Evaluando la Creación de Pruebas Unitarias para un Sistema Legado mediante Microsoft Fakes y Dobles de Prueba

Mario Barrantes[1*] y Alexandra Martínez[2]

[1] Intel, Estados Unidos
mario.alberto.barrantes.quesada@intel.com
[2] Escuela de Ciencias de la Computación e Informática,
Centro de Investigaciones en Tecnologías de la Información y Comunicación,
Universidad de Costa Rica, Costa Rica
alexandra.martinez@ecci.ucr.ac.cr

Resumen Este artículo evalúa la creación de pruebas unitarias para un sistema de código legado usando dos técnicas alternativas: el *framework* de aislamiento Microsoft Fakes y el método de dobles de prueba. La organización en la cual se efectuó este estudio no realiza pruebas unitarias, lo cual deriva en un alto costo de mantenimiento para sus sistemas pues debe ejecutar pruebas de regresión manuales cuando se realiza un cambio. Se usó entonces uno de sus sistemas legados como caso de estudio para evaluar cuál técnica de inclusión de pruebas unitarias resultaba más ventajosa para la organización. En primer lugar, se analizó la mantenibilidad del sistema legado. En segundo lugar, se seleccionaron los métodos a probar con pruebas unitarias y finalmente, se desarrollaron las pruebas usando el *framework* Microsoft Fakes y el método de dobles de prueba. Durante el desarrollo y la ejecución de las pruebas, se recolectaron las métricas necesarias para la comparación de ambos métodos. Los resultados de este caso de estudio muestran que Microsoft Fakes posibilita el desarrollo de pruebas unitarias en código legado sin necesidad de cambiar el código productivo y en un menor tiempo de desarrollo que dobles de prueba, pero a un mayor tiempo de ejecución.

Keywords: Código legado, pruebas unitarias, Microsoft Fakes, dobles de prueba

1. Introducción

Las pruebas unitarias son una parte importante del proceso de desarrollo de software pues constituyen la retroalimentación más temprana que tiene el desarrollador sobre su código. Recordemos que un defecto detectado en desarrollo es de 3 a 10 veces más barato de arreglar que si es detectado en producción [4]. Adicionalmente, durante la etapa de mantenimiento del software, las pruebas

[*] La afiliación actual del autor es Intel, pero esta investigación se desarrolló como parte de su investigación de posgrado en la Universidad de Costa Rica

unitarias son un buen mecanismo para identificar rápidamente regresiones en el software. De ahí el interés en incorporar pruebas unitarias no sólo al nuevo software que se desarrolle sino también al software existente que está en mantenimiento. La creación de pruebas unitarias automatizadas es una práctica estándar en desarrollos modernos de software, no obstante, su incorporación a software existente sigue siendo un reto debido a que los componentes de software deben aislarse en unidades que puedan ser probadas de forma independiente y no es usual que el software existente soporte tal nivel de aislamiento si no fue diseñado para ello.

En este trabajo utilizaremos el término "código legado" para referirnos a código (software) sin pruebas unitarias, siguiendo la definición dada por Feathers [7]. El código legado generalmente tiene que ser adaptado para que se puedan ejecutar pruebas unitarias sobre él. Esta adaptación se conoce como refactorización, que es un proceso de reestructuración interna del código tendiente a mejorar su diseño, legibilidad o mantenibilidad, sin alterar su comportamiento externo [7].

Las técnicas tradicionales para resolver el problema de aislamiento de unidades de software en código legado se conocen con el término genérico de "dobles de prueba" (*test doubles* en inglés) [3] y consisten en simular componentes de software que interactúan con el software a probar en forma de dependencia. Existen distintos tipos de dobles de prueba, entre los cuales están: maniquí (*dummy*), auxiliar (*stub*), espía (*spy*), falsificación (*fake*) y simulacro (*mock*) [6]. Los maniquíes son el tipo más sencillo de dobles de prueba pues no contienen implementación y se usan como valores de parámetros (bien pueden ser nulos). Los auxiliares son implementaciones mínimas de interfaces o clases base y normalmente devuelven valores preestablecidos, que obligan al software bajo prueba a ejecutar caminos específicos. Los espías son auxiliares con capacidad de grabación y generalmente guardan información sobre las llamadas a métodos hechas por el software bajo prueba, para ser verificado luego por las pruebas. Las falsificaciones son implementaciones más complejas que ofrecen la misma funcionalidad que el componente real pero de una forma más sencilla y se usan principalmente cuando el componente real no está disponible aún, es muy lento o no puede usarse en el ambiente de pruebas debido a sus efectos secundarios. Los simulacros son creados dinámicamente mediante una biblioteca, en lugar de ser codificados por un desarrollador de pruebas (como los otros dobles de prueba).

Microsoft Fakes es el nuevo *framework* de aislamiento que introdujo Microsoft en el 2012, mediante el cual se pueden reemplazar partes de código con *stubs* o *shims* para fines de pruebas [2]. Un *stub* remplaza una clase con un sustituto pequeño que implementa la misma interfaz y típicamente se usa para llamadas dentro de una solución que se pueden desacoplar usando interfaces. Un *shim* modifica en tiempo de ejecución el código compilado para inyectar un sustituto y se usa para llamadas a ensamblados de referencia para los que el código no está bajo nuestro control. La mayor diferencia entre ambos elementos es que los *shims* modifican el ensamblado de la dependencia que necesitamos sustituir, mientras los *stubs* únicamente inyectan una dependencia distinta (ajustada a lo que se necesita) de una interfaz o abstracción existente.

2. Trabajo Relacionado

El problema de incluir pruebas unitarias en código legado ha existido desde hace tiempo. Algunas investigaciones se han enfocado en desarrollar y evaluar herramientas para crear automáticamente pruebas unitarias. Un ejemplo de esto es el estudio realizado por Ramler y otros [8], el cual desarrolló un experimento que comparaba pruebas manuales contra pruebas generadas automáticamente. Las pruebas unitarias manuales las realizaron 48 estudiantes de maestría por un lapso de 60 minutos, mientras que las pruebas automatizadas las generaron mediante la herramienta de pruebas unitarias Randoop. El estudio arrojó que los 48 estudiantes en una hora cubrieron el 32 % del código y encontraron 24 errores, mientras Randoop cubrió el 29 % del código y encontró 9 errores.

Otros trabajos de investigación se han centrado en desarrollar técnicas de objetos de simulación. Estos objetos frecuentemente se necesitan para realizar pruebas unitarias, así que se han desarrollado *frameworks* que facilitan su inclusión en pruebas unitarias a partir de clases ya existentes. Un ejemplo de esta línea de investigación es el modelo tabular propuesto por Kim y otros [5], que permite tener trazabilidad entre las diferentes llamadas a métodos o funciones del objeto de simulación mediante una pila de llamados manejados por una máquina de estados de interacción. Sin embargo, a pesar de que este modelo ayuda a crear más pruebas unitarias, el esfuerzo que requiere la preparación consume mucho tiempo. Otra técnica de objetos de simulación propuesta por Samimi y otros [9] es la simulación declarativa, un esquema en el que los desarrolladores escriben especificaciones de métodos en un lenguaje de alto nivel para el API simulado y un solucionador de restricciones ejecuta dichas especificaciones cuando los métodos son invocados. Si bien este enfoque puede acelerar la creación de pruebas unitarias, solamente existe una extensión para Java.

Todas estas técnicas intentan agilizar el proceso de creación de pruebas unitarias, sin embargo, están orientadas -en su mayoría- al desarrollo de software guiado por pruebas. Lastimosamente, en código legado no se puede aplicar el desarrollo guiado por pruebas. Es por ello que ha surgido interés en investigar sobre cómo incorporar pruebas unitarias a código legado. Un ejemplo de esta línea de investigación es el trabajo de Shihab y otros [11], que propone un nuevo concepto llamado “Mantenimiento guiado por pruebas”, el cual provee una serie de heurísticas para trabajar en un proceso similar al desarrollo guiado por pruebas, pero con código legado.

3. Contexto

La Universidad de Costa Rica cuenta con un conjunto de sistemas informáticos creados por varias unidades de desarrollo de software institucional. El presente estudio fue desarrollado en una de estas unidades, que posee una extensa base de código legado en producción. La mayoría de este código no fue diseñado para ser probado mediante pruebas unitarias, sin embargo, está en constante mantenimiento, lo cual implica un alto costo pues se deben ejecutar pruebas de regresión manuales cada vez que se realiza un cambio sobre el código.

La unidad de desarrollo en cuestión, de ahora en adelante llamada "la organización", se beneficiaría en gran medida de la incorporación de pruebas unitarias a sus sistemas legados, ya que servirían como pruebas de regresión automatizadas, reduciendo el tiempo y el costo de mantenimiento de sus sistemas. Uno de los retos de incorporar pruebas unitarias a este código legado es el fuerte acoplamiento entre sus componentes de software. Para solventarlo, se debe recurrir a la refactorización, o bien, a un *framework* de aislamiento como Microsoft Fakes.

Esta investigación propone entonces utilizar uno de los sistemas legados de la organización como caso de estudio para evaluar dos técnicas de creación de pruebas unitarias: el *framework* Microsoft Fakes y el método de dobles de prueba con refactorización. Se espera que las conclusiones derivadas de este caso de estudio ayuden a la organización a elegir la técnica que les resulte más ventajosa.

4. Metodología

4.1. El sistema legado

El sistema legado escogido para este estudio es una aplicación web desarrollada en ASP.Net utilizando el lenguaje de programación Visual Basic. El sistema cuenta con 87 casos de pruebas manuales y funcionales documentados, pero ninguna prueba unitaria. El sistema utiliza un modelo típico de tres capas: la capa de presentación, la capa de lógica de negocios y la capa de acceso a datos. La capa de lógica de negocios accede a varios servicios web de otros sistemas de la organización. La mayor parte de la lógica del sistema se encuentra en la capa de lógica de negocios (como es usual), por lo que para efectos de los análisis realizados, métricas recolectadas y pruebas realizadas en el estudio, se usó solamente el proyecto de lógica de negocios del sistema legado.

4.2. Pasos de la metodología

El primer paso de la metodología de investigación fue realizar un análisis de mantenibilidad del sistema legado. El segundo paso fue seleccionar los métodos a probar del sistema legado. El tercer paso fue determinar las métricas que se recolectarían y utilizarían en la comparación de las técnicas de pruebas unitarias. El cuarto paso fue la creación y ejecución de las pruebas unitarias bajo ambas técnicas: Microsoft Fakes y dobles de prueba. A continuación se detalla cada paso de la metodología.

Análisis de mantenibilidad del sistema legado. La organización nos facilitó el código fuente del sistema legado así como documentos de diseño, de arquitectura y de pruebas del sistema. Con estos insumos, se procedió a estudiar la arquitectura del sistema y el código mismo. Se extrajeron las siguientes métricas de mantenibilidad: el índice de mantenibilidad, la complejidad ciclomática y el acoplamiento de clases. El 'índice de mantenibilidad'combina funciones de

la complejidad ciclomática, el volumen Halstead y la cantidad de líneas de código por módulo [12]. La 'complejidad ciclomática' mide la cantidad de caminos linealmente independientes en el código [1]. El 'acoplamiento de clases' mide la cantidad de entidades que dependen de un determinado miembro de una clase [4]. Todas estas métricas permiten evaluar la facilidad o dificultad de realizar pruebas unitarias sobre un código.

Selección de los métodos a probar. Para poner en contexto el sistema bajo estudio, se sabe que este sistema fue desarrollado y ha sido parte de una organización dentro de la UCR por varios años y a través del tiempo se le han implementado cambios y arreglos. Para seleccionar los métodos a los cuales se les aplicarían pruebas unitarias, se definieron los siguientes criterios a satisfacer:

- El método debía estar en el proyecto de Lógica de Negocios del sistema, ya que en esta capa es donde se realiza la mayor parte de lógica del sistema.
- El método debía tener lógica y no ser un simple método de paso de datos. Durante la investigación se detectó que mucho código era generado automáticamente mediante plantillas básicas de conexión y acceso a la base de datos, por lo que esos métodos básicos fueron ignorados.
- El método debía tener más de un caso de prueba posible. En otras palabras, debía ser lo suficientemente complejo (en cantidad de ramificaciones) para aplicar al menos dos casos de prueba.
- El método debía poseer dependencias. Todos los métodos de la Lógica de Negocios tenían al menos una dependencia.

Después de aplicar estos criterios a los métodos de la capa de lógica de negocio del sistema legado, quedaron seleccionados los siguientes métodos:

1. **DeshacerRegistroTramitado**: este método tiene la lógica de deshacer una transacción realizada. Fue seleccionado debido a que pueden existir diferentes casos de prueba, por ejemplo, si se intenta deshacer un registro que no existe o que está vació, o cuando el registro tiene un monto.
2. **InsertarNotificar**: este método inserta un registro y notifica por correo electrónico. Fue seleccionado por que existen muchos casos de prueba que se pueden automatizar y además porque agrega varias dependencias, entre ellas: la base de datos y el servidor de correos electrónicos.
3. **ModificarNotificar**: este método modifica un registro y notifica por correo electrónico. Fue seleccionado por las mismas razones que el anterior.
4. **TotalRubrosPorEntidad**: este método calcula el total de rubros por entidad. Tiene muchas dependencias con tablas y estructuras de datos de la aplicación y también tiene muchas ramificaciones y posibles casos de prueba.

Elección de las métricas a recolectar. Se eligió recolectar las siguientes métricas sobre la creación y ejecución de las pruebas unitarias:

a. **Tiempo invertido en el desarrollo de las pruebas (TD)**: mide el tiempo de desarrollo de las pruebas unitarias, incluyendo el arreglo de los datos, la

refactorización (de ser necesaria), las aserciones y la depuración. La medición de este tiempo se hizo de forma manual.

b. **Tiempo de ejecución de las pruebas (TE)**: mide el tiempo que toma ejecutar las pruebas unitarias. Como el tiempo de ejecución puede variar por factores no controlables (el estado del sistema operativo en el momento de correr las pruebas unitarias, el uso de la red, el uso del procesador, etc.), se tomó la decisión de ejecutar las pruebas diez veces y tomar el promedio de estas corridas.
c. **Cobertura de código (CC)**: mide el porcentaje de líneas de código del sistema legado cubiertas (o ejercitadas) por las pruebas unitarias. Este porcentaje se obtuvo mediante la herramienta de análisis de cobertura que ofrece la herramienta Visual Studio.
d. **Cantidad de líneas de código productivo modificadas (CCM)**: cuenta la cantidad de líneas de código productivo que tuvieron que ser modificadas, eliminadas o agregadas para incorporar pruebas unitarias.
e. **Total de líneas de código productivo (TC)**: cuenta el total de líneas de código productivo bajo prueba (en nuestro caso, se contabilizó sólo el código pertinente de la capa de Lógica de Negocios).
f. **Porcentaje de código modificado (PCM)**: es la razón de CCM a TC.

Creación y ejecución de pruebas unitarias. Para la creación de pruebas unitarias, primeramente se estudió la documentación tanto de Microsoft Fakes como del método de dobles de pruebas con refactorización. Una vez estudiadas y entendidas ambas técnicas, se procedió a aplicarlas sobre el código legado en cuestión. Las pruebas unitarias se diseñaron intentando alcanzar la mayor cobertura de código posible. Todas las pruebas fueron desarrolladas por un solo desarrollador (primer autor del artículo). Cada prueba se ejecutó diez veces, para efectos de medición del tiempo de ejecución.

5. Resultados y Discusión

Mantenibilidad del sistema legado. El cuadro 1 muestra el resumen (promedio, valor máximo y valor mínimo) de las métricas de mantenibilidad obtenidas sobre la capa de lógica de negocios del sistema legado, que contenía 172 miembros. Las métricas se obtuvieron mediante análisis estático de código usando la herramienta Visual Studio. Del cuadro 1 se observa que el promedio del índice de mantenibilidad es normal (el rango de valores de esta métrica es de 0 a 100), pero existen 22 miembros de clases con un índice de mantenibilidad por debajo de 35 (valores bajos representan código poco mantenible). La complejidad ciclomática promedio es baja, pero hay 28 miembros de clases que presentan una complejidad ciclomática mayor a 10 (un valor de complejidad ciclomática de 10 o menor es un buen indicador de código mantenible [13]). Por otro lado, el promedio de acoplamiento de clases es alto, lo cual refleja la existencia de 40 miembros de clases con acoplamiento mayor a 20 (un valor de acoplamiento de clases de 9 o menor indica código mantenible [10]).

Cuadro 1. Resumen de las métricas de mantenibilidad del sistema legado.

	Promedio	Max	Min
Indice de mantenibilidad	56.1	98	23
Complejidad Ciclomática	5.0	30	1
Acoplamiento de clases	10.3	42	0

Un estudio más detallado del código del sistema legado determinó que el alto acoplamiento de clases se debe a las dependencias que tiene la capa de lógica de negocios con la capa de acceso a datos, servicios externos que consume y otros utilitarios. Todas estas dependencias son consumidas utilizando implementaciones concretas y no existe inyección de dependencias.

Los resultados arrojados por este análisis de mantenibilidad muestran que el sistema legado tiene un alto acoplamiento de clases, lo cual advierte que el proceso de refactorización no será fácil. El nivel de acoplamiento de clases indirectamente permite estimar qué tanta refactorización de código será necesaria al crear pruebas unitarias con las técnicas tradicionales (como dobles de pruebas).

Creación y ejecución de las pruebas unitarias. Se diseñaron 14 pruebas unitarias sobre los 4 métodos seleccionados del sistema legado, para un total de 28 pruebas unitarias implementadas (14 con cada técnica). El cuadro 2 muestra las métricas recolectadas durante el desarrollo y la ejecución de las pruebas unitarias, tanto usando el *framework* Microsoft Fakes como el método de dobles de prueba con refactorización. Aquí, TD es el tiempo de desarrollo de la prueba unitaria en minutos y TE es el tiempo de ejecución de la prueba unitaria en milisegundos. Del cuadro 2 se concluye que el tiempo de desarrollo de las pruebas unitarias fue 3.3 veces mayor usando dobles de prueba, mientras que el tiempo de ejecución de las pruebas fue 2.8 veces mayor usando Microsoft Fakes.

Por otro lado, el porcentaje de cobertura de código (métrica CC) fue de 100 % en ambos casos. El total de líneas de código productivo (métrica TC) en la capa de lógica de negocio fue 35,347. La cantidad de líneas de código productivo modificadas (métrica CCM) fue 0 cuando se utilizó Microsoft Fakes y 3,445 cuando se usó dobles de prueba. El porcentaje de código modificado (métrica PCM) resultó en 0 % para Microsoft Fakes y 9.7 % para dobles de prueba.

Las diferencias en tiempo de desarrollo y tiempo de ejecución entre ambas técnicas se deben a la forma como trabaja cada una de ellas. En el caso de los dobles de prueba, el desarrollador debe crear manualmente los objetos de simulación y refactorizar el código, lo cual incrementa el tiempo de desarrollo, pero mejora el tiempo de ejecución. El tiempo de ejecución mejora porque el código con los objetos de simulación es compilado y por ende el ejecutor de pruebas no tiene sobrecarga adicional en tiempo de ejecución. En contraposición, cuando se usa Microsoft Fakes, el ejecutor de pruebas debe ensamblar los proyectos *fakes* creados por el *framework* junto con el código de la prueba unitaria antes de ejecutar la prueba. Entre más grande sea el ensamblado que se está falsificando con Microsoft Fakes, mayor será el tiempo de ejecución.

Cuadro 2. Métricas recolectadas sobre las pruebas unitarias. TD es el tiempo de desarrollo en minutos y TE es el tiempo de ejecución en milisegundos.

	Microsoft Fakes		Dobles de prueba	
Prueba unitaria	**TD**	**TE**	**TD**	**TE**
1	25	21	90	6
2	20	1	15	1
3	15	3	90	1
4	15	1	12	1
5	20	2	70	2
6	10	8	10	6
7	20	1	125	1
8	8	324	12	33
9	45	1	180	1
10	5	6	10	5
11	6	15	10	56
12	5	1	15	1
13	4	125	15	67
14	5	5	15	3
Total	203	514	669	184
Promedio	14.5	36.71	47.79	13.14

Vale la pena mencionar que el alto acoplamiento de clases detectado durante el análisis de mantenibilidad incidió en el tiempo de desarrollo de las pruebas que se hicieron con dobles de pruebas, pues estas requirieron una refactorización del 9.7 % del código legado productivo. Como el *framework* de aislamiento Microsoft Fakes no requirió modificar el código productivo (evitó refactorizar), el tiempo de desarrollo de las pruebas con esta técnica fue menor.

Valoración de las técnicas de pruebas. Las ventajas encontradas al utilizar el *framework* Microsoft Fakes fueron:

- Se pudieron crear pruebas unitarias sobre código legado sin necesidad de modificar el código productivo original, lo cual elimina el riesgo de regresiones.
- El tiempo de desarrollo de las pruebas fue bajo en comparación con dobles de prueba.

Las desventajas encontradas al utilizar el *framework* Microsoft Fakes fueron:

- El tiempo de ejecución de las pruebas fue alto en comparación con dobles de prueba.
- Al omitir el proceso de refactorización (el cual es saludable), permitió probar código que incumplía con principios de buen diseño de software.

Las ventajas encontradas al utilizar la técnica de dobles de prueba fueron:

- El tiempo de ejecución de las pruebas fue bajo en comparación con el de Microsoft Fakes.
- Requirió del proceso de refactorización, el cual es conveniente desde la perspectiva de buen diseño de software.

Por otro lado, las desventajas que se encontraron con dobles de prueba fueron:

- El tiempo de desarrollo de las pruebas fue alto en comparación con el de Microsoft Fakes.
- El riesgo a introducir nuevos defectos es mayor que con Microsoft Fakes, pues la refactorización es obligatoria para crear las pruebas.

6. Conclusiones

Esta investigación comparó el uso del *framework* de aislamiento Microsoft Fakes contra la técnica tradicional de dobles de prueba con refactorización, para agregar pruebas unitarias a un sistema de código legado. El sistema legado es una aplicación web para uso interno de la Universidad de Costa Rica, desarrollado por una unidad de desarrollo de software institucional.

En este trabajo se analizó la mantenibilidad del sistema legado, utilizando las métricas de índice de mantenibilidad, complejidad ciclomática y acoplamiento de clases. Dicho análisis señaló que el mayor problema de mantenibilidad del sistema legado en estudio era su alto acoplamiento de clases, el cual es producido por la falta de inyección de dependencias. Luego de este análisis, se seleccionaron cuatro métodos de la capa de lógica de negocio para ser probados mediante pruebas unitarias. Sobre estos métodos se implementaron 14 pruebas unitarias con Microsoft Fakes y con dobles de prueba. Durante su implementación, se recolectaron métricas sobre el tiempo de desarrollo y ejecución de las pruebas unitarias, así como sobre la cantidad de líneas de código que tuvieron que ser modificadas para crear las pruebas.

Los resultados indican que el *framework* Microsoft Fakes logró aislar las dependencias que tenía el código con servicios externos a los que la capa de lógica de negocio se conectaba, posibilitando así la creación de pruebas unitarias automatizadas sin necesidad de cambiar una sola línea de código productivo. (Se optó por aislar y no simular estos servicios ya que la simulación hubiese sido más costosa.) Los resultados también indican que la técnica de dobles de prueba logró crear pruebas unitarias automatizadas sobre el código legado, pero requirió la creación manual de los objetos de simulación y la respectiva refactorización del código productivo. En este caso de estudio, para usar la técnica de dobles de prueba se tuvo que modificar un 9.7 % del código productivo. Otro resultado interesante para este caso de estudio es que el tiempo de desarrollo de las pruebas unitarias con Microsoft Fakes fue 3.3 veces menor que con dobles de prueba, mientras que el tiempo de ejecución de las pruebas con Microsoft Fakes fue 2.8 veces mayor que con dobles de pruebas.

Finalmente, nuestra recomendación es utilizar el *framework* Microsoft Fakes en código legado que posea un alto nivel de acoplamiento de clases (mayor a 9) y que esté sujeto a mantenimientos pequeños. Por el contrario, en código legado con buenos indicadores de mantenibilidad (en especial, bajo acoplamiento de clases), se recomienda usar dobles de prueba ya que el esfuerzo de refactorización no sería muy grande. Similarmente, si el código legado va a sufrir cambios grandes, se

recomienda utilizar dobles de pruebas para obtener un mejor diseño del software a través de la refactorización. Finalmente, para desarrollo de código nuevo se recomienda usar métodos tradicionales de dobles de prueba y refactorización, que ayuden a producir software con bajo acoplamiento de clases.

Agradecimientos

Agradecemos al Centro de Investigaciones en Tecnologías de la Información y Comunicación de la Universidad de Costa Rica por servir de enlace con la organización donde se efectuó el estudio, proveer la infraestructura tecnológica y brindar apoyo técnico durante la investigación. Agradecemos también a la organización que desarrolló el sistema legado aquí estudiado.

Referencias

1. J.L. Anderson. Using software tools and metrics to produce better quality test software. In *AUTOTESTCON*, pages 293–297, 2004.
2. Microsoft Corporation. *Exploring The Continuum Of Test Doubles*. Microsoft Corporation, 2007.
3. Microsoft Corporation. *Better Unit Testing with Microsoft Fakes*. Microsoft Corporation, 2013.
4. Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004.
5. Taeksu Kim, Chanjin Park, and Chisu Wu. Mock object models for test driven development. In *4th International Conference on Software Engineering Research, Management and Applications*, pages 221–228, 2006.
6. Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.
7. Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.
8. R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *38th Conference on Software Engineering and Advanced Applications*, pages 286–293, 2012.
9. Hesam Samimi, Rebecca Hicks, Ari Fogel, and Todd Millstein. Declarative mocking. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 246–256, New York, NY, USA, 2013. ACM.
10. R. Shatnawi. A quantitative investigation of the acceptable risk levels of object-oriented metrics in open-source systems. *IEEE Transactions on Software Engineering*, 36(2):216–225, March 2010.
11. E. Shihab, Z. M. Jiang, B. Adams, A.E. Hassan, and R. Bowerman. Prioritizing unit test creation for test-driven maintenance of legacy systems. In *10th International Conference on Quality Software*, pages 132–141, 2010.
12. Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison-Wesley Professional, 2006.
13. Arthur H. Watson, Thomas J. Mccabe, and Dolores R. Wallace. Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.