso (por ejemplo, tc_1). Si la intersección de los mutantes muertos por tc_1 y tc_2 no es vacía, entonces el algoritmo devuelve un conjunto formado por tc_1 y tc_2.

Cuando se encuentran los casos que matan a *n* mutantes, se añaden a *requiredTC* (línea 9) y se eliminan los mutantes muertos del conjunto de mutantes (líneas 10-13). El valor de *n* se actualiza al número de mutantes que quedan vivos.

En la implementación real del algoritmo, la ejecución del conjunto complete de casos contra la clase bajo prueba y sus mutantes se realice en una función separada (función *execute*, llamada en la línea 2). Esta función devuelve un conjunto de objetos de tipo *TestCaseResult*, compuestos por el nombre del caso de prueba y la lista de mutantes a los que matan (Figura 3).

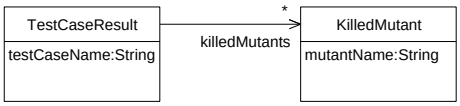

Figura 3. Estructura de los elementos devueltos por la función *execute* función (línea 2 de la Figura 2)

A la función encargada de recoger los casos que matan *n* mutantes se la llama en la línea 7 de la Figura 2, y aparece detallada en la Figura 4: recorre los elementos incluidos en *testCaseResults* y toma aquellos casos cuya lista de mutantes muertos (elemento *killedMutants* de la Figura 3) tienen *n* elementos. Para garantizar que no se seleccionan dos casos que matan a los mismos mutantes, la

función elimina los mutantes muertos del conjunto cada vez que elige un caso de prueba.

```
1.getTestCasesThatKillN(completeTC :
      SetOfTestCases, n : int, mutants : SetOfMutants,
      mutantsNowKilled : SetOfMutants,
      testCaseResults: SetOfTestCaseResults)
2.    testCasesThatKillN = ∅
3.    for i=1 to |testCaseResults|
4.      testCaseResult = testCaseResults[i]
5.      if |testCaseResult.killedMutants| == n then
6.        testCasesThatKillN = testCasesThatKillN ∪
      testCaseResult.testCaseName
7.        mutantsNowKilled = mutantsNowKilled ∪
      testCaseResult.killed.Mutants
8.        mutants = mutants – mutantsNowKilled
9.        for j=1 to | testCaseResults |
10.           aux = testCaseResults[j]
11.           if aux.testCaseName ≠
                 testCaseResult.testCaseName then
12.              aux.remove(mutantsNowKilled[i]
13.           end_if
14.       next
15.     end_if
16.   next
17.   return testCasesThatKillN
18. end
```

Figura 4. Función que devuelve los casos que matan *n* mutantes

3.1. Análisis de coste

El coste real de ejecución del algoritmo depende de la clase sobre la que se aplica.

Un caso extremo sucede cuando cada caso de prueba mate solo un mutante (lo cual es prácticamente imposible). En esta situación, la función de la Figura 2 hará *n* iteraciones (*n* es el número de mutantes), pero la instrucción condicional de la línea 8 no será cierta hasta que *n=1*. En este caso, el coste computacional del algoritmo es:

$$O(n\text{-}1)\cdot O(getTestCasesThatKillN) + O(1)\cdot O(getTestCasesThatKillN) + O(|testCases|)$$

El coste de la función de la Figura 4 hace siempre dos iteraciones (bucles de las líneas 3 y 9): su coste, en el caso peor, será siempre $O(|testCaseResults|)$.

Ya que $|testCaseResults| \pounds |testCases|$, el coste del algoritmo en el peor caso es:

$$O(reduceTestSuite) = O(|testCases|)^3$$

Otro caso extremo (también casi imposible) sucederá cuando un caso de prueba mate todos los mutantes. En esta situación, los bucles de la Figura 2 harán una sola iteración, con lo que el coste en este caso es $O(|testCases|)^2$.

En otras situaciones, el coste computacional depende también del número de mutantes, como se muestra en la Figura 5.

$$O(reduceTestSuite) = O(|mutants|) \cdot O(|testCases|)^3$$

Figura 5. Coste General de *reduceTestSuite*

En la implementación, el coste del algoritmo se reduce respecto del mostrado en la Figura 5 debido al uso de estructuras de datos que permiten búsquedas no secuenciales. Así, el conjunto de objetos de tipo *TestCaseResult* se implementa como una tabla hash indexada por el número de mutantes muertos, lo que hace innecesario recorrer el conjunto completo cada vez que se ejecuta la función de la Figura 4.

4. "A motivational example" (Un ejemplo motivador)

En su artículo [11], Jeffrey y Gupta incluyen una sección con el mismo título de esta (*A motivational example)* en el que muestran un pequeño programa, al que aplican su algoritmo de reducción de casos mediante redundancia selectiva. Para nuestro caso, hemos traducido su código al pequeño programa Java que se muestra en la Figura 6. Para este sencillo programa, MuJava genera 48 mutantes tradicionales (entre los que hay 15 funcionalmente equivalentes) y 6 mutantes de clase.

Usando los valores {-1.0, 0.0, -1.0} como valores de prueba para los cuatro parámetros de la función *f*, y generando los casos de prueba con la estrategia *All combinations* algoritmo, la herramienta *testooj* genera un fichero JUnit y otro MuJava con 3×3×3×3=81 casos de prueba equivalentes, que consiguen matar al 100% de los mutantes no equivalentes.

Tras aplicar el algoritmo de reducción, el *test-suite* de 81 casos se consigue reducir a otro compuesto por solo 7 casos de prueba sin perder cobertura, lo que supone una disminución del tamaño del conjunto al 8.6% del tamaño original.

5. Implementación

La herramienta que implementa los algoritmos mostrados anteriormente requiere un fichero con los casos de prueba en formato MuJava (en la figura, el fichero *MujavaJGExample_1.class)*, la

configuración correspondiente del entorno (variable CLASSPATH), el nombre de la clase bajo prueba (en la figura, *paper.JGExample.class)* y la ubicación de los mutantes de la clase bajo prueba.
De acuerdo con el algoritmo de la Figura 7, la herramienta ejecuta la clase con los casos de prueba contra todas las versiones que va encontrando de la clase bajo prueba (lo que incluye, desde luego, a la clase original): en el caso del *JGExample*, se crean 81 instancias de la clase najo prueba (correspondientes a los 81 casos de prueba) para los 54 mutantes no equivalentes, lo que significa que se crean 81×54=4374 instancias de la clase bajo prueba.

```
public class JGExample {
   float returnValue;
   public float f(float a, float b, float c, float d) {
      float x=0, y=0;
      if (a>0)  x=2;
      else  x=5;
      if (b>0)   y=1+x;
      if (c>0)
         if (d>0)
            returnValue=x;
         else
            returnValue=10;
      else
         returnValue=(1/(y-6));
      return returnValue;
   }
}
```

Figura 6. El "ejemplo motivador" de Jeffrey y Gupta

El resultado de ejecutar cada caso de prueba contra las versiones del programa se coloca en objetos de tipo *ExecutionResult* (cuya estructura se corresponde con la Figura 7), que guardan el estado de la instancia de la clase bajo prueba en forma de cadena junto al nombre del caso de prueba. Con objeto de preservar la memoria del computador, estos resultados de ejecución se guardan en disco, para realizar posteriormente las comparaciones los cálculos mostrados en los algoritmos presentados anteriormente.

Cuando todos los casos han sido ejecutados contra la clase original y los mutantes, la herramienta muestra un resumen de los resultados en una matriz de "mutantes muertos", cuyas celdas indican que el caso de prueba que se muestra en la columna de la tabla ha matado al mutante de la fila correspondiente. Este resultado se puede exportar en formatos *html* o *txt* para facilitar otros análisis con, por ejemplo, una hoja de cálculo.
Como se ha explicado, la herramienta construye un fichero JUnit con los casos de prueba correspondientes al conjunto reducido.

```
for each cutFile in folders
    cut = load the CUT corresponding to cutFile using
a class loader
    mutantName = name of the folder
    results = ∅
    for each testCase in Suite
        cutInstance = execute testCase on cut
        executionResult = new
            ExecutionResult(testCase.name, cutIn-
stance)
        results = results ∪ {executionResult}
    next
    save results in a file called mutantName.ser
next
```

Figura 7. Algoritmo de ejecución de casos de prueba

6. Validación adicional

Además de al ejemplo ilustrativo de Jeffrey y Gupta, la implementación del algoritmo se ha aplicado a un conjunto de programas que se pueden clasificar en tres categorías:

- *Programas de "juguete"*: se han utilizado los seis programas que Pargas y Harrold [12] utilizan para validar su algoritmo de generación de casos de prueba.
- *Programas industriales*: se ha utilizado la clase *PluginTokenizer* de la aplicación *jtopas*, incluida en la infraestructura de testing de [13] y una clase de tipo contenedor (el *Vector* del paquete *java.util)*, también utilizadas en publicaciones sobre testing [14].
- *Programas de estudiantes*: se ha utilizado una clase *Sudoku* escrita por un grupo de alumnos. La clase tiene un constructor sin parámetros que inicializa una matriz de 9x9 enteros; su método *setNumber(int fila, int columna, int numero)* comprueba si es posible poner el *numero* en la *fila* y *columna* según las reglas del sudoku.

Para estos programas, se han generado casos de prueba utilizando *testooj* con *All combinations.* Los resultados se muestran en la Tabla 2. Como se observa, se consiguen en muchos casos reducciones próximas al 90%.

ISSN 1988-3455

		Nº de casos	
Programa	**Mutantes**	**Original**	**Reducido**
Bisect	66	6	1
Bub	86	84	2
Find	166	252	2
Fourballs	190	64	4
Mid	172	12	2
TriTyp	258	216	22
PluginTokenizer	102	14	1
Vector	444	56	5
Sudoku	135	64	5

Tabla 2. Resultados obtenidos en diversos casos

7. Conclusiones

Este artículo ha presentado un algoritmo de reducción del conjunto de casos de prueba. El algoritmo utiliza el porcentaje de mutantes muertos como criterio de inclusión. El algoritmo se ha implementado como parte de la herramienta *testooj* (http://alarcos.inf-cr.uclm.es/testooj3).
De acuerdo con la revisión realizada a la literatura especializada, esta la primera herramienta que incluye un algoritmo de reducción de casos aplicable a herramientas de prueba ampliamente extendidas, como JUnit.

8. Agradecimientos

Trabajo parcialmente financiado por la Dir. Gral. de Investigación/FEDER, TIN2006-15175-C05-05 y TIN2005-24792-E.

9. Referencias

1. Polo M, Tendero S, and Piattini M, *Integrating techniques and tools for testing automation.* Software Testing, Verification and Reliability, 2007. **15**(1), 3-39.
2. Ng SP, Murnane T, Reed K, Grant D, and Chen TY. *A Preliminary Survey on Software Testing Practices in Australia*. Proceedings of the Australian Software Engineering Conference (ASWEC 2004). 2004. Melbourne, Australia: IEEE Computer Society, pp. 116-125.
3. Do H, Rothermel G, and Kinner A, *Prioritizing JUnit test cases: An empirical assessment and cost-benefit analysis.* Empirical Software Engineering, 2005.
4. Ma Y-S, Offutt J, and Kwon YR, *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability, 2005. **15**(2), 97-133.
5. Polo M and Piattini M. *Automatización del proceso de pruebas unitarias.* PRIS 2006: Taller sobre pruebas en Ingeniería del Software. 2006. Sitges (Barcelona), pp.
6. Jones JA and Harrold MJ, *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage.* IEEE Transactions on Software Engineering, 2003. **29**(3), 195-209.
7. Harrold M, Gupta R, and Soffa M, *A methodology for controlling the size of a test suite.* ACM Transactions on Software Engineering and Methodology, 1993. **2**(3), 270-285.
8. Heimdahl MPE and George D. *Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing*. 19th IEEE International Conference on Automated Software Engineering (ASE'04). 2004, pp. 176-185.
9. McMaster S and Memon AM. *Call Stack Coverage for Test Suite Reduction*. 21st IEEE International Conference on Software Maintenance. 2005. Budapest (Hungary), pp. 539-548.
10. Grindal M, Offutt J, and Andler SF, *Combination testing strategies: a survey.* Software Testing, Verification and Reliability, 2005(15), 167-199.
11. Jeffrey D and Gupta N. *Test suite reduction with selective redundancy*. International Conference on Software Maintenance. 2005. Budapest (Hungary): IEEE Computer Society Press, pp. 1-10.
12. Pargas RP, Harrold MJ, and Peck RR, *Test-Data Generation Using Genetic Algorithms.* Software Testing, Verification and Reliability, 1999(9), 263-282.
13. Do H, Elbaum SG, and Rothermel G, *Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact.* Empirical Software Engineering, 2005. **10**(4), 405-435.
14. Ball T, Hoffman D, Ruskey F, Webber R, and White L, *State generation and automated class testing.* Software Testing, Verification and Reliability, 2000. **10**, 149-170.

Un experimento controlado sobre pruebas de consultas SQL

Javier Tuya[1], Javier Dolado[2], Mª José Suárez-Cabal[1], Claudio de la Riva[1]

(1) Departamento de Informática
Universidad de Oviedo
Campus Universitario de Gijón
33204 Gijón
[tuya | cabal | claudio]@uniovi.es

(2) Departamento de Lenguajes y Sistemas Informáticos
Universidad del País Vasco
P.M. Lardizabal, 1
20.009 San Sebastián
dolado@si.ehu.es

Resumen

Este artículo presenta los resultados de la validación experimental de una técnica de diseño de casos de prueba basada en un criterio de cobertura estructural para consultas escritas en el lenguaje SQL. Se describe un experimento controlado realizado con el objeto de comparar dicho criterio con criterios convencionales basados en partición de clases de equivalencia y análisis de valores límite. Los resultados de un análisis preliminar muestran que la utilización del criterio de cobertura estructural permite que los usuarios realicen casos de prueba más efectivos y además se obtienen recomendaciones adicionales para mejorar el propio criterio.

1. Introducción

Desde la aparición de los primeros lenguajes de consulta a bases de datos, se han realizado un gran número de estudios empíricos para analizar la efectividad y eficiencia en la construcción de sentencias escritas en estos lenguajes y en especial en SQL. Estudios iniciales (como los recopilados por Reisner en [6]) han permitido comprender los problemas que el usuario se encuentra al escribir consultas SQL y mejorar el propio lenguaje. Otros estudios recientes analizan aspectos relacionados con la normalización [1], la complejidad de la consulta [7], o los defectos típicos que introducen los usuarios al realizar consultas [2].

Por otra parte, en la comunidad de las pruebas del software se han realizado multitud de estudios consistentes en la validación empírica de diferentes técnicas de prueba desde diversos puntos de vista [3] y para diferentes tipos de lenguajes. Sin embargo, no se han realizado estudios empíricos que puedan contribuir a entender y mejorar las técnicas y procesos de prueba para las consultas SQL que acceden a la información almacenada en bases de datos. Esto contrasta con el inmenso número de aplicaciones que utilizan bases de datos, y manejan la información utilizando SQL.

El lenguaje SQL fue desarrollado y comenzó a ser utilizado de forma industrial a partir de finales de la década de 1970, y continúa en evolución y uso en la actualidad. A pesar de ello, la mayor parte de la investigación en pruebas relacionadas con este tipo de lenguaje pertenece a la década actual y la validación de resultados se suele realizar mediante casos de estudio de pequeño tamaño [5]. En estos casos la validación de resultados no suele estar basada en experimentos controlados que permitan analizar y comprender de una forma rigurosa las ventajas e inconvenientes de unas técnicas frente a otras. Este es el ámbito en el que se enmarca este artículo.

El objetivo del presente artículo es comparar mediante un experimento controlado la efectividad de diferentes técnicas de prueba para sentencias SQL cuando éstas son utilizadas por personas que utilizan dichos criterios para guiar el diseño de los casos de prueba. En concreto se comparará un criterio sistemático y particular para SQL que está basado en cobertura estructural con la utilización de criterios convencionales como la partición en clases de equivalencia y análisis de valores límite.

En la Sección 2 se describe el criterio de cobertura estructural a utilizar. En la Sección 3 se describe el diseño del experimento cuyos resultados serán analizados en la Sección 4.

ISSN 1988-3455

Finalmente en la Sección 5 se presentan brevemente las principales conclusiones.

2. Criterio de cobertura estructural para consultas SQL

El criterio de cobertura estructural para sentencias SQL se basa en la consideración de que una consulta realiza comparaciones sobre valores que se encuentran en diversas filas de una o varias tablas y por tanto, las comparaciones en general se realizan entre valores no escalares. Se establece una diferenciación entre las comparaciones de izquierda a derecha y de derecha a izquierda, así como la consideración de la existencia de valores desconocidos (valores nulos).

El criterio se describe con más detalle en [8]. Se basa en organizar las condiciones presentes en las cláusulas JOIN y WHERE en un árbol de cobertura que se representa gráficamente en la Figura 1.

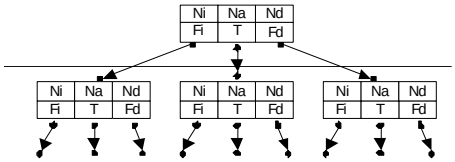

Figura 1. Representación gráfica de un árbol de cobertura

Cada condición se sitúa en un nivel del árbol y su evaluación se representa mediante uno o más "nodos de cobertura". La evaluación de la condición que se representa dependerá de la evaluación de los nodos superiores y se realizará respecto de los valores nulos (Ni, Na, Nd) y respecto de los valores no nulos (Fi, T, Fd), teniendo en cuenta el sentido en el que se realiza: de izquierda a derecha (i) y viceversa (d).

Cuando la consulta se ejecuta con un caso de prueba se evalúan cuáles de las diferentes situaciones representadas en los nodos de cobertura se cumplen y cuales no (seis en cada nodo). El porcentaje de cobertura será entonces el porcentaje de situaciones cubiertas respecto del total de situaciones posibles representadas en el árbol.

En el caso de la cláusula HAVING se realiza un árbol independiente para las condiciones incluidas en dicha cláusula y se evalúa a partir del resultado obtenido tras la ejecución de las cláusulas FROM, WHERE y GROUP BY de la consulta.

La representación gráfica es muy visual pero difícil de incorporar en una herramienta, por lo que para la realización del experimento se ha optado por representar cada una de las situaciones a cubrir mediante una consulta SQL denominada regla de cobertura. Cada una se ejecuta contra la base de datos de prueba, detectando que la situación representada está cubierta cuando la regla devuelve alguna fila. Un comentario textual descriptivo acompaña a cada una de estas reglas informando al usuario de la situación que se debe tener en los datos de prueba para que se cumpla la regla.

A continuación se muestra un ejemplo de una consulta realizada sobre tres tablas (proyecto, empleado y trabajo):

```
SELECT e.nombre , t.mes , t.horas , p.nombre
FROM empleado e
LEFT JOIN trabajo t ON t.idempleado=e.idempleado
LEFT JOIN proyecto p ON p.idproyecto=t.idproyecto
```

El árbol de cobertura tiene dos niveles (para las condiciones de las cláusulas JOIN). Una regla de cobertura para las situaciones T (en el primer nivel) y Fl (en el segundo) es la siguiente:

```
SELECT * FROM empleado e
INNER JOIN trabajo t on e.idempleado=t.idempleado
LEFT JOIN proyecto p on t.idproyecto=p.idproyecto
WHERE t.idproyecto is not null and p.idproyecto is null
```

La primera cláusula JOIN obliga a obtener registros de tabla trabajo con empleado relacionados, mientras que la segunda permite que un trabajo no tenga proyecto seleccionado. La cláusula WHERE impone una restricción adicional para que solamente se extraigan aquellas filas donde no hay un registro correspondiente a la tabla proyecto. Esta regla se presenta textualmente como “Debe existir un empleado que tenga relacionado un trabajo pero que no tiene relacionado un proyecto”.

En [9] se enumeran algunas mejoras del criterio encaminadas a mejorar la eficiencia y la eficacia. Respecto de la eficiencia, para evitar la expansión de los árboles se recomienda podar las ramas, por lo en las reglas utilizadas en el experimento se utilizará un subconjunto de todas las posibles: Se genera un primer conjunto de

reglas conteniendo exclusivamente los niveles correspondientes a las cláusulas JOIN, otro conjunto de reglas donde las condiciones de los JOIN solamente continúan hacia el WHERE en la rama para la que éstas están evaluadas a cierto y reglas adicionales en las que alguna de las situaciones correspondientes a las cláusulas JOIN son Fl o Fr. En relación a la eficacia, se ha mostrado que ésta puede aumentar cuando se contemplan los valores límite de las condiciones. Esto se ha incorporado añadiendo reglas que se cumplen cuando los valores en una condición están en los límites que hacen que esta cambie de cierto a falso y viceversa.

3. Diseño del experimento

El objetivo del experimento es determinar si la técnica utilizada en el diseño de casos de prueba para sentencias de SQL influye en la eficacia de los casos para detectar defectos en las consultas. En concreto se compararán dos técnicas:

- Basada en el criterio de cobertura estructural descrito en la sección 2 (denominada en lo sucesivo CC).
- Basada en otras técnicas de prueba convencionales: partición en clases de equivalencia y análisis de valores límite (denominada en lo sucesivo OT).

Así, se establecerá la hipótesis nula H_0 como "La utilización de una técnica para la elaboración de casos prueba de sentencias SQL (CC frente a OT) no influye en la capacidad de detección de defectos de los casos de prueba elaborados".

La variable independiente del experimento es, por tanto, la técnica utilizada (CC u OT), y la variable dependiente la capacidad de detección de defectos (Score), denominado en adelante, M (mutation score). Este se obtiene al ejecutar los casos de prueba respecto de un conjunto de consultas SQL con fallos (mutantes) obtenidos con la herramienta SQLMutation[1] [10]. Los operadores de mutación para consultas SQL son descritos con detalle en [11] y se agrupan en cuatro categorías:

- *SC - SQL clause mutation operators*: Realizan mutaciones en las cláusulas principales: SELECT, JOIN, GROUP BY, UNION, ORDER BY, subconsultas y funciones agregado.
- *NL - NULL mutation operators*: Relativas al manejo de los valores nulos, tanto en las condiciones como en las salidas de la consulta.
- *OR - Operator replacement mutation operators*: Similares a los operadores de modificación de expresiones descritos en [4] más operadores adicionales para los predicados BETWEEN y LIKE.
- *IR - Identifier replacement mutation operators*: Reemplazo de columnas, constantes y parámetros presentes en la consulta o en las tablas utilizadas por ésta.

El experimento se realizó con un conjunto de 20 alumnos de un curso de extensión universitaria sobre pruebas del software. Todos ellos recibieron formación básica en técnicas de pruebas, así como formación específica en la particularización de éstas técnicas a las pruebas de consultas SQL.

Los alumnos se organizaron en dos grupos (A y B) con 10 alumnos cada uno. Cada grupo realizó dos sesiones de pruebas de 45 minutos cada una. En cada una de las sesiones se realizaron las pruebas de dos consultas SQL diferentes, utilizando para el diseño de los casos de prueba una técnica diferente en cada sesión. La Tabla 1 resume este diseño

Grupo.	Sesión	Consultas	Técnica
A	1	Q1, Q3	CC
B	1	Q1, Q3	OT
A	2	Q2, Q4	OT
B	2	Q2, Q4	CC

Tabla 1. Diseño del experimento

Todas las consultas contienen cláusulas JOIN con tres tablas. En cada una de las sesiones, la primera consulta (Q1 ó Q2) es la más simple, incluyendo una cláusula WHERE y ORDER BY. La segunda (Q3 ó Q4) incluye además GROUP BY y HAVING.

La Tabla 2 muestra para cada consulta el número de mutantes no equivalentes generados junto con el número de reglas de cobertura utilizadas (criterio CC).

[1] SQLMutation está disponible a través de un interfaz web y como servicio web en http://in2test.lsi.uniovi.es/sqlmutation/

	Q1	Q2	Q3	Q4
Mutantes no equivalentes	89	106	125	122
Reglas de cobertura (CC)	18	18	20	22

Tabla 2. Número de mutantes y reglas de cobertura para cada consulta SQL

Para la realización de las pruebas, todos los alumnos utilizaron una herramienta denominada SQTest (http://in2test.lsi.uniovi.es/sqltest/) que permite introducir desde un único formulario los datos de entrada y visualizar los resultados. En el caso de utilizar la técnica CC, además existe la posibilidad de evaluar la cobertura de los casos de prueba y visualizar cuáles son las situaciones no cubiertas.

4. Resultados del experimento

Tras la realización de las dos sesiones del experimento, se generaron los mutantes para todas las consultas y se evaluó el "mutation score" (M) obtenido por cada una, así como la cobertura conseguida (C). La Figura 2 muestra el diagrama de cajas correspondiente a la variable dependiente (Score). La leyenda correspondiente a cada una de las cajas representa la variable independiente que se está midiendo (M, C), la consulta (Q1 a Q4) y técnica utilizada (CC, OT) en el diseño de los casos de prueba.

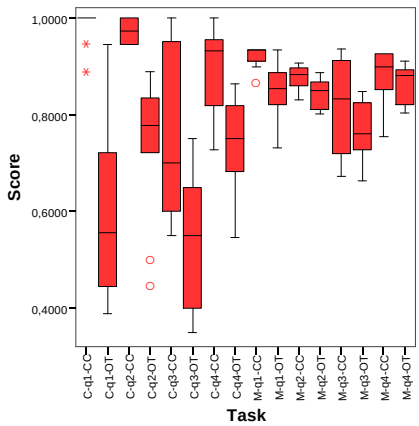

Figura 2. Diagrama de cajas correspondiente a las consultas Q1 a Q4

En la primera mitad de la figura se puede apreciar cómo para los casos diseñados utilizando CC se obtiene un porcentaje mucho mayor de cobertura C, lo cual es lógico puesto que estas pruebas han sido elaboradas utilizando este mismo criterio como guía.

La observación de los valores del "mutation score" (M) en la segunda mitad de la Figura 2 muestra que aparentemente se obtiene un mayor valor en las pruebas realizadas utilizando el criterio CC que en las realizadas utilizando OT, aunque las diferencias son mayores en las consultas Q1 y Q2 que en el resto.

Con el objeto de determinar si estas diferencias son estadísticamente significativas, para cada una de las consultas se realizará una prueba T para muestras independientes con un factor (la técnica utilizada: CC u OT) y variable dependiente M. La Tabla 3 muestra los resultados del estadístico (t), el grado de significación (p) y la diferencia de medias.

	t	p	Diferencia de medias
Q1	3,624	0,004	7,08%
Q2	2,471	0,024	1,22%
Q3	1,142	0,272	3,78%
Q4	0,685	0,502	2,15%

Tabla 3. Resultados del análisis para Q1 a Q4

Se aprecia que tanto en Q1 como Q2 la diferencia es significativa. Hay que hacer notar que la distribución de M en Q1 no cumple el requisito de normalidad (la mayor parte de los valores de M son iguales). Por ello se realiza un test no paramétrico alternativo (Mann-Whitney) que muestra que la diferencia es significativa con p=0,002. Ello permite rechazar la hipótesis nula de que ambas medias son diferentes.

Pero en las consultas Q3 y Q4, aunque los valores medios de M son superiores en ambos casos cuando se utiliza CC, esa diferencia no es significativa. Se pueden buscar diferentes justificaciones a ello que se discuten a continuación.

Una primera causa puede ser la fiabilidad de los casos de prueba realizados ya que estas consultas son más complejas, realizadas al final de cada sesión, y no siempre se completan. En la Figura 2 se puede apreciar que el porcentaje de cobertura C utilizado al aplicar la técnica CC tiene

mucho mayor rango de variación y valores inferiores para Q3 y Q4 que para Q1 y Q2. Además las reglas que no han sido cubiertas son las últimas en ser utilizadas para realizar las pruebas, por lo que no se puede considerar que estas pruebas no hayan sido completadas en su totalidad. Estos resultados deben aplicarse con precaución, aunque esta influencia afecta en principio por igual a ambos grupos.

Sin embargo, se ha encontrado otra justificación más plausible relacionada con el propio diseño de la técnica de cobertura CC. Examinando en detalle las situaciones cubiertas y los mutantes que quedan vivos se aprecia que se produce un enmascaramiento de los defectos que son capaces de detectar los casos de prueba.

Estas consultas (Q3 y Q4) tienen una cláusula HAVING que selecciona filas tras haber realizado JOIN, seleccionado con WHERE y agrupado con GROUP BY. Cuando se usa el criterio CC, el sujeto experimental centra su atención en cumplir las reglas de cobertura que se le presentan. La técnica CC no considera la existencia de la cláusula HAVING para la evaluación de la cobertura en cláusulas JOIN y WHERE. Por tanto, es posible que se cubran situaciones en éstas que cumplen las reglas de cobertura, pero que luego son filtradas por el HAVING, y por tanto no se traducen en un fallo detectable. Sin embargo, cuando se utiliza OT, el sujeto experimental centra su atención en toda la consulta y se tiene menor tendencia a este fenómeno. Es decir, muchas de las reglas que se cumplen utilizando el criterio CC no se traducen en una salida diferente, y por tanto, no matan determinados mutantes. El enmascaramiento es más grave en Q4, debido a que además de HAVING tiene una cláusula WHERE (que no tiene Q3).

Para comprobar lo anterior se añadirán dos consultas más denominadas Q5 y Q6 que son una copia de las Q3 y Q4 a las que se les ha eliminado la cláusula HAVING. Estas consultas son ejecutadas con los mismos casos de prueba diseñados para Q3 y Q4, y el valor de M evaluado. Los resultados de los valores de M se presentan en la Figura 3.

Gráficamente se puede apreciar que cuando se compara CC con OT, las diferencias parecen mayores en Q5 y Q6 que en Q3 y Q4 respectivamente.

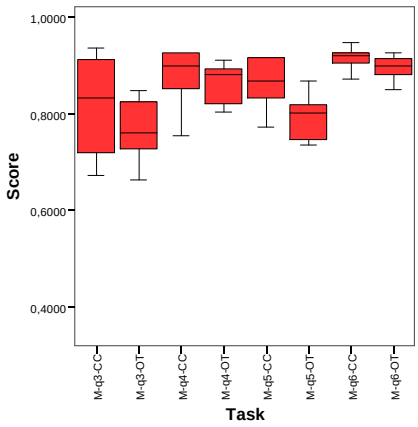

Figura 3. Diagrama de cajas comparativo entre Q3, Q4 y Q5, Q6.

Una nueva prueba T proporciona los resultados que se presentan en la Tabla 4.

	t	p	Diferencia de medias
Q5	12,966	0,008	6,50%
Q6	1,947	0,067	2,04%

Tabla 4. Resultados del análisis para Q5 y Q6

Por tanto, en ambos casos se rechaza la hipótesis nula (si bien en el caso de Q6 de forma marginal, pues p=0,067).

Esto sugiere una mejora de la implementación del criterio de cobertura CC, de forma que cuando se evalúe la cobertura del JOIN y WHERE en una consulta que tiene HAVING, se debe imponer una restricción adicional para que la cláusula HAVING sea cierta.

5. Conclusiones

En este artículo se ha mostrado empíricamente mediante un experimento controlado que el uso sistemático de un criterio de cobertura estructural para SQL para el diseño de casos de prueba consigue que se obtengan casos de pruebas capaces de detectar más defectos que cuando se utilizan otros criterios convencionales.

El examen de los casos en los que las diferencias no eran significativas ha permitido

asimismo detectar un punto de mejora en el propio criterio aplicable a las consultas que tienen cláusula HAVING.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Programa Nacional de I+D+i del Ministerio de Educación y Ciencia y fondos FEDER, con los proyectos IN2TEST (TIN2004-06689-C03-02), IN2QUANT (TIN2004-06689-C03-01) y la Red para la Promoción y Mejora de las Pruebas en Ingeniería del Software: RePRIS (TIN2005-24792-E).

Referencias

[1] P.L. Bowen, F.H. Rohde, Further evidence of the effects of normalization on end-user query errors: an experimental evaluation, International Journal of Accounting Information Systems 3(4) (2002) 255–290.

[2] S. Brass, C. Goldberg, Semantic Errors in SQL Queries: A Quite Complete List. Journal of Systems and Software 79(5) (2006) 630-644.

[3] N. Juristo, A.M. Moreno, S Vegas, Reviewing 25 Years of Testing Technique Experiments, Empirical Software Engineering 9(1-2) (2004) 7–44.

[4] K.N. King, A.J. Offutt. A Fortran Language System for Mutation-Based Software Testing. Software Practice and Experience 21(7) (1991) 686-718.

[5] N.F. Moratinos, M.J. Suárez-Cabal, J. Tuya, Evaluación de la investigación en el campo de pruebas de aplicaciones con base de datos, Taller sobre Pruebas en Ingeniería del Software (PRIS 2006), Sitges, 2006.

[6] P. Reisner, Use of Psychological Experimentation as an Aid to Development of a Query Language, IEEE Transactions on Software Engineering 3(3) (1977) 218-229.

[7] K. L. Siau, H.C. Chan, K.L. Wei, Effects of Query Complexity and Learning on Novice User Query Performance With Conceptual and Logical Database Interfaces, IEEE Transactions on Systems, Man and Cybernetics-Part A: Systems and Humans, 34(2) (2004) 276-281.

[8] M.J. Suárez-Cabal, J. Tuya, Using a SQL Coverage Measurement for Testing Database Applications, 12th ACM SIGSOFT Symposium on Foundations of Software Engineering. ACM Software Engineering Notes 19(6) 2004, 253-262.

[9] M.J. Suárez-Cabal, Mejora de casos de prueba en aplicaciones con bases de datos utilizando medidas de cobertura de sentencias SQL. Tesis doctoral, Universidad de Oviedo, 2006.

[10] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, SQLMutation: a Tool to Generate Mutants of SQL Database Queries, Second Workshop on Mutation Analysis (Mutation 2006), Raleigh, NC, 2006.

[11] J. Tuya, M.J. Suárez-Cabal, C. de la Riva, Mutating Database Queries, Information and Software Technology, 49(4) (2007) 398-417.

Un experimento sobre hábitos de pruebas artesanales de software: Resultados y Conclusiones

Pedro J. Lara Bercial, Luis Fernández Sanz
Departamento de Sistemas Informáticos
Universidad Europea de Madrid
pedro.lara@uem.es, luis.fernandez@uem.es

Resumen

En el mundo del desarrollo de software, los retrasos en las etapas previas hacen que sea frecuente que el tiempo dedicado a pruebas sea más reducido incluso que lo escasamente planificado inicialmente. Desde esta perspectiva, resulta de vital importancia conocer cómo piensan y abordan las pruebas los desarrolladores peculiarmente cuando adoptan una filosofía no sistemática de diseño de casos. Así, se aprovechó la oportunidad de otras investigaciones realizadas por los autores en el área de la automatización de pruebas para incrementar el conocimiento sobre otros aspectos prácticos implicados en las pruebas. De esta forma, además de la información necesaria para demostrar de AQUABUS (nuevo algoritmo de generación automática de pruebas a partir de especificaciones UML), se obtuvo información relevante sobre cómo los diferentes profesionales del desarrollo y gestión de proyectos de software llevaron a cabo el diseño de pruebas de un software específicamente pensado para evaluar su proceder y actitud frente a las pruebas planteadas.

1. Introducción

AQUABUS[1] [1] es un algoritmo diseñado por los autores para generar casos de prueba a partir de especificaciones UML; en concreto, a partir de un tipo especial de diagrama de actividad que incluye información adicional para permitir la priorización de los casos generados en función del riesgo derivado de dejarlos sin probar. La validación de este algoritmo como parte de la tesis doctoral de Pedro J. Lara [2] incluyó también esta serie de objetivos:

- Validar la importancia de una buena especificación
- Validar la compleción de AQUABUS en cuanto a la consecución de una lista suficiente de casos de prueba y la necesidad de priorización de los mismos.
- Validación de las posibilidades de aplicación de AQUABUS en proyectos reales.

Para cada objetivo se llevo a cabo una actividad o experimento diseñado expresamente para el mismo, aunque en algunos casos, los resultados de una determinada actividad sirvieron no sólo para apoyar el objetivo perseguido, sino también para facilitar la consecución de alguno de los otros objetivos.

En esta comunicación se explica brevemente en qué consistieron dichos experimentos y se detallan aquellos resultados que aportan información relevante para entender cómo los profesionales del desarrollo de software suelen llevar a cabo las tareas de prueba. Dicha información incluye resultados acerca de

- Las diferencias entre diseñar pruebas con una buena especificación y una mala.
- Comparación de datos cuantitativos acerca del número de pruebas realizadas manualmente, frente al número de pruebas posibles (estimado con AQUABUS)
- Datos acerca del número de veces que se repiten las mismas pruebas o pruebas equivalentes.
- Comparativas entre las pruebas que se creen a priori más importantes y las que se realizan en realidad.
- Valoraciones acerca de la rentabilidad de u automatismo de pruebas.

[1] Del Inglés: “Algorithm for Quality Assurance Based on UML Specification”

ISSN 1988-3455

2. Descripción de las actividades de validación de AQUABUS

En la investigación original [2] se perseguía, en resumen, comprobar las siguientes afirmaciones:

1. El diseño de casos de prueba es más completo en cuanto más completa sea la especificación del sistema.
2. El método propuesto (AQUABUS) es capaz de generar al menos todos los casos de prueba relacionados con datos que diseñaría un experto en desarrollo de software de manera artesanal.
3. El estudio del riesgo derivado del hecho de dejar una determinada prueba sin realizar implica un esfuerzo de análisis que ayuda a la correcta priorización de las mismas, reduciendo las probabilidades de estar probando inconscientemente aquello que es menos importante, aunque más evidente.
4. El esfuerzo de integración de herramientas de aseguramiento de calidad y del método propuesto en un proceso de desarrollo no solo es posible desde el punto de vista teórico, sino que además es rentable de cara a las organizaciones desde el punto de vista práctico.

Para conseguirlo se diseñaron las actividades experimentales que se describen a continuación.

2.1. Validar la influencia de la especificación

Para evaluar en qué medida afecta el tipo de especificación de partida al resultado final de la fase de pruebas, se diseñó un primer experimento que consistía en la realización de las actividades que pueden verse en la Figura 1 y que se explican a continuación:

1. Se entregaron las especificaciones de una aplicación sencilla en tres formatos distintos a sendos grupos distintos de programadores (fundamentalmente alumnos de 4º y 5º de Ingeniería Informática y 3º de Ingeniería Técnica de Gestión y Sistemas):
   - Una especificación meramente textual.
   - Una especificación basada en una plantilla de Casos de Uso pero sin diagramas UML adicionales.
   - Una especificación basada en una plantilla de Casos de Uso que incluía diagramas UML de actividad.
2. Cada programador, a partir de la especificación que les fue asignada en función del grupo al que pertenecía, se encargó de diseñar los casos de prueba.
3. Se compararon los casos de prueba generados por cada grupo con el fin de validar la afirmación de que con una mejor especificación de requisitos se consiguen mejores diseños de prueba (afirmación número 1).

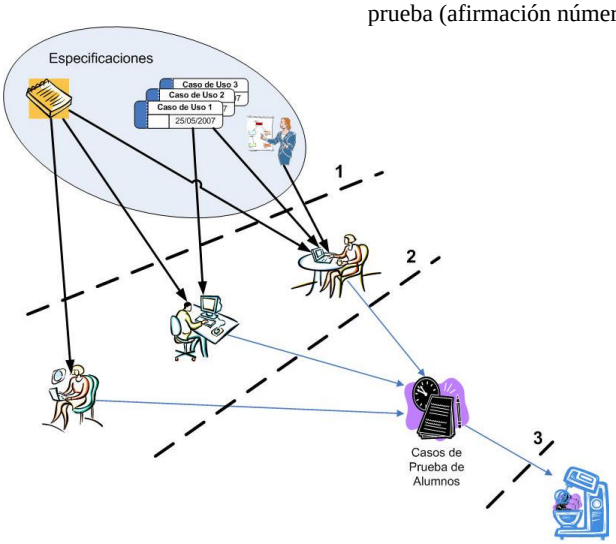

Figura 1. Esquema de validación de la especificación

2.2. Validación de AQUABUS

A la hora de validar el método que se proponía se pensó, fundamentalmente, en dos objetivos que coinciden con la segunda y tercera afirmación de las comentadas anteriormente. Para ello se llevaron a cabo las siguientes actividades previas:

1. Se diseñó e implementó una aplicación PHP de gestión de una base de datos de DVDs con el fin de que pudiera ser ejecutada a través de Internet desde cualquier navegador.
2. Se construyo una librería PHP para registrar en una base de datos, el paso por cada una de las actividades incluidas en los diagramas de actividad utilizados durante el diseño, aplicable a cualquier aplicación PHP y no sólo a la construida para esta validación.
3. Se añadieron, a dicha librería, funciones para registrar los datos introducidos por el usuario durante la ejecución de la aplicación.
4. Se integró la aplicación en un conjunto de páginas web desde las cuales se explicaba al usuario cómo utilizar y probar la aplicación.

Una vez construida la plataforma PHP de validación se llevo acabo el experimento según se describe a continuación (Figura 2):

1. En la página principal se mostraba al participante una breve introducción del marco de la experiencia junto con la especificación de la aplicación a probar y se le daba paso a iniciar la misma.
2. Una vez que el participante decidía empezar, se le explicaba cómo realizar adecuadamente el experimento.
3. Después se le pedía rellenar un formulario con una serie de información básica que servía para identificar el perfil profesional del participante.

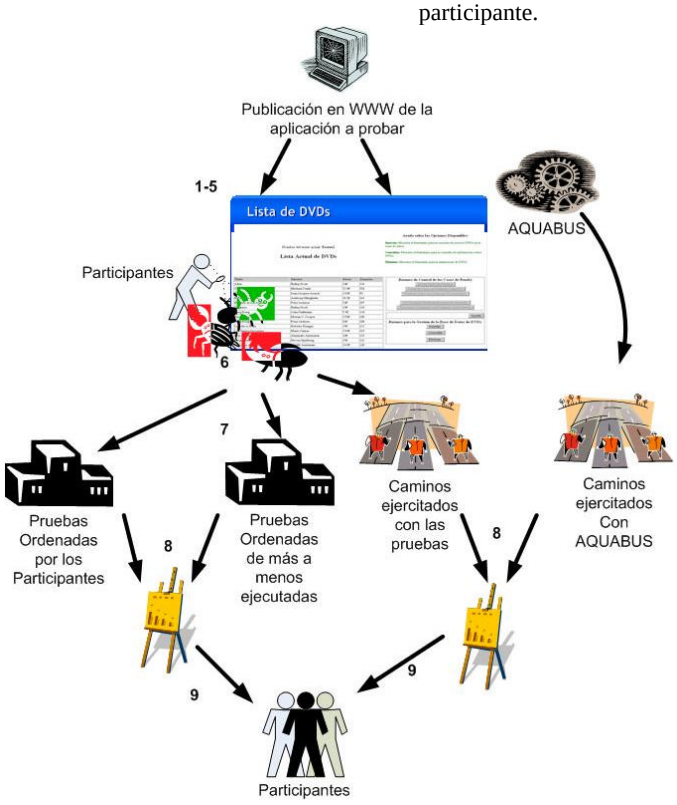

Figura 2. Esquema de validación del método AQUABUS

4. Una vez registrada la información anterior, el participante podía comenzar a realizar las pruebas sobre la aplicación desde una pantalla como la de la Figura 3 en la que tenía la opción de probar los tres tipos de casos de uso (inserción, consulta y borrado) de la aplicación.
5. Según se realizaban las pruebas, y dado que éstas se almacenaban en la base de datos, el participante podía consultarlas en cualquier momento con objeto de saber qué había probado ya y qué no. En un principio, se pensó en no darle esta opción al usuario puesto que era una manera de dar una herramienta de registro de pruebas de la que, en muchos casos, no se dispone cuando se prueba una aplicación.
6. Finalmente, se optó por la opción de dar un mínimo de información acerca de las pruebas que se iban realizando, para descargar de algo de trabajo a los participantes; uno de los principales riesgos del experimento era la posibilidad de que muchos de los participantes no lo acabaran porque les supusiese demasiada dedicación.
7. Cada vez que el participante realizaba una prueba debía indicar si el resultado de la prueba era o no exitoso; es decir, debía indicar si el sistema se había comportado o no tal y como se esperaba que lo hiciera (se habían insertado a propósito varios defectos de funcionamiento en el comportamiento del programa). Esto da origen a varios tipos de resultados:
   - Casos evaluados por el participante como pruebas que detectaron defectos y que, sin embargo, no lo hicieron porque la especificación así lo determina explicita o implícitamente.
   - Casos evaluados como pruebas que no detectan defectos y que, sin embargo, las especificaciones indican lo contrario. Por ejemplo, el hecho de no poder insertar un DVD con precio 0, cuando las especificaciones indican que sí debe permitirse para representar el caso de DVDs que se regalan con otras compras.
   - Casos evaluados correctamente por el participante, tanto en sentido positivo o negativo, en cuanto a la detección de defectos.
8. Cuando el usuario decidía que la aplicación estaba suficientemente probada, finalizaba la primera parte de la prueba y se le pedía que priorizase las pruebas que él mismo había realizado con el objetivo de evaluar si realmente se prueba con más insistencia lo que se considera más importante o más peligroso.
9. Se mostraban los resultados obtenidos y la comparación con la solución "buena" generada con AQUABUS.

Figura 3. Pantalla principal de prueba

2.3. Validación de la aplicabilidad

Para validar la necesidad y la aplicabilidad de un método automático de diseño de pruebas y en particular de AQUABUS, se pidió a los mismos participantes en el experimento anterior que respondieran un cuestionario que incluía dos tipos de preguntas:

- unas relacionadas con el modelo de diseño, ejecución y gestión de pruebas seguido en el entorno de trabajo del experimento
- otro grupo de cuestiones orientado a evaluar el grado de aplicabilidad y rentabilidad de la automatización del diseño de pruebas.

3. Resultados relevantes

Como ya se ha dicho al principio, los resultados que se muestran a continuación son un subconjunto de todos los obtenidos (ver [2]). En concreto, corresponden con aquellos resultados de los que se obtiene información adicional a la meramente necesaria para la validación del método que se proponía como objetivo inicial.

3.1. A mejor especificación mejores pruebas

Uno de los primeros datos, cuyo análisis va más allá de lo relacionado con AQUABUS, son los obtenidos durante el análisis de la influencia de la especificación en las pruebas. En este caso se comprueba claramente cómo una mejor especificación que sirva como punto de partida para la generación de casos de prueba, da como resultado un diseño más completo. La muestra de participación en esta fase de análisis está especificada en la Tabla 1 para los distintos grupos de trabajo creados para la comparación de resultados.

Se observa, en la Figura 4, que **cuando se dispone del diagrama de actividad** como parte de las especificaciones (grupo 3), **se produce un aumento ostensible tanto en el número de casos de prueba diseñados** (en color azul) **como en el número de valores probados para los datos** (en color granate)**.**

Tabla 1. Datos de la muestra de participación

Nº participantes	28
Nº particip. Grupo 1. (Espec. Textual)	8
Nº particip. Grupo 2. (Espec. Casos Uso)	8
Nº particip. Grupo 3. (Espec. Diag. Actividad)	12
Casos de prueba sin repeticiones[2] y sin incluir combinaciones de valores[3]	14
Total de posibles casos de prueba sin repeticiones y sin incluir combinaciones de valores	15
Total de casos de prueba con repeticiones[4] y sin incluir combinaciones de valores	52
Total de casos de prueba con repeticiones, incluidas las combinaciones de valores de datos[5]	569
Valores utilizados en alguna combinación de prueba sin repeticiones	19
Total de valores posibles en alguna combinación de prueba sin repeticiones	35
Valores utilizados en alguna combinación de prueba con repeticiones	62

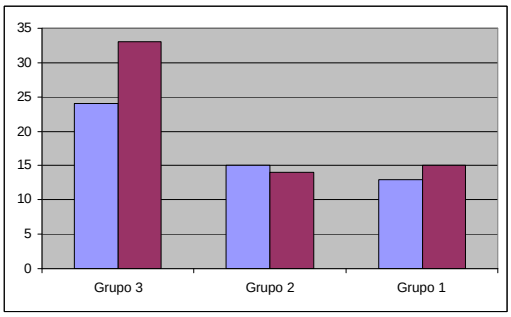

Figura 4. Datos obtenidos.

Por otro lado, al contrario de lo que hubiera podido pensarse, la diferencia entre tener una simple especificación textual (Grupo 1) y una especificación basada en casos de uso algo más elaborada (Grupo 2) no es demasiado importante si ésta no incluye diagramas de actividad.

[2] Contando una sola vez aquellos casos descubiertos por más de un participante
[3] Contando una sola vez aquellos casos que aparezcan varias veces con valores de prueba distintos para los datos
[4] Contando cada caso tantas veces como participantes lo hayan diseñado
[5] Contando cada caso tantas veces como combinaciones de valores se hayan diseñado para el mismo.

3.2. Prueba no sistemática: escasa y sin criterio

En este mismo experimento, se puede observar que frente al valor de referencia (total de casos generados con AQUABUS) los alumnos del grupo 3, en media, alcanzan el 71% del total de casos generados con el método, mientras que los de los otros dos grupos se quedan por debajo del 50%. Es decir que **la prueba no sistemática, incluso con una buena especificación tampoco alcanza un nivel adecuado**.

Cuando se pide a alguien que diseñe las pruebas para una aplicación, tal y como se le planteó a los participantes del primer experimento, implícitamente se está obligando al *tester* a que se pare a pensar, analizar y diseñar las pruebas; de otra manera, especialmente en una aplicación simple como ésta, seguramente nunca nadie diseñaría sino que las pruebas serían realizadas directamente "al vuelo".

Por este motivo se realizó el segundo experimento, en el que simplemente se les pedía probar, sin hablar en ningún momento de diseñar.

Los resultados son aún más significativos. De un total de 71 *testers*, tan **sólo 1 probó más del 75% de los caminos de prueba obtenidos por AQUABUS**, mientras que **el 56% de los participantes se quedaron en menos de la mitad de caminos**.

Otro resultado curioso es que, en media, **la mitad de las pruebas realizadas, probaban caminos ya probados previamente por ese mismo *tester*.**

Dado que también se les pedía priorizar sus pruebas, indicando cuáles realizarían en el caso de no tener tiempo o recursos para realizarlas todas, se obtuvieron resultados acerca de la coherencia de esta priorización. Comparando la lista de casos de prueba ordenada por prioridad, con la lista de casos de prueba ordenados por el número de veces que se repitió la prueba de dicho caso, se confirmó que a diferencia de lo que se pudiera pensar**, entre los diez caminos más probados, tan solo aparece uno de los diez más prioritarios** (en el puesto octavo)**.** Pero es que **entre los diez caminos menos probados aparecen tres de los 10 caminos valorados como más prioritarios.**

3.3. Automatización de Pruebas: Necesario y, en muchas ocasiones, rentable

Por último, de las respuestas a las preguntas de la encuesta, se obtuvieron los siguientes resultados en relación con la aplicabilidad y rentabilidad de la automatización de pruebas (la muestra es la descrita en la Tabla 2):

Tabla 2. Distribución de la muestra por perfiles

Perfil	%	Media de años de experiencia
Programadores	4%	2.5
Analistas	3%	4.5
Ingenieros de Software	6%	3.3
Jefes de Proyecto	8%	5.5
Testers	13%	3
Profesores de Ingeniería del Sw	3%	10.5
Profesores de Tecn. Información	10%	8.4
Estudiantes de últimos cursos	40%	
Otros	13%	

- Hay un 44% de encuestados que utiliza diagramas UML para diseñar, más un 21% que lo hace dependiendo del proyecto.
- Un 64% no utiliza UML para diseñar pruebas en modo alguno. Siendo un 55% el que no utiliza tampoco ningún otro método definido de diseño de pruebas.
- El 77% de los encuestados considera que a pesar de tener que elaborar diagramas adicionales para cada caso de uso la aplicación de algún tipo de método automático en sus organizaciones seguiría siendo rentable.
- El 70% considera rentable realizar una priorización previa sobre los casos generados.
- El 79% de los participantes consideran útil o muy útil el uso de un mecanismo que automatice el diseño y ejecución de las pruebas, siendo un 9% de los encuestados los que lo consideran imprescindible

4. Conclusiones

Como parte de una serie de experimentos de validación de un nuevo método automático de generación de casos de prueba, se ha podido obtener información valiosa sobre cómo los desarrolladores de software plantean las pruebas. Se ha podido constatar la influencia de una buena especificación en el diseño de pruebas a la vez que las limitaciones de eficacia (pobres resultados de exploración de las distintas opciones de funcionamiento y escenarios) y de eficiencia (repetición innecesaria de casos e insistencia en los menos importantes). De hecho, los propios profesionales han hecho ver que a menudo se prueba más lo que es más fácil de probar y no lo que se cree más importante.

Por último es importante destacar que no solo parece necesario la existencia de este tipo de métodos sino que, además una amplia mayoría de los profesionales participantes en la experiencia lo considerarían rentable dentro de sus organizaciones, lo cual permite despejar las dudas acerca de la posibilidad o no de asumir el sobreesfuerzo de la realización de un extenso diseño de pruebas.

Agradecimientos

Este trabajo ha sido parcialmente financiado por el Programa Nacional de I+D+i del Ministerio de Educación y Ciencia y fondos FEDER, con la Red para la Promoción y Mejora de las Pruebas en Ingeniería del Software: RePRIS (TIN2005-24792-E).

Referencias

[1] Fernández, L., Lara, P. J., Cuadrado, J.J., “Efficient software quality assurance approaches oriented to UML models in real life” en Dasso A. y Funes A. (eds.) *Verification, validation and testing in software engineering*, Idea Group, 2006, p. 341-377.

[2] Lara, P., “Método de diseño y priorización de casos de uso a partir de especificaciones UML”, Tesis doctoral, Universidad de Alcalá, 29 de marzo de 2007.

Modelo para la Capacitación de los Especialistas en Pruebas de Sistemas Software

Miguel Ángel García Palomo
Consultor Senior de Ingeniería Software
Responsable Formación
Métodos y Tecnología
Paseo de la Castellana, 182 Planta 10
28046 Madrid
miguel.garcia@mtp.es

Mamdouh Elcuera
Director de Ingeniería Software
Métodos y Tecnología
Paseo de la Castellana, 182 Planta 10
28046 Madrid
melcuera@mtp.es

Resumen

Se presenta un Modelo de Capacitación para los profesionales que han orientado su carrera profesional en una de las áreas de mayor demanda actualmente dentro del mercado TI, pero a su vez la menos formalizad especialmente en lo referente a formación.

El Modelo de Capacitación presentado está basado en roles, que habilitan para la realización de una serie de actividades y tareas dentro del Proceso de Pruebas.

Cada rol tiene asociados una serie de capítulos formativos específicos que le permiten la realización de dichas tareas, pero también es función de la experiencia individual.

1. Introducción

Las Organizaciones, especialmente las de tamaño medio y alto, basan cada vez más su Negocio en los Sistemas TIC. Además de esto, hay varios factores que han hecho de las Pruebas SW y de Sistemas una necesidad: por un lado la creciente complejidad, heterogeneidad y variabilidad de las tecnologías, que incrementa el riesgo. Por otro lado la presión del *time to market* y también la madurez del mercado (véase el incremento creciente de las certificaciones CMMI e ITIL por poner un ejemplo).

Por consiguiente, y es un hecho, en los últimos años han aumentado los presupuestos relacionados con las actividades de Pruebas y por consiguiente de la demanda de los Servicios relacionados. Esto no ha llevado parejo en cambio una mejora de la Calidad, y en algunos casos incluso hay retrocesos.

Las causas de esa falta (o no mejora) de Calidad son varias y tienes ciertas particularidades dependiendo de la organización concreta, pero siempre están presentes los denominadores comunes de *Procesos, Tecnología y Personas*, tal y como se muestra en la figura 1.

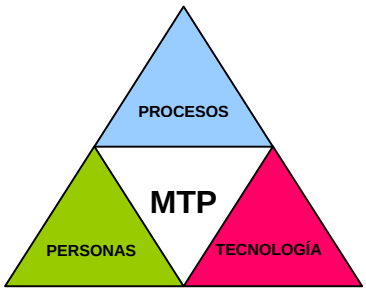

Figura 1. Tres dimensiones del Testing

En lo referente a Procesos y Tecnología existen varias aproximaciones, muchas válidas, siempre que se adapten de forma coherente a las necesidades y particularidades concretas de cada organización, pero lo que sigue siendo un verdadero agujero es la dimensión Personas.

ISSN 1988-3455

Aunque modelos como *People CMM* tratan el aspecto humano de madurez, su aplicación, por las dificultades que conlleva, sigue siendo residual.

Las actividades de pruebas son tan especializadas, o más, que otras dentro del ámbito TIC: desarrollo, operación, administracion de sistemas, etc. Sin embargo, la formación necesaria para alcanzar ese grado de especialización no se imparte en los centros habituales de formación, ni las empresas presentan planes de carrera para esa especialización.

Este hecho, unido a la creciente demanda de servicios, ya comentada, hace que cada vez se necesiten más profesionales y que éstos deban ser cada vez más competentes. Esta falta de recursos humanos plantea a las empresas la inmediata necesidad de formar a sus empleados.

Además, poco a poco el incremento de la demanda se va compensando con un incremento de la oferta, traducido en un número creciente de empresas que ofertan servicios de testing. Por tanto, la competencia entre las empresas que ofertan estos servicios es mayor, y es aquí donde la formación cobra una nueva dimensión más profunda que la de cubrir una necesidad inmediata.

También es necesario fidelizar a los empleados para evitar su marcha a la competencia y un recurso para ello es ofrecer perspectivas de futuro y de diversidad laboral mediante la capacitación en nuevas áreas.

No es suficiente con una formación ad hoc, impartida de cualquier modo o sobre cualquier tema. Es necesario orientar la formación a la consecución de los objetivos de pruebas, al empleo de la metodología de pruebas definida y al conocimiento de la tecnología.

2. Las tres dimensiones del Testing

El Modelo que permita la consecución de los objetivos de testing debe basarse en tres aspectos:

Procesos. Dentro de los procesos se definirán las fases de pruebas, las actividades y el lugar que ocupan en el ciclo de vida del software.

Tecnología. Incluye las infraestructuras de pruebas, los entornos y datos de prueba, las herramientas y los entornos.

Personas. Cubre los aspectos relativos a la organización de las pruebas, como los roles en un proyecto de pruebas y posibles modelos de organización.

Los tres aspectos están relacionados entre sí. Sin embargo, el punto de vista que interesa en este momento es el de *Personas*. Son las personas las que van a realizar las tareas y actividades definidas por los procesos y para ello tendrán que trabajar en los entornos de pruebas, haciendo uso de las herramientas, entornos, datos etc.

Por lo tanto, desde el punto de vista formativo, *Procesos* y *Tecnología* dirigirán las acciones de formación orientadas a que aumenten las competencias de las *Personas*.

3. El Modelo en V

La base de la definición de los procesos en esta solución metodológica es el *Modelo en V*. Este modelo considera las pruebas como un proceso que corre en paralelo con el análisis y el desarrollo, en lugar de constituir una fase aislada al final del proyecto.

En la representación gráfica clásica del Modelo en V (véase Figura 2), las fases de desarrollo de software aparecen a la izquierda y los correspondientes niveles de pruebas a la izquierda. Cada organización puede utilizar su versión del Modelo en V, basándolo en su propia terminología.

Partiendo de los requisitos o del diseño (a la izquierda) se deben planificar y preparar los niveles de pruebas correspondientes (a la derecha). En general, cada actividad de pruebas a la derecha valida la actividad enfrentada de la izquierda.

También en general, los niveles superiores de pruebas en el diagrama están basados en caja negra (pruebas basadas en las especificaciones) y los niveles inferiores están basados en caja blanca (basadas en la estructura interna de los componentes del sistema).

Cada uno de los niveles de pruebas de la Figura 2 tiene distintos objetivos, entornos, perfiles de personal, etc. Sin embargo, las distintas fases (que se verán a continuación) se repiten para cada nivel y tipo de pruebas.

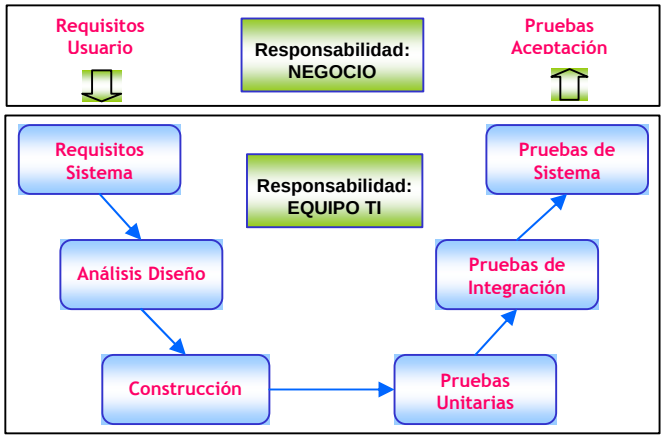

Figura 2. El Modelo en V

Además, existen distintos tipos de pruebas (funcionales, prestaciones, instalación, seguridad, usabilidad, instalación,…) y cada uno de los niveles de pruebas tiene asociados uno o varios de estos tipos.

4. Los *Procesos*

Además de definir los niveles de pruebas y los tipos de pruebas asociados, es necesario definir qué actividades es necesario realizar.

En la solución propuesta hay siete fases (véase Figura 3), cada una de las cuales se corresponde con un conjunto de actividades. Éstas, a su vez, tienen asignados roles, responsabilidades y entregables. Estas fases, con sus correspondientes actividades se repiten en cada iteración, tipo de pruebas o nivel de pruebas.

Un proyecto siempre comienza con la fase de *Planificación* para determinar el alcance de las pruebas, los recursos necesarios, el tiempo que se empleará, etc.

En la fase de *Diseño* se compone el plan de pruebas para cada iteración, tipo de prueba o nivel de pruebas, detallándose los elementos a probar y los casos o escenarios diseñados.

En la fase de *Construcción* se incluyen todas las actividades que permitirán la ejecución de los casos diseñados. Tras esta fase quedarán preparados los entornos, los datos de prueba y toda la infraestructura de pruebas.

Los casos diseñados y construidos se ejecutan en la fase de *Ejecución*. Los resultados se analizan e informan en tres ciclos: informe de los defectos, corrección de defectos y reejecución de las pruebas.

En la fase de *Cierre*, se evalúa el proceso completo y se preparan los entregables finales.

La *Gestión del Proyecto de Pruebas* es un conjunto continuo de actividades: seguimiento del proyecto, informes de estado, gestión del alcance, y entrega de entregables de calidad y en fechas.

El propósito de la *Gestión de la Calidad* es gestionar, dirigir y mejorar la calidad de los entregables del proceso (casos de prueba, escenarios, documentos, etc.).

ISSN 1988-3455

5. La *Tecnología*

Del mismo modo que los Procesos están presentes a lo largo de todo el ciclo de pruebas, la Tecnología también debe ser vista como algo que afecta a todas las fases de proceso.

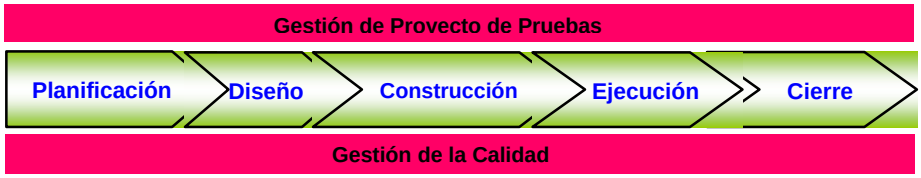

Figura 3. Fases del Ciclo de Pruebas

Así, cada una de las fases precisará de la utilización de diversas herramientas (de diseño de casos, de planificación, de gestión de pruebas, de edición de informes, etc.) que habilitan o simplemente facilitan la realización de las actividades asociadas.

Esto es especialmente cierto durante la *Fase de Ejecución*, en la que será necesario disponer de un entorno de pruebas que permita la ejecución de los casos. Este entorno debe constar de:

- Hardware y su configuración (PC, mainframe, etc.)
- Sistema operativo y software de sistema
- Red
- Interfaces con otras aplicaciones instaladas, stubs, drivers, etc.
- Datos de prueba
- Herramientas de prueba

Los requisitos de entorno serán diferentes dependiendo del tipo de pruebas o de niveles de prueba que se vayan a realizar. En unas pruebas de sistema conviene disponer de un entorno aislado, no compartido con otras aplicaciones. En unas pruebas de aceptación, se necesitará un entorno tan parecido al real como sea posible.

Los entornos de pruebas deben ser estables y gestionables, tan representativos del sistema bajo prueba como sea posible y deben permitir las modificaciones necesarias derivadas de los cambios en requisitos.

Estos cambios que pueden producirse en los entornos deben ser gestionados con los procedimientos adecuados de gestión de configuración.

La ejecución de las pruebas puede provocar cambios en los datos o incluso llevar a cabo acciones no deseadas, por lo cuál es necesario disponer de herramientas de backup y de restore.

6. Las *Personas*

A la hora de describir este aspecto del modelo de pruebas, el primer paso es definir qué roles van a desempeñar las distintas actividades. Es muy conveniente también definir los posibles itinerarios y relaciones entre los roles definidos.

6.1. El trabajo diario: los roles

Una propuesta de roles mediante los cuales podría implantarse el modelo de capacitación, con una breve descripción de sus responsabilidades, podría ser:

- *Ejecutor de pruebas*. Ejecuta casos o escenarios de prueba y documenta la ejecución.
- *Ingeniero de pruebas*. Diseña casos y escenarios de prueba, los documenta y los construye.
- *Jefe de Proyecto de Pruebas*. Planifica las pruebas, coordina el proyecto y resuelve los

problemas técnicos que puedan surgir, encargándose también del cierre.

- *Consultor de pruebas*. Puede realizar labor de consultoría en diversas áreas, como Metodología, Automatización, Infraestructura o Soporte de negocio.
- *Especialista en automatización de pruebas.* Tiene un conocimiento alto de los aspectos "teóricos" de las pruebas para las que se usan sus herramientas. Tiene conocimientos de desarrollo y/o arquitectura y está capacitado para crear y gestionar un test harness. Maneja una o varias herramientas de pruebas.
- *Gestor de Calidad*. Se encarga del control de la calidad de los entregables y de la definición y obtención de métricas de proceso y de producto.

Cada uno de estos roles tiene unas actividades asignadas dentro de un proyecto. Para desempeñar esas tareas requiere de unos conocimientos y de una experiencia.

La acción de mapear las actividades con los conocimientos necesarios para realizarlas permitirá:

- Identificar el rol o los roles que cada empleado es capaz de desempeñar en un proyecto.
- Determinar el grado de conocimientos que debe alcanzar para desempeñar nuevos roles.
- Establecer y planificar las acciones formativas necesarias para capacitar a los empleados.
- Establecer itinerarios de carrera para los empleados.

Las acciones formativas pueden ser llevadas a cabo por las propias empresas internamente, o contratar servicios de formación en empresas que ofrezcan cursos, seminarios, workshops o certificaciones oficiales.

La diferencia entre certificaciones oficiales y cursos internos no oficiales puede llevar a establecer niveles dentro de los distintos roles (por ejemplo, Ingeniero de Pruebas *Tipo A* y *Tipo B*), para identificar diferencias en la formación o en la experiencia sin que haya sin embargo una diferencia en las actividades.

6.2. La carrera profesional: los itinerarios

Una vez definidos los roles es posible identificar qué tareas puede realizar cada empleado y, por tanto, cuáles son los roles que puede desempeñar en un proyecto.

Los itinerarios son conjuntos de conocimientos y experiencia que deben adquirirse para que un empleado pase de un rol a otro.

Dos roles pueden necesitar capacidades comunes, con lo que puede ser fácil pasar de uno a otro. Sin embargo lo normal será que dos roles realicen tareas completamente distintas. Por lo tanto, para alcanzar la capacitación necesaria para un determinado rol, habrá que impartir unas acciones formativas, al tiempo que se dejan de lado otras, que capacitan para un rol distinto.

Los itinerarios permitirán a los empleados identificar la consecuencia lógica, en forma de acciones formativas futuras, de la realización de unas actividades concretas de formación o estimar el esfuerzo necesario para alcanzar la capacitación que desean.

En definitiva, les permitirá participar en el proceso continuo de maduración profesional.

6.3. Un ejemplo.

Una persona que comience su carrera profesional en una empresa de testing (o en general en cualquier empresa) necesita una capacitación mínima para poder empezar a trabajar. Parte de esa capacitación la ha obtenido durante sus estudios, pero el resto debe ser proporcionado por la empresa para comenzar con el rol de *Ejecutor de Pruebas.*

Para que pueda llevar a cabo las tareas propias de ese rol será necesario formarle en Pruebas (qué son y para qué sirven), Proceso de pruebas (fases, roles, actividades), Herramienta de gestión y seguimiento de pruebas, la arquitectura en la que van a tener lugar las pruebas y en manejo de bases de datos a nivel básico.

Esto se conseguiría proporcionándole unos cursos de *Introducción a Pruebas*, *Introducción a Herramientas de Gestión de Pruebas*,

ISSN 1988-3455

Introducción a Arquitectura de Sistema Bajo Pruebas e *Introducción a Manejo de Base de Datos.*

El rol inmediatamente superior al de *Ejecutor de Pruebas* sería *Ingeniero de Pruebas*. Para que un Ejecutor pueda pasar a Ingeniero de Pruebas se establece que debe acumular una experiencia de un año. En ese año, además, es necesario que amplíe sus conocimientos con un curso de Técnicas de Diseño de casos de Prueba, que le capacite para diseñar los casos y que obtenga la certificación oficial *ISTQB Certified Tester*.

Además, es posible ampliar sus conocimientos con cursos de mejora en *Arquitectura* y *Bases de Datos* o con nuevos cursos de iniciación a distintas herramientas de pruebas, en función de las necesidades.

Una vez que un empleado sea Ingeniero de Pruebas, deberá pasar entre uno y dos años desempeñando ese rol, periodo en el que deberá orientar su formación hacia el rol de *Consultor de pruebas* en alguno de sus áreas o hacia el de *Especialista en Herramientas de pruebas*, recibiendo los cursos necesarios para alcanzar la capacitación elegida.

En el caso de que empleados con experiencia previa en desarrollo puede que no sea necesario empezar como *Ejecutor de Pruebas*. Será necesario analizar la experiencia del empleado y sus preferencias. Así se podrán seleccionar acciones formativas concretas que permitan alcanzar un perfil adecuado.

Un camino natural para un desarrollador es la automatización de pruebas funcionales. También puede ser posible acceder de manera rápida al perfil *Ingeniero de Pruebas.*

7. Conclusión.

El modelo que se ha presentado plantea un objetivo global que una empresa debe marcarse a medio o largo plazo.

Es una tarea complicada pues exige un esfuerzo considerable en la definición de las tres dimensiones básicas al comienzo y después un esfuerzo para los Departamentos de Formación, que deberán gestionar las capacitaciones que permitan que el modelo funcione.

Por último, exigirá un esfuerzo continuo de mantenimiento por parte de los Departamentos de Formación, que deberán mantener actualizada su oferta formativa y adecuarla respondiendo a cualquier cambio que pueda producirse en la definición de las tres dimensiones de pruebas.

A cambio, la empresa contará con una importante base de conocimiento, residente en sus empleados, lo que la hará más competitiva comercial y técnicamente.

Además, la participación de los empleados en su formación, el establecimiento de metas profesionales que puedan traducirse en mejoras de empleo y salario, y el hecho de que la formación sea continua y de calidad, hará que el compromiso de éstos con la empresa crezca, evitándose los indeseables problemas que genera la rotación de personal.

Referencias

[1] Chrissis, Mary Beth, Mike Konrad y Sandy Shrum. *CMMI®: Guidelines for Process Integration and Product Improvement*, Addison Westley, 21 de Febrero de 2003

[2] IEEE 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology*

[3] IEEE 730-2002, *IEEE Standard for Software Quality Assurance Plans*

[4] ISO 9126-1:2001, *Software Engineering – Product Quality – Part 1: Quality Model*

[5] Spillner, Andreas, Tito Linz y Hans Schaefer, *Software Testing Foundations*, 2ª Edición, Rocky Nook Inc., Santa Barbara, 2007.

[6] *Standard Glossary of Terms Used in Software Testing*, International Software Testing Qualifications Board, versión 1.2, 4 de Junio de 2006

[7] van Veenendaal, Erik y Ron Swinkels. *Guidelines for Testing Maturity*, partes 1 y 2. Artículos publicados en *Professional Tester*, Tercer Volumen, números 1 y 2, March 2002.