



Incluye CD

# Pruebas de Software y JUnit

Un análisis en profundidad y ejemplos prácticos

Daniel Bolaños Alonso  
Almudena Sierra Alonso  
Miren Idoia Alarcón Rodríguez

PEARSON  
Prentice  
Hall



# **Pruebas de Software y JUnit**

**Un análisis en profundidad y ejemplos prácticos**



# **Pruebas de Software y JUnit**

## **Un análisis en profundidad y ejemplos prácticos**

**Daniel Bolaños Alonso**

*Departamento de Ingeniería Informática  
Universidad Autónoma de Madrid*

**Almudena Sierra Alonso**

*Departamento de Ciencias de la Computación  
Universidad Rey Juan Carlos*

**Miren Idoia Alarcón Rodríguez**

*Departamento de Ingeniería Informática  
Universidad Autónoma de Madrid*



Madrid • México • Santa Fe de Bogotá • Buenos Aires • Caracas • Lima • Montevideo  
San Juan • San José • Santiago • São Paulo • White Plains

**PRUEBAS DE SOFTWARE Y JUNIT  
UN ANÁLISIS EN PROFUNDIDAD Y EJEMPLOS PRÁCTICOS**

Bolaños Alonso, D.; Sierra Alonso, A.; Alarcón Rodríguez, M. I.

PEARSON EDUCACIÓN, S.A., Madrid, 2007

ISBN: 978-84-8322-354-3

Materia: Informática, 004

Formato: 195 × 250 mm

Páginas: 368

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual (*arts. 270 y sgts. Código Penal*).

**DERECHOS RESERVADOS**

© 2007 por PEARSON EDUCACIÓN, S. A.

Ribera del Loira, 28

28042 Madrid (España)

**PRUEBAS DE SOFTWARE Y JUNIT  
UN ANÁLISIS EN PROFUNDIDAD Y EJEMPLOS PRÁCTICOS**

Bolaños, D.; Sierra, A.; Alarcón, M. I.

**ISBN:** 978-84-8322-354-3

Depósito Legal: M.

PEARSON PRENTICE HALL es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

**Equipo editorial:**

**Editor:** Miguel Martín-Romo

**Técnico editorial:** Marta Caicoya

**Equipo de producción:**

**Director:** José A. Clares

**Técnico:** José A. Hernán

**Diseño de cubierta:** Equipo de diseño de PEARSON EDUCACIÓN, S. A.

**Composición:** JOSUR TRATAMIENTOS DE TEXTOS, S.L.

**Impreso por:**

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

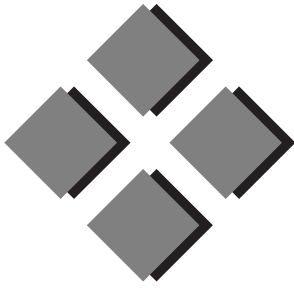
A Jesús, Genoveva y Berta.  
*Daniel Bolaños Alonso*

A Jorge y Alejandro  
*Almudena Sierra Alonso*

A Ángel y Paula  
*Miren Idoia Alarcón*







# Índice

---

<b>Prefacio</b> .....	xv
<b>Capítulo 1. Fundamentos de las pruebas de software</b> .....	1
1.1. Introducción .....	1
1.2. Principios básicos .....	2
1.2.1. Verificación y validación .....	3
1.3. Tareas básicas .....	4
1.4. Inspecciones de código .....	5
1.5. Pruebas basadas en la ejecución del código: técnicas .....	5
1.5.1. Pruebas de caja blanca .....	5
1.5.1.1. Pruebas de interfaces entre módulos o clases .....	6
1.5.1.2. Prueba de estructuras de datos locales .....	6
1.5.1.3. Prueba del camino básico .....	6
1.5.1.4. Pruebas de condiciones límite .....	8
1.5.1.5. Pruebas de condición .....	9
1.5.2. Pruebas de caja negra .....	12
1.5.2.1. Partición de equivalencia .....	13
1.5.2.2. Análisis de valores límite .....	13
1.6. Diseño de casos de prueba .....	13
1.7. Estrategia de pruebas .....	18
1.7.1. Pruebas unitarias .....	20
1.7.2. Pruebas de integración .....	21
1.7.2.1. Pruebas de integración ascendentes .....	21
1.7.2.2. Pruebas de integración descendentes .....	21
1.7.2.3. Pruebas de integración sandwich .....	22
1.7.2.4. Elección del módulo/clase crítico/a .....	22
1.7.2.5. Acoplamiento y cohesión .....	22

1.7.3.	Pruebas de validación	23
1.7.3.1.	Pruebas alfa y beta	24
1.7.4.	Pruebas del sistema	24
1.7.5.	Pruebas de aceptación	24
1.8.	Pruebas de sistemas orientados a objetos	25
1.8.1.	Pruebas de clases de objetos	26
1.8.2.	Pruebas de integración de objetos	26
1.8.2.1.	Pruebas de interfaces	27
1.8.3.	Pruebas del sistema	27
1.9.	Depuración de errores	27
1.10.	Otras pruebas	28
1.10.1.	Pruebas de regresión	28
1.10.2.	Pruebas de estrés	28
1.10.3.	Pruebas de interfaz de usuario	28
1.11.	Criterios para dar por finalizadas las pruebas	28
1.12.	Equipo de pruebas	29
1.13.	Errores más comunes que se cometen en la fase de pruebas	30
1.14.	Documentación de pruebas	30
1.15.	Bibliografía	33
<b>Capítulo 2.</b>	<b>Pruebas unitarias: JUnit</b>	<b>35</b>
2.1.	Introducción	35
2.1.1.	Aportaciones de JUnit	36
2.1.2.	Versiones	37
2.2.	Instalación	38
2.2.1.	Comprobación de la correcta instalación de JUnit	39
2.3.	Primera toma de contacto con JUnit	39
2.4.	Creación de una clase de prueba	41
2.4.1.	Creación de una clase de pruebas con JUnit 3.8.1	43
2.4.2.	Creación de una clase de pruebas con JUnit 4.x	47
2.5.	Conceptos básicos	50
2.5.1.	Prueba de constructores	51
2.5.1.1.	Procedimiento de prueba de un constructor	51
2.5.2.	Prueba de métodos get y set	54
2.5.2.1.	Prueba de métodos get y set mediante la técnica de caja blanca	56
2.5.2.2.	Prueba de métodos get y set mediante la técnica de caja negra	57
2.5.3.	Prueba de métodos convencionales	57
2.5.3.1.	Casos particulares:	59

2.6.	Organización de las clases de prueba .....	60
2.7.	Ejecución de los casos de prueba .....	62
2.7.1.	Mecanismos de ejecución de los casos de prueba .....	62
2.7.1.1.	Ejecución en modo texto .....	63
2.7.1.2.	Ejecución en modo gráfico .....	64
2.7.2.	Interpretación de los resultados obtenidos .....	67
2.7.2.1.	Concepto de error en JUnit .....	67
2.7.2.2.	Concepto de fallo en JUnit .....	68
2.8.	Conceptos avanzados en la prueba de clases Java .....	69
2.8.1.	Prueba de excepciones .....	70
2.8.1.1.	Excepciones esperadas .....	70
2.8.1.2.	Excepciones no esperadas .....	76
2.8.2.	Prueba de métodos que no pertenecen a la interfaz pública .....	76
2.8.2.1.	Prueba de forma indirecta .....	77
2.8.2.2.	Modificar el atributo de privacidad de modo que los métodos sean accesibles desde el paquete .....	78
2.8.2.3.	Utilizar clases anidadas .....	79
2.8.2.4.	Utilizar la API de Reflection de Java .....	79
2.9.	Bibliografía .....	81
<b>Capítulo 3.</b>	<b>Ant .....</b>	<b>83</b>
3.1.	Introducción .....	83
3.2.	Instalación y configuración .....	84
3.3.	Conceptos básicos .....	85
3.3.1.	Propiedades .....	87
3.3.1.1.	Estructuras <i>Path-Like</i> .....	87
3.3.1.2.	Grupos de ficheros y directorios .....	88
3.3.2.	Objetivos .....	90
3.3.3.	Tareas .....	90
3.3.3.1.	Tareas sobre ficheros .....	91
3.3.3.2.	Tareas de acceso al sistema de ficheros .....	92
3.3.3.3.	Tareas de compilación .....	93
3.3.3.4.	Tareas de documentación .....	93
3.3.3.5.	Tareas de ejecución .....	94
3.3.3.6.	Tareas para la definición de propiedades .....	95
3.3.3.7.	Tareas relacionadas con las pruebas .....	95
3.3.3.8.	Tareas definidas por el usuario .....	96
3.3.3.9.	Otras tareas .....	98
3.4.	Creación de un proyecto básico .....	99
3.5.	Ejecución de los casos de prueba mediante Ant .....	101
3.6.	Bibliografía .....	108

<b>Capítulo 4. Gestión de la configuración del Software</b>	109
4.1. Introducción	109
4.2. Principios básicos	110
4.3. Objetivos	111
4.4. Líneas base	112
4.4.1. Elementos de configuración	112
4.4.2. Líneas base	112
4.5. Actividades	114
4.6. Control de cambios	115
4.6.1. Motivos del cambio	117
4.7. Herramientas de GCS	118
4.8. Documentación	119
4.8.1. Plan de GCS	120
4.8.2. Formulario de Petición de Cambios	120
4.8.3. Informes de Cambios	121
4.8.4. Otros documentos	123
4.9. Bibliografía	123
<b>Capítulo 5. Herramientas de control de versiones: Subversion (SVN)</b>	125
5.1. Introducción	125
5.2. ¿Por qué utilizar Subversion?	126
5.3. Descripción general de Subversion	126
5.4. Instalación	128
5.4.1. Servidor basado en svnserve	128
5.4.1.1. Autenticación con svnserve	129
5.4.2. Servidor basado en Apache	130
5.5. Creación de repositorios	132
5.5.1. La estructura del repositorio	133
5.5.2. Acceso al repositorio	133
5.5.3. Mantenimiento del repositorio	134
5.6. Autenticación	134
5.7. Autorización	137
5.7.1. Control de acceso general	137
5.7.2. Control de acceso basado en directorios	137
5.7.2.1. Definición del fichero de control de acceso	139
5.8. Puesta en marcha	141
5.9. Trabajando con Subversion: TortoiseSVN	141
5.9.1. Instalación	142
5.9.2. Conexión con el repositorio	142
5.9.3. Ciclo de trabajo	142

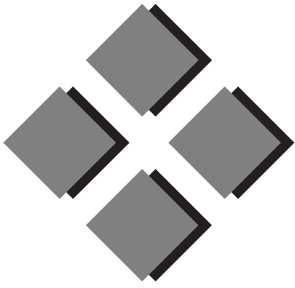
5.9.4.	Operaciones básicas .....	143
5.9.4.1.	Importar datos al repositorio: Importar... ..	143
5.9.4.2.	Obtener una copia de trabajo: SVN Obtener... ..	145
5.9.4.3.	Enviar los cambios al repositorio: SVN Confirmar... ..	146
5.9.4.4.	Actualizar una copia de trabajo: SVN Actualizar ....	149
5.9.4.5.	Resolver conflictos: Editar conflictos y Resuelto... ..	150
5.9.4.6.	Registro de revisiones: Mostrar registro .....	151
5.9.4.7.	Otras operaciones .....	152
5.10.	Bibliografía .....	154
<b>Capítulo 6.</b>	<b>Generación de informes sobre las pruebas .....</b>	<b>155</b>
6.1.	Introducción .....	155
6.2.	Informes con los resultados de ejecución de las pruebas .....	156
6.2.1.	Utilización de la tarea JUnitReport de Ant para generar informes con los resultados de ejecución de las pruebas .....	156
6.2.2.	Otras librerías de interés: JUnit PDF Report .....	160
6.3.	Informes sobre alcance de las pruebas .....	162
6.3.1.	Utilización de la herramienta Cobertura para generar informes de cobertura .....	163
6.3.1.1.	Indicar a Ant la localización de las nuevas tareas ....	164
6.3.1.2.	Instrumentalización de las clases que van a ser probadas .....	165
6.3.1.3.	Ejecución de las pruebas sobre las clases instrumentadas .....	165
6.3.1.4.	Generación de los informes de cobertura .....	167
6.3.1.5.	Establecimiento y verificación de umbrales de cobertura .....	169
6.3.2.	Interpretación de los informes de cobertura .....	172
6.3.2.1.	Estimación de recursos .....	174
6.3.2.2.	Aseguramiento de la calidad de componentes software ..	174
6.4.	Bibliografía .....	174
<b>Capítulo 7.</b>	<b>Pruebas unitarias en aislamiento mediante Mock Objects: JMock y EasyMock .....</b>	<b>175</b>
7.1.	Introducción .....	175
7.2.	Diferencias entre Mock Objects y Stubs .....	178
7.3.	Filosofía de funcionamiento de los Mock Objects .....	179
7.4.	Procedimiento general de utilización de Mock Objects .....	179
7.5.	Herramientas para la puesta en práctica de la técnica de Mock Objects: EasyMock y JMock .....	181
7.5.1.	EasyMock .....	182
7.5.1.1.	Instalación .....	182
7.5.1.2.	Ejemplo de utilización de EasyMock .....	183

7.5.2.	JMock .....	191
7.5.2.1.	Instalación .....	191
7.5.2.2.	Ejemplo de utilización de JMock .....	192
7.6.	Comparativa entre EasyMock y JMock .....	196
7.7.	Bibliografía .....	196
<b>Capítulo 8.</b>	<b>Mejora de la mantenibilidad mediante JTestCase .....</b>	<b>197</b>
8.1.	Introducción .....	197
8.2.	Conceptos básicos .....	200
8.2.1.	Creación del documento XML con los casos de prueba .....	200
8.2.2.	Acceso desde los métodos de prueba a los casos de prueba definidos en los documentos XML .....	204
8.2.3.	Tratamiento de casos especiales .....	207
8.3.	Definición de parámetros complejos con JICE .....	211
8.4.	JTestCase como herramienta de documentación de los casos de prueba ..	218
8.5.	Bibliografía .....	218
<b>Capítulo 9.</b>	<b>Prueba de aplicaciones que acceden a bases de datos: DBUnit .....</b>	<b>219</b>
9.1.	Introducción .....	219
9.2.	Técnicas de prueba .....	221
9.2.1.	Utilización de Mock Objects .....	221
9.2.2.	Utilización de una base de datos real .....	222
9.2.3.	Procedimiento y recomendaciones .....	223
9.3.	Prueba del código perteneciente a la interfaz de acceso a la base de datos: DBUnit .....	225
9.3.1.	Introducción .....	225
9.3.2.	Creación de una clase de pruebas con DBUnit .....	226
9.3.2.1.	Definición de la clase de prueba .....	226
9.3.2.2.	Definición de los métodos de prueba .....	229
9.3.3.	Definición de los casos de prueba .....	244
9.3.4.	Recomendaciones .....	245
9.4.	Bibliografía .....	246
<b>Capítulo 10.</b>	<b>Pruebas de documentos XML: XMLUnit .....</b>	<b>247</b>
10.1.	Introducción .....	247
10.2.	Configuración de XMLUnit .....	248
10.3.	Entradas para los métodos de XMLUnit .....	250
10.4.	Comparación de documentos XML .....	250
10.4.1.	¿Qué métodos de aserción utilizar para comparar código XML? .....	252
10.5.	Cómo salvar diferencias superficiales .....	253

10.5.1.	Ignorar los espacios en blanco .....	253
10.5.2.	Ignorar los comentarios .....	254
10.5.3.	La interfaz <i>DifferenceListener</i> . . . . .	254
10.6.	Prueba de transformaciones XSL .....	255
10.7.	Validación de documentos XML durante el proceso de pruebas .....	256
10.7.1.	Validación frente a un DTD .....	257
10.7.2.	Validación frente a un esquema XML .....	258
10.8.	Bibliografía .....	258
<b>Capítulo 11.</b>	<b>Prueba de aplicaciones Web .....</b>	<b>259</b>
11.1.	Introducción .....	259
11.2.	Herramientas para la automatización de la prueba .....	260
11.2.1.	HttpUnit .....	262
11.2.2.	HtmlUnit .....	262
11.2.3.	JWebUnit .....	262
11.3.	Prueba de un sitio Web .....	263
11.3.1.	Pruebas de navegación .....	263
11.3.1.1.	Procedimiento general de prueba .....	264
11.3.1.2.	Utilización de JWebUnit y JtestCase para realizar las pruebas de navegación .....	265
11.3.2.	Prueba de enlaces rotos .....	273
11.3.3.	Pruebas de estructura y contenido .....	274
11.3.3.1.	Prueba de páginas Web dinámicas .....	275
11.3.3.2.	Prueba de páginas HTML estáticas .....	281
11.3.3.3.	Prueba de documentos XML generados dinámicamente .....	283
11.4.	Bibliografía .....	289
<b>Capítulo 12.</b>	<b>Pruebas de validación .....</b>	<b>291</b>
12.1.	Introducción .....	291
12.2.	JFunc .....	292
12.2.1.	Procedimiento de utilización de JFunc .....	293
12.2.2.	Ejecución de las clases de prueba mediante JFunc .....	295
12.2.3.	Mensajes detallados .....	296
12.3.	JUnitPerf .....	297
12.3.1.	Instalación y configuración de JUnitPerf .....	297
12.3.2.	Creando pruebas con JUnitPerf .....	298
12.3.2.1.	Pruebas de tiempo de respuesta: TimedTest ....	298
12.3.2.2.	Pruebas de carga: LoadTest .....	300
12.3.2.3.	Pruebas combinadas de tiempo y carga .....	303
12.3.2.4.	Ejecución de las pruebas de rendimiento .....	306
12.4.	JMeter .....	306

12.4.1.	Instalación y configuración de JMeter .....	307
12.4.2.	Elementos de un plan de pruebas .....	309
12.4.2.1.	ThreadGroup .....	309
12.4.2.2.	Controllers .....	310
12.4.2.3.	Listeners .....	312
12.4.2.4.	Timers .....	312
12.4.2.5.	Assertions .....	312
12.4.2.6.	Configuration Elements .....	312
12.4.2.7.	Pre-Processor Elements .....	313
12.4.2.8.	Post-Processor Elements .....	313
12.4.2.9.	Reglas de alcance .....	313
12.4.2.10.	Orden de ejecución .....	314
12.4.2.11.	WorkBench .....	316
12.4.3.	Creando pruebas con JMeter .....	316
12.4.3.1.	Una prueba simple .....	316
12.4.3.2.	Uso de parámetros en las peticiones .....	318
12.4.3.3.	Una prueba con varias peticiones .....	320
12.5.	Bibliografía .....	322
<b>Apéndice A. Variables de entorno .....</b>		<b>323</b>
A.1	Linux .....	323
A.2	Windows .....	324
<b>Apéndice B. Sistema a probar .....</b>		<b>327</b>
B.1	Descripción general del sistema .....	328
B.2	Arquitectura del sistema .....	329
B.3	Configuración del sistema .....	330
B.4	Características del sistema y su relevancia en las pruebas de software ...	331
B.5	Arquitectura interna del sistema .....	332
B.6	Documento de especificación de requisitos software .....	334
<b>Apéndice C. Estándares de nomenclatura y normas de estilo en Java .....</b>		<b>337</b>
C.1	Conceptos de clase y objeto .....	337
C.2	Normas de estilo .....	338
<b>Apéndice D. Novedades en Java 5.0 .....</b>		<b>341</b>
D.1	Introducción .....	341
D.2	Anotaciones en Java .....	341
D.3	<i>Import</i> estático .....	342





# Prefacio

---

A lo largo de los últimos años, debido al rápido desarrollo de la tecnología, la complejidad de los sistemas software desarrollados se ha incrementado de una forma muy notable. Por este motivo, la realización de un correcto proceso de pruebas, que permita culminar el proceso de desarrollo de un producto satisfaciendo enteramente las necesidades del cliente, cada día tiene mayor importancia.

Dentro de la fase de desarrollo de cualquier sistema software siempre se producen errores que, independientemente de la fase (análisis, diseño o codificación) en que se introduzcan, en última instancia siempre quedan plasmados sobre el código de la aplicación. Por tanto, es preciso revisar ese código con el objetivo de detectar y eliminar dichos errores. Todos aquellos errores que no se identifiquen antes de la entrega del producto aparecerán durante su uso, con el consiguiente perjuicio para el usuario y para el desarrollador. Por un lado el usuario se sentirá insatisfecho y frustrado, mientras que por el otro lado, el equipo de desarrollo tendrá que volver atrás y buscar donde se ha producido el fallo para posteriormente corregirlo.

Tradicionalmente, la fase de pruebas del código de un proyecto software tenía lugar después de la fase de codificación y antes de la distribución del producto al usuario. El fin era encontrar los errores acumulados en las fases anteriores de análisis y diseño y que han quedado plasmados en el código fuente, además de los errores propios introducidos durante la codificación. Sin embargo, se ha demostrado en numerosos casos prácticos, que realizar un proceso de pruebas en paralelo a toda la fase de desarrollo en lugar de únicamente al final, permite detectar los errores de una forma más temprana y solucionarlos a menor coste. A este respecto cabe destacar la aparición de una metodología de desarrollo software llamada *Test Driven Development* (desarrollo guiado por las pruebas) en la que las pruebas pasan a ser el centro del proceso de desarrollo, en el que el resto de tareas giran a su alrededor. Esta visión puede en un principio parecer un tanto radical, sin embargo, si el objetivo de cualquier proyecto software es producir un software que satisfaga las expectativas del cliente y esto solo se puede alcanzar mediante un adecuado proceso de pruebas ¿Por qué no centrar la fase de desarrollo en este proceso? Con este enfoque, las pruebas se definen a la vez que se va definiendo el producto y se automatizan de modo que pueden repetirse tantas veces como sea necesario: durante la puesta a punto del producto o durante las sucesivas acciones de mantenimiento que seguramente sufrirá la aplicación.

Un producto software se puede considerar un producto de calidad si cumple con todos los requisitos de usuario, software y requisitos implícitos. Tales como que el sistema se comporte de

la forma esperada, que la interacción con otros sistemas o elementos externos sea la adecuada, que sea robusto, que cumpla ciertos parámetros de rendimiento, etc. Justamente, verificar que el producto desarrollado cumple todos estos requisitos, es la finalidad de las pruebas. Sin ninguna duda, la fase de pruebas incide de forma directa en la calidad, seguridad, eficiencia, corrección y completitud del producto software desarrollado.

Actualmente un gran porcentaje del esfuerzo, y por tanto del coste, que una empresa realiza durante el ciclo de vida de un proyecto software recae sobre la fase de mantenimiento. Durante esta fase, el software evoluciona incorporando nuevas funcionalidades para dar respuesta a nuevos requisitos del usuario, y corrigiendo errores que van apareciendo durante su uso. Por este motivo, realizar una correcta fase de pruebas que permita producir un sistema fácilmente mantenible y, por tanto, permita minimizar este esfuerzo, es de una gran importancia. Cuando se modifica un producto durante su mantenimiento, hay que someterlo a las pruebas de regresión para garantizar que los cambios introducidos no traigan consigo ningún defecto. Si estas pruebas ya están definidas y automatizadas, los recursos requeridos para las pruebas durante el mantenimiento se reducen considerablemente.

Una buena estrategia de pruebas incluye pruebas unitarias, de integración, de sistema, funcionales o de validación y de aceptación. A lo largo de este texto se verán todas ellas. Inicialmente, se ofrece una visión general de la fase de pruebas definiendo y explicando los conceptos más relevantes de dicha fase, sus actividades, el diseño de casos de pruebas, principios, estrategia, técnicas, roles, productos generados, etc. Asimismo se describe el proceso de escritura del Plan de Pruebas, documento indispensable al inicio de toda fase de pruebas. Una vez realizada esta introducción, el lector se encuentra en disposición de afrontar el resto de los capítulos con una base sólida.

Estos capítulos se centran en describir todo tipo de técnicas de prueba sobre software Java mediante la utilización de herramientas pertenecientes a la familia JUnit. Estas técnicas comprenden: realizar pruebas unitarias, pruebas en aislamiento, pruebas de aplicaciones que acceden a bases de datos, pruebas de aplicaciones WEB, pruebas de documentos estáticos, pruebas de rendimiento, etc. A la hora de describir estas técnicas se ha seleccionado un conjunto de herramientas, todas ellas disponibles bajo licencias de software libre, que permiten ponerlas en práctica de una forma efectiva y cómoda para el desarrollador. Sin embargo, a pesar de que el procedimiento de utilización de estas herramientas ocupa una buena parte de este libro, se ha tratado en la medida de lo posible de aislar las técnicas de prueba en sí mismas, de las herramientas que las llevan a la práctica. En el mundo del desarrollo software las herramientas de prueba cambian constantemente, sin embargo, la técnica de prueba en la que se basan raras veces lo hace. Por este motivo, el objetivo de este libro ha sido realizar una colección atemporal de técnicas y procedimientos de prueba que conserve su valor y utilidad independientemente de las herramientas utilizadas.

A lo largo de este libro se han descrito en profundidad más de 10 herramientas de prueba pertenecientes a la familia JUnit. Todas ellas permiten automatizar diferentes aspectos de la prueba de código Java: JUnit para las pruebas unitarias, JUnitReport para generar informes sobre los resultados de las pruebas, facilitando así su análisis y almacenamiento, JMock y EasyMock para realizar pruebas de aislamiento de clases, JTestCase para separar el código asociado a los casos de prueba de los datos usados para ejecutarlos, DBUnit para probar código que accede a bases de datos, HTTPUnit, HTMLUnit y JWEBUnit para probar aplicaciones Web, XMLUnit para comprobar la correcta generación de código XML y JFunc, JUnitPerf y JMeter para las pruebas de validación.

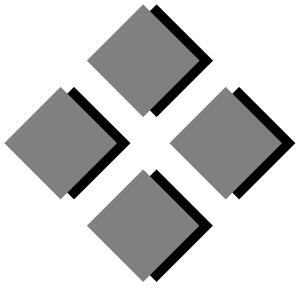
Este texto también incluye otras dos herramientas que proporcionan un importante apoyo durante la fase de pruebas: Ant y Subversión (SVN). La primera de ellas es una herramienta bien conocida en el mundo de desarrollo Java y que presenta una serie de características que la hacen especialmente adecuada para la compilación y creación de código Java en cualquier plataforma de desarrollo. Sin embargo este libro no pretende ser un manual de Ant sino introducir los mecanismos que esta tecnología proporciona para la realización de pruebas con JUnit. Por otro lado SVN, es una herramienta de control de versiones muy poderosa y actualmente utilizada en infinidad de proyectos software. Junto con su *front-end* gráfico conocido como TortoiseSVN, permite almacenar y recuperar versiones de código fuente (en este caso código de pruebas y código de producción) así como de otros elementos software como archivos de datos, configuración, etc.

Asimismo, cabe destacar que todos los capítulos de este libro se apoyan en un único caso práctico: una aplicación Java (entregada en forma de CD junto con este libro y descrita en el Apéndice B del mismo) que proporciona información sobre el estado del tráfico de carreteras. Esta aplicación ha sido desarrollada exclusivamente con el objetivo de proporcionar un marco común sobre el que demostrar la utilización de las diferentes técnicas de prueba tratadas a lo largo de este libro. De este modo la lectura del texto es didáctica y comprensible, asistiendo en todo momento al lector y al desarrollador mediante la explicación por medio de ejemplos analizados hasta el más mínimo detalle. Se ha cuidado la elección de los ejemplos para que resulten amenos e ilustrativos, a la vez que se ha tratado de no pasar por alto aquellos aspectos en la aplicación de determinadas técnicas que resultan más arduos.

Este libro viene a cubrir la escasez de literatura especializada en pruebas de software, realizando un análisis comprometido y en profundidad sobre la materia con el objetivo de proporcionar una visión global de este singular proceso, tan complejo como necesario.

Finalmente es importante destacar que este texto no ha sido escrito, en absoluto, en exclusiva para la enseñanza universitaria. La intención es que sirva de apoyo para todos aquellos desarrolladores del mundo Java que desean incorporar a su filosofía de trabajo una completa metodología de pruebas de software con la que probar su código de forma más efectiva y alcanzar así sus propósitos y objetivos, garantizando siempre la máxima calidad.





# Capítulo 1

# Fundamentos de las pruebas de software

---

## SUMARIO

<b>1.1.</b> Introducción	<b>1.9.</b> Depuración de errores
<b>1.2.</b> Principios básicos	<b>1.10.</b> Otras pruebas
<b>1.3.</b> Tareas básicas	<b>1.11.</b> Criterios para dar por finalizadas las pruebas
<b>1.4.</b> Inspecciones de código	<b>1.12.</b> Equipo de pruebas
<b>1.5.</b> Pruebas basadas en la ejecución del código: técnicas	<b>1.13.</b> Errores más comunes que se cometen en la fase de pruebas
<b>1.6.</b> Diseño de casos de prueba	<b>1.14.</b> Documentación de pruebas
<b>1.7.</b> Estrategia de pruebas	<b>1.15.</b> Bibliografía
<b>1.8.</b> Pruebas de sistemas orientados a objetos	

## 1.1. Introducción

Las pruebas de software se definen como el proceso que ayuda a identificar la corrección, completitud, seguridad y calidad del software desarrollado. De forma genérica, por software se entiende el conjunto de datos, documentación, programas, procedimientos y reglas que componen un sistema informático. Este libro, y consecuentemente este capítulo, se centra en las pruebas de programas, es decir, de código.

Este capítulo cubre, entre otras cosas, los conceptos, principios y tareas básicos de la fase de pruebas de un ciclo de vida de un proyecto informático, el diseño de casos de prueba utilizando técnicas que lleven a obtener un conjunto de casos con alta probabilidad de encontrar

errores en el código, fin principal de las pruebas, y la estrategia de pruebas más adecuada y comúnmente aceptada para probar de forma completa y consistente un sistema software partiendo tanto de un diseño estructurado como de un diseño orientado a objetos. Finalmente, se explican algunos conceptos adicionales, tales como la depuración de errores, los criterios para dar por finalizada la etapa de pruebas y los errores que se cometen más habitualmente en esta fase. Por último se menciona la documentación que se genera durante el proceso de pruebas software.

Se pretende, por tanto, proporcionar al lector una visión genérica de la fase de pruebas software. Con este capítulo queda enmarcado el resto del libro, que se centra principalmente en la verificación del código y más particularmente utilizando JUnit.

## 1.2. Principios básicos

Como definición básica del contenido de este libro, hay que mencionar, en primer lugar, que prueba es el proceso de ejecutar un conjunto de elementos software con el fin de encontrar errores. Por tanto, probar no es demostrar que no hay errores en el programa ni únicamente mostrar que el programa funciona correctamente, ambas son definiciones incorrectas y, sin embargo, comúnmente utilizadas.

Otra definición esencial es la de caso de prueba, término que se utilizará ampliamente a lo largo de todo este texto. Se define caso de prueba software como un conjunto de condiciones, datos o variables bajo las cuales el desarrollador determinará si el o los correspondientes requisitos de un sistema software se cumplen de forma parcial, de forma completa o no se cumplen.

Otros conceptos fundamentales son los de error, fallo y defecto software. Se define formalmente error como la discrepancia entre un valor o condición calculado, observado o medido y el valor o condición específica o teóricamente correcta. Es un fallo que comete el desarrollador. Defecto técnico es la desviación en el valor esperado para una cierta característica. Defecto de calidad. Desviación de estándares y, finalmente, fallo es la consecuencia de un defecto. En muchas ocasiones estos términos se confunden y se utiliza uno de ellos de forma genérica.

Así, un error y un defecto software (y como consecuencia un fallo) existen cuando el software no hace lo que el usuario espera que haga, es decir, aquello que se ha acordado previamente en la especificación de requisitos. En una gran parte de los casos, esto se produce por un error de comunicación con el usuario durante la fase de análisis de requisitos o por un error de codificación.

El objetivo de las pruebas es, por tanto, descubrir los errores y fallos cometidos durante las fases anteriores de desarrollo del producto.

Se listan seguidamente un conjunto de principios básicos para llevar a cabo el proceso de pruebas de forma efectiva y con éxito:

- Las pruebas de código están muy ligadas al diseño realizado del sistema. De hecho, para generar los casos de prueba hay que basarse en el diseño observando tanto el diseño de los componentes u objetos como de la integración de éstos que componen el sistema com-

pleto. Por ello, es conveniente definir la mayor parte de los casos de prueba y, al menos, esbozar el plan de pruebas en la fase de diseño.

- Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error o fallo no descubierto hasta entonces. Por tanto, la prueba tiene éxito si descubre un error o un fallo no detectado hasta entonces. Así que es fundamental seleccionar de forma adecuada el conjunto de casos de pruebas que se va a utilizar y seguir las técnicas correctas para elaborar un juego de ensayo que permita obtener el objetivo mencionado.
- La definición del resultado esperado del programa es una parte integrante y necesaria de un caso de prueba. Es imprescindible conocer el resultado esperado de la prueba que se está llevando a cabo para poder discernir si el resultado obtenido es el correcto o no. En este sentido, hay que inspeccionar cuidadosamente el resultado de cada prueba. Típicamente y como se verá a lo largo de este libro, la forma de comprobar resultados esperados es mediante la definición de condiciones a verificar sobre los resultados obtenidos.
- Un programador debe evitar probar su propio programa. De forma natural, un programador tiende a no encontrar los errores en su propio código. Normalmente, el ser humano percibe una gran parte de su valor como persona asociado a la calidad de su trabajo. Es por ello que inconscientemente tiende a pasar por alto los errores de su código. Es mucho más efectivo que el grupo de pruebas esté compuesto, por una parte, por programadores no involucrados en la fase de codificación del proyecto y, por otra, por un conjunto de potenciales usuarios con el rol más similar posible al perfil del usuario real.
- Los casos de prueba deben ser escritos tanto para condiciones de entrada válidas y esperadas como para condiciones inválidas e inesperadas. Más aún, examinar un programa para comprobar si hace o no lo que se supone que debe hacer es sólo la mitad del trabajo. También hay que comprobar que no haga aquello que se supone que no debe hacer y que los errores y fallos estén controlados.
- Hay que tener en cuenta que la probabilidad de encontrar errores adicionales en una función del programa o método de una clase es proporcional al número de errores ya encontrados en esa misma función o método. Esto deriva en que cuanto más se modifiquen los elementos presentes en el código fuente de una función o programa, más hay que probarlo.
- Es imprescindible documentar todos los casos de prueba. Esto permite, por una parte, conocer toda la información sobre cada caso de prueba y, por otra, volver a ejecutar en el futuro aquellos casos que sean necesarios (pruebas de regresión).

## 1.2.1. Verificación y validación

Otros conceptos importantes para el entendimiento del texto de este libro son los de validación y verificación, que se definen a continuación en esta sección.

La verificación comprueba el funcionamiento del software, es decir, asegura que se implemente correctamente una funcionalidad específica. En definitiva, responde a la pregunta ¿se ha construido el sistema correctamente?

Por su parte, la validación comprueba si los requisitos de usuario se cumplen y los resultados obtenidos son los previstos. Responde a la pregunta ¿se ha construido el sistema correcto?

Ambos conceptos se usarán ampliamente a lo largo de los siguientes capítulos.

## 1.3. Tareas básicas

Las tareas principales que hay que realizar en la etapa de pruebas son:

1. **Diseño del plan de pruebas.** Esta tarea, como ya se ha mencionado, debe realizarse en la fase de diseño dentro del ciclo de vida de un proyecto software. Pero, en muchas ocasiones, esto no ocurre y se lleva a cabo en la fase de pruebas. En este caso, debe hacerse al principio de la etapa de pruebas. El plan de pruebas consta de la planificación temporal de las distintas pruebas (cuándo, cómo y quién va a llevarlas a cabo), definición de la estrategia de pruebas (ascendente, descendente, sándwich, etc.), procedimiento a seguir cuando una prueba no obtiene el resultado esperado, asignación de responsabilidades, etc. Al final de este capítulo se ofrece un esquema a modo de ejemplo de un plan de pruebas estándar.
2. **Diseño de casos de prueba.** En esta fase se definirán los casos de prueba de tal forma que con un número de casos cuidadosamente seleccionados se realice un número de pruebas que alcancen el nivel de cobertura<sup>1</sup> deseado. El diseño de casos de prueba se detallará más adelante en este capítulo.
3. **Prueba.** Se lleva a cabo la escritura del código de pruebas encargado de la ejecución de los casos de prueba anteriormente diseñados. Posteriormente, se realiza la ejecución de la prueba propiamente dicha<sup>2</sup>.
4. **Comparación y evaluación de resultados.** Teniendo en cuenta los resultados esperados, se comparan éstos con los obtenidos. Si son iguales, la prueba se considera válida, si no es así se aplican los procedimientos definidos en el plan de pruebas.
5. **Localización del error.** En el caso en el que la ejecución de ciertos casos de prueba produzca resultados no esperados, es necesario encontrar el segmento de código fuente en el que se encuentra el error. Actualmente existen herramientas capaces de proporcionar al desarrollador la localización exacta en el código fuente de los errores detectados. Como se verá en el Capítulo 2, la herramienta JUnit es un gran aliado al respecto. Sin embargo, en ocasiones hay errores que son detectados en zonas del código diferentes a las zonas en que se producen, en estos casos las herramientas de depuración son de gran ayuda. El proceso de depuración se lleva a cabo mediante herramientas automáticas y consiste en ejecutar el programa y detenerlo en un punto determinado cuando se cumplan ciertas condiciones o a petición y conveniencia del programador. El objetivo de esta interrupción es que el programador examine aspectos tales como las variables o las llamadas que están relacionadas con el error o que ejecute partes internas de una función o incluso las sentencias de forma independiente. Se dedica una breve sección de este capítulo a la depuración del código. Para localizar el error también se puede acudir a las inspecciones de código que se describen a continuación en el Apartado 1.4 de este capítulo.

---

<sup>1</sup> En el Capítulo 6, Generación de informes sobre las pruebas, se presenta una herramienta capaz de generar automáticamente informes de cobertura. Estos informes son un indicador de la calidad de los casos de prueba diseñados y puede utilizarse para la monitorización y reasignación de recursos durante la fase de pruebas.

<sup>2</sup> Como se verá a lo largo del Capítulo 2, JUnit es una herramienta ideal para asistir en la creación y ejecución del código de pruebas. En general, esta herramienta permite automatizar todos las tareas posteriores al diseño de los casos de prueba.



En las siguientes secciones se define la actividad de inspección de código como técnica de pruebas no basada en el ordenador y la técnica basada en ordenador: caja blanca y caja negra.

## 1.4. Inspecciones de código

Las inspecciones de código, en caso de llevarse a cabo, suelen realizarse por un equipo que incluye personas ajenas al proyecto, participantes en otras fases del proyecto y personal del grupo de calidad. Consisten en la revisión de los listados de programas y el resultado que producen es un registro de los errores encontrados. En algunos casos se ha demostrado que con las inspecciones de código se pueden encontrar más del 80% de los errores cometidos a pesar de no ser muy común la aplicación de esta técnica. El motivo es que se trata de una técnica no automatizable ya que debe ser realizada por una persona. Esto es un gran inconveniente ya que multiplica el coste de las pruebas de regresión, especialmente durante la fase de mantenimiento del producto.

## 1.5. Pruebas basadas en la ejecución del código: técnicas

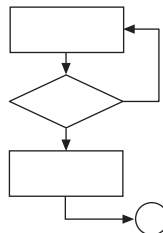
En esta categoría se incluyen las denominadas pruebas de caja blanca y pruebas de caja negra que se describen a continuación.

### 1.5.1. Pruebas de caja blanca

Estas pruebas se centran en probar el comportamiento interno y la estructura del programa examinando la lógica interna, como muestra la Figura 1.1. Para ello:

- Se ejecutan todas las sentencias (al menos una vez).
- Se recorren todos los caminos independientes de cada módulo.
- Se comprueban todas las decisiones lógicas.
- Se comprueban todos los bucles.

Finalmente, en todos los casos se intenta provocar situaciones extremas o límites.



**Figura 1.1.** Lógica interna de un programa.

Nótese que al utilizar técnicas de caja blanca se está llevando a cabo una verificación del código. Por lo tanto, éste tiene que estar disponible para la realización de este tipo de pruebas.

En los siguientes apartados se describen las técnicas de caja blanca más habituales.

#### 1.5.1.1. Pruebas de interfaces entre módulos o clases

Este tipo de prueba analiza el flujo de datos que pasa a través de la interfaz (tanto interna como externa) del módulo (en un lenguaje de programación estructurada) o la clase (en un lenguaje orientado a objetos) objetivo de la prueba. Se distingue entre interfaces internas y externas.

En las pruebas de interfaces internas entre funciones o métodos es necesario comprobar que los argumentos de llamadas a funciones y la consistencia de las definiciones de variables globales entre los módulos. Las pruebas de interfaces internas corresponden al conjunto de pruebas unitarias, que están enfocadas a verificar el correcto funcionamiento de un módulo o clase aisladamente del resto.

Para probar adecuadamente las interfaces externas se ha de verificar que el flujo de datos intercambiado entre clases o módulos es el correcto. Este tipo de pruebas es una parte de las pruebas de integración.

#### 1.5.1.2. Prueba de estructuras de datos locales

Estas pruebas aseguran la integridad de los datos durante todos los pasos de la ejecución del módulo. Se comprueban las referencias de datos, la posible utilización de variables no inicializadas, no salirse del límite de matrices o vectores, la correcta declaración de datos y el hecho de no realizar comparaciones entre variables de distinto tipo, como aspectos más importantes. Además se localizan errores derivados del uso de variables, tales como *overflow*, *underflow*, división por cero, etc.

#### 1.5.1.3. Prueba del camino básico

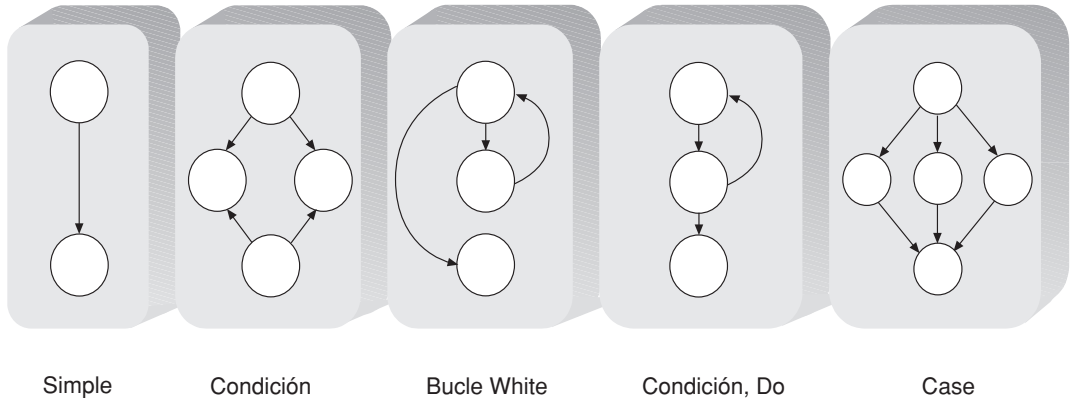
Se definen para este tipo de pruebas un conjunto básico de caminos de ejecución usando una medida calculada previamente de la complejidad del módulo llamada *complejidad ciclomática*, propuesta por McCabe, que se basa en el grafo de flujo. La Figura 1.2 representa los grafos de los distintos caminos básicos.

La complejidad ciclomática indica el número de caminos básicos a probar y responde a la siguiente fórmula:

$$V(G) = \text{Aristas} - \text{Nodos} + 2$$

Los pasos a seguir para realizar las pruebas de camino básico son:

1. Dibujar el grafo de flujo.
2. Determinar la complejidad ciclomática del grafo.
3. Determinar los caminos linealmente independientes.
4. Preparar los casos de prueba que forzarán la ejecución de cada camino.



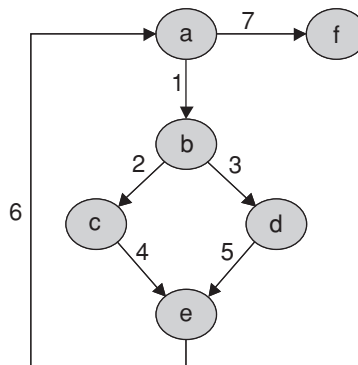
**Figura 1.2.** Caminos básicos.

Existen correlaciones entre la complejidad de un módulo y el número de errores en el módulo, el tiempo para encontrar errores, el esfuerzo de prueba y el esfuerzo de mantenimiento. Así, un módulo con complejidad ciclomática mayor de 10 debe ser examinado para posibles simplificaciones o contemplar la posibilidad de partirlo en varias subrutinas.

Se muestra, a continuación, un ejemplo de este método de pruebas.

```
public int MCD(int x, int y) {
    while (x != y)    // a
    {
        if (x > y)    // b
            x = x - y; // c
        else
            y = y - x; // d
    }                // e
    return x;        // f
}
```

Seguidamente, se realiza el grafo correspondiente,



Y se halla la complejidad ciclomática:  $V(G) = \text{Aristas} - \text{Nodos} + 2$ . En este caso,  $V(\text{MCD}) = 7 - 6 + 2 = 3$ . La complejidad ciclomática en este caso es 3, lo que significa que hay tres caminos linealmente independientes.

La Tabla 1.1 muestra los tres posibles caminos hallados con los valores correspondientes de “x” y de “y”, así como el valor de retorno para cada uno de ellos.

Tabla 1.1. Caminos hallados.

	1	2	3	4	5	6	7	Caso de Prueba
af	0	0	0	0	0	0	1	x=1, y=1, ret=1
abdeaf	1	0	1	0	1	1	1	x=1, y=2, ret=1
abceaf	1	1	0	1	0	1	1	x=2, y=1, ret=1

Para seguir completando las pruebas con técnicas de caja blanca, es necesario ejecutar las pruebas de bucles que se describen a continuación.

1.5.1.4. Pruebas de condiciones límite

En primer lugar es necesario representar de forma gráfica los bucles para, posteriormente, validar la construcción de los bucles, que pueden ser simples, anidados, concatenados y no estructurados. Nótese que estas clasificaciones no son excluyentes entre sí.

A continuación se describe la forma óptima de tratar cada tipo de bucle para ejecutar las pruebas de condiciones.

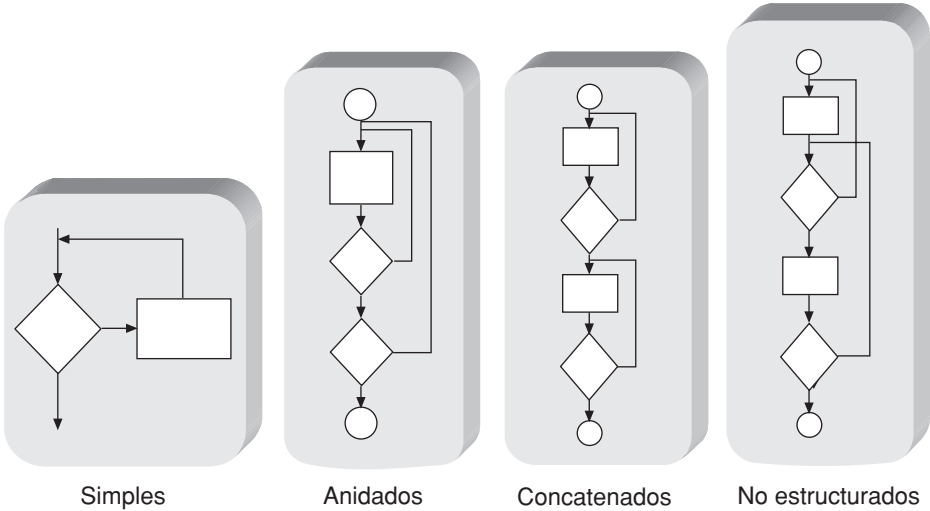


Figura 1.3. Tipos de bucles.

Para los bucles simples, siendo  $n$  el número máximo de pasos, hay que realizar las siguientes iteraciones, con sus correspondientes pruebas para garantizar que cada tipo de bucle queda adecuadamente probado:

- Pasar por alto el bucle.
- Pasar una sola vez.
- Pasar dos veces.
- Hacer  $m$  pasos con  $m < n$ .
- Hacer  $n-1$ ,  $n$  y  $n+1$  pasos.

En el caso de los bucles anidados, se comienza por el bucle más interno progresando hacia fuera.

Los bucles concatenados pueden ser independientes. En este caso, se realizan las mismas pruebas que si fueran de bucles simples. En el caso de que no sean independientes, se aplica el enfoque de los bucles anidados.

Los bucles no estructurados necesitan ser rediseñados ya que comprometen la calidad del diseño y, una vez hecho esto, se tratan como el tipo de bucle que haya resultado. A día de hoy, aunque instrucciones como `goto` todavía forman parte de muchos lenguajes de programación, la programación no estructurada en lenguajes de alto nivel está completamente desaconsejada. Un código no estructurado es difícil de leer y, por tanto, difícil de mantener.

#### 1.5.1.5. Pruebas de condición

Las condiciones en una sentencia pueden ser, esencialmente, simples o compuestas. Una condición simple puede ser una variable lógica (TRUE o FALSE) o una expresión relacional de la siguiente forma:

$E1$  (operador relacional)  $E2$ ,  
donde  $E1$  y  $E2$  son expresiones aritméticas y el operador relacional es del tipo  
 $<$ ,  $<=$ ,  $>$ ,  $>=$ ,  $=$ ,  $?#$ .

Las condiciones compuestas están formadas por varias condiciones simples, operadores lógicos del tipo NOT, AND, OR y paréntesis.

Los errores más comunes que se producen en una condición y que, por tanto, hay que comprobar son:

- Error en el operador lógico: que sean incorrectos, desaparecidos, sobrantes, etc.
- Error en la variable lógica.
- Error en la expresión aritmética.
- Error en el operador relacional.
- Error en los paréntesis.

En las decisiones, es necesario hacer pruebas de ramificaciones, que consisten en probar la rama verdadera y la rama falsa y cada condición simple.

En general, existen los siguientes tipos de pruebas relacionadas con las condiciones y decisiones:

- De cobertura de decisión.
- De cobertura de condición.
- De cobertura de decisión/condición.

El siguiente ejemplo define los casos de prueba aplicando las coberturas de decisión y de decisión/condición a partir de este fragmento de código:

```
public void comprobarHora(int h, int m, int s) {
    if ((h >= 0) && (h <= 23)) {
        if ((m >= 0) && (m <= 59)) {
            if ((s >= 0) && (s <= 59)) {
                System.out.println("La hora marcada es correcta");
            }
        }
    }
    System.out.println("La hora marcada es incorrecta");
}
```

Se generan a continuación los casos de prueba necesarios para obtener una cobertura completa de decisiones.

En el código hay tres decisiones:

D1  $\rightarrow (h \geq 0) \text{ and } (h \leq 23)$

D2  $\rightarrow (m \geq 0) \text{ and } (m \leq 59)$

D3  $\rightarrow (s \geq 0) \text{ and } (s \leq 59)$

Cada decisión debe tomar al menos una vez el valor verdadero y otra el valor falso. Los datos concretos para los casos de prueba podrían ser los siguientes:

	Valor Verdadero	Valor Falso
D1	h=10	h=24
D2	m=30	m=60
D3	s=59	s=70

Para cubrir todas las decisiones se deben definir los siguientes casos:

Caso de prueba 1: D1 = Verdadero; D2 = Verdadero; D3 = Verdadero  
h=10; m=30; s=50

- Caso de prueba 2: D1 = Verdadero; D2 = Verdadero; D3 = Falso  
h=10; m=30; s=70
- Caso de prueba 3: D1 = Verdadero; D2 = Falso  
h=10; m=60
- Caso de prueba 4: D1 = Falso  
h=24

Se generan seguidamente los casos de prueba necesarios para obtener una cobertura completa de decisión/condición.

En el código hay tres decisiones. Cada una de ellas comprende dos condiciones.

- D1  $\rightarrow$  (h>=0) and (h<=23)  
C1.1  $\rightarrow$  h>=0  
C1.2  $\rightarrow$  h<=23
- D2  $\rightarrow$  (m>=0) and (m<=59)  
C2.1  $\rightarrow$  m>=0  
C2.2  $\rightarrow$  m<=59
- D3  $\rightarrow$  (s>=0) and (s<=59)  
C3.1  $\rightarrow$  s>=0  
C3.2  $\rightarrow$  s<=59

Hay que garantizar que cada condición tome al menos una vez el valor verdadero y otra el valor falso, garantizando además que se cumpla la cobertura de decisión. Los datos concretos para los casos de prueba podrían ser los siguientes:

	Valor Verdadero	Valor Falso
C1.1	h=10	h=-1
C1.2	h=10	m=24
C2.1	m=30	m=-1
C2.2	m=30	m=60
C3.1	s=50	s=-1
C3.2	s=50	s=70

Si se utilizan los datos de C11 y C12 que hacen que tomen los valores VERDADERO simultáneamente, la decisión D1 tomará también el valor VERDADERO. Para que tome el valor FALSO, no se puede hacer que C11 y C12 tomen los valores FALSO simultáneamente, y habrá que considerar dos casos: C11=V, C12=F y C11=F, C12=V para que la decisión D1 tome también el valor FALSO cubriendo todas las condiciones.

Lo mismo ocurre con C2.1 y C2.2 y con C3.1 y C3.2.

- Caso de prueba 1: C1.1= Verdadero, C1.2= Verdadero, C2.1= Verdadero, C2.2= Verdadero, C3.1= Verdadero, C3.2= Verdadero  
h=10; m=30; s=50

- Caso de prueba 2: C1.1= Verdadero, C1.2= Verdadero, C2.1= Verdadero, C2.2= Verdadero, C3.1= Verdadero, C3.2= Falso  
h=10; m=30; s=70
- Caso de prueba 3: C1.1= Verdadero, C1.2= Verdadero, C2.1= Verdadero, C2.2= Verdadero, C3.1= Falso, C3.2= Verdadero  
h=10; m=30; s=-1
- Caso de prueba 4: C1.1= Verdadero, C1.2= Verdadero, C2.1= Verdadero, C2.2= Falso  
h=10; m=30
- Caso de prueba 5: C1.1= Verdadero, C1.2= Verdadero, C2.1= Falso, C2.2= Verdadero  
h=10; m=-1
- Caso de prueba 6: C1.1= Verdadero, C1.2= Falso  
h=10
- Caso de prueba 7: C1.1= Falso, C1.2= Verdadero  
h=-1

### 1.5.2. Pruebas de caja negra

Las pruebas de caja negra están conducidas por los datos de entrada y salida. Así, los datos de entrada deben generar una salida en concordancia con las especificaciones considerando el software como una caja negra sin tener en cuenta los detalles procedimentales de los programas.

Las pruebas consisten en buscar situaciones donde el programa no se ajusta a su especificación. Para ello, es necesario conocer de antemano la especificación o salida correcta y generar como casos de prueba un conjunto de datos de entrada que deben generar una salida en concordancia con la especificación. Cada vez que se genera una salida es necesario cuestionarse si la salida resultante es igual a la salida prevista. Si es así, se puede continuar con la siguiente prueba. Si no, se ha hallado un error que habrá que investigar y corregir antes de continuar con el proceso de pruebas.

Las dos técnicas más comunes de las pruebas de caja negra son la técnica de partición de equivalencia y la de análisis de valores límites, que se explican en las siguientes secciones.



**Figura 1.4.** Esquema de pruebas de caja negra.



### 1.5.2.1. Partición de equivalencia

Dado que es imposible realizar pruebas que contemplen absolutamente todos los casos y caminos posibles, el propósito de esta técnica y la siguiente es cubrir las pruebas de la forma más amplia posible diseñando y utilizando un número de casos de prueba manejable. Para ello, en primer lugar y como objetivo de la partición de equivalencia, se divide la entrada de un programa en clases de datos de los que se puedan derivar casos de prueba para reducirlos.

Así, se toma cada valor de entrada y se divide en varios grupos utilizando las siguientes recomendaciones:

- Si la entrada es un **rango**, se define una clase de equivalencia válida y dos inválidas.
- Si la entrada es un **valor específico**, se define una clase de equivalencia válida y dos inválidas.
- Si la entrada es un **valor lógico**, se define una clase de equivalencia válida y otra inválida.

### 1.5.2.2. Análisis de valores límite

Después de haber definido y probado los distintos casos siguiendo la técnica de partición de equivalencia, se prueban los valores límite de cada clase de equivalencia, es decir, los valores fronterizos de cada clase por ambos lados.

En el ejemplo mostrado en la siguiente sección se utiliza esta técnica. En dicha sección se muestra la estrategia a seguir para definir de forma más óptima el conjunto de casos de prueba que se va a utilizar de tal forma que se hagan las pruebas de la forma más amplia posible dentro del límite de recursos que se tiene.

## 1.6. Diseño de casos de prueba

Dado que, como ya se ha mencionado, es inviable probar todos los posibles casos de prueba que cabría definir para un sistema software dado, el objetivo para realizar un buen diseño de casos es crear un subconjunto de los casos de prueba que tiene la mayor probabilidad de detectar el mayor número posible de errores y fallos.

Esto se realiza aplicando, en primer lugar, las técnicas de caja negra y después completar creando casos suplementarios que examinen la lógica del programa, es decir, de caja blanca.

El propósito de una buena estrategia de pruebas es crear un conjunto de casos que tengan una alta probabilidad de encontrar errores en el software, pero que supongan un esfuerzo manejable para el ingeniero de software. El objetivo es alcanzar un grado de cobertura suficiente sobre el código a probar minimizando los recursos empleados para ello.

En particular, la estrategia habitual para conseguir este propósito sigue los pasos enumerados a continuación:

- Aplicar análisis del valor límite.
- Diseñar casos con la técnica de partición de equivalencia.

- Utilizar la técnica de conjetura de error:
  - Combinar las entradas.
  - Incluir casos típicos de error (por ejemplo, 0 en la entrada o salida, listas vacías, listas de un elemento, etc.), también llamado heurística de malicia, de la que se hablará un poco más adelante.
- Usar técnicas de caja blanca (para pruebas unitarias, como se verá más adelante o para funciones especialmente críticas):
  - Ejecutar cada sentencia al menos una vez.
  - Invocar cada subrutina al menos una vez.
  - Aplicar cobertura de decisión. Asignar a cada decisión al menos una vez un resultado verdadero y otro falso.
  - Aplicar cobertura de decisión/condición. Asignar a cada condición al menos una vez un resultado verdadero y otro falso.
  - Probar los bucles.

La heurística de malicia consiste en probar con casos típicos de error no comprendidos en los casos generados pero susceptibles de provocar un error, como por ejemplo los datos fronterizos. Por ejemplo, el cero como valor es un buen candidato que se debe utilizar como caso de prueba o como parte de uno. Para los casos de ficheros con listas o simplemente listas, conviene probar:

- Listas vacías.
- Listas de un elemento.
- Listas de  $n-1$ ,  $n$  y  $n+1$  elementos, siendo  $n$  el número máximo de elementos de la lista.

Además, en el caso, por ejemplo de una función que añade elementos a una lista es interesante agregar datos a las listas de la siguiente forma:

- Añadir un elemento a una lista vacía.
- Añadir un elemento cualquiera al principio de una lista unitaria, es decir, de un elemento.
- Añadir un elemento cualquiera al final de una lista unitaria.
- Añadir un elemento igual al existente en una lista unitaria.
- Añadir un elemento al principio de una lista de al menos tres elementos. En este caso se trata de comprobar el funcionamiento de los punteros. Así, se asegura que el segundo de los elementos de esta lista tendrá punteros al elemento anterior y al siguiente, que el primero de los elementos dispondrá de punteros al siguiente y que el último y tercero de los elementos tendrá como puntero únicamente al anterior.
- Añadir un elemento al final de una lista de al menos tres elementos. Por el mismo motivo que el caso anterior.
- Añadir un elemento en el medio de una lista de al menos seis elementos. Este caso tiene como fin, nuevamente comprobar el uso de punteros en un caso un poco más complejo que la lista de tres pero que genera casos de prueba diferentes. El método consiste en hacer pruebas introduciendo datos con el mismo fin que para la lista de tres elementos, es decir,

al principio y al final de la misma. También es interesante añadir un elemento justo en medio de los seis datos de la lista.

En cualquier caso, para que el método de generación de casos de pruebas sea eficiente es necesario que el número de casos no se dispare explotándose de forma combinatoria y que el esfuerzo necesario para llevar a cabo esta tarea sea manejable.

Las siguientes clases Java representan respectivamente un puesto determinado en el organigrama de una empresa y el código de error obtenido al dar de alta un empleado en la base de datos de empleados de la empresa.

```
public class Puesto {

public static final int ERROR_CONNECT_BD    -1;
public static final int MAL_NOMBRE          -2
public static final int MAL_DESPACHO        -3
public static final int MAL_PUESTO          -4
public static final int MAL_SUELDO          -5
public static final int MAL_EDAD            -6
}

Public class CodigoError {

public static final String JEFE_AREA = JEFE_AREA;
public static final String DIRECTOR_COMERCIAL = DIRECTOR_COMERCIAL;
public static final String JEFE_PROYECTO = JEFE_PROYECTO;
public static final String ANALISTA = ANALISTA;
public static final String PROGRAMADOR = PROGRAMADOR;
}
```

El siguiente método se encarga de dar de alta un empleado (registro) en la base de datos de empleados.

```
public int Alta (char nombre[256], char despacho [5], Puesto p, int
edad, double sueldo_netto, double * retencion );
```

Esta función da de alta un registro en la base de datos. Los parámetros de entrada del método son los siguientes:

Nombre: nombre del empleado, cadena de letras o espacios (no números, ni caracteres ascii que no sean letras), de longitud en el intervalo [4...255].

Despacho: despacho asignado, el primer carácter es 'A' o 'B', los tres siguientes son números.

P: El puesto asignado, de acuerdo al tipo enumerado "Puesto"

Edad: Edad del empleado, número en el intervalo [18..., 67].

Sueldo\_netto: Sueldo neto mensual del empleado, en el intervalo [1000...6000]

Retención: parámetro de salida con el cálculo de la retención a efectuar. Esta se calcula de la siguiente forma:

Si el sueldo está entre 1000 y 2000 (incluido) entonces la retención es del 8.0. La retención sube 1,5 cada 1000 euros. Además, dependiendo del puesto (debido a las primas), hay que añadir las siguientes cantidades: si el puesto es jefe de área, se añade un 3.5, si es director comercial un 3, si es jefe de proyecto y la edad es mayor que 30, un 2. Analistas y programadores no tienen primas.

La función devuelve los códigos definidos (de  $-1$  a  $-5$ ) en caso de error, o 1 si todo ha ido bien. El orden de comprobación de errores en los parámetros es el indicado en los “defines”.

Las clases válidas y las inválidas para cada dato siguiendo la técnica de caja negra son:

Dato	Clase válida	Clase no válida
Conectividad	Sí	No
Nombre	Sólo letras o espacios	Contiene caracteres $\neq$ de letras o espacios
	$4 \leq \text{longitud} \leq 255$	longitud $< 4$
		longitud $> 255$
Despacho	Annn	1. <sup>er</sup> carácter $\neq$ A o B
	Bnnn	2. <sup>o</sup> , 3. <sup>o</sup> , 4. <sup>o</sup> caracteres $\neq$ número
	Longitud = 4	longitud $< 4$
		longitud $> 4$
Puesto	cadena $\in$ {JEFE_AREA, DIRECTOR_COMERCIAL, JEFE_PROYECTO, ANALISTA, PROGRAMADOR}	cadena $\notin$ {JEFE_AREA, DIRECTOR_COMERCIAL, JEFE_PROYECTO, ANALISTA, PROGRAMADOR}
Edad	[18,67]	$< 18$
		$> 67$
Sueldo neto	[1000...6000]	$< 1000$
		$> 6000$
Retención	*0,08 si sueldo [1000...2000] *0,095 si sueldo (2000...3000] *0,11 si sueldo (31000...4000] *0,125 si sueldo (4000...5000] *0,14 si sueldo (5000...6000]	Cualquier otro valor
	+ 3,5% si puesto JA + 3% si puesto DC + 2% si puestp JP y edad $> 30$ + 0 en otro caso	
Código error	1	$-1, -2, -3, -4, -5, -6$

Se presentan en la siguiente página los correspondientes casos de prueba utilizando las clases de equivalencia especificadas anteriormente componiendo los casos con valores válidos y no válidos, así como el resultado generado por cada caso como parte integrante del mismo.

SALIDA			ENTRADA				
Código error	Retención	Conectividad	Nombre	Despacho	Puesto	Sueldo neto	Edad
ERROR_CONNECT_BD	—	No	—	—	—	—	—
MAL_NOMBRE	—	Sí	Lui3 Alonso	—	—	—	—
MAL_NOMBRE	—	Sí	Lui	—	—	—	—
MAL_NOMBRE	—	Sí	Arturo... (más de 255)	—	—	—	—
MAL_DESPACHO	—	Sí	Juan López	Z321	—	—	—
MAL_DESPACHO	—	Sí	Alberto Pérez	AB35	—	—	—
MAL_DESPACHO	—	Sí	Luisa Ruiz	B12	—	—	—
MAL_DESPACHO	—	Sí	Ana Blanco	A3456	—	—	—
MAL_PUESTO	—	Sí	Ana Blanco	B345	COMERCIAL	—	—
MAL_SUELDO	—	Sí	Ana Blanco	B345	JEFE_AREA	999	—
MAL_SUELDO	—	Sí	Luis Alonso	A792	DIRECTOR_COMERCIAL	6001	—
MAL_EDAD	—	Sí	Ana Blanco	B345	PROGRAMADOR	1900	17
MAL_EDAD	—	Sí	Ana Blanco	B345	JEFE_PROYECTO	4200	68
1	1001*0,08	Sí	Pepe Pérez	A234	PROGRAMADOR	1001	56
1	2001*0,095	Sí	Santiago Aler	B567	ANALISTA	2001	43
1	3001*(0,11+2)	Sí	Alberto Ruiz	B345	JEFE_PROYECTO	3001	31
1	3001*0,11	Sí	Alberto Ruiz	B345	JEFE_PROYECTO	3001	30
1	4001*(0,125+3)	Sí	Juan Álvarez	A792	DIRECTOR_COMERCIAL	4001	55
1	5001*(0,14+3,5)	Sí	Ana Gómez	A234	JEFE_AREA	5001	67

Es importante señalar que además de las clases válidas y no válidas, es necesario probar con los valores fronterizos cuando un parámetro pueda tomar un rango de valores. Por ejemplo, si un parámetro puede tomar valores de 1 a  $n$ , habrá que probar con las cadenas de los siguientes caracteres: 0, 1, 2,  $n-1$ ,  $n$  y  $n+1$ .

Como se verá a lo largo de este texto y, en particular, en el Capítulo 8 (Mejora de la mantenibilidad mediante JTestCase), es posible ahorrar tiempo en la definición de los casos de prueba utilizando el formato XML de JTestCcase. Además, utilizando esta herramienta se garantiza que la definición de un caso de prueba es única y entendible por la herramienta y el diseñador. Por otra parte, la ventaja de definir los casos de prueba en XML según el formato de dicha herramienta (JTestCase) es que se evita tener que escribirlos por duplicado en otros documentos. En este sentido, basta con hacer una descripción del caso de prueba y asignarle un identificador de forma que la definición en detalle del caso de prueba (entradas salidas y condiciones a verificar) se encuentre en un archivo XML que tiene la ventaja de ser legible por las personas (equipo de pruebas) y por el código de pruebas.

## 1.7. Estrategia de pruebas

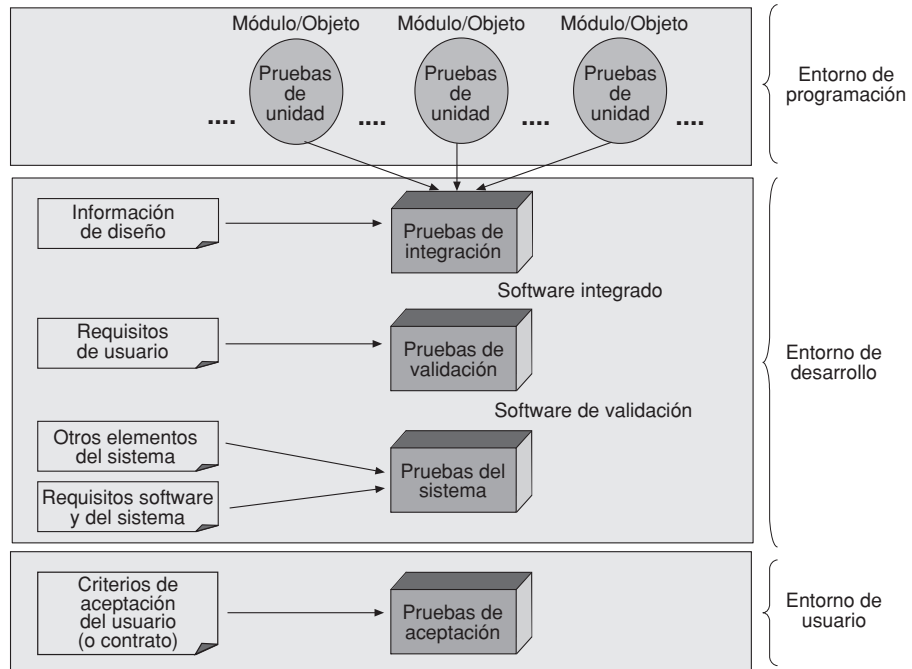
Esta sección explica cuál es la estrategia adecuada para llevar a cabo las pruebas en todo el sistema software de tal forma que finalice la fase con la satisfacción de los ingenieros de software y de los usuarios. Así, para realizar una buena estrategia de pruebas que cubra todo el software desarrollado, es necesario realizar las pruebas desde dentro hacia fuera, es decir, comenzar con módulos unitarios y acabar con el sistema completo.

Las pruebas que se realizan para conseguir llevar a cabo la estrategia completa se enumeran a continuación y se desarrollan en los capítulos siguientes.

- Pruebas unitarias. Se comprueba la lógica, funcionalidad y la especificación de cada módulo o clase aisladamente respecto del resto de módulos o clases.
- Pruebas de integración. Se tiene en cuenta la agrupación de módulos o clases y el flujo de información entre ellos a través de las interfaces.
- Pruebas de validación. Se comprueba la concordancia respecto a los requisitos de usuario y software.
- Pruebas del sistema. Se integra el sistema software desarrollado con su entorno hardware y software.
- Pruebas de aceptación. El usuario valida que el producto se ajuste a los requisitos del usuario.

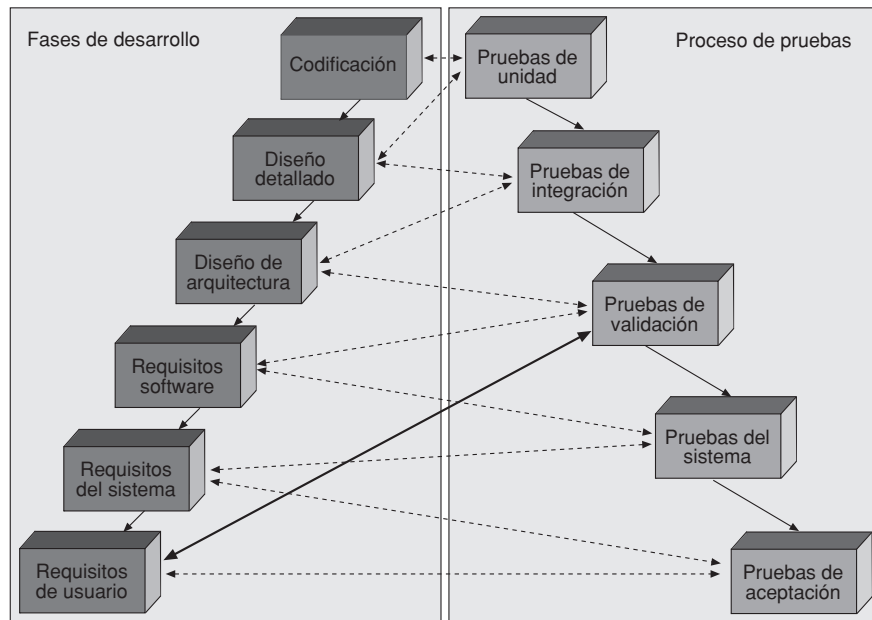
Es importante comprobar el resultado generado por cada fase del ciclo de vida de un proyecto de software antes de pasar a la siguiente fase. De este forma, no se arrastran errores cuyo coste de corrección será mayor cuanto más tarde se detecten.

La estrategia de pruebas descrita se representa en la Figura 1.5. Lo que se pretende con esta estrategia es obtener una estabilidad incremental del sistema software. Sin embargo, hay que destacar que cada nivel de prueba se corresponde con una fase del proceso de desarrollo del pro-



**Figura 1.5.** Estrategia de pruebas.

ducto software, correspondiendo los niveles más altos de las pruebas a las primeras fases del desarrollo, como se representa en la Figura 1.6. Esto deriva directamente en el hecho de que el



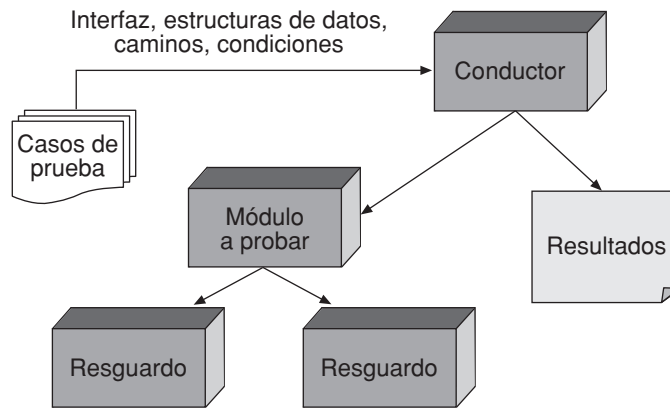
**Figura 1.6.** Relación Nivel de pruebas-Fase de desarrollo.

coste de encontrar y corregir un error aumenta a medida que avanzamos en la estrategia de pruebas. Así, si se detecta y corrige un error en la fase de pruebas del sistema, habrá que modificar los requisitos del sistema, los requisitos software, el diseño y el código de forma consistente y coherente.

### 1.7.1. Pruebas unitarias

Las pruebas unitarias se corresponden con la prueba de cada uno de los módulos o clases del programa de forma independiente y es realizada por el programador en su entorno de trabajo. En definitiva, consiste en probar los bloques más pequeños con identidad propia presentes dentro del programa. De esta forma, si una prueba descubre un nuevo error, este está más localizado. Además, se pueden probar simultáneamente varios módulos.

El enfoque de las pruebas unitarias es el de las técnicas de caja blanca. Para ello se crean módulos conductores y módulos resguardo. Un módulo conductor o módulo impulsor es un módulo específicamente creado para la prueba que llama al módulo a probar. Un módulo resguardo o módulo auxiliar es un módulo específicamente creado para la prueba que es llamado por el módulo a probar, tal y como se muestra en la siguiente figura.



**Figura 1.7.** Esquema de pruebas unitarias.

Se construyen, por tanto, módulos resguardo o módulos conductores cuya función es el paso de parámetros o variables o hacer las llamadas necesarias al módulo que se desea probar de tal forma que se pruebe el módulo de forma unitaria pero con el paso de parámetros real o desde otros módulos. De esta forma, se prueba el módulo en cuestión y se corrigen los errores que surjan de dicho módulo, de tal manera que cuando se pase a la siguiente etapa de pruebas, los módulos estén todos probados de forma independiente.

En general, en un lenguaje orientado a objetos, los resguardos son llamados clases colaboradoras y se caracterizan por mantener una relación de asociación con la clase a probar. En el Capítulo 7, Objetos Mock, se presentan técnicas para la creación de resguardos de forma completamente automatizada.



## 1.7.2. Pruebas de integración

Esta etapa consiste en integrar los módulos o clases<sup>3</sup>, ya probados de forma independiente en las pruebas unitarias centrándose en probar sus interfaces. Habitualmente se utiliza el enfoque de caja negra.

La cuestión principal es determinar la manera en la que se combinan los distintos módulos. Hay dos estrategias básicas.

- Prueba no incremental o big bang, en la que no hay un procedimiento establecido y se van integrando los módulos sin ningún orden establecido. Se desaconseja totalmente.
- Prueba incremental, en la que se supone un diseño en forma jerárquica o de árbol y establece un orden que puede ser descendente, ascendente o sandwich. Los módulos se van integrando poco a poco con unas especificaciones establecidas. Estos tres tipos de integración incremental se describen en las próximas secciones.

### 1.7.2.1. Pruebas de integración ascendentes

Se comienza a integrar por los módulos terminales del árbol y se continúa integrando de abajo hacia arriba hasta llegar al módulo raíz. En esta estrategia se utilizan módulos conductores y el procedimiento básico es el siguiente:

1. Se combinan módulos del nivel más bajo en grupos.
2. Se construye un conductor para coordinar la E y S de los casos de prueba.
3. Se prueba el grupo.
4. Se eliminan los conductores sustituyéndolos por los módulos reales y se combinan los grupos moviéndose hacia arriba por la estructura del programa.
5. Se hacen pruebas de regresión: repetir ciertos casos de prueba que funcionaban con el software antes de sustituir los módulos reales para asegurar que no se introducen nuevos errores.

### 1.7.2.2. Pruebas de integración descendentes

En este caso se comienza con el módulo superior y se continúa hacia abajo por la jerarquía de control bien en profundidad, bien en anchura. Se utilizan módulos resguardo.

Las fases que se siguen son:

1. Se usa el módulo de control principal como conductor de pruebas, construyendo resguardos para los módulos inmediatamente subordinados.
2. Se sustituyen uno a uno los resguardos por los módulos reales.
3. Se prueba cada vez que se integra un nuevo módulo.
4. Se continúa reemplazando módulos resguardo por módulos reales hasta llegar a los nodos terminales.

---

<sup>3</sup> En adelante, se utilizará el termino módulo para referirse a módulos o clases dependiendo de si el lenguaje de programación es estructurado u orientado a objetos respectivamente.

5. Se hacen pruebas de regresión: repetir ciertos casos de prueba que funcionaban con el software antes de sustituir los módulos reales para asegurar que no se introducen nuevos errores.

### 1.7.2.3. Pruebas de integración sandwich

Esta estrategia combina las dos anteriores, es decir las aproximaciones ascendentes y descendentes.

De esta forma, se aplica la integración ascendente en los niveles inferiores de la jerarquía de módulos y paralelamente, se aplica la integración descendente en los niveles superiores de la jerarquía de módulos. La integración termina cuando ambas aproximaciones se encuentran en un punto intermedio de la jerarquía de módulos.

### 1.7.2.4. Elección del módulo/clase crítico/a

Para decidir si se utiliza una aproximación ascendente, una descendente o una de tipo sándwich es imprescindible investigar sobre cuál es el módulo o los módulos más adecuados para comenzar a integrar. En general, los módulos a los que se les da preferencia por ser más propensos a contener errores son los módulos de entrada o salida y los módulos críticos, que son aquellos que cumplen con alguna de las siguientes características:

- Está dirigido a varios requisitos del software.
- Tiene un nivel de control alto.
- Es complejo o contiene un algoritmo nuevo.
- Es propenso a errores.
- Tiene unos requisitos de rendimiento muy definidos o muy estrictos.

Así, se comenzarán las pruebas por el o los módulos que cumplan alguna de las condiciones anteriormente señaladas. Si estos se corresponden con los módulos superiores, se utilizará una estrategia descendente. Si, por el contrario, se trata de los módulos inferiores, se seguirá una estrategia ascendente y si los módulos críticos se encuentran tanto en la parte superior como en la inferior del diseño jerárquico, la aproximación más adecuada es la de tipo sándwich.

### 1.7.2.5. Acoplamiento y cohesión

El acoplamiento es una medida de la interconexión entre los módulos de un programa. El diseño es la fase en la que se puede y debe tener en cuenta esta medida. Así, hay que tender a un bajo acoplamiento, ya que, entre otras cosas, se minimiza el efecto onda (propagación de errores), se minimiza el riesgo al coste de cambiar un módulo por otro y se facilita el entendimiento del programa.

La cohesión mide la relación, principalmente funcional pero no sólo, de los elementos de un módulo. Hay que conseguir un alto grado de cohesión ya que conlleva un menor coste de programación y consecuentemente una mayor calidad del producto.

En definitiva, hay que diseñar haciendo que los módulos sean tan independientes como sea posible (bajo acoplamiento) y que cada módulo haga (idealmente) una sola función (alta cohesión).

Ambas medidas tienen una gran repercusión en las pruebas, ya que facilitan las mismas tanto a la hora de detección de un error, como a la hora de corregirlo y realizar las correspondientes pruebas de regresión.

### 1.7.3. Pruebas de validación

El objetivo de las pruebas de validación o pruebas funcionales es comprobar que se cumplen los requisitos de usuario. Es por tanto una fase en la que interviene tanto el usuario como el desarrollador y se realiza en el entorno de desarrollo. Los criterios de validación que se utilizan son aquellos que se acordaron al inicio del proyecto en la fase de definición de requisitos y que deben constar en el correspondiente documento de especificación de requisitos. Dado que se está probando el sistema completo y que lo relevante es la salida producida, la técnica que se utiliza es la de caja negra comprobando los resultados obtenidos con los resultados esperados.

Las pruebas de validación constan de dos partes:

- La validación por parte del usuario para comprobar si los resultados producidos son correctos.
- La utilidad, facilidad de uso y ergonomía de la interfaz de usuario.

Ambas son necesarias y se deben llevar a cabo para completar las pruebas de validación de manera satisfactoria.

Es frecuente y recomendable realizar una matriz de trazabilidad donde se ponen los requisitos en las filas y los módulos de los que se compone el sistema en las columnas. Se traza una cruz en aquellas celdas en las que el requisito correspondiente a esa fila se encuentra representado en el módulo de esa columna. Un requisito puede encontrarse en varios módulos y un módulo puede contener varios requisitos. Esta matriz es muy útil ya que permite comprobar que todos los requisitos están cubiertos por el sistema. Además, en las pruebas se reconoce qué requisito no se cumple cuando un módulo falla o, al contrario, qué módulo hay que corregir cuando un requisito no se cumple. Así mismo para mantenimiento y gestión de cambios es de gran utilidad. Permite saber dónde hay que corregir y modificar en cada caso asegurando que las modificaciones se realizan en todas las partes del sistema donde está involucrado el error o cambio que se esté tratando. A continuación, se muestra un ejemplo de una matriz de trazabilidad.

**Tabla 1.2.** Tabla de referencias cruzadas entre módulos y requisitos para verificar que se han contemplado todos los requisitos.

<b>Módulo</b> <b>Párrafo/Req.</b>	<b>Módulo A</b>	<b>Módulo B</b>	<b>Módulo C</b>	<b>...</b>	<b>Módulo n</b>
Párrafo/Req. 1.1.1.	X				
Párrafo/Req. 1.1.2.		X	X		
Párrafo/Req. 1.1.3.			X		
...					
Párrafo/Req. x.y.z.					

### 1.7.3.1. Pruebas alfa y beta

Dentro de las pruebas de validación, merece la pena destacar las pruebas alfa y las pruebas beta. Las primeras las lleva a cabo el cliente en el lugar de desarrollo y se prueba el sistema aunque no estén las funcionalidades finalizadas al cien por cien. De hecho, es posible añadir nuevas funcionalidades durante el proceso de pruebas alfa.

Las segundas, las pruebas beta, las llevan a cabo los potenciales consumidores en su entorno y el desarrollador habitualmente no está presente.

Por ejemplo, para herramientas software con un mercado potencial muy extenso, los productos de las empresas desarrolladoras son probados por miles de personas que realizan pruebas beta (casi siempre de forma voluntaria) y reportan los errores encontrados antes de lanzar la herramienta al mercado.

### 1.7.4. Pruebas del sistema

En los pasos anteriores el sistema de pruebas es, habitualmente, distinto al de producción. Es aquí cuando se prueba el sistema integrado en su entorno hardware y software para verificar que cumple los requisitos especificados.

En ocasiones se realiza antes que las pruebas de validación.

Entre otras pruebas consta de:

- Pruebas de interfaces externas (HW, SW, de usuario).
- Pruebas de volumen.
- Pruebas de funcionamiento (funciones que realiza).
- Pruebas de recuperación.
- Pruebas de seguridad.
- Pruebas de resistencia (en condiciones de sobrecarga o límites).
- Pruebas de rendimiento/comportamiento (en condiciones normales).
- Pruebas de fiabilidad.
- Pruebas de documentación (adecuación de la documentación de usuario).

En algunas ocasiones las pruebas de validación y las del sistema se realizan conjuntamente.

### 1.7.5. Pruebas de aceptación

Es el último paso antes de la entrega formal del software al cliente y se realiza, normalmente, en el entorno del usuario. Consiste en la aceptación por parte del cliente del software desarrollado.

El cliente comprueba que el sistema está listo para su uso operativo y que satisface sus expectativas. Habitualmente es el usuario quien aporta los casos de prueba.

En el caso en el que se trate de una herramienta desarrollada para salir al mercado, se suelen llevar a acabo las pruebas RTM testing (Release To Market). En estas pruebas se comprueba cada funcionalidad del sistema completo en el entorno real de producción.

Una vez finalizadas las pruebas de aceptación de forma satisfactoria, se procede a la entrega formal de la aplicación o al empaquetamiento y distribución del sistema y se da por concluido el proceso de desarrollo para pasar a la fase de mantenimiento. Durante la fase de mantenimiento también existirán pruebas de aceptación asociadas a las nuevas versiones del software que se hayan creado en respuesta a las correspondientes peticiones de mantenimiento.

En las pruebas unitarias, de integración y del sistema participa únicamente el equipo desarrollador. En las pruebas de validación y de aceptación está, además, involucrado el usuario final o usuario potencial.

## 1.8. Pruebas de sistemas orientados a objetos

Se muestra en este capítulo las diferencias de la estrategia explicada anteriormente cuando se trata de un sistema orientado a objetos. Nótese que este capítulo es un capítulo introductorio que pretende dar una visión global sobre las pruebas de software en todo tipo de desarrollos. Sin embargo, el grueso de este texto está enfocado a las pruebas orientadas a objetos. El motivo es que, actualmente, la inmensa mayoría de los sistemas están desarrollados utilizando lenguajes orientados a objetos como Java, C++ o .Net. En particular, en el Capítulo 2 se hace un análisis en profundidad del procedimiento de pruebas unitarias orientadas a objetos.

Tal y como define Sommerville en [2], un objeto es una entidad que tiene un estado y un conjunto de operaciones definidas que operan sobre ese estado. El estado se representa como un conjunto de atributos del objeto. Las operaciones que son asociadas al objeto proveen servicios a otros objetos, denominados clientes, que solicitan dichos servicios cuando se requiere llevar a cabo algún cálculo. Los objetos se crean conforme a una definición de clases de objetos. Una definición de clases de objetos sirve como una plantilla para crear objetos. Esta incluye las declaraciones de todos sus atributos y operaciones asociadas con un objeto de esa clase.

Las pruebas orientadas a objetos siguen básicamente las mismas pautas y fases que las expuestas anteriormente en este capítulo con algunas diferencias. Lógicamente, en lugar de módulos, se prueban clases u objetos pertenecientes a clases que, en muchas ocasiones, están constituidos por más de una función con lo que, en general, son, en tamaño, mayores que los módulos. Esto implica, por ejemplo, que la técnica de caja blanca haya que llevarla a cabo con objetos de grano más grueso.

Teniendo esto en cuenta, los niveles o etapas de las pruebas orientadas a objetos se pueden clasificar en los siguientes pasos:

1. Pruebas de los métodos y operaciones individuales de las clases.
2. Pruebas de las clases individuales.
3. Pruebas de agrupaciones de objetos (integración).
4. Pruebas del sistema entero.

### 1.8.1. Pruebas de clases de objetos

El objetivo de estas pruebas es asegurar que una clase y todas sus instancias cumplen con el comportamiento definido en la definición de requisitos realizada en la fase de análisis del ciclo de vida del desarrollo. Al igual que en las pruebas unitarias, aquí también el proceso consiste en ejecutar todas las instrucciones de un programa al menos una vez y ejecutar todos los caminos del programa. Para cubrir las pruebas completamente es necesario:

- Probar todas las operaciones asociadas a los objetos de forma aislada.
- Probar todos los atributos asociados al objeto.
- Ejecutar los objetos en todos los estados posibles. Para ello se simulan todos los eventos que provocan un cambio de estado en el objeto en cuestión. Por ello, es común utilizar el diagrama de estados como punto de partida de las pruebas.

Al igual que en las pruebas unitarias de programas no orientados a objetos, se utiliza la complejidad ciclomática como medida de los caminos que se deben probar y responde a la misma fórmula:

$$V(G) = \text{Aristas} - \text{Nodos} + 2$$

En el caso en el que en lugar de un grafo sea necesario realizar multigrafos, esta fórmula y el consecuente método siguen siendo válidos.

### 1.8.2. Pruebas de integración de objetos

Consiste en asegurar que las clases, y sus instancias, conforman un software que cumple con el comportamiento definido. Para ello, se realizarán pruebas que verifiquen que todas las partes del software funcionan juntas de la forma definida en especificación de requisitos. Esta etapa recibe también el nombre de pruebas de cluster ya que el método más común es integrar utilizando el criterio de agrupar las clases que colaboran para proveer un mismo conjunto de servicios.

Las pruebas ascendentes, descendentes y sándwich no cobran sentido en las pruebas de integración a objetos. Hay dos tipos de pruebas que se realizan generalmente en la integración de objetos:

- Uno es el basado en escenarios. En este caso, se definen los casos de uso o escenarios que especifican cómo se utiliza el sistema. A partir de aquí, se comienza probando los escenarios más probables, siguiendo a continuación con los menos comunes y finalizando con los escenarios excepcionales.
- El otro es el basado en subprocesos, también llamado pruebas de cadenas de eventos. Estas pruebas se basan en probar las respuestas obtenidas al introducir una entrada específica o un evento del sistema o un conjunto de eventos de entrada.

En este apartado cabe destacar las pruebas de interfaces que, si bien son necesarias en cualquier programa, cobran especial relevancia en los sistemas orientados a objetos.

### 1.8.2.1. Pruebas de interfaces

De acuerdo con R. R. Lutz en su publicación *Analysing software requirements error in safety-critical embedded system* los errores de interfaces más comunes se dividen en tres tipos, que se describen a continuación y las pruebas deben ir dirigidas a la búsqueda de estos tipos de errores.

- Abuso de interfaces. Este error se basa en una inadecuada utilización de la interfaz. En muchas ocasiones es debido al paso de parámetros erróneos, bien porque estos se pasen en un orden incorrecto, bien porque se pasa un número erróneo de parámetros o bien porque los propios parámetros son de tipo erróneo.
- Malentendido de interfaces. En una llamada o invocación de un componente a otro, la especificación de la interfaz del componente invocado es malentendida por el componente que invoca y provoca un error.
- Errores en el tiempo. Este es un error mucho más inusual y casi específico de un sistema de tiempo real que utiliza memoria compartida o una interfaz para paso de mensajes. El problema surge cuando las velocidades a las que operan el productor de datos y el consumidor de datos son distintas, lo que genera un error.

En las pruebas de interfaz es también conveniente hacer uso de la heurística mailiciosa pasando, por ejemplo, parámetros con apuntadores nulos, o variar el orden de objetos que interactúan mediante memoria compartida. Además también se deben diseñar casos de prueba en los que los valores de los parámetros sean valores frontera (los primeros y últimos de su rango).

### 1.8.3. Pruebas del sistema

Estas pruebas no difieren de las ya mencionadas como pruebas del sistema en la sección 1.7.4. de este capítulo y se remite a dicha sección.

## 1.9. Depuración de errores

Es el proceso de eliminación de errores software mediante la detección de las causas a partir de los síntomas.

Un breve repaso de los pasos a seguir para llevar a cabo la depuración de errores es:

1. Especificación de la desviación producida (¿qué, cuándo y en qué condiciones?).
2. Determinación de la ubicación y de la causa. Se suelen usar herramientas automáticas. Los pasos básicos son:
  - Trazar.
  - Realizar volcados de memoria.
  - Volver atrás hasta determinar la causa.
3. Corregir.

## 1.10. Otras pruebas

### 1.10.1. Pruebas de regresión

Cuando se realiza un cambio en el software, es necesario volver a probar comportamientos y aspectos ya probados de forma satisfactoria anteriormente y volver a utilizar algunos del conjunto de los casos de prueba diseñados para asegurar que el cambio efectuado en el software no ha interferido en el correcto comportamiento del sistema software y no se han producido efectos colaterales. Este tipo de pruebas son, por tanto, muy comunes durante la fase de mantenimiento de un sistema software.

### 1.10.2. Pruebas de estrés

Se introduce en el sistema un elevado número de datos, preguntas, transacciones, usuarios, etc. para asegurar que el sistema funciona como se espera bajo grandes volúmenes de entradas, muchas más de las esperadas. Se realizan durante la etapa de pruebas del sistema.

### 1.10.3. Pruebas de interfaz de usuario

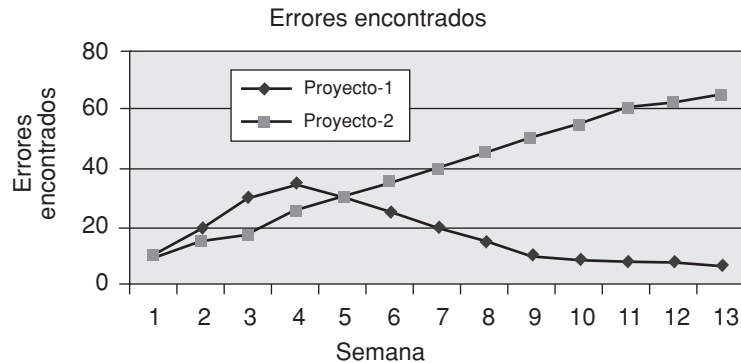
Estas pruebas se refieren a la interfaz hombre-máquina. Existen dos roles bien diferenciados:

- Uno de ellos consiste en probar la interfaz de usuario para garantizar que funciona correctamente desde el punto de vista técnico y que cumple los estándares y requerimientos definidos. La realización de estas pruebas se lleva a cabo utilizando todos los conceptos definidos anteriormente en este capítulo puesto que se trata de probar un código y comprende los términos de validación y de verificación.
- El otro rol, menos habitual pero tan importante o más que el anterior, es la usabilidad de la interfaz. La usabilidad se deriva de una comunicación efectiva de la información. Para ello, es necesario tener muy en cuenta los requisitos de usuario y desarrollar la interfaz de usuario de acuerdo con ellos. Es muy recomendable realizar una maqueta que nos permita validar los requisitos de usuario desde fases tempranas del proceso de desarrollo. La usabilidad tiene que tener en cuenta, además del perfil de los potenciales usuarios aspectos tales como, número de ventanas, uso de ratón, colores utilizados, font, diseño de gráficos, etc. Si un sistema funciona correctamente y desde el punto de vista técnico está perfectamente verificado pero al usuario final le es incómoda o poco manejable la interfaz, ello dará lugar a que no use el producto software y al consecuente fracaso del proyecto.

## 1.11. Criterios para dar por finalizadas las pruebas

La pregunta ¿cuándo se dan por acabadas las pruebas? es una pregunta común entre los ingenieros de software que, lamentablemente, no tiene una respuesta contundente. La afirmación más habitual es: las pruebas finalizan cuando el tiempo para realizarlas se acabe (o el dinero, pero de





**Figura 1.8.** Gráfico-Criterio para finalizar las pruebas.

forma más habitual el tiempo). Sin embargo esta respuesta es poco precisa, aún siendo la más usada.

Partiendo de la base de que estadísticamente no hay ningún software entregado y dado por finalizado que esté libre de errores, sí es necesario que exista un alto porcentaje de confianza en el sistema desarrollado. Estadísticamente se acepta un 95% como porcentaje válido.

En cualquier caso, un buen diseño de casos de prueba ayuda a elevar este grado de confianza escogiendo aquellos casos de prueba que tengan más probabilidades de tener errores y que cubran el espectro de pruebas de la forma más amplia posible.

En ocasiones se puede dibujar una gráfica con el número de errores encontrados en relación al tiempo y determinar como final de las pruebas cuando se lleve un tiempo predeterminado con un número de errores encontrados estable y bajo. Esto se muestra en la Figura 1.8. En este ejemplo, las pruebas del proyecto 1 se pueden dar por finalizadas después de la semana 13, pero no así las del proyecto 2.

## 1.12. Equipo de pruebas

El equipo de pruebas teóricamente ideal está formado por parte del personal de aseguramiento de calidad, externo al proyecto donde se ha realizado el sistema que se está probando, por desarrolladores que han intervenido en la programación del sistema, por programadores que no han intervenido en dicha codificación y por personal externo al proyecto que tenga un perfil similar al usuario final. Sin embargo, en ocasiones esto no es posible, bien porque no existe departamento de calidad, bien porque no se encuentran personas internas en la organización que tengan un perfil similar al usuario, etc. En estos casos es imprescindible que el equipo de pruebas lo formen, al menos desarrolladores involucrados en el proyecto, desarrolladores no involucrados en el proyecto, o al menos no en la etapa de codificación, y personal externo al proyecto que no sean programadores.

En particular, para las pruebas de unidad y de integración es suficiente con desarrolladores que hayan y que no hayan participado en la codificación del sistema. Para el resto de las pruebas,

se introduciría el tercer elemento, el personal ajeno al proyecto y con un perfil lo más similar posible al usuario final. En cualquier caso, si en la organización existe personal exclusivo de aseguramiento de la calidad, debe participar en todo momento.

Además, en las pruebas de validación y en las de aceptación, el usuario debe intervenir para validar los resultados del sistema. En particular, en las de aceptación el usuario es el gran protagonista que decide finalmente si está conforme con el sistema desarrollado o no.

Finalmente, hay organizaciones donde existen equipos de pruebas independientes de los proyectos cuya misión es, justamente, realizar las pruebas de todos los sistemas.

## 1.13. Errores más comunes que se cometen en la fase de pruebas

Como resumen de este capítulo genérico de pruebas software, se listan en este apartado los errores más comunes que se cometen en la fase de pruebas (normalmente por los desarrolladores):

- Suponer que no se encontrarán errores.
- Efectuar las pruebas sólo por el programador que ha realizado el programa.
- No especificar el resultado esperado.
- No probar condiciones inválidas.
- No definir las entradas, acciones y salidas esperadas.
- No construir un conjunto de casos de pruebas completo.
- No llevar a cabo un enfoque basado en el riesgo, y no concentrarse en los módulos más complejos y en los que podrían hacer el mayor daño si fallan.
- No recordar que también se pueden cometer errores durante las pruebas.

## 1.14. Documentación de pruebas

Tomando como referencia el estándar IEEE 829-1983, la documentación mínima que se debe generar durante la fase de pruebas de un ciclo de vida software es:

- Plan de pruebas.
- Especificación de los requerimientos para el diseño de los casos de prueba.
- Caso de prueba, que incluye, además, la descripción del procedimiento de prueba y la descripción del ítem a probar (véase Capítulo 8).
- Reporte de incidentes de prueba.
- Resumen de pruebas.

Basándose nuevamente en la definición del IEEE Standard Glossary of Software Engineering Technology [1], un plan de pruebas es un documento que describe el alcance, enfoque, recursos

y planificación de las actividades de prueba deseadas. Identifica los elementos y características a probar, las tareas de prueba, quién realizará cada tarea y cualquier riesgo que requiera planes de contingencia. Cabe destacar que, a pesar de ser un documento que se utiliza en la fase de pruebas, el plan de pruebas debería escribirse en la etapa de diseño, que es cuando se especifica la arquitectura del sistema y se establecen y detallan los módulos/objetos que van a integrar el sistema. En este momento es mucho más fácil realizar un diseño de los casos de prueba y definir el plan de pruebas.

Seguidamente se lista y se explica un ejemplo estándar del índice de un plan de pruebas:

1. Introducción al documento
2. Alcance de la fase de pruebas
3. Requisitos del entorno de pruebas (hardware y software)
4. Ítems a probar
5. Planificación de las pruebas
  - 5.1. Calendario
  - 5.2. Equipo de pruebas
  - 5.3. Responsabilidades (para cada una de las tareas previstas en el plan)
  - 5.4. Manejo de riesgos (identificación de riesgos y planes de contingencia)
  - 5.5. Técnicas
  - 5.6. Estrategia de integración utilizada
6. Para cada paso de integración
  - 6.1. Orden de integración
  - 6.2. Módulos a ser probados
  - 6.3. Pruebas de unidad para los módulos a ser probados
  - 6.4. Resultados esperados
  - 6.5. Entorno de pruebas
7. Resultados de las pruebas de verificación
  - 7.1. Datos
  - 7.2. Condiciones
8. Resultados de las pruebas de validación
9. Pruebas de estrés
10. Monitorización de las pruebas
11. Referencias y apéndices

El apartado 1, introducción al documento recoge, por una parte, una breve descripción del sistema que se está probando incluyendo sus funcionalidades y, por otra, introduce brevemente cada uno de los capítulos del plan para que el lector se haga una idea genérica de qué va a encontrar en dicho documento.

El apartado 2 del plan define qué se va a probar y qué no, de forma que queda claramente acotado el alcance de las pruebas que se van a realizar.

En el apartado 3 se recoge el entorno tanto hardware como software donde se van a llevar a cabo las pruebas. Es conveniente indicar, además, el entorno de operación donde se va a utilizar el sistema una vez en producción.

En el apartado 4 se define qué se va a probar en relación a qué objetos o módulos, o incluso, funciones. Este apartado es opcional porque habitualmente se prueba todo el sistema comenzando con las pruebas unitarias de todos los objetos/módulos.

El apartado 5 es propiamente el plan. Se debe especificar tanto el plan temporal como la estrategia que se seguirá durante el proceso, las técnicas que se utilizarán en cada caso (caja blanca, caja negra) y una descripción del equipo encargado de llevar a cabo las pruebas haciendo especial hincapié en las responsabilidades y tareas de cada miembro, las relaciones entre ellos y los canales de comunicación.

En el apartado 6 se describen las pruebas de unidad y las de integración explicando paso a paso cómo se van a llevar a cabo, el resultado esperado y el resultado obtenido. En ocasiones, aquí se define únicamente cómo se van a realizar y los resultados se especifican junto con los casos de prueba en el correspondiente documento.

Los apartados 7 y 8 del documento de pruebas se pueden tomar como el resumen de las pruebas, que en ocasiones forma un documento independiente. Si no es así, en ocasiones el apartado 7 sobre verificación desaparece porque ya se recoge en los distintos apartados correspondientes a los diferentes niveles de pruebas. El apartado sobre validación se mantiene y especifica cómo se van a llevar a cabo las pruebas de validación o pruebas funcionales incluyendo los resultados esperados de dichas pruebas. Esto último puede aparecer en el correspondiente informe junto con los casos para las pruebas de validación.

Las pruebas de estrés se detallan posteriormente especificando qué se va a tener en cuenta (un gran volumen de datos, una velocidad alta de introducción de los datos de entrada, etc.) Esta es una forma de garantizar la gestión de errores.

La monitorización de las pruebas, el apartado 10 del documento, explica qué procedimientos se van a utilizar para realizar un seguimiento del proceso de pruebas y poder conocer en todo momento su estado. Algo similar ocurre con los informes de cobertura. Esta tarea se ve facilitada si se utiliza la herramienta Cobertura, descrita también en este texto, que indica la cobertura alcanzada por los casos de prueba creados en un momento dado de la fase de pruebas. Se trata, por tanto, de una herramienta muy útil para la monitorización y reasignación de recursos dependiendo de cómo vaya el proceso en cada momento.

Por último, todas las referencias y apéndices que se deseen introducir se incluyen al final del documento.

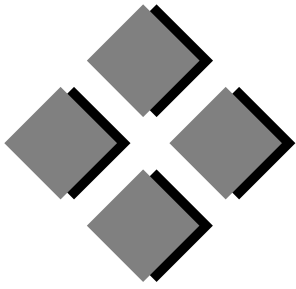
Como ya se ha mencionado en la documentación que se genera en la fase de pruebas, existe un documento donde se definen todos los casos de prueba y para cada caso, habitualmente, se realiza un informe donde se especifique el identificador de la prueba, cómo se va a llevar a cabo la prueba, la fecha en la que se ha realizado, el autor de la misma, el resultado obtenido, comentarios adicionales y por quién ha sido aprobada o supervisada. En este mismo informe se pueden incluir los incidentes generados y los procedimientos definidos para su resolución.

Sin embargo, tal y como se comenta en el Capítulo 8, existen herramientas como JTestCase que permiten definir los casos de prueba en documentos XML mediante una sintaxis específica. De esta forma, es posible diseñar los casos de prueba en un formato (XML) legible al mismo tiempo por el diseñador y por el código encargado de ejecutarlos. Así, además, se mejora sustancialmente la mantenibilidad.

## 1.15. Bibliografía

- IEEE Standard Glossary of Software Engineering Terminolgy [IEEE, 1990]
- Sommerville, I.: *Ingeniería del Software*, Capítulos 19 y 20, sexta edición. Addison Wesley, 2002.
- Pressman, R.: *Ingeniería del Software: Un enfoque práctico*. Capítulos 13 y 14, sexta edición. McGrawHill, 2006.
- Lutz, R. R.: *Analysing software requirements error in safety-critical embedded systems*. *Proc. RE'93*, San Diego, CA, IEEE, 1993.
- Myers, G.: *The art of software testing*. Wiley, 2004.
- Hailpern, B. y Santhanam, P.: «Software Debugging, Testing and Verification», en *IBM Systems Journal*, vol. 41, número 1, 2002. <http://www.research.ibm.com/journal/sj/411/hailpern.html>.





# Capítulo 2

# Pruebas unitarias: JUnit

---

## SUMARIO

<b>2.1.</b> Introducción	<b>2.6.</b> Organización de las clases de prueba
<b>2.2.</b> Instalación	<b>2.7.</b> Ejecución de los casos de prueba
<b>2.3.</b> Primera toma de contacto con JUnit	<b>2.8.</b> Conceptos avanzados en la prueba de clases Java
<b>2.4.</b> Creación de una clase de prueba	<b>2.9.</b> Bibliografía
<b>2.5.</b> Conceptos básicos	

## 2.1. Introducción

Como se ha visto en el capítulo anterior, cuando se habla de pruebas de software, las pruebas unitarias son aquellas cuyo objetivo es probar partes indivisibles del software de forma aislada. En un lenguaje orientado a objetos en general y en el lenguaje Java en particular, estas unidades básicas e indivisibles son las clases, por lo tanto las pruebas unitarias están enfocadas a probar clases. Típicamente a todo proceso de pruebas unitarias le sigue un proceso de pruebas de integración, que tiene como objetivo verificar la correcta interacción entre las diferentes clases que componen un sistema software.

A lo largo de los últimos años han aparecido diferentes entornos para la realización de pruebas unitarias en el lenguaje Java, sin embargo, no cabe duda de que hablar de pruebas unitarias en Java es hablar de JUnit (<http://www.junit.org>).

JUnit es un framework que permite la automatización de pruebas unitarias sobre clases desarrolladas en el lenguaje Java. Fue creado en 1997 por Erich Gamma y Kent Beck basándose inicialmente en un framework para Smalltalk llamado SUnit y que fue desarrollado por este úl-

timo. JUnit es un proyecto de código abierto escrito en Java y distribuido bajo la licencia Common Public License Version 1.0. Está alojado en el sitio Web SurceForge (<http://sourceforge.net/projects/junit/>) y disponible para libre descarga.

Desde su aparición JUnit ha sido utilizado en multitud de proyectos software como una herramienta capaz de asistir eficazmente al desarrollador software en la realización de pruebas unitarias. Y ya desde hace varios años, JUnit se ha convertido en la herramienta de referencia para la realización de pruebas unitarias en el mundo de desarrollo Java. Para hacerse una idea del impacto que ha tenido JUnit en el mundo Java en particular y en el mundo de desarrollo software en general, basta con echar un vistazo a la siguiente cita dedicada a JUnit:

*Never in the field of software development was so much owed by so many to so few lines of code.*

MARTIN FOWLER <sup>1</sup>

cuya traducción al castellano es:

*Nunca en el campo del desarrollo software tantos le han debido tanto a tan pocas líneas de código.*

Sin embargo JUnit no ha recorrido todo este camino en solitario, sino que desde sus inicios han surgido multitud de herramientas para potenciar y complementar su funcionalidad en todos los aspectos imaginables. Desde librerías que permiten la generación automática de documentación con los resultados de las pruebas hasta librerías para la prueba de aplicaciones de acceso a bases de datos, pasando por librerías que facilitan la prueba de interfaces de usuario basadas en AWT o Swing. A modo de anécdota cabe comentar que incluso existen extensiones de JUnit destinadas a la prueba de aplicaciones que utilizan el protocolo SIP para prestar servicios de telefonía por Internet a través de VoIP. Mientras que este capítulo trata las pruebas unitarias con JUnit en profundidad, otros capítulos de este libro exponen la forma en que algunas de estas herramientas pueden aplicarse a las pruebas de software Java para complementar la funcionalidad de JUnit.

Por otra parte, JUnit ha contribuido enormemente al desarrollo de la técnica de pruebas unitarias, sirviendo de inspiración para la creación de herramientas similares destinadas a la automatización de pruebas unitarias en todo tipo de lenguajes de programación: C++, .Net, Delphi, ASP, PHP, Python, PL/SQL, JavaScript, TCL/TK, Perl, Visual Basic, etc.

### 2.1.1. Aportaciones de JUnit

En este punto y una vez que esta herramienta ha sido presentada, se va a hacer un repaso de las características que la han convertido en una herramienta extremadamente útil y popular en la tarea de automatización de pruebas unitarias.

- Se encarga de resolver las tareas repetitivas asociadas al proceso de pruebas como son la organización de las clases de prueba y el manejo de las situaciones de error.

---

<sup>1</sup> Martin Fowler es actualmente uno de los autores más influyentes en el mundo de la programación orientada a objetos. Es especialista en las corrientes metodológicas ágiles y Extreme Programming.



- Delimita claramente las tareas del desarrollador, que se restringen a plasmar la información contenida en los casos de prueba en forma de código Java.
- Proporciona un conjunto de métodos que facilitan la tarea de comprobación de las condiciones contenidas en los casos de prueba definidos. Estos métodos en adelante serán llamados métodos assert.
- Proporciona un mecanismo para ejecutar los casos de prueba de forma ordenada a la vez que mantiene información en tiempo de ejecución de los defectos software encontrados. Dicha información es mostrada al usuario al final del proceso.
- Consta de un muy reducido número de clases y métodos por lo que la curva de aprendizaje es bastante plana. Siendo esta una de las principales razones de la enorme popularidad que ha alcanzado la herramienta.

## 2.1.2. Versiones

A lo largo del tiempo, JUnit ha ido evolucionando, y nuevas versiones han aparecido en SourceForge para ser utilizadas por los miembros de la comunidad Java. Sin embargo, sin duda alguna la versión que alcanzó una mayor popularidad y que quizás más fama ha dado a JUnit, ha sido la versión 3.8.1. Esta versión apareció en Octubre de 2002 y ha sido descargada hasta la fecha en más de un millón de ocasiones. Se trata de una versión madura y estable, que ha resistido el paso del tiempo y la competencia de otros frameworks de pruebas que han aparecido por el camino.

No obstante y casi 3 años y medio después de la aparición de la versión 3.8.1, una nueva versión de JUnit, la versión 4.x, ha hecho aparición. Se trata de una versión que presenta novedades sustanciales respecto a versiones precedentes y que saca provecho de los mecanismos de anotaciones e import estático<sup>2</sup> en la JDK versión 5.0. El resultado es que JUnit vuelve a aparecer en primer plano después de que en los últimos tiempos, y debido a un prolongado periodo de inactividad del proyecto JUnit, otras herramientas de prueba como TestNG<sup>3</sup> le hubieran restado protagonismo.

A lo largo de este capítulo se va a trabajar con ambas versiones de JUnit, la 3.8.1 y la 4.x. El motivo es que ambas versiones son actualmente muy utilizadas. La primera porque ha estado mucho tiempo en el mundo de desarrollo Java, de forma que aun hoy muchos desarrolladores la siguen utilizando casi por inercia. La segunda, porque es una versión completamente novedosa y revolucionaria que pronto se convertirá en el nuevo estándar.

En este capítulo se va a describir cómo realizar pruebas unitarias sobre código Java con la ayuda de JUnit. Sin embargo, antes de continuar conviene tener presente la diferencia entre dos conceptos fundamentales. Por un lado está la técnica de pruebas unitarias, que es universal y se ha ido depurando con el paso del tiempo, y por otro lado se tiene la herramienta, en este caso JUnit, que permite poner dicha técnica en práctica de una forma eficaz y cómoda para el

---

<sup>2</sup> Para aquellos que no estén familiarizados con las anotaciones o el import estático en Java, se recomienda la lectura del Apéndice F. Novedades en Java 5.0.

<sup>3</sup> TestNG es un conjunto de herramientas que, al igual que JUnit, asisten al desarrollador software en la realización de pruebas unitarias. Se trata de un proyecto de código abierto (licencia Apache) y se encuentra alojado en el sitio Web <http://testng.org/>.

desarrollador. En este capítulo se pretende presentar ambos conceptos. Por un lado se va a explicar en detalle en qué consiste la técnica de pruebas unitarias, cuál es el procedimiento general de prueba, qué prácticas son recomendables y cuáles no, en qué aspectos de la prueba se ha de poner mayor esfuerzo y cuáles de ellos minimizar, etc. Por otro lado se van a aportar directrices, sugerencias y ejemplos de uso de JUnit para poner en práctica dicha técnica y procedimientos de prueba. A menudo ambos conceptos van a aparecer entremezclados dentro del capítulo, sin embargo corresponde al lector diferenciarlos y de esta manera formarse un espíritu crítico que le permita conocer a fondo el mundo de las pruebas unitarias y a la vez crecer como desarrollador de software.

Inicialmente, se va a describir el procedimiento de instalación de JUnit así como los pasos previos para comenzar a utilizar dicha herramienta. Posteriormente, se va a pasar a describir el proceso de prueba de una clase con JUnit acompañado de los ejemplos pertinentes, así como de la correcta organización y ejecución de los casos de prueba definidos. Finalmente, se va a hablar de conceptos avanzados en la prueba de elementos del lenguaje Java como puede ser la prueba de excepciones o la prueba de métodos que no pertenecen a la interfaz pública de una clase.

## 2.2. Instalación

El procedimiento de instalación de JUnit es muy sencillo. Simplemente se ha de descargar la versión de JUnit con la que se desea trabajar (recordemos que las versiones con las que se trabajará en este libro son la 3.8.1 y la 4.2) y añadir un fichero .jar con las clases de JUnit a la variable de entorno CLASSPATH<sup>4</sup>. Los pasos a seguir son los siguientes:

1. Descargar la versión de JUnit con la cual se desea trabajar desde el sitio Web de SourceForge (<http://sourceforge.net>). Se trata de un archivo .zip que contiene todo el software distribuido en la versión: clases de JUnit, código fuente, extensiones, javadoc, ejemplos, etc.
2. Descomprimir el archivo descargado, por ejemplo junit4.2.zip, a un directorio creado para tal efecto (por ejemplo c:\descargas en el caso de que se este trabajando con Windows o /home/usuario/descargas en el caso de Linux<sup>5</sup>). Es importante que la ruta completa de ese directorio no contenga espacios ya que si así fuera, pueden ocurrir problemas en la ejecución de los tests. Un error típico derivado de esta circunstancia es el siguiente: `java.lang.ClassNotFoundException: junit.tests.runner.LoadedFromJar`.
3. Al descomprimir el fichero .zip automáticamente se crea una carpeta con todo su contenido (que será c:\descargas\junit4.2 en Windows o /home/usuario/descargas/junit4.2 en Linux). Dentro de esta carpeta se encuentra un archivo .jar (junit.jar en la versión 3.8.1 y junit-4.2.jar en la versión 4.2) que contiene las clases de JUnit, necesarias para la ejecución. Dicho archivo, así como la ruta del directorio actual, es decir “.”, ha de añadirse a la variable de entorno CLASSPATH.

<sup>4</sup> Esta tarea se puede realizar de diferentes formas y es una tarea dependiente del sistema operativo, para obtener información en detalle véase Capítulo 3 y Apéndice A.

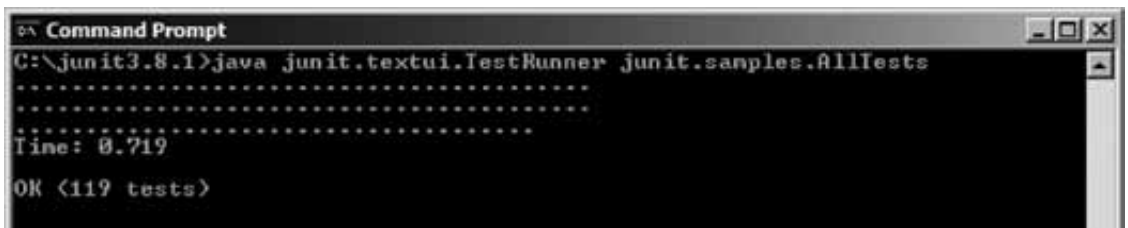
<sup>5</sup> Más adelante, en el punto 2.5 se discutirá cual es la mejor forma de organizar en disco las librerías necesarias para realizar las pruebas.

### 2.2.1. Comprobación de la correcta instalación de JUnit

La distribución de JUnit contiene una serie de ejemplos cuyo objetivo es proporcionar al desarrollador sencillos casos de uso con los que empezar a conocer la herramienta. Sin embargo, tienen otra utilidad, y es que es posible servirse de estos ejemplos para comprobar la correcta instalación de JUnit en la máquina. Para ello el primer paso es abrir una ventana de comandos, ya sea en Windows o Linux (a partir de ahora y siempre que no se indique lo contrario, todos los ejemplos o procedimientos son válidos para Linux o Windows), moverse a la carpeta en la que se haya descomprimido JUnit y escribir lo siguiente:

```
> java junit.textui.TestRunner junit.samples.AllTests
```

Con este comando lo que se consigue es ejecutar los tests de ejemplo utilizando un Runner (una clase que permite ejecutar tests) basado en texto, cuyo nombre es `junit.textui.TestRunner`. En caso de que la instalación se haya realizado correctamente, aparecerá en la ventana de comandos el siguiente mensaje “OK (119 tests)”. En caso contrario se deberá revisar el proceso de instalación en busca de la causa que ha originado el problema.



## 2.3. Primera toma de contacto con JUnit

Con el objetivo de ilustrar adecuadamente mediante ejemplos los contenidos de este libro, se ha desarrollado y probado, mediante JUnit y un buen número de sus extensiones, un sistema software completo. Dicho sistema software se describe con todo detalle en el apéndice B de este libro. Sin embargo, una forma rápida de empezar a conocer JUnit es mediante los ejemplos que acompañan a la distribución. Aunque más adelante se explicará paso por paso y en detalle cómo utilizar JUnit, este apartado tiene como propósito presentar una idea general sobre cómo funciona JUnit y qué es lo que el desarrollador puede esperar de ella.

Desafortunadamente, las distribuciones de JUnit correspondientes a las versiones 4.x no contienen ejemplos hasta la fecha<sup>6</sup>. A continuación, se lista una versión reducida del código de uno de los ejemplos contenidos en la distribución 3.8.1 de JUnit. Dicho ejemplo, pertenece al paquete `junit.samples` y consiste en la prueba de la clase `Vector`. Dicha clase representa un array dinámico de objetos, y es perfectamente conocida por cualquier desarrollador Java.

<sup>6</sup> La versión 4.3 contiene ejemplos, pero están basados en las versiones previas a la versión 4.0 por lo que no muestran cómo utilizar las novedades presentes en las versiones 4.x.

```

package junit.samples;

import junit.framework.*;
import java.util.Vector;

public class VectorTest extends TestCase {

    protected Vector fEmpty;
    protected Vector fFull;

    public static void main (String[] args) {
        junit.textui.TestRunner.run (suite());
    }

    protected void setUp() {
        fEmpty= new Vector();
        fFull= new Vector();
        fFull.addElement(new Integer(1));
        fFull.addElement(new Integer(2));
        fFull.addElement(new Integer(3));
    }

    public static Test suite() {
        return new TestSuite(VectorTest.class);
    }

    public void testCapacity() {
        int size= fFull.size();
        for (int i= 0; i < 100; i++)
            fFull.addElement(new Integer(i));
        assertTrue(fFull.size() == 100+size);
    }

    public void testClone() {
        Vector clone= (Vector)fFull.clone();
        assertTrue(clone.size() == fFull.size());
        assertTrue(clone.contains(new Integer(1)));
    }
}

```

Este ejemplo consiste en una clase llamada `VectorTest` que se encarga de probar la clase `Vector`. Para ello esta clase define dos métodos de prueba: `testCapacity` y `testClone`, que se encargan respectivamente de probar los métodos `capacity` y `clone` de la clase `Vector`. Adicionalmente, el método `setUp` se encarga de crear dos instancias de la clase `vector`. Una de ellas con 3 elementos y otra de ellas vacía, que van a ser utilizadas por los métodos de prueba. Finalmente el método `suite` y la función `main`, hacen uso de los mecanismos que proporciona JUnit para ejecutar los métodos de prueba definidos en esta clase y mostrar los resultados por pantalla. Es decir, si se ha detectado algún fallo durante la prueba o no. Como puede observarse, el “verdadero” código de prueba se encuentra dentro de los métodos `testCapacity` y `testClone`, que, en el ejemplo, se encargan de ejecutar un caso de prueba. El caso de prueba que tiene lugar en `testCapacity` consiste en comprobar si el método `capacity` de la clase `Vector` realmente devuelve el número exacto de elementos contenidos en él.

El proceso se realiza en dos pasos, inicialmente se añaden 100 elementos a una de las instancias creadas en el método `setUp` (método de inicialización) y se invoca el método a probar (`capacity`) para finalmente comparar el valor devuelto con el valor esperado haciendo uso del método `assertTrue` proporcionado por JUnit. Nótese que el método `testClone` realiza un procedimiento análogo, que, en general, y como se verá a lo largo de este capítulo, se resume en tres pasos:

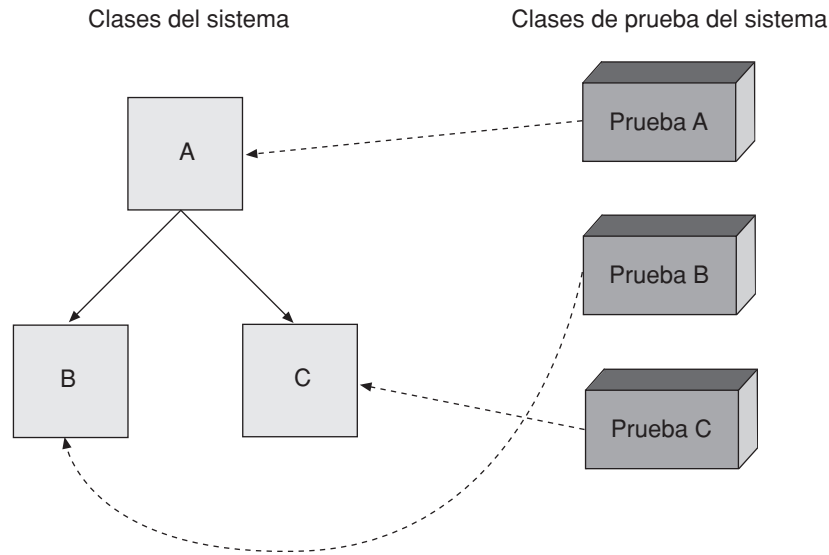
1. Carga de los datos correspondientes al caso de prueba (en el ejemplo, este paso se realiza parcialmente dentro del método `setUp` que JUnit siempre invoca automáticamente antes de invocar cada método de prueba).
2. Ejecución del método de prueba en las condiciones provistas en el punto anterior.
3. Comparación del resultado obtenido con el resultado esperado utilizando la familia de métodos `assert` proporcionada por JUnit.

## 2.4. Creación de una clase de prueba

Una vez puesto a punto el entorno de pruebas, se está en disposición de comenzar a desarrollar el software de pruebas. Antes de nada, conviene dejar clara la diferencia entre dos conceptos que se utilizarán con frecuencia en lo sucesivo: software de producción y software de pruebas. El software de producción no es otro que el software objetivo de las pruebas, es decir, aquel software que será entregado al cliente y al usuario al final de la fase de desarrollo y cuyo correcto funcionamiento se desea verificar. El software de pruebas es el conjunto de elementos software utilizados durante la fase de pruebas, es decir: código de pruebas, herramientas y librerías utilizadas durante la prueba, documentación utilizada y generada, fuentes de datos, etc.

En un lenguaje orientado a objetos, siempre y cuando la fase de diseño se haya llevado a cabo correctamente, es decir, atendiendo al principio de maximización de la cohesión y minimización del acoplamiento, las clases se pueden entender como unidades independientes con una funcionalidad bien delimitada. Todos los elementos definidos dentro de una clase (constructores, métodos, atributos, etc.) están muy ligados entre sí y no dependen del funcionamiento interno o detalles de implementación de otras clases. Por tanto, cuando se habla de pruebas en un lenguaje orientado a objetos, se ha de tener claro que a pesar de que los métodos son las unidades mínimas de ejecución, las clases constituyen el marco fundamental de la prueba. La razón es que al realizar pruebas sobre los métodos de un objeto, es necesario conocer en qué estado se encuentra dicho objeto. O lo que es equivalente, los métodos de un objeto no pueden ser probados (invocados) en cualquier orden, ya que los objetos tienen memoria, es decir, mantienen un estado interno que normalmente reside en sus atributos y que se ve alterado cuando sus métodos son invocados.

Por ejemplo la clase `Log` del sistema presentado en el Apéndice B, se encarga de escribir eventos en un archivo en disco. Una vez esta clase es instanciada el método `inicializar` ha de ser llamado de forma que se cree un archivo vacío. En el momento en que el método `inicializar` se ha ejecutado con éxito, ya es posible utilizar el método `nuevoError` para añadir eventos al archivo creado. Como puede comprobarse, los métodos del objeto deben ser invocados en determinado orden para que todo funcione correctamente. Por tanto la prueba de dichos métodos ha de hacerse conjuntamente dentro del contexto de la prueba de la clase en la que han sido definidos.



**Figura 2.1.** Relaciones entre las clases de un sistema y las correspondientes clases de pruebas unitarias.

En este punto parece claro que la unidad básica, objetivo de la prueba es la clase, pero, ¿qué procedimiento se ha de seguir para probar una clase con JUnit? Cuando se trabaja con JUnit se ha de crear una clase para cada clase que se desee probar. Dicha clase, será llamada en adelante “clase de pruebas” y contendrá todo el código necesario para la ejecución de los diferentes casos de prueba definidos para la clase a probar también conocida como “clase objetivo”. Si el objetivo es probar un sistema software completo, como por ejemplo el descrito en el Apéndice B, se han de crear tantas clases de prueba como clases tenga dicho sistema<sup>7</sup>.

En la Figura 2.1 se muestran las clases correspondientes a un sistema junto con las clases creadas para realizar las pruebas unitarias de ese sistema. El sistema está compuesto por las clases A, B y C, existiendo dos relaciones de asociación indicadas por flechas de línea continua. Puesto que el sistema consta de tres clases, se han definido tres clases (PruebaA, PruebaB y PruebaC) que van a realizar pruebas unitarias sobre ellas. Las flechas de línea discontinua indican la relación entre la clase de pruebas y la clase a probar. Esta relación es una relación de asociación ya que es necesario instanciar la clase a probar dentro de la clase de pruebas. Nótese además que no existe ninguna relación entre las diferentes clases de pruebas unitarias.

A continuación, se va a describir el procedimiento de prueba de una clase mediante JUnit en las dos versiones más populares de la herramienta. La versión 3.8.1, que ha sido la versión más popular hasta fechas recientes, y la versión 4.2, que esta suponiendo actualmente una revolución en el mundo JUnit.

<sup>7</sup> Nótese que estas consideraciones estarán siempre sujetas al alcance de las pruebas definido en el Plan de Pruebas, véase Capítulo 1.

## 2.4.1. Creación de una clase de pruebas con JUnit 3.8.1

Para la creación de una clase de pruebas utilizando esta herramienta se ha de seguir el siguiente procedimiento:

1. Definición de una clase de pruebas de modo que herede de la clase `TestCase`, perteneciente al paquete `junit.framework`. Esto es necesario debido a que la clase `TestCase` contiene una serie de mecanismos que van a ser utilizados por la clase de prueba, como son los métodos `assert`. Nótese que la clase `TestCase` hereda de la clase `Assert` y, por tanto, todos los métodos `assert` (`assertNull`, `assertEquals`, `AssertTrue`, ...) van a estar disponibles dentro de la clase de pruebas. Por motivos de legibilidad, es importante seguir ciertas convenciones de nomenclatura cuando se trabaja con JUnit. En este caso el nombre de una clase de prueba debe estar formado por el nombre de la clase a probar seguido del sufijo `Test`. Por ejemplo, la clase de prueba de una clase llamada `Log`, se llamara `LogTest`. Es importante no olvidar importar el paquete `junit.framework` o bien importar aisladamente las clases que se vayan a utilizar de este paquete.
2. Definición de los métodos para la inicialización y liberación de los recursos a utilizar durante las pruebas de la clase: JUnit permite al desarrollador definir un método que se encargue de inicializar los recursos que se utilizarán durante la ejecución de un método de prueba, así como otro método para la liberación de dichos recursos una vez que la ejecución del método de prueba termina. Dichos métodos se llaman `setUp` y `tearDown` respectivamente y se han de definir en la clase de prueba correspondiente. Estos métodos están definidos en la clase `TestCase`, y dado que la clase de prueba hereda de `TestCase`, al definirlos lo que ocurre realmente es que se están sobrescribiendo. La gran utilidad de estos métodos es que son invocados automáticamente por JUnit antes y después de la ejecución de cada uno de los métodos de prueba, por lo que son ideales para la definición y liberación de recursos. Sin embargo, existe un inconveniente, en ocasiones los recursos a inicializar dependen de los casos de prueba asociados definidos para el método de prueba a ejecutar, por lo que la inicialización y la liberación no es común a todos ellos. En estos casos específicos los métodos `setUp` y `tearDown` no resultan de gran utilidad.
3. Definición de los métodos de prueba: para cada uno de los métodos y constructores que se deseen probar (típicamente al menos aquellos pertenecientes a la interfaz pública de la clase objetivo) se ha de definir un método de prueba correspondiente. Estos métodos se van a encargar de ejecutar uno a uno todos los casos de prueba asociados al método a probar. Es obligatorio que el nombre de estos métodos se construya con el prefijo `test` seguido del nombre del método a probar comenzando con mayúscula. Por ejemplo, el método definido para la prueba de un método llamado `procesarPeticiónHTTP` se llamará `testProcesarPeticiónHTTP`. Mientras que el prefijo `test` es obligatorio ya que permite a JUnit descubrir en tiempo de ejecución cuáles son los métodos de prueba; el nombre del método a probar se añade por motivos de legibilidad. De esta forma al ver el nombre del método de prueba, inmediatamente se conoce cual es el método que prueba. A continuación, se describe la secuencia de tareas que los métodos de prueba han de llevar a cabo:
  - a. Obtener los datos asociados a un caso de prueba<sup>8</sup>. Estos datos varían de unos métodos a otros pero a grandes rasgos<sup>9</sup> serán los parámetros de entrada con los que se

<sup>8</sup> Recuérdese que para cada método de prueba se definen típicamente uno o más casos de prueba, para más información véase Capítulo 1.

<sup>9</sup> Más adelante se hará una descripción en detalle de todos los aspectos que se han de tener en cuenta en la prueba.

invocará al método y los valores esperados de retorno así como las condiciones a comprobar sobre los mismos <sup>10</sup>.

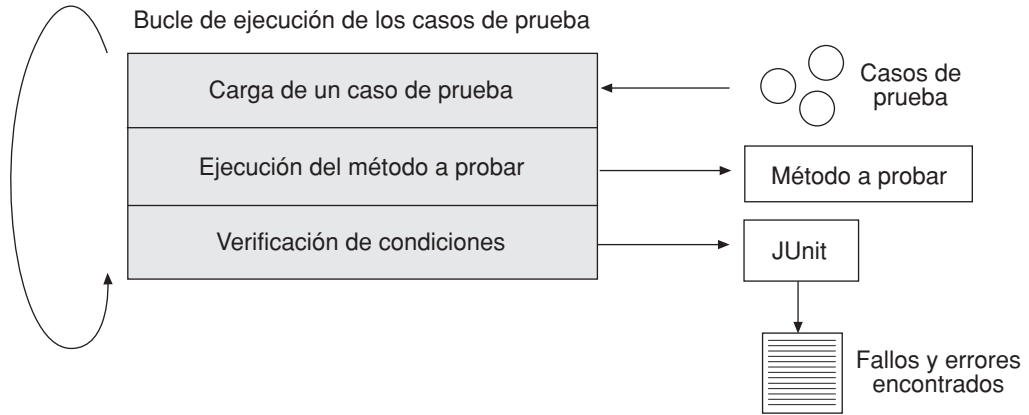
- b. Invocar el método a probar con los datos del caso de prueba. Para ello es necesario crear una instancia de la clase a probar. Estas instancias u objetos se crearán dentro del método `setUp` y se almacenarán en variables de objeto en caso de que vayan a ser reutilizadas desde diferentes métodos de prueba. En caso contrario, lo ideal es crearlas *ad-hoc* para el caso de prueba.
- c. Comprobar ciertas condiciones sobre los valores devueltos por el método invocado así como sobre su código de retorno y sobre el estado de otras entidades externas que se puedan ver afectadas por la ejecución del método, tal y como venga descrito en el caso de prueba. El resultado de dichas comprobaciones determinarán si el caso de prueba ha detectado un fallo o por el contrario todo ha ido como se esperaba. Nótese que las condiciones a verificar no han de restringirse a los parámetros de salida de un método o a su valor de retorno, de hecho, estos pueden no existir. La verificación de las condiciones deberá realizarse sobre todas las salidas que produzca el método, lo que incluye, por ejemplo, creación o escritura en ficheros, aperturas de conexiones de red, acceso a bases de datos, etc. <sup>11</sup>. JUnit, a través de la clase `junit.framework.Assert`, proporciona una serie de métodos, comúnmente llamados métodos `assert`, que facilitan la realización de tales comprobaciones. La clase `Assert` es la clase base de `TestCase` por lo que los métodos definidos en la clase de pruebas, dado que esta hereda de `TestCase`, pueden utilizar los métodos `assert` directamente. Estos métodos básicamente reciben uno o dos objetos (o tipos básicos) como parámetros de entrada y comprueban una condición sobre ellos, como por ejemplo si son iguales. En caso de que la condición no se cumpla, el método `assert` comunicará un fallo utilizando la clase `AssertionFailedError`, cuya clase base es `Assert`.
- d. En este punto la ejecución del caso de prueba ha terminado por lo que retornará al punto a. hasta que se hayan ejecutado todos los casos de prueba asociados al método.

La clase `java.lang.Error`, cuya clase base es `Throwable`, pertenece a la API de Java. Típicamente los errores en Java (no confundir la clase `Error` en Java con el concepto de error en JUnit) se utilizan para comunicar a la máquina virtual de Java (JVM) que un error no recuperable ha ocurrido. Puesto que estos errores son de naturaleza irrecuperable, no deben ser capturados mediante un bloque `trycatch` como ocurre con las excepciones `java` (clase `java.lang.Exception`). Cuando uno ocurre, la aplicación típicamente ha de terminar. Sin embargo JUnit utiliza la clase `AssertionFailedError` (que descende de la clase `java.lang.Error`) como si de una excepción `Java` se tratase. Los objetos de esta clase que son lanzados cuando un método `assert` falla al verificar una condición son capturados por el framework de JUnit interrumpiendo la ejecución del método de prueba. Sin embargo la ejecución de los siguientes métodos de prueba prosigue.

<sup>10</sup> Existen varias posibilidades a la hora de cargar los datos de los casos de prueba, la más sencilla e inmediata es incluir los datos dentro del propio código. Sin embargo, como se verá en el Capítulo 8 existen mejores alternativas desde el punto de vista de la mantenibilidad.

<sup>11</sup> La información detallada acerca de los efectos producidos por la ejecución de un método se puede encontrar en el Apartado 2.5 Conceptos básicos.





**Figura 2.2.** Esquema de ejecución de un método de prueba.

En la Figura 2.2 se muestra el esquema general de funcionamiento interno de un método de prueba. Básicamente consiste en un bucle de ejecución de los casos definidos para el método a probar. En cada iteración se cargan los datos asociados a un caso de prueba (esta carga puede ser obteniendo los datos desde el exterior, por ejemplo mediante el uso de archivos de definición de casos de prueba, o bien definiendo los datos en el propio caso de prueba), se ejecuta el método a probar con dichos datos y se verifican las condiciones asociadas al caso de prueba mediante JUnit (mediante los métodos `assert`). De esta forma JUnit es capaz de obtener el resultado de las verificaciones y generar información que es mostrada al final de la ejecución al desarrollador. El último paso es revisar dicha información y corregir los defectos software encontrados.

A continuación, se incluye un ejemplo de clase de prueba utilizando la versión 3.8.1 de JUnit. Se trata de la clase de prueba<sup>12</sup> de la clase `Registro` perteneciente al sistema descrito en el Apéndice B. En particular esta clase realiza la prueba de los métodos `comprobarFormato` y `obtenerLongitud` pertenecientes a la clase `Registro`.

`pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java`

```

package pruebasSistemaSoftware.junit381;

import junit.framework.*;
import servidorEstadoTrafico.Registro;
import servidorEstadoTrafico.RegistroMalFormadoException;
import servidorEstadoTrafico.Tramo;

public class RegistroTest extends TestCase {

    private Registro m_registro;

    public void setUp() {
  
```

<sup>12</sup> Por motivos de espacio se trata de una versión reducida de la clase original `RegistroTest`.

```

        String strCarretera = "M-40";
        String strHora = "12:23:45";
        String strFecha = "1/3/2007";
        String strClima = "Nublado";
        String strObras = "No";

        m_registro = new Registro(strCarretera, strHora, strFecha,
            strClima, strObras);
        Tramo tramo1 = new Tramo("0", "10", "3", "1", "Retenciones",
            "Sin accidentes");
        Tramo tramo2 = new Tramo("10", "12", "2", "0", "Retenciones",
            "Sin accidentes");
        Tramo tramo3 = new Tramo("12", "15", "3", "1", "Retenciones",
            "Sin accidentes");
        m_registro.anadirTramo(tramo1);
        m_registro.anadirTramo(tramo2);
        m_registro.anadirTramo(tramo3);
    }

    public void tearDown() {

    }

    public void testComprobarFormato() {

        try {

            m_registro.comprobarFormato();
        }
        catch (RegistroMalFormadoException e) {

            fail("Se ha originado una excepcion inesperada" +
                e.toString());
        }
    }

    public void testObtenerLongitud() {

        assertEquals(m_registro.obtenerLongitud(), 10+2+3);
    }
}

```

Como puede observarse, se define un método `setUp` en el que se crea una instancia de la clase `Registro` que contiene tres objetos de la clase `Tramo`, es decir, representa una carretera con tres tramos. Esta clase `Registro` será posteriormente utilizada en los métodos de prueba `testComprobarFormato` y `testObtenerLongitud`. El método `tearDown` no tiene cuerpo pues no hay recursos que liberar. El método `testComprobarFormato` simplemente ejecuta el método a probar y comprueba que la excepción `RegistroMalFormadoException` no es lanzada, que equivale a que el registro está bien formado y es en última instancia consistente con los datos del caso de prueba con los que el objeto `Registro` fue construido. Por último, el método `testObtenerLongitud` simplemente comprueba que el método `obtenerLongitud` devuelve un valor equivalente a la suma de las longitudes de los objetos `tramo` con los que ha sido creado.

## 2.4.2. Creación de una clase de pruebas con JUnit 4.x

La principal novedad que introduce JUnit 4.x es el uso de anotaciones. Las anotaciones son un mecanismo proporcionado por Java a partir de la versión 5.0 de la JDK que permite asociar información o metadatos a diferentes elementos del código fuente, como son métodos, atributos, constantes, etc. JUnit define y pone a disposición del desarrollador un conjunto de etiquetas que permiten asociar información a los métodos que componen una clase de prueba y a la propia clase de prueba. En tiempo de ejecución, JUnit es capaz de interpretar esta información y con ella llevar a cabo la ejecución de la prueba en los términos que el desarrollador dispone. Por este motivo, a la hora de utilizar JUnit 4.x para la prueba de una clase, la sintaxis cambia ligeramente de forma y el proceso se simplifica. Inicialmente se va a describir punto por punto los pasos necesarios para realizar la prueba de una clase con esta versión de la herramienta. Finalmente se hará un repaso de las principales novedades respecto a anteriores versiones y se sacaran conclusiones al respecto.

Antes de continuar con la lectura de este apartado, y para aquellos no familiarizados con las novedades que incorpora la versión 5.0 de Java, que son principalmente el import estático y las anotaciones, se recomienda realizar, al menos, una lectura rápida del Apéndice F. Dichos conceptos son fundamentales para comprender las aportaciones de esta versión sobre versiones precedentes.

A continuación, se describe el procedimiento de creación de una clase de prueba. Como se puede observar es en esencia bastante similar al descrito para la versión 3.8.1.

1. Definición de una clase de pruebas cuyo nombre debe estar formado por el nombre de la clase a probar seguido del sufijo `Test`. En este caso la clase de pruebas no debe heredar de la clase `TestCase`. Ante esta novedad surge la siguiente pregunta: ¿al no heredar de `TestCase` los métodos `assert` dejan de estar disponibles para ser usados en los métodos de prueba? La respuesta es no, los métodos `assert` siguen estando disponibles, pero esta vez gracias al import estático. La forma de hacer uso de ellos es importarlos estáticamente. Por ejemplo, si se pretende usar el método `assertEquals`, se ha de importar añadiendo la siguiente línea de código: `import static org.junit.Assert.assertEquals`.
2. Definición de los métodos para la inicialización y liberación de los recursos a utilizar durante las pruebas de la clase: en este caso el hecho de que la clase de prueba no herede de `TestCase`, hace imposible redefinir los métodos `setUp` y `tearDown` que tradicionalmente se utilizaban para inicializar y liberar recursos para la prueba. Ahora los métodos de inicialización y liberación pueden tener cualquier nombre siempre y cuando se declaren con la etiqueta `@Before` y `@After` respectivamente. No obstante y a pesar de que no existe una restricción para el nombre de estos métodos, se recomienda utilizar nombres que faciliten la legibilidad. Estos nombres pueden ser `setUp` y `tearDown`, para aquellos acostumbrados a la nomenclatura tradicional, o bien `inicializar` y `liberar`, o cualquier nombre en esa línea. Una característica interesante de esta nueva versión de JUnit es la posibilidad de definir métodos para la inicialización y liberación de recursos al inicio y final de la ejecución de la clase de prueba. Es decir, un método que es llamado antes de ejecutar los métodos de prueba y debe ser definido con la etiqueta `@BeforeClass` y otro método que se llama justo después de que el último método de prueba haya sido ejecutado y debe estar definido con la etiqueta `@AfterClass`. Nótese que mientras que para una clase de prueba dada solo un méto-

do puede estar anotado con la etiqueta `@BeforeClass` o `@AfterClass`, puede haber múltiples métodos anotados con las etiqueta `@Before` o `@After`.

3. Definición de los métodos de prueba: para cada uno de los métodos y constructores que se vayan a probar se ha de definir un método de prueba en la clase. Dicho método ahora no necesita comenzar con el prefijo `test` ya que Java ya no utiliza el mecanismo de Reflection para descubrir los métodos de prueba en tiempo de ejecución. En su lugar, cada método de prueba ha de definirse con la etiqueta `@Test` de forma que JUnit pueda encontrarlo utilizando el mecanismo de anotaciones. Típicamente los nombres de métodos de prueba se llamarán de forma idéntica al método a probar, para así ser fácilmente reconocibles. Además, existen dos nuevas características interesantes en los métodos de prueba que JUnit se encarga de gestionar y que se listan a continuación:
  - a. La anotación `@Test` puede recibir un parámetro llamado `timeout` indicando el tiempo máximo en milisegundos que debe esperar JUnit hasta que la ejecución del método de prueba termine. En caso de que la ejecución del método de prueba se prolongue mas allá del tiempo límite, JUnit considerará que un fallo ha ocurrido y dará por terminada la ejecución del método de prueba. Este mecanismo es especialmente útil a la hora de probar métodos que pueden incluir operaciones bloqueantes o bucles potencialmente infinitos. También es útil en situaciones en las que la obtención de los datos asociados al caso de prueba requiere de operaciones que pueden bloquear o demorar en exceso la prueba, como por ejemplo acceder a una base de datos que está caída. Como norma general este parámetro es de gran utilidad en las pruebas funcionales ya que dentro de las cuales se incluyen las pruebas de rendimiento.
  - b. La anotación `@Test` puede recibir un parámetro llamado `expected` que indica que una excepción se espera que se origine al invocar el método de prueba. Se trata de un mecanismo que trata de facilitar la prueba de excepciones esperadas<sup>13</sup>.

A continuación se muestra un ejemplo de clase de prueba utilizando la versión 4.2 de JUnit. Se trata de la clase de prueba de la clase Registro por lo que este ejemplo es equivalente al expuesto anteriormente para la versión 3.8.1 de JUnit. De esta forma es posible ver más claramente las diferencias entre ambas versiones.

pruebas sistema software/src/pruebasSistemaSoftware/junit42/RegistroTest.java

```
package pruebasSistemaSoftware.junit42;

import java.lang.*;
import java.util.*;

import org.junit.Test;
import org.junit.After;
import org.junit.Before;
```

<sup>13</sup> Para obtener información detallada acerca de la prueba de excepciones esperadas y una discusión en detalle de las ventajas e inconvenientes de la solución propuesta en JUnit 4.x, consultar el Apartado 2.8 Conceptos avanzados en la prueba de clases Java.

```

import static org.junit.Assert.*;
import junit.framework.JUnit4TestAdapter;

import servidorEstadoTrafico.Registro;
import servidorEstadoTrafico.RegistroMalFormadoException;
import servidorEstadoTrafico.Tramo;

public class RegistroTest {

    private Registro m_registro;

    @Before public void inicializar() {

        String strCarretera = "M-40";
        String strHora = "12:23:61";
        String strFecha = "1/3/2007";
        String strClima = "Nublado";
        String strObras = "No";

        m_registro = new Registro(strCarretera, strHora, strFecha, strClima,
            strObras);
        Tramo tramo1 = new Tramo("0", "10", "3", "1", "Retenciones",
            "Sin accidentes");
        Tramo tramo2 = new Tramo("10", "12", "2", "0", "Retenciones",
            "Sin accidentes");
        Tramo tramo3 = new Tramo("12", "15", "3", "1", "Retenciones",
            "Sin accidentes");
        m_registro.anadirTramo(tramo1);
        m_registro.anadirTramo(tramo2);
        m_registro.anadirTramo(tramo3);
    }

    @After public void liberar() {

    }

    @Test(expected=RegistroMalFormadoException.class) public void comprobarFormato() throws RegistroMalFormadoException {

        m_registro.comprobarFormato();
    }

    @Test(timeout=1000) public void obtenerLongitud() {

        assertEquals(m_registro.obtenerLongitud(), 10+2+3);
    }
}

```

Como puede observarse se ha definido un método de inicialización de recursos, inicializar, anotado con la etiqueta @Before. De igual forma, se han utilizado anotaciones en la definición de los métodos de prueba comprobarFormato y obtenerLongitud, que esta vez reciben el mismo nombre del método que prueban. En la definición del método comprobarFormato se ha añadido el parámetro expected con el valor RegistroMalFormadoException.class,

lo que significa que JUnit debe reportar un fallo en caso de que dicha excepción no se produzca. Es muy importante declarar el método con la sentencia `throws RegistroMalFormadoException` en lugar de utilizar una sentencia `try-catch` para capturar la excepción, de otra forma JUnit sería incapaz de observar la excepción cuando esta se produzca. Adicionalmente nótese que los datos con los que se ha definido el caso de prueba han sido elegidos apropiadamente para que dicha excepción salte, ya que el valor del segundo de captura es 61 y el mayor valor válido es obviamente 59<sup>14</sup>. Finalmente, cabe destacar que se ha añadido a la anotación `@Test` del método `obtenerLongitud`, el parámetro `timeout`<sup>15</sup> con un valor de 2000, lo que significa que la ejecución de dicho método de prueba no puede prolongarse más allá de 2 segundos o JUnit abortará la prueba del método y reportará un fallo.

A continuación, se listan las principales novedades de esta versión de JUnit respecto a versiones precedentes.

- Es necesario utilizar la versión 5 de la JDK para ejecutar los tests.
- Las clases de prueba no necesitan heredar de la clase `junit.framework.TestCase`.
- Los métodos de inicialización y liberación pueden ser definidos con cualquier nombre siempre y cuando sean etiquetados adecuadamente con `@Before` y `@After`. Además puede existir más de un método de inicialización y liberación simultáneamente.
- Los nombres de los métodos de prueba no necesitan contener el prefijo `test`, sin embargo es necesario que sean definidos con la etiqueta `@Test`, la cual permite ser utilizada con parámetros que enriquecen las posibilidades de la prueba.
- Existe la posibilidad de declarar métodos de inicialización y liberación globales a la clase de pruebas mediante las etiquetas `@BeforeClass` y `@AfterClass`.

Como se puede observar, las diferencias entre estas dos versiones de JUnit son notables. JUnit 4.x no solo dota de una mayor simplicidad, flexibilidad y extensibilidad a JUnit gracias al uso de anotaciones, sino que corrige aquellos defectos de versiones preliminares a los que el desarrollador casi se había acostumbrado. La única desventaja, por citar alguna, es el uso del `import` estático que tan duramente ha sido criticado desde su inclusión en el lenguaje Java debido a los problemas de legibilidad que causa. El principal problema es que a simple vista no se puede saber la clase a la que estos métodos pertenecen.

Una vez que se ha descrito el procedimiento de creación de una clase de prueba desde un punto de vista global, en el siguiente apartado de este libro, titulado Conceptos Básicos, se describirá como realizar la prueba del código contenido en dichas clases, tanto dentro de métodos como de constructores.

## 2.5. Conceptos básicos

A continuación, se van a tratar cuestiones básicas en la prueba de código Java mediante JUnit como son la prueba de métodos y constructores. Inicialmente, se va a describir cómo realizar la

---

<sup>14</sup> El código fuente correspondiente a la clase `Registro` no se ha incluido en este capítulo por motivos de espacio, sin embargo dicho código está disponible en la ruta `Sistemasoftware/src/servidorEstadoTrafico/Registro.java` dentro del CD que acompaña a este libro.

<sup>15</sup> En este método de prueba en particular el parámetro `timeout` no es realmente necesario y simplemente se ha añadido como ejemplo de uso.

prueba de un constructor así como de métodos `get` y `set`<sup>16</sup>, ya que son casos particulares que presentan una menor complejidad. Finalmente, se discutirá en detalle cómo realizar la prueba de un método Java convencional y qué factores se han de tener en cuenta para llevar este procedimiento a buen fin.

## 2.5.1. Prueba de constructores

Los constructores en cualquier lenguaje orientado a objetos en general y en Java en particular, tienen como objetivo permitir la instanciación de clases, es decir, crear objetos. Cada vez que el desarrollador utiliza la palabra reservada `new`, el constructor de la clase correspondiente es invocado. Típicamente los constructores son utilizados para inicializar las propiedades de un objeto que acaba de ser creado.

En caso de que ningún constructor haya sido definido por el desarrollador, el compilador de Java crea automáticamente un constructor llamado constructor por defecto. Este tipo de constructor carece de argumentos y se encarga de inicializar los atributos del objeto: a `null` las referencias a objetos, a `false` los atributos de tipo `boolean` y a 0 los atributos de tipo numérico. Los constructores definidos por defecto no han de incluirse dentro de los objetivos de la prueba, ya que no pueden fallar. Sin embargo, comúnmente el desarrollador define constructores que reciben ciertos parámetros y que permiten inicializar ciertas propiedades del objeto convenientemente. En el cuerpo de dichos constructores, el desarrollador escribe una serie de líneas de código responsables de la inicialización, dichas líneas son obviamente susceptibles de contener errores y por tanto han de ser probadas.

Los constructores presentan una serie de características especiales respecto a los métodos que deben ser tenidas en cuenta durante la prueba:

- Carecen de valor de retorno. Existe una restricción en el lenguaje Java según la cual la palabra reservada `return` no puede aparecer dentro del cuerpo de un constructor. Por lo tanto, estos no devuelven ningún valor al finalizar su ejecución. Sin embargo, los constructores a menudo utilizan las excepciones para indicar que ha habido un problema en la inicialización del objeto. Estas excepciones han de ser probadas mediante la técnica de prueba de excepciones esperadas<sup>17</sup>.
- A la hora de probarlos existe un problema que es inherente a su naturaleza, y es que comprobar que se han ejecutado correctamente no es fácilmente observable. Esta característica quedará expuesta en detalle a continuación.

### 2.5.1.1. Procedimiento de prueba de un constructor

Probar un constructor, según el procedimiento de prueba de métodos convencionales en JUnit, consistiría en invocar dicho constructor y comparar el objeto obtenido con el objeto esperado, que no es otro que un objeto correctamente inicializado tal y como se supone que el constructor debería producir. El problema es que, obviamente, la única forma de obtener el objeto esperado es utilizando el constructor de la clase que se está probando, por lo que este procedimiento no tiene sentido. Es como si, por poner un ejemplo, a la hora de definir una palabra se utilizase esa misma palabra dentro de la definición.

<sup>16</sup> Los métodos `get` y `set` son típicamente utilizados en la programación orientada a objetos para acceder a las propiedades de un objeto sin romper el principio de encapsulación.

<sup>17</sup> Esta técnica se describe en profundidad en el Apartado 2.8 Conceptos avanzados en la prueba de clases Java.

Es necesario, por tanto, encontrar una forma de probar un constructor sin realizar comparaciones sobre el objeto creado. Esa forma no es otra que preguntar al objeto creado acerca de sus propiedades a través de su interfaz pública y comprobar que han sido inicializadas correctamente. En este contexto aparecen una serie de incógnitas: ¿qué propiedades se van a ver modificadas durante la inicialización llevada a cabo en el constructor? ¿es posible acceder al valor de estas propiedades para comprobar su correcta inicialización? La respuesta a ambas incógnitas depende en gran medida de la técnica de pruebas utilizada. Mientras que en el caso de la técnica de pruebas de caja blanca (véase Capítulo 1) conocer dichas propiedades es automático, puesto que se dispone del código fuente, en el caso de la caja negra no lo es. El motivo es que al desconocerse los detalles de implementación del constructor, establecer una asociación entre un valor que se le pasa al constructor y un valor de una variable de objeto de su interfaz pública puede ser cuando menos, muy arriesgado.

A continuación, se recogen recomendaciones para ambos casos:

#### 2.5.1.1.1. PRUEBA DE UN CONSTRUCTOR MEDIANTE LA TÉCNICA DE CAJA BLANCA

En este caso se conocen con exactitud las propiedades que se ven afectadas durante la inicialización del objeto, por tanto basta con invocar al constructor con una serie de valores y comparar el valor de tales propiedades con dichos valores. Los métodos `get` (véase el siguiente punto) a menudo son grandes aliados en la prueba de constructores ya que permiten fácilmente acceder al valor de las propiedades del objeto. De esta forma es posible realizar la prueba de los métodos `get` a la vez que la prueba del constructor, solventándose ambos problemas de una vez.

En el ejemplo que se lista a continuación puede observarse como la prueba de un constructor se resume en tres pasos:

1. Carga de los datos correspondientes al caso de prueba. Nótese que en el ejemplo sólo un caso de prueba ha sido definido. Aunque en la práctica puede ser necesario definir más de un caso de prueba para la prueba de un constructor, típicamente no son necesarios muchos casos diferentes. Esto es debido a que un constructor normalmente sólo realiza asignaciones de variables, por lo que un único caso de prueba muchas veces es más que suficiente para comprobar que dichas asignaciones se han realizado correctamente y que no se ha omitido ninguna, etc.
2. Invocación del constructor con lo que se crea una instancia de la clase sobre la que posteriormente realizar las comparaciones.
3. Realización de las comparaciones sobre las propiedades del objeto instanciado utilizando para ello los métodos `get`. Básicamente para cada propiedad del objeto cuyo valor ha de ser verificado, se utiliza el método `assertEquals` que compara su valor con el valor con el que se invocó al constructor.

```
pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java
```

```
public void testRegistro() {

    //Carga de los datos correspondientes al caso de prueba
    String strCarretera = "M-40";
    String strHora = "12:23";
```



```

String strFecha = "1-3-2006";
String strClima = "Nublado";
String strObras = "No";

//Instanciacion de las clases necesarias
Registro registro = new Registro(strCarretera,strHora,strFecha,
    strClima,strObras);

//Invocacion de los metodos de prueba y verificacion de las
    condiciones
assertEquals(registro.obtenerCarretera(),strCarretera);
assertEquals(registro.obtenerHora(),strHora);
assertEquals(registro.obtenerFecha(),strFecha);
assertEquals(registro.obtenerClima(),strClima);
assertEquals(registro.obtenerObras(),strObras);

return;
}

```

Este procedimiento es muy sencillo y cómodo, sin embargo existe un inconveniente. En ocasiones, a pesar de que las propiedades del objeto que se ven modificadas en el constructor son perfectamente conocidas ya que el código fuente está disponible, pueden no ser accesibles desde el exterior del objeto. El mecanismo de encapsulacion proporcionado por Java permite al desarrollador ocultar el valor de ciertas propiedades simplemente cambiando su atributo de privacidad y de esta forma ocultar detalles de implementación del objeto. Típicamente los objetos hacen públicas ciertas propiedades a través de los métodos *get* (véase el siguiente punto) o directamente declarándolas públicas, sin embargo, otras propiedades simplemente no pueden ser accesibles desde el exterior mientras que su valor es inicializado dentro del constructor. A pesar de que existen mecanismos para salvar este obstáculo<sup>18</sup> basados en sortear el mecanismo de encapsulacion de la máquina virtual de Java, no se recomienda su uso para la prueba de constructores. Esto es debido al elevado coste en tiempo que conlleva en comparación con el escaso beneficio que se puede obtener atendiendo a la escasa complejidad y por tanto baja probabilidad de error de la mayoría de los constructores que se definen.

#### 2.5.1.1.2. PRUEBA DE UN CONSTRUCTOR MEDIANTE LA TÉCNICA DE CAJA NEGRA

En el caso de la caja negra, dado que el código fuente no está disponible, es necesario suponer qué propiedades del objeto se van a ver modificadas al invocar el constructor. Normalmente, estas suposiciones pueden ser acertadas si se ha respetado un estándar de nomenclatura durante la fase de diseño del objeto, pero dado que los detalles de implementación del constructor no son visibles, siempre existe un riesgo y en ocasiones dichas suposiciones pueden conducir a falsas alarmas. Una falsa alarma debe entenderse como una “detección” de un error que no es tal. Ha de tenerse presente que el retorno de un método *get* no representa necesariamente el valor de una propiedad o variable sino que el valor devuelto puede haber sido calculado en tiempo de ejecución al ser invocado. La forma en la que la información pasada al constructor en forma de parámetros se almacena en el objeto no es transparente para el desarrollador y forma parte de los detalles de implementación del objeto, por tanto, no debe ser objeto de suposiciones. Por este motivo se desaconseja realizar pruebas sobre constructores bajo la técnica de pruebas de caja ne-

<sup>18</sup> Véase Apartado 2.8 Conceptos avanzados en la prueba de clases Java.

gra salvo en un caso excepcional, que es aquel en el que el constructor pueda lanzar excepciones. En este caso las excepciones sí deberán ser probadas. Esto se realizará mediante la técnica de prueba de excepciones esperadas descrita en el Apartado 2.8 Conceptos avanzados en la prueba de clases Java.

## 2.5.2. Prueba de métodos get y set

Los métodos `get` y `set`<sup>19</sup> son comúnmente utilizados en la programación orientada a objetos para obtener y asignar el valor de las propiedades de un objeto respectivamente. La primera vez que se descubre la existencia de este tipo de métodos surge la siguiente pregunta: ¿por qué no realizar la asignación o lectura de las propiedades del objeto directamente sobre las variables, reduciendo de esta forma el tiempo de acceso y el tamaño del código fuente a escribir? La respuesta es sencilla, los métodos `get` y `set` permiten ocultar los detalles de implementación del objeto, en particular, permiten ocultar la forma en la que la información se almacena en el interior del objeto. Es decir, constituyen un mecanismo de encapsulación de las propiedades internas del objeto.

Supóngase que un objeto es modificado de forma que el tipo de dato de una de sus propiedades cambia de `String` a `int`, para, por ejemplo, ahorrar espacio de almacenamiento. En este caso todo objeto que acceda a dicha propiedad directamente dejará de compilar, con los problemas de mantenibilidad que eso conlleva. En el caso en el que los objetos accedan a esta propiedad a través de métodos `get` y `set` en lugar de hacerlo directamente, se evitará tener que modificar el código de dichos objetos que accedan a esta propiedad simplemente cambiando la implementación del método `get` y `set` de forma que realicen la conversión de tipos al acceder a la propiedad del objeto. Por tanto, los métodos `get` y `set` tienen como misión facilitar la encapsulación, que es uno de los pilares de la programación orientada a objetos.

Sin embargo, a pesar de sus ventajas, existen detractores de los métodos `get` y `set` ya que en el fondo no solucionan todos los problemas de encapsulación derivados del uso de variables públicas. El motivo es que realmente exponen de alguna manera los detalles de implementación de un objeto y, por tanto, su uso debe ser minimizado al máximo. Por ejemplo, si cambia el tipo de dato de una propiedad de `int` a `double` porque la nueva implementación del objeto proporciona mayor precisión para dicha propiedad, si se realiza la conversión inversa de tipo de dato en los métodos `get` y `set`, aunque todo compilara perfectamente, la ventaja de la nueva implementación quedará sin efecto. La cuestión es ¿existe realmente una buena solución para este problema? ¿es realmente posible solventar este tipo de soluciones garantizando una alta mantenibilidad? La respuesta en casi todas las situaciones es sí, y como se verá a continuación, no es un problema de codificación sino de diseño.

El argumento principal que sostiene la postura en contra de los métodos `get` y `set` consiste en que se debe limitar al máximo la información intercambiada entre los objetos, porque solo así podrá garantizarse la mantenibilidad del código. Este objetivo solo se puede conseguir a través de un diseño realmente modular, que va en contra de la mala costumbre de diseño que tiene por norma hacer privadas todas las variables de un objeto y exponerlas al exterior a través de métodos `get` y `set` públicos.

---

<sup>19</sup> En este libro se ha escogido la nomenclatura `get` y `set` por consistencia con la literatura en lengua inglesa dedicada a este campo. Prefijos alternativos para este tipo de métodos son, por ejemplo, `obtener` y `modificar`.

Esta discusión sobre la conveniencia de usar métodos get y set, a primera vista puede parecer no estar muy relacionada con lo que son las pruebas en sí, pero realmente sí lo está. A menudo los diseñadores tienden a exponer las propiedades de los objetos a través de métodos get y set mas allá de lo que sería necesario, según el diseño realizado. Esto es debido a dos motivos fundamentalmente:

- **Mantenibilidad preventiva:** nunca se sabe a ciencia cierta si determinada propiedad de un objeto va a ser de utilidad en el futuro para las clases que hagan uso de este objeto a pesar de que en el momento de la implementación del objeto no lo sea.
- **Facilitar la fase de pruebas:** puesto que las pruebas normalmente se basan en realizar ciertas operaciones sobre los objetos (invocar sus métodos y constructores acorde a los casos de prueba y observar el comportamiento de ciertas propiedades, códigos de retorno, etc.), facilitar el acceso a las propiedades del objeto a menudo facilita enormemente el proceso de la prueba.

Sin embargo, en general, no se ha de caer en el error de romper la encapsulación de un objeto con la consiguiente pérdida de mantenibilidad para facilitar las pruebas. Esto no quiere decir que no se haya de pensar en las pruebas durante la codificación, que obviamente sería un error. En efecto, las pruebas se han de tener siempre en mente ya que el objetivo de las mismas es el objetivo de todo proyecto software, es decir, garantizar el cumplimiento de todos y cada uno de los requisitos software<sup>20</sup>.

El procedimiento general para la prueba de una pareja de métodos get y set asociados a una propiedad es muy sencillo y se describe a continuación.

1. Se instancia un objeto de la clase.
2. Se almacena un valor en la propiedad con el método set de acuerdo al caso de prueba.
3. Se obtiene el valor de la propiedad con el método get.
4. Se comprueba que ambos valores, el obtenido y el almacenado, son idénticos para lo cual se utiliza un método assert, típicamente el método `assertEquals`.

A continuación, se muestra un ejemplo en el que se puede ver dicho procedimiento. Se trata de la prueba de los métodos get y set de la clase `Registro`, perteneciente al sistema software presentado en el Apéndice B.

pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java

```
public void testGetSet() {

    String strCarretera = "M-40";
    String strHora = "12:23";
    String strFecha = "1-3-2006";
    String strClima = "Nublado";
    String strObras = "No";
```

<sup>20</sup> De hecho existe una fuerte corriente llamada "Desarrollo guiado por las pruebas" (Test Driven Development) que surge como uno de los pilares de la programación extrema (Extreme Programming) a principios del año 2000, en la que todo gira alrededor de las pruebas.

```

//Se crea una instancia con valores diferentes a los del caso de prueba
Registro registro = new Registro("", "", "", "", "");

//Se modifican las propiedades del objeto mediante los metodos set
registro.modificarCarretera(strCarretera);
registro.modificarHora(strHora);
registro.modificarFecha(strFecha);
registro.modificarClima(strClima);
registro.modificarObras(strObras);

//Se comprueba que las propiedades tienen los valores esperados
mediante los metodos get
assertEquals(registro.obtenerCarretera(), strCarretera);
assertEquals(registro.obtenerHora(), strHora);
assertEquals(registro.obtenerFecha(), strFecha);
assertEquals(registro.obtenerClima(), strClima);
assertEquals(registro.obtenerObras(), strObras);

return;
}

```

Como puede observarse, en este caso la prueba de todos los métodos get y set de la clase se realiza simultáneamente.

Nótese que normalmente un único caso de prueba es suficiente para probar exhaustivamente este tipo de métodos. Sin embargo, esto no siempre es correcto y va a depender de la implementación de los mismos. Además, a la hora de probar este tipo de métodos se han de tener en cuenta ciertas particularidades dependiendo de la técnica empleada.

### 2.5.2.1. Prueba de métodos get y set mediante la técnica de caja blanca

Esta situación se da cuando el código fuente está disponible a la hora de planificar la prueba. Se pueden dar los siguientes casos:

- Los métodos get y set no acceden directamente al valor de la propiedad sino que existe algún tipo de traducción o procedimiento intermedio en el acceso a la información contenida en el objeto. En este caso, la prueba tiene un objetivo claro, probar el correcto funcionamiento de dicho procedimiento intermedio. Simplemente, se seguirá el procedimiento de prueba general descrito anteriormente.
- Los métodos get y set acceden directamente al valor de la propiedad, es decir, el parámetro que se le pasa al método set será siempre exactamente el nuevo valor de la propiedad del mismo modo que el valor devuelto por el método get refleja el valor real de la propiedad. En este caso parece un tanto absurdo realizar una prueba del correcto funcionamiento de la pareja de métodos get y set asociados a una propiedad. Este tipo de pruebas innecesarias son comúnmente llamadas “probar la plataforma” ya que en realidad solo pueden fallar si la plataforma (el compilador o la máquina virtual de Java) falla. A pesar de todo, existe una razón para no descartar la prueba de estos métodos. Esta razón no es otra que la mantenibilidad preventiva, es decir, si la implementación de los métodos get y set evoluciona a una mayor complejidad, el software de pruebas estaría preparado para ello. En cualquier caso, debe ser el programador, basándose en su experiencia personal, el que decida cuándo y cuándo no vale la pena realizar este tipo de pruebas.

### 2.5.2.2. Prueba de métodos get y set mediante la técnica de caja negra

En este caso se realizará la prueba de los métodos get y set según el procedimiento general anteriormente descrito y siempre que el desarrollador lo considere suficientemente útil.

### 2.5.3. Prueba de métodos convencionales

Normalmente, el grueso del código de una aplicación y por tanto el grueso de las pruebas se concentra en los métodos llamados “convencionales”, es decir, aquellos que no son get ni set. Los métodos get y set, así como los constructores, presentan siempre una estructura muy similar entendida como el papel que desempeñan dentro de una clase así como la sintaxis que habitualmente se emplea para escribirlos. Por este motivo, como ya se ha visto, resulta relativamente sencillo establecer una serie de pasos que, sin pérdida de generalidad, permitan guiar al desarrollador para realizar una prueba efectiva.

Desafortunadamente, para los llamados métodos convencionales, estas circunstancias no se dan. Por lo que el proceso de pruebas reviste mayor complejidad y son más los detalles que normalmente se han de tener en cuenta.

Un problema derivado de la falta de experiencia en el mundo de las pruebas de software es la tendencia a pensar en los métodos como entidades que reciben parámetros y devuelven valores de retorno en función de esos parámetros de forma invariable. Según esta simplificación del concepto de método, las pruebas han de limitarse a la mera invocación de métodos con determinados parámetros y a la comparación del valor de retorno de dichos métodos con el valor esperado. Todo ello tal y como haya sido descrito en los casos de prueba diseñados. Nada más lejos de la realidad, los métodos son habitualmente entidades muy complejas y los resultados que producen dependen de muchos factores más allá de los parámetros que reciben. A continuación, se discutirá esto en detalle.

En todo proceso de pruebas la unidad básica de prueba, aun estando enmarcado en la prueba de una clase, es el método. Obviamente, las pruebas de software van mucho más allá de la prueba de métodos aisladamente, sin embargo esto es la base de todo. De modo que si el objetivo es verificar el correcto comportamiento de un método, la única forma de llevar a cabo esta tarea es atendiendo a los efectos visibles y “no visibles” que dicho método provoca al ser invocado. A continuación, se van a plantear una serie de preguntas y respuestas que ayudarán a clarificar todas estas cuestiones.

#### A. ¿CUÁLES SON LOS FACTORES QUE CONDICIONAN LA EJECUCIÓN DE UN MÉTODO Y POR TANTO DETERMINAN LOS RESULTADOS QUE ESTE MÉTODO PRODUCE?

La respuesta a esta cuestión se lista a continuación:

- Parámetros de entrada: obviamente el valor de los parámetros de entrada con los que se invoca un método van a influir en la ejecución de dicho método.
- Estado interno del objeto: debe entenderse como el valor que presentan las propiedades del objeto en el momento en que el método es invocado. Es muy común encontrar objetos que utilizan ciertas propiedades, típicamente variables de objeto, para almacenar información relativa a su estado interno. Esta información puede servir al objeto, por ejemplo, para conocer qué llamadas de métodos han sido realizadas sobre él y de esta forma conocer qué

otras llamadas pueden ser efectuadas y cuáles no. Por ejemplo, supóngase el caso de un objeto que representa un reproductor de audio, en este caso el objeto puede realizar una serie de tareas como son iniciar la reproducción, pausarla o detenerla. Sin embargo, suponiendo que para cada una de estas operaciones existiera un método asociado, dichos métodos no podrían ser invocados en cualquier orden. No tiene sentido invocar el método que pausa la reproducción de audio si no hay reproducción en curso. Por tanto, este objeto debe hacer uso de variables (típicamente privadas) que almacenan su estado proveyéndole de “memoria”.

- Estado de los objetos externos: es habitual que un método interactúe con otros objetos, comúnmente llamados objetos colaboradores<sup>21</sup>, para, por ejemplo, obtener cierta información o realizar una cierta tarea en un momento dado de su ejecución. El estado interno de dichos objetos obviamente va a influir en el resultado de la ejecución del método en cuestión. Nótese que en caso de que el estado de dichos objetos esté completamente determinado por el estado del objeto donde el método está definido, se estaría hablando del punto anterior.
- Entidades externas que constituyen el entorno de ejecución: el entorno de ejecución software y hardware en el que un método es ejecutado, influye de lleno en los resultados obtenidos y por tanto ha de considerarse en la prueba. Este entorno no es ni más ni menos que el estado interno y las propiedades de aquellas entidades que lo componen. Este aspecto, a menudo considerado como marginal, juega en ocasiones un papel fundamental. Dos buenos ejemplos en los que el entorno juega un papel decisivo es en métodos que interactúan con dispositivos externos como el sistema de archivos o bien con una base de datos. Obviamente, si determinado archivo no está presente en la ruta adecuada o si la base de datos contiene ciertos registros en unas tablas u otras, el resultado de la ejecución de un método va a ser completamente diferente<sup>22</sup>. A primera vista considerar este elemento como un ingrediente más a tener en cuenta durante el proceso de prueba de un método, puede resultar desalentador debido al potencialmente elevadísimo coste en tiempo que supone. No obstante, será la experiencia del desarrollador así como el alcance de las pruebas establecido en el plan de pruebas, lo que determine qué elementos es necesario considerar y cuáles no. En cualquier caso a lo largo de este libro se verán numerosos ejemplos en este sentido que permitirán a los más neófitos adquirir un criterio al respecto con el que sobrevivir con éxito en la primera toma de contacto.

## **B. ¿CUÁLES SON LOS EFECTOS QUE SE PRODUCEN COMO CONSECUENCIA DE INVOCAR UN MÉTODO?**

- Valor de retorno del método: habitualmente los métodos devuelven un valor con la información requerida. Un ejemplo inmejorable son los métodos `get` que devuelven el valor de una propiedad del objeto.
- Efectos producidos sobre los parámetros: a menudo cuando se invoca un método, este realiza una serie de cambios sobre los parámetros con los que ha sido invocado. Un ejemplo sería el caso en el que un método recibe un `Vector` (clase `java.util.Vector`) de objetos y los ordena mediante un algoritmo de ordenación dado. Dicho méto-

---

<sup>21</sup> El Capítulo 7 Mock Objects está dedicado a describir una técnica de pruebas que permite probar métodos que hacen uso de objetos colaboradores en el contexto de relaciones de agregación entre objetos.

<sup>22</sup> El Capítulo 9 está enteramente dedicado a la prueba de métodos que interaccionan con una base de datos.

do está por tanto modificando los parámetros de entrada, y una forma de comprobar su correcto funcionamiento sería verificar la correcta ordenación de los objetos del `Vector`.

- Excepciones lanzadas por el método: Java proporciona un mecanismo de excepciones que es utilizado por los métodos para comunicar al método llamante situaciones anómalas que se producen en tiempo de ejecución <sup>23</sup>.
- Cambio del estado interno del objeto: a menudo los objetos mantienen información de su estado en variables privadas. La ejecución de un método puede alterar el valor de dichas variables y por tanto el estado interno del objeto. ¿Por qué es importante tener en cuenta el cambio del estado interno de un objeto? Obviamente porque el estado interno del objeto va a determinar el resultado de pruebas sucesivas.
- Efectos externos al objeto: modificaciones sobre el estado interno o las propiedades de entidades que compongan el estado de ejecución software y hardware del método en cuestión.
- Una vez repasada esta lista resulta asombrosa la cantidad de efectos que puede producir la ejecución de un método y por tanto las situaciones que se han de tener en cuenta durante su prueba. Sin embargo, esta lista no es más que una recopilación de todos los posibles efectos. A menudo, los métodos a probar solo son capaces de producir uno o dos de los efectos recogidos en ella, por lo que las pruebas resultan bastante sencillas.

Una vez planteadas estas dos cuestiones y sus correspondientes respuestas, el siguiente paso es ver la forma en que se ha de utilizar toda esta información durante las pruebas. La respuesta a la cuestión A es básicamente una enumeración de los factores que se han de tener en cuenta a la hora de diseñar los casos de prueba para un método. Si el objetivo de los casos de prueba es proporcionar una adecuada cobertura para la prueba de un método, dicha cobertura se alcanzará atendiendo a las posibles variaciones de los elementos que determinan los resultados de la ejecución del método. Por otro lado, en todo caso de pruebas se ha de hacer una descripción del comportamiento esperado del método acorde al contexto en el que es invocado. La respuesta a la cuestión B no es otra que un desglose de los elementos que constituyen dicho comportamiento. Determinar si la ejecución de un caso de prueba es satisfactoria o no consistirá, por tanto, en examinar estos elementos.

### 2.5.3.1. Casos particulares:

En este apartado se van dar algunas recomendaciones sobre ciertos casos particulares con los que el desarrollador se puede encontrar.

#### 2.5.3.1.1. PRUEBA DE UN MÉTODO QUE NO TIENE PARÁMETROS DE ENTRADA

En este caso la “dificultad” reside en la definición del caso de prueba. Puesto que no hay parámetros de entrada los casos de prueba definidos deberán centrarse en aquellos otros factores que condicionan la ejecución del método. Para ello nada mejor que revisar la respuesta a la cuestión A, planteada en el Apartado 2.5.3.

---

<sup>23</sup> En el Apartado 2.8 Conceptos avanzados en la prueba de clases Java, se incluye una completa descripción sobre como manejar este tipo de circunstancias en las pruebas.

### 2.5.3.1.2. PRUEBA DE UN MÉTODO QUE NO TIENE VALOR DE RETORNO

Que un método no tenga valor de retorno es equivalente a decir que su tipo de retorno es `void`. En este caso la prueba del método simplemente deberá enfocarse sobre aquellos otros efectos observables de su ejecución que si están presentes. Para ello nada mejor que revisar la respuesta a la cuestión B, planteada en el Apartado 2.5.3.

## 2.6. Organización de las clases de prueba

Durante el proceso de creación del conjunto de clases de prueba asociadas al sistema software que se está probando, es necesario organizar dichas clases de alguna forma. Existen diferentes consideraciones que se han de tener en cuenta a este respecto. En primer lugar, es importante mantener separado el código de producción del código de pruebas, de esta forma se facilita la gestión de configuraciones y la mantenibilidad. Por otra parte, conviene que todo el código de pruebas esté agrupado de forma que sea más fácil realizar una ejecución conjunta de todas las pruebas. Por este motivo, normalmente, si el código del sistema software a probar está organizado en un paquete<sup>24</sup>, se suele crear otro paquete en paralelo con las clases de prueba correspondientes. Es importante establecer esta separación entre el código de producción y el código de pruebas desde las primeras fases de la escritura de este último.

Por ejemplo, en el sistema software (de nombre SET) descrito en el Apéndice B, sobre el cual se realizan pruebas a lo largo de este libro, se puede comprobar que todas sus clases pertenecen a un paquete llamado `servidorEstadoTrafico` que se corresponde con el archivo `servidorEstadoTrafico.jar`. En paralelo, el conjunto de clases de prueba encargadas de probar dicho sistema se encuentran contenidas en otro paquete llamado `pruebasSistemaSoftware` que a su vez está contenido en el archivo `servidorEstadoTraficoTest.jar`.

El procedimiento para la creación de un paquete de clases en Java es muy sencillo, simplemente se ha de declarar la sentencia `package` seguida del nombre del paquete en cada uno de los archivos correspondientes a las clases que se pretende pertenezcan al paquete. Por ejemplo, en el sistema SET cada una de las clases incluyen la siguiente sentencia: `package servidorEstadoTrafico;`, por lo que todas ellas pertenecerán al paquete `servidorEstadoTrafico`.

Una vez que se han desarrollado y compilado las clases correspondientes a un paquete, tiene sentido agrupar los archivos `.class` correspondientes en un archivo `.jar`. Estos archivos permiten compactar y comprimir las clases de un paquete de forma que estas son más fácilmente distribuibles y desplegables. La forma de crear un archivo `.jar` es muy sencilla y se puede hacer desde la línea de comandos por medio del comando `jar` incluido en la JDK o bien por medio de un archivo `Ant`<sup>25</sup>. A continuación se describe el proceso de creación de un archivo `.jar` desde la ventana de comandos:

1. Moverse al directorio donde se encuentren los archivos `.class` que se pretende formen parte del archivo `.jar`.

<sup>24</sup> En Java, el término paquete se corresponde con `package` que representa un conjunto de clases vinculadas entre sí.

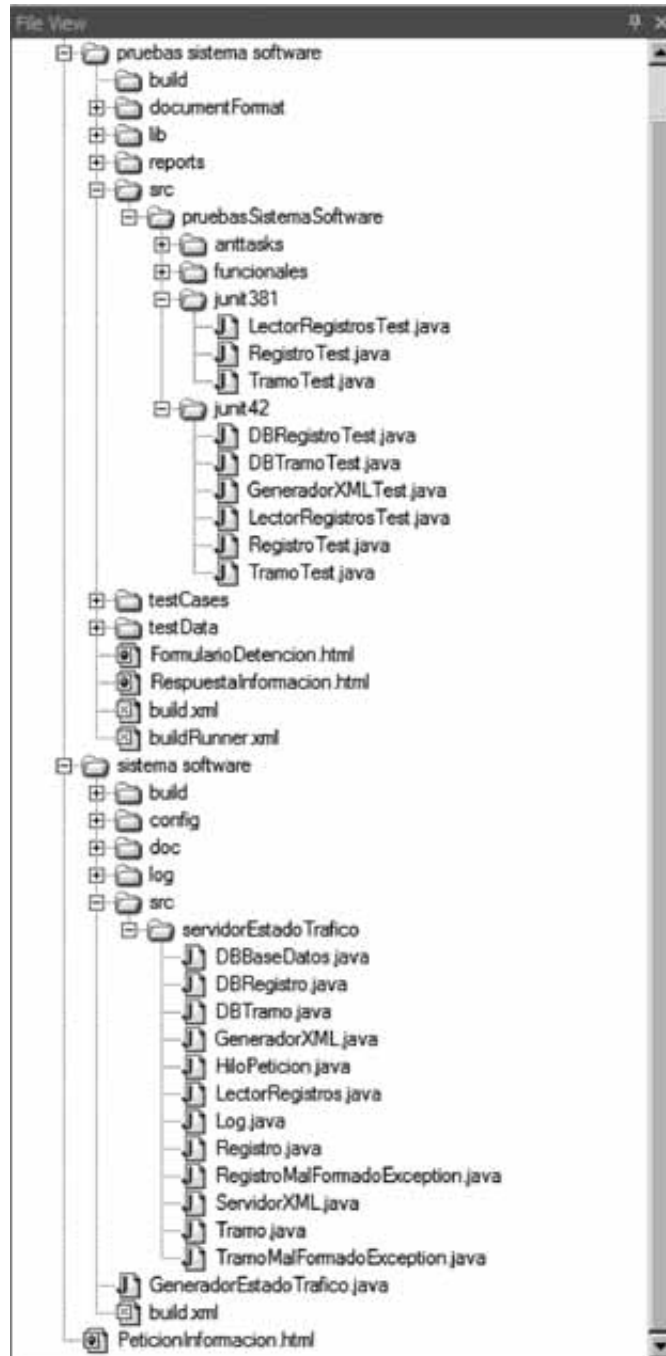
<sup>25</sup> En el Capítulo 3 de este libro, se presentan las ventajas de la utilización de archivos `Ant` para el despliegue del software Java así como las facilidades que aporta a la hora de manejar el software de pruebas.



2. Escribir el siguiente comando:

```
jar cf nombre-del-archivo-jar *.class
```

En la Figura 2.3 se muestra el árbol de directorios y ficheros correspondiente al código del sistema descrito en el Apéndice B y al código de pruebas encargado de realizar las pruebas de dicho



**Figura 2.3.** Estructura de directorios del código de pruebas y del código del sistema a probar.

sistema. El directorio “pruebas sistema software” contiene todo el software de pruebas. Los directorios a destacar son “junit381” y “junit42” que contienen el código fuente de las clases encargadas de realizar pruebas unitarias mediante las versiones 3.8.1 y 4.2 de Junit respectivamente. El otro directorio desplegado es “servidorEstadoTrafico” que contiene el sistema a probar. Nótese que no existen clases de pruebas unitarias para todas las clases del sistema, esto es debido a que no ha sido necesario escribirlas para cubrir los objetivos de este libro. En este caso la creación de los archivos .jar a partir de las clases generadas en la compilación y almacenadas en la carpeta “build”, se realiza mediante los respectivos documentos Ant de nombre build.xml.

## 2.7. Ejecución de los casos de prueba

Una vez que el código de las pruebas ha sido desarrollado incluyendo todos los casos de prueba relativos a cada método, el siguiente paso es ejecutar el código de pruebas y analizar los resultados. En la práctica, y aunque en este libro se haya descrito de forma lineal, la creación de código de pruebas y su ejecución, se suele hacer de forma paralela y escalonada. Normalmente, se escribe una clase de pruebas, se ejecuta y se observan los resultados obtenidos. Si ha habido algún fallo, se corrige y se vuelve a ejecutar la clase de pruebas hasta que no queden más fallos. En ese momento se pasa a la siguiente clase, así sucesivamente hasta completar la prueba de todas las clases del sistema. De hecho una buena costumbre de trabajo colaborativo es no subir nunca al repositorio una clase sin su correspondiente clase de pruebas. Y lo que es más, no se debe subir una clase cuya correspondiente clase de pruebas detecte algún error que aún no ha sido corregido.

JUnit tiene incorporados ciertos mecanismos que facilitan la ejecución de las clases de prueba. A pesar de que en el Capítulo 3 de este libro se presentará una forma más profesional de llevar a cabo la ejecución de las pruebas por medio de la tarea `<junit>` en el contexto de un documento Ant, los mecanismos proporcionados por JUnit son de gran utilidad cuando el código de las pruebas es de un volumen reducido. Este apartado va a estar dedicado a describir cómo utilizar dichos mecanismos presentes en JUnit para ejecutar el código de pruebas y visualizar los resultados obtenidos, es decir, si los casos de prueba definidos se han ejecutado con éxito o si se han detectado fallos. Finalmente, se discutirá la forma correcta de interpretar los resultados obtenidos.

### 2.7.1. Mecanismos de ejecución de los casos de prueba

JUnit incorpora diferentes mecanismos para la ejecución del código de pruebas. Las diferencias entre ellos radica básicamente en el modo de presentar la información resultante de la ejecución: en modo texto o en modo gráfico. La disponibilidad de estos mecanismos de ejecución depende de la versión de JUnit que se esté utilizando. JUnit en su versión 3.8.1 provee tres diferentes mecanismos para la ejecución de las pruebas, uno en modo texto y dos en modo gráfico. Sin embargo, los mecanismos de ejecución en modo gráfico han sido eliminados a partir de la versión 4.0. Si bien es cierto que dicha funcionalidad está disponible mediante el uso de extensiones<sup>26</sup>.

---

<sup>26</sup> JUnitExt es un proyecto de código libre con licencia CPL 1.0 alojado en <http://www.junitext.org/>. Se trata de un conjunto de extensiones de la funcionalidad básica de las versiones 4.x de JUnit. Entre la funcionalidad que añade a JUnit destaca la definición de nuevos tipos de anotaciones así como mecanismos de ejecución (test runners) en modo gráfico equivalentes a los disponibles para las versiones anteriores de JUnit.

### 2.7.1.1. Ejecución en modo texto

El primer paso para utilizar un ejecutor de pruebas (también conocido como `TestRunner`, ya que este es el nombre que reciben las clases que realizan la ejecución de las pruebas) en modo texto es declarar una función `main` en la clase de pruebas. Dentro de esta función se creará un objeto de la clase `TestSuite`, perteneciente al paquete `junit.framework`, que se utiliza para agrupar métodos de prueba relacionados. En este caso, pasándole al constructor de `TestSuite` la clase de pruebas automáticamente todos los métodos de prueba definidos en esa clase pasan a formar parte del objeto `TestSuite` creado. El objeto `TestSuite` es capaz de descubrir dinámicamente dichos métodos basándose en que el nombre de todos ellos incorpora el prefijo `test`. El último paso consiste en invocar el método `run` del objeto `TestRunner` perteneciente al paquete `junit.textui` pasándole como parámetro el objeto `TestSuite` anteriormente creado. Dicho método `run` se encarga de tomar los métodos de prueba contenidos en el `TestSuite` y de ejecutarlos, mostrando los resultados en modo texto en la ventana de comandos.

A continuación se muestra con un ejemplo la ejecución de los casos de prueba correspondientes a la clase `RegistroTest`, perteneciente al sistema software presentado en el Apéndice B.

La función `main` se define en la clase `RegistroTest` tal y como se acaba de comentar:

```
pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java
```

```
public static void main(String args[]) throws Exception {
    TestSuite testSuite = new TestSuite(RegistroTest.class);
    junit.textui.TestRunner.run(testSuite);
    return;
}
```

Para llevar a cabo la ejecución en modo texto basta con ejecutar el siguiente comando<sup>27</sup>:

```
> java pruebasSistemaSoftware.junit381.RegistroTest
```

El resultado que aparece en la ventana de comandos es el siguiente:

```
...
Time: 0.016

OK (3 tests)
```

El mensaje indica, por un lado, el tiempo de ejecución de los tests (en este caso 16 milisegundos) y el número de tests ejecutados (en este caso 3 ya que la clase `RegistroTest` solamente consta de tres métodos de prueba). La palabra `OK` indica que todo ha ido correctamente, es decir, no se han detectado fallos ni errores. Los tres puntos de la primera línea se corresponden con cada uno de los tres métodos de prueba existentes, y los escribe el `TestRunner` a modo de barra de progreso en formato texto a medida que los tests se van completando.

<sup>27</sup> Nótese que para la exitosa ejecución de este comando las correspondientes librerías deben haber sido añadidas a la variable de entorno `CLASSPATH` previamente.

En caso de producirse algún fallo el resultado obtenido sería, por ejemplo, el siguiente:

```
...F
Time: 0.015
There was 1 failure:
1)
testObtenerLongitud(pruebasSistemaSoftware.junit381.RegistroTest) junit.
framework.AssertionFailedError: expected:<15> but was:<16>
    at pruebasSistemaSoftware.junit381.RegistroTest.testObtener
        Longitud(Unknown Source)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at pruebasSistemaSoftware.junit381.RegistroTest.main(Unknown Source)

FAILURES!!!
Tests run: 3, Failures: 1, Errors: 0
```

Las diferencias son: la F que aparece en la primera línea indicando que se ha encontrado un fallo en la ejecución de un método de prueba y el mensaje descriptivo del fallo. Este mensaje indica el método de prueba en el que se ha producido el fallo (en este caso el método `testObtenerLongitud` perteneciente a la clase `RegistroTest` contenida en el paquete `pruebasSistemaSoftware`) junto con el fallo que se ha producido, es decir, los parámetros que ha recibido el método `assert` que ha fallado (en este caso indica que se esperaba un valor 16 cuando el valor obtenido ha sido 15). Nótese que cuando un método de prueba detecta un fallo, la ejecución de dicho método se detiene, sin embargo la ejecución del resto de métodos de prueba continúa.

### 2.7.1.2. Ejecución en modo gráfico

Para la ejecución en modo gráfico, existen dos posibilidades, utilizar un ejecutor de pruebas basado en ventanas de tipo AWT o bien utilizar uno basado en ventanas de tipo Swing<sup>28</sup>. Para cada uno de ellos existe un ejecutor de pruebas o test runner asociado. La forma de utilizarlos es muy sencilla y se presenta a continuación.

#### 2.7.1.2.1. AWT

En este caso el test runner a utilizar es la clase `TestRunner` perteneciente al paquete `junit.awtui`. Dicha clase `TestRunner` posee un método `run` que recibe la clase de prueba y se encarga de su ejecución mostrando los resultados en una ventana al estilo AWT. Nótese que en este caso no es necesario utilizar un objeto de la clase `TestSuite`.

A continuación, se muestra un ejemplo equivalente al utilizado para la ejecución en modo texto. La función `main` definida tiene el siguiente aspecto:

```
pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java
```

```
public static void main(String args[]) throws Exception {
    junit.awtui.TestRunner.run(RegistroTest.class);
}
```

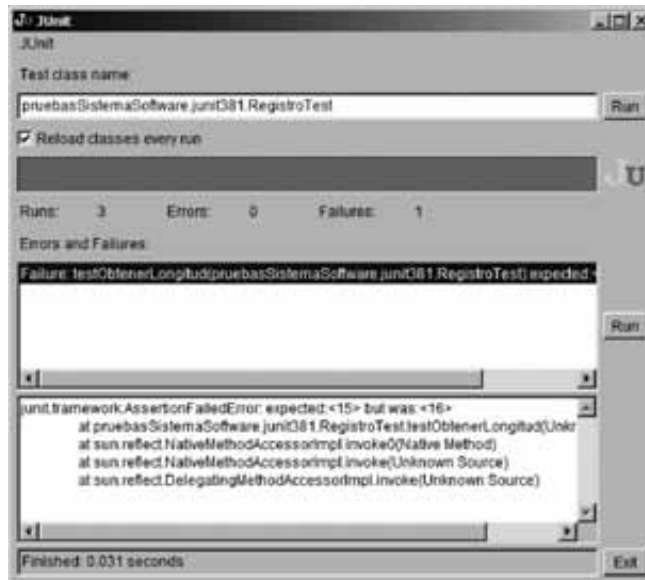
<sup>28</sup> AWT y Swing son dos conjuntos de herramientas de desarrollo de interfaces de usuario en Java. Ambos constan de un conjunto de controles gráficos predefinidos fácilmente extensibles y adaptables. Swing es actualmente mucho más utilizado ya que es más flexible y ofrece muchas más posibilidades y controles más especializados que AWT.

```

    return;
}

```

Una vez ejecutada aparecerá una ventana como la de la Figura 2.4 o Figura 2.5, mostrando la información resultante. Como se puede observar, la información contenida en dicha ventana es equivalente a la presentada en modo texto anteriormente con una salvedad, existe una barra de



**Figura 2.4.** Ventana tipo AWT que muestra información sobre el resultado de la ejecución de la clase de pruebas `RegistroTest` (se ha producido un fallo).



**Figura 2.5.** Ventana tipo AWT que muestra información sobre el resultado de la ejecución de la clase de pruebas `RegistroTest` (no se han producido fallos ni errores).

color rojo o verde que indica si ha habido fallos o no respectivamente en la ejecución de las pruebas<sup>29</sup>. Una vez que el código está libre de errores y fallos, la barra aparecerá de color verde.

### 2.7.1.2.2. SWING

Utilizar este ejecutor de pruebas es completamente equivalente al anterior por lo que simplemente se ha de utilizar la clase `TestRunner` perteneciente al paquete `junit.swingui` en lugar del paquete `junit.awtui`. Sin embargo, y como puede observarse en la Figura 2.6, la ventana Swing presenta mas posibilidades que la ventana AWT, por lo que se recomienda su uso. Por un lado se muestra la jerarquía de pruebas, es decir, las clases de prueba que se han ejecutado con sus métodos correspondientes indicando cuáles de ellos han fallado. Por otro lado, desde esta ventana es posible seleccionar un archivo `.class` desde el sistema de archivos y ejecutar sus métodos de prueba asociados pulsando el botón etiquetado con la palabra “Run”.



**Figura 2.6.** Ventana tipo Swing con información sobre el resultado de la ejecución de la clase de pruebas `RegistroTest` (se ha producido un fallo).

<sup>29</sup> Una frase muy conocida que aparece en el sitio Web de JUnit ([www.junit.org](http://www.junit.org)) dice así “*keep the bar green to keep the code clean...*”, cuya traducción al castellano es “*mantén la barra verde para mantener el código libre de defectos*”. Dicha frase se refiere justo a la barra que aparece en esta ventana.

## 2.7.2. Interpretación de los resultados obtenidos

La ejecución de los casos de prueba permite conocer qué casos de prueba han concluido con éxito y cuales han fallado. Aquellos casos de prueba que hayan fallado indican al desarrollador qué partes del código de producción, y bajo qué circunstancias, presentan defectos que han de ser corregidos. En cualquier caso lo primero que se ha de tener claro a la hora de interpretar correctamente los resultados de la ejecución de las pruebas, es la diferencia que JUnit establece entre errores y fallos.

### 2.7.2.1. Concepto de error en JUnit

Los errores son eventos que ocurren durante la ejecución de un método de prueba y que el desarrollador no ha previsto, es decir, no ha tenido en cuenta la posibilidad de que ocurrieran y por tanto no ha definido un método `assert` que los anticipe. Los errores pueden tener diferentes orígenes:

- Se trata de excepciones que ocurren en el código de producción al ser ejecutado durante el proceso de prueba y que no son capturadas dentro del método de prueba. Este tipo de excepciones (por ejemplo `ArrayIndexOutOfBoundsException`) van a pertenecer siempre al grupo de excepciones no comprobadas ya que el compilador de Java no obliga al desarrollador a capturarlas mediante un bloque `try-catch`. Normalmente estas excepciones denotan un defecto en el código de producción. Nótese que estas excepciones son siempre de naturaleza no comprobada ya que el compilador de Java obliga al sistema software de producción a gestionar internamente las excepciones de tipo comprobado.
- Se trata de excepciones que ocurren en el código de la prueba y que no han sido capturadas. Este tipo de excepciones están originadas por defectos en el código de pruebas y no en el código de producción.

Es conveniente recordar la diferencia entre los dos tipos de excepciones en Java, es decir, excepciones comprobadas y no comprobadas:

- Excepciones no comprobadas: se trata de un tipo de excepciones que heredan de la clase `RuntimeException` y que el desarrollador no está obligado a tratar. Es decir, el compilador no obliga a definir un bloque `try-catch` para su captura ni tampoco una sentencia `throws` en la declaración del método que las lanza. Esto es debido a que son excepciones que se producen comúnmente en situaciones irreversibles y ocasionadas por un defecto en el código de producción. Por ejemplo, si se produce una excepción del tipo `NullPointerException` porque en el interior de un método público no se ha verificado que todos los parámetros de entrada están inicializados, dicha excepción indica un defecto en el método.
- Excepciones comprobadas: a diferencia de las anteriores descienden directamente de la clase `Exception` y el desarrollador está obligado a tratarlas bien definiendo un bloque `try-catch` o bien incluyendo en la declaración del método que las lanza una sentencia `throws`. Se trata de excepciones que se producen en situaciones típicamente recuperables por lo que suele interesar capturarlas para realizar operaciones de contingencia. Por ejemplo en caso de que falle una operación de lectura sobre un fichero, típicamente se produce la excepción comprobada `IOException`, cuya captura puede servir para buscar otra fuente de información de la que leer los datos o bien advertir al usuario del problema y buscar otro camino.

A modo de ejemplo, las siguientes líneas de código dentro de un método de prueba provocan que se produzca una excepción del tipo `java.lang.NullPointerException` lo que hace que JUnit reporte un error (siempre y cuando la excepción no sea capturada).

```
Vector v = null;
assertEquals(v.capacity(),0);
```

El error es reportado por JUnit de la siguiente forma:

```
.E
Time: 0
There was 1 error:
1) testPrueba(RegistroTest)java.lang.NullPointerException
    at pruebasSistemaSoftware.junit381.RegistroTest.testPrueba
        (Unknown Source)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod
        AccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delegating
        MethodAccessorImpl.java:25)
    at pruebasSistemaSoftware.junit381.RegistroTest.main(Unknown
        Source)
```

```
FAILURES!!!
Tests run: 1, Failures: 0, Errors: 1
```

### 2.7.2.2. Concepto de fallo en JUnit

Los fallos en JUnit son eventos que se producen cuando un método de tipo `assert` utilizado dentro de un método de prueba falla al verificar una condición sobre uno o más objetos o tipos básicos. Es decir, indican que un caso de prueba ha fallado. Cuando un fallo es detectado por JUnit, el desarrollador inmediatamente sabe que el código de producción presenta un defecto que ha de ser corregido. Los fallos, a diferencia de los errores, son eventos a los que el desarrollador se ha anticipado previendo su posibilidad de ocurrencia.

Por ejemplo, la siguiente línea de código produciría un fallo ya que obviamente la sentencia `v.capacity` devuelve 1 en lugar de 2.

```
Vector<String> v = new Vector<String>();
v.addElement(new String("elemento 1"));
assertEquals(v.capacity(),2);
```

El fallo será reportado por JUnit de la siguiente forma:

```
.F
Time: 0
There was 1 failure:
1) testPrueba(pruebasSistemaSoftware.junit381.RegistroTest)junit.fra-
mework.AssertionFailedError: expected:<10> but was:<2>
    at pruebasSistemaSoftware.junit381.RegistroTest.testPrueba(Unknown
        Source)
```



```

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethod
    AccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Delegating
    MethodAccessorImpl.java:25)
at pruebasSistemaSoftware.junit381.RegistroTest.main(Unknown
    Source)

```

FAILURES!!!

Tests run: 1, Failures: 1, Errors: 0

Curiosamente, con la aparición de la versión 4.0 de JUnit, la diferencia entre errores y fallos ha desaparecido, es decir, ambos eventos son reportados de la misma forma. El motivo exacto no está claro, pero parece que se debe a lo siguiente: la diferencia entre errores y fallos no es trivial, de hecho, la existencia de esta distinción siempre ha sido una importante causa de confusión y controversia entre los desarrolladores que utilizan esta herramienta. Prueba de ello es la cantidad de foros en Internet en la que aparecen dudas y aclaraciones acerca de la forma adecuada de tratar cada uno de estos eventos.

El problema de fondo es que tanto errores como fallos pueden indicar defectos en el software de producción mientras que los defectos en el código de pruebas son siempre reportados en forma de errores. Por este motivo, conocer el número de errores y fallos que se han producido tras la ejecución de los casos de prueba, no permite hacerse una idea de dónde está el origen de los defectos software. Es decir, no es un indicador fiel del número de defectos presentes en el código de producción ni tampoco del número de defectos presentes en el código de pruebas.

## 2.8. Conceptos avanzados en la prueba de clases Java

Este apartado se va a dedicar a cuestiones relacionadas con la prueba de aspectos inherentes al lenguaje de programación Java en particular y a cualquier lenguaje de programación orientado a objetos en general. Se trata de aspectos como la prueba de excepciones y prueba de métodos no pertenecientes a la interfaz pública de la clase, cuyo procedimiento de prueba no es ni mucho menos obvio y merece una discusión aparte. Antes de comenzar con este capítulo el lector debe estar completamente familiarizado con el uso de JUnit y debe conocer perfectamente el procedimiento de creación de una clase de pruebas comentado en apartados anteriores.

Nótese que el título de este apartado es bastante genérico, y aunque el apartado esté contenido dentro del capítulo de JUnit, este apartado no está directamente relacionado con JUnit. El motivo de incluirlo dentro de este capítulo es porque las técnicas que aquí se van a discutir permiten completar la prueba de una clase Java, cubriendo aquellos problemas no resueltos cuando en anteriores apartados se habló de la utilización de JUnit para realizar esta tarea.

Básicamente, se va a explicar detalladamente la técnica de prueba sobre aspectos del lenguaje que merecen una atención especial ya que no pueden probarse de la manera “tradicional”. Adicionalmente, se aportarán ejemplos que hacen uso de determinadas herramientas para poner en práctica las técnicas de prueba comentadas.

## 2.8.1. Prueba de excepciones

Las excepciones en Java son eventos que ocurren durante la ejecución de un programa y que alteran su flujo normal de ejecución. Las excepciones están siempre asociadas a métodos en el sentido de que una excepción siempre se origina dentro de un método que la lanza. Cuando una excepción ocurre dentro de un método, este tiene tres posibilidades, capturarla y actuar en consecuencia mediante el uso de las sentencias `try`, `catch` y `finally`, lanzarla utilizando la sentencia `throws` o bien no hacer nada si se trata de una excepción de tipo no comprobado. La prueba de excepciones tiene sentido en el segundo caso, es decir cuando un método puede lanzar una excepción y tal evento se puede asociar a un caso de prueba.

Es importante notar que las excepciones, aun estando asociadas a situaciones anómalas, no tienen por qué estar asociadas a errores en el código fuente, como podría ser utilizar un objeto no inicializado, sino que muchas veces son ocasionadas debido a circunstancias que suceden de forma imprevista en el entorno de ejecución. Estas circunstancias pueden ser de muy diversa índole, como por ejemplo que una conexión de red no esté disponible, que un fichero no pueda ser encontrado en disco o que falle la inicialización de una determinada librería.

Las excepciones son una parte importante del lenguaje Java, y como tal, han de ser probadas. Atendiendo a la forma en que son probadas, podemos clasificarlas en dos categorías: excepciones esperadas y excepciones no esperadas.

### 2.8.1.1. Excepciones esperadas

Se trata de aquellas excepciones que constituyen el objetivo del caso de prueba, es decir, se diseña un caso de prueba en el que la excepción debe producirse y de no producirse constituiría un fallo. Estas excepciones son siempre excepciones comprobadas, es decir, excepciones definidas o no por el desarrollador pero que heredan directamente de la clase `Exception`. Se utilizan normalmente para comunicar situaciones imprevistas que pueden ser resueltas en tiempo de ejecución de una forma razonable. Por ejemplo, una entrada de datos por parte del usuario con información incorrecta.

Nótese que no tiene sentido realizar un caso de prueba de excepciones esperadas para excepciones no comprobadas. Esto es así porque definir tal caso de prueba es lo mismo que decir que una excepción no comprobada se espera.

Por otro lado, diseñar un caso de prueba en el que se invoca un método de un objeto que no ha sido inicializado previamente, es decir, con valor `null` no tiene ninguna utilidad. Obviamente, una excepción del tipo `NullPointerException` se va a producir ya que la máquina virtual de Java funciona correctamente. Este tipo de pruebas se conocen comúnmente como “probar la plataforma” de desarrollo, y obviamente, este no es el objetivo de las pruebas. Quizás sea el objetivo de los desarrolladores de la plataforma Java, pero eso es otra historia diferente.

#### 2.8.1.1.1. PROCEDIMIENTO GENERAL

A continuación, se verá el patrón típico de la prueba de excepciones esperadas mediante un ejemplo. Supóngase que se está probando el método `conectar` de una clase que es nuestra interfaz con una base de datos. Típicamente este método recibe una cadena de caracteres con información acerca de la localización en red del gestor de base de datos así como el nombre de la base de datos y los detalles de autenticación. De esta forma invocar el método `conectar` con

una cadena de caracteres vacía, parece una buena forma de realizar un caso de prueba en el que se espera una excepción del tipo `FalloDeConexionException`. A continuación, se lista el código fuente del método `testConectar`.

```
public void testConectar() throws Exception {
    try {
        interfazBD.conectar("");
        fail("El metodo deberia haber lanzado una excepcion.");
    } catch (FalloDeConexionException e) {
        assertTrue(true);
    }
}
```

En caso de que el método `conectar` funcione correctamente, deberá saltar la excepción `FalloDeConexionException` y, por tanto, la sentencia `fail` no se ejecutará, ya que el flujo de ejecución pasará automáticamente a la primera instrucción del bloque `catch`. En caso contrario la sentencia `fail` producirá un fallo que JUnit reportará junto con el mensaje descriptivo del fallo, en este caso *“El método debería haber lanzado una excepción”*.

El procedimiento general para realizar la prueba de excepciones esperadas, consta de los siguientes pasos:

1. Incluir la llamada al método a probar (aquel que ha de lanzar la excepción) dentro de un bloque `try-catch`.
2. Escribir una sentencia `fail` justo a continuación de la llamada al método a probar. Esta sentencia `fail` es la que actuará de testigo en caso de que la excepción no se produzca, e informará a Junit de que se ha producido un fallo.
3. Crear un bloque `catch` a continuación, en el que se capturará la excepción esperada. Adicionalmente, se podrán comprobar determinadas condiciones sobre el objeto que representa la excepción (el objeto `e` en el ejemplo).
4. Declarar el método de prueba de forma que lance la excepción `Exception`. Esto no es estrictamente necesario pero es una forma de programación preventiva. Véase en el ejemplo que el método `conectar` podría ser modificado en el futuro de forma que lanzara una excepción comprobada distinta de `FalloDeConexionException`. En este caso, si el método de prueba no captura ni lanza dicha excepción se produciría un error de compilación. Por otra parte, hay opiniones en contra del uso de esta técnica ya que en el fondo este tipo de error de compilación es beneficioso puesto que avisa al desarrollador de que un nuevo caso de prueba para la excepción recientemente añadida necesita ser creado. Por tanto, este último paso se propone únicamente como una opción para el desarrollador.

Como se ha visto, este procedimiento general es válido para cualquier versión de JUnit, sin embargo diferentes versiones de JUnit aportan mecanismos que lo facilitan. En los siguientes puntos se realizara una discusión en detalle de dichos mecanismos para finalmente extraer conclusiones y recomendaciones de uso.

#### 2.8.1.1.2. PRUEBA DE EXCEPCIONES ESPERADAS MEDIANTE LA CLASE `ExceptionTestCase`

Una forma, *a priori*, alternativa de hacer pruebas de excepciones esperadas es mediante el uso de la clase `ExceptionTestCase`, perteneciente al paquete `junit.extensions` de la ver-

sión 3.8.1 de JUnit. Nótese que la clase `ExceptionTestCase` hereda de `TestCase`. A continuación, se muestra un ejemplo de utilización.

```
public class InterfazBDTest() extends ExceptionTestCase {

    //...

    public InterfazBDTest(String nombre, Class excepcion) {
        super(nombre, excepcion);
    }
    public void testConectar() {
        interfazBD.conectar("");
    }
}
```

Se trata simplemente de hacer que la clase de prueba herede de la clase `ExceptionTestCase` en lugar de heredar directamente de `TestCase`. Además, se ha de declarar un constructor que llame a la superclase pasándole la clase de la excepción esperada como parámetro. El procedimiento es el siguiente:

1. Importar el paquete `junit.extensions`.
2. Hacer que la clase de prueba herede de la clase `ExceptionTestCase` en lugar de `TestCase`.
3. Definir un constructor que reciba dos parámetros, un `String` con el nombre de la clase de prueba y un objeto de tipo `Class` con la clase de la excepción esperada.
4. Definir el método de prueba de forma que llame al método a probar que lanzará la excepción esperada.

La forma de añadir esta clase de prueba a un objeto de la clase `TestSuite` es la siguiente:

```
TestSuite suite = new TestSuite();
suite.addTest(new MyTest("testConectar", FalloDeConexion.class));
```

Lo que llama la atención es que, adentrándose en el funcionamiento interno de la clase `ExceptionTestCase`, es posible comprobar que lo que realmente hace esta clase es completamente equivalente al procedimiento general anteriormente visto. A continuación, se lista el código perteneciente a la clase `ExceptionTestCase` perteneciente a la versión 3.8.1 de Junit, donde puede observarse que efectivamente el procedimiento es análogo.

```
public class ExceptionTestCase extends TestCase {

    Class fExpected;

    public ExceptionTestCase(String name, Class exception) {

        super(name);
        fExpected= exception;
    }
    protected void runTest() throws Throwable {
```

```

        try {
            super.runTest();
        } catch (Exception e) {
            if (fExpected.isAssignableFrom(e.getClass()))
                return;
            else
                throw e;
        }
        fail("Expected exception " + fExpected);
    }
}

```

El funcionamiento es sencillo; simplemente declara un bloque `try` desde el que llama al método `runTest` de la clase `TestCase` (este método es el encargado de ejecutar los métodos de prueba pertenecientes a la clase de prueba). Asimismo declara un bloque `catch` en el que se captura la excepción esperada (o cualquiera de sus descendientes) o bien se lanza en caso de no ser del tipo adecuado. En caso de que ninguna excepción se produzca, la sentencia `fail` indica a JUnit el fallo.

Sin embargo, como puede observarse, este método de prueba de excepciones mediante la clase `ExceptionTestCase` no es nada recomendable ya que presenta una serie de desventajas sobre el procedimiento general anteriormente visto, son las siguientes:

- Todos los métodos de prueba declarados en la clase de prueba han de lanzar forzosamente la excepción esperada o bien la prueba fallará.
- Si existe más de un caso de prueba dentro del método de prueba y la prueba falla, no es fácil determinar qué caso de prueba es el que ha dado lugar al fallo.
- No se puede probar más de una clase de excepción simultáneamente.

Con toda seguridad, estos enormes problemas de flexibilidad son los que han motivado que la clase `JUnitExtensions` haya sido eliminada de las versiones 4.x de Junit.

#### 2.8.1.1.3. PRUEBA DE EXCEPCIONES ESPERADAS MEDIANTE JUNIT 4.0

JUnit 4.0 proporciona un mecanismo que simplifica la prueba de excepciones esperadas, este mecanismo se basa una vez más en anotaciones. Se trata de un método mucho más compacto y legible que el anteriormente comentado, que sin duda facilita la prueba y mejora la mantenibilidad.

Véase el ejemplo anterior, pero esta vez utilizando Junit 4.0 y el parámetro `expected` de la etiqueta `@Test`.

```

@Test(expected=FalloDeConexionException.class)
public void testConectar() throws Exception {

    interfazBD.conectar("");
}

```

Como puede observarse, basta con añadir el parámetro `expected`, con la clase de la excepción a probar como valor, en la anotación `@Test` y JUnit se encarga de hacer el resto. En general los pasos a seguir son los siguientes:

1. Añadir el atributo `expected` a la anotación `@Test` del método de prueba y asignarle como valor la clase de la excepción esperada.
2. Declarar el método de prueba de forma que lance la excepción `Exception`.

Sin embargo, este mecanismo presenta de nuevo un inconveniente respecto al método general anteriormente comentado, y es que no es posible comprobar condiciones sobre el objeto de la excepción una vez que esta se ha producido. El motivo es bien sencillo, dentro del método de prueba no se puede definir un bloque `try-catch` para capturar la excepción esperada ya que si se captura, tal excepción no podrá ser observada por JUnit a no ser que se lance de nuevo con la instrucción `throw`, lo cual complica todo muchísimo.

Echando un vistazo atrás, al ejemplo anterior, supóngase que la excepción `FalloDeConexionException` contiene información de bajo nivel acerca de los motivos por los que la conexión con el gestor de base de datos ha fallado. En este caso sería interesante probar no solamente si la excepción esperada se produce, sino también si la información asociada a los motivos por los que se ha producido es coherente con el caso de prueba que la originó.

El siguiente ejemplo trata de ilustrar exactamente la situación que se acaba de comentar. Se trata de la prueba del método `comprobarFormato` definido en la clase `Tramo` del sistema presentado en el Apéndice B. Dicho método se encarga de comprobar que la información con la que ha sido construido el objeto `Tramo` es válida y por tanto el formato del objeto es correcto. En caso de que no lo sea, lanza una excepción del tipo `TramoMalFormadoException` que contiene un mensaje descriptivo del error encontrado. A continuación, se lista el código del método `comprobarFormato`:

sistema software/src/servidorEstadoTrafico/Tramo.java

```
public void comprobarFormato() throws TramoMalFormadoException {
    try {
        int iInicio = Integer.parseInt(this.m_strKMInicio);
        int iFin = Integer.parseInt(this.m_strKMFin);
        int iCarriles = Integer.parseInt(this.m_strCarriles);
        int iCarrilesCortados = Integer.parseInt(this.m_strCarrilesCortados);

        if (iInicio < 0)
            throw new TramoMalFormadoException("Valor de kilometro inicial negativo.");
        if (iCarrilesCortados < 0)
            throw new TramoMalFormadoException("Valor de carriles cortados negativo.");
        if (iInicio > iFin)
            throw new TramoMalFormadoException("Valor de kilometro inicial superior a valor de kilometro final.");
    }
}
```

```

        if (iCarrilesCortados > iCarriles)
            throw new TramoMalFormadoException("Valor de carriles cortados superior a valor de carriles totales.");

    } catch (NumberFormatException e) {

        throw new TramoMalFormadoException("La informacion asociada al objeto tramo es inconsistente.");
    }
}

```

El siguiente código corresponde al método de prueba del método anterior.

pruebas sistema software/src/pruebasSistemaSoftware/junit381/RegistroTest.java

```

public void testComprobarFormato() {

    String strKMInicio = "0";
    String strKMFin = "14";
    String strCarriles = "3";
    String strCarrilesCortados = "4";
    String strEstado = "Retenciones";
    String strAccidentes = "Sin accidentes";

    //Instanciacion de un objeto con los datos del caso de prueba
    Tramo tramo = new Tramo(strKMInicio, strKMFin, strCarriles, strCarrilesCortados, strEstado, strAccidentes);

    try {
        tramo.comprobarFormato();
        fail("La excepcion esperada no se ha producido");
    }
    catch (TramoMalFormadoException e) {
        //Verificacion de la causa de la excepcion
        String strCausaEsperada = "Valor de carriles cortados superior a valor de carriles totales.";
        String strCausa = e.toString();
        assertEquals(strCausaEsperada, strCausa);
    }
    return;
}

```

Como puede observarse el caso de prueba no se limita a comprobar que una excepción se ha producido sino que además el mensaje que acompaña a la excepción es el correspondiente al caso de prueba. En este caso, como se ha creado un tramo con cuatro carriles cortados y el tramo solo tiene tres carriles, el mensaje<sup>30</sup> de error asociado a la excepción debe describir tal situación.

---

<sup>30</sup> Nótese que los mensajes de error deben estar definidos como constantes de forma que solo se escriban una vez. Se ha elegido esta forma de representarlos por claridad en el ejemplo.

### 2.8.1.2. Excepciones no esperadas

Son aquellas que simplemente aparecen durante la ejecución de un método de prueba sin que ello esté previsto en la definición de ninguno de sus casos de prueba. Cuando esto ocurre, la excepción es atrapada por el framework JUnit y un error es reportado. Por este motivo no es necesario que el desarrollador proporcione ningún mecanismo para prever este tipo de situaciones, JUnit en este caso hace todo el trabajo.

## 2.8.2. Prueba de métodos que no pertenecen a la interfaz pública

Al igual que todo lenguaje orientado a objetos, Java consta de un mecanismo de encapsulación. La encapsulación tiene como objetivo ocultar los detalles de implementación de una clase de forma que las clases que la utilizan solo conocen su interfaz pública. Esta interfaz pública es todo lo que se necesita saber de una clase para poder utilizarla y sacar partido de ella. El objetivo de la encapsulación no es otro que garantizar la mantenibilidad del código cuando las clases evolucionan. De esta forma una clase puede ser modificada para optimizar su implementación y, siempre que la interfaz pública original no se vea modificada, el sistema seguirá compilando y funcionando como hasta entonces.

Java define 4 atributos diferentes de privacidad, por orden de mayor a menor visibilidad son los siguientes:

- **public:** cuando el elemento es visible desde cualquier nivel.
- atributo por defecto, cuando el elemento no es visible desde fuera del paquete al que pertenece.
- **project:** cuando el elemento no es visible más que dentro de la propia clase y subclases.
- **private:** cuando el elemento es solo visible desde dentro de la propia clase en que es definido.

Nadie duda de las ventajas que aporta el uso de la encapsulación en un lenguaje orientado a objetos, sin embargo, a la hora de realizar las pruebas surge un problema, y es que solo los métodos de la interfaz pública pueden, por definición, ser invocados desde fuera de la clase a probar y por tanto, *a priori*, solo esos métodos pueden ser probados. En realidad esto no es totalmente cierto, ya que, como se verá a continuación, todo método en Java puede ser probado, aunque lógicamente si el método es público todo resulta mas sencillo.

En primer lugar, se ha de reflexionar acerca de qué es un método privado y cuál es el origen típico de este tipo de métodos. Normalmente, los métodos privados no son más que segmentos de código duplicado que han sido extraídos de otros métodos mediante un proceso de refactorización. Es decir, cuando una serie de líneas de código se repiten con frecuencia, tiene sentido tomar esas líneas de código y crear un método con ellas. De esta forma el código queda más limpio y menos redundante. Esta reflexión lleva a pensar que un método privado no necesita ser probado ya que al probar los métodos que hacen uso de él, de forma indirecta, se está probando dicho método. Bueno, esto en principio tiene sentido, pero no siempre se cumple.



A continuación, se van a detallar las diferentes técnicas para probar métodos no pertenecientes a la interfaz pública de una clase<sup>31</sup>. Desde la prueba indirecta hasta el uso de herramientas específicas. Antes de continuar nótese que hablar de prueba de métodos privados solo tiene sentido desde una perspectiva de caja blanca en la que es posible conocer la funcionalidad de dicho método y por tanto probarlo.

### 2.8.2.1. Prueba de forma indirecta

Se entiende por probar un método de forma indirecta o implícita a probarlo mediante las llamadas que otros métodos, que sí son probados directamente, realizan sobre el método en cuestión. Puesto que el método no puede ser invocado directamente desde el código de pruebas debido a su atributo de privacidad, al menos probarlo indirectamente mediante los métodos que hacen uso de él. De esta forma, la diferencia entre la prueba directa e indirecta es que para esta última no es necesario definir casos de prueba. Los casos de prueba quedan definidos implícitamente al definir los casos de prueba de los métodos que hacen uso del método probado indirectamente.

En general, existen fuertes razones que desaconsejan la prueba de métodos no pertenecientes a la interfaz pública de una clase de forma explícita, es decir, directa. Son las siguientes:

- Cuando un método privado es suficientemente sencillo, normalmente puede ser probado eficazmente de forma indirecta. Es decir, a través de los métodos públicos desde los cuales es invocado.
- Cuando un método privado no es lo suficientemente sencillo como para ser probado indirectamente de forma satisfactoria, esto suele revelar un problema de diseño. La razón es que cuando un método alcanza un nivel de complejidad relativamente alto, este método debe ser promocionado para convertirse en una nueva clase. Es decir, el método en sí mismo tiene una entidad propia y por tanto, por motivos de modularidad, es buena idea extraerlo de la clase original y convertirlo en una nueva clase que será utilizada por la clase original.
- En la gran mayoría de los casos, los métodos privados provienen de segmentos de código que aparecen de forma repetida en métodos públicos. Típicamente mediante un proceso de refactorización, el desarrollador convierte estos segmentos de código en métodos privados.

Sin embargo, en ocasiones el proceso de prueba indirecta de un método no público a través de la interfaz pública de la clase es incapaz de detectar defectos existentes en ese método, creando en el desarrollador una falsa sensación de confiabilidad en el software desarrollado. Esto es debido a que en lugar de definirse casos de prueba que permitan probar el método de una forma adecuada, los casos de prueba permanecen implícitos y están restringidos a la forma en que otros métodos llaman al método en cuestión.

Este problema es especialmente grave en el caso particular de los métodos protegidos. Supóngase que, en un momento dado, un desarrollador decide crear una clase que herede de la clase en la que el método protegido fue definido. Esta nueva clase posiblemente hará uso del método protegido, y posiblemente lo hará dentro de un contexto y con unos parámetros de entrada totalmente diferentes a aquellos con los que este método era invocado desde su propia clase. De esta forma aquellos errores latentes no detectados en la prueba indirecta podrían sin duda apa-

<sup>31</sup> Nótese que en java estos métodos son todos aquellos con un atributo de privacidad distinto de `public`.

recer y causar problemas. Además la resolución de estos errores, al haber sido detectados posiblemente en una fase posterior del desarrollo del software, tendría un coste significativamente superior.

En la práctica se dan situaciones en las que un método es lo suficientemente complejo para desaconsejar una prueba indirecta y al mismo tiempo no tiene la entidad suficiente para ser refactorizado en una nueva clase colaboradora.

Normalmente, para determinar la forma en la que un método no público ha de ser probado parece una buena idea, primero, hacer un buen proceso de diseño y refactorización. Una vez hecho esto, aquellos métodos privados sencillos serán probados indirectamente, mientras que aquellos que presenten una cierta complejidad se deberán probar de forma directa o bien ser promocionados al nivel de clase colaboradora<sup>32</sup>.

### 2.8.2.2 Modificar el atributo de privacidad de modo que los métodos sean accesibles desde el paquete

El procedimiento consiste en lo siguiente:

1. Convertir todos los métodos no públicos de la clase en métodos de paquete, es decir, con el atributo de privacidad por defecto. De esta forma, esos métodos serán visibles desde cualquier clase contenida en el mismo paquete al que pertenece la clase en la cual fueron definidos.
2. Incluir la clase de pruebas dentro del paquete de la clase a probar e instantáneamente los problemas de accesibilidad desaparecen.

Sin embargo, es evidente que este procedimiento presenta ciertos inconvenientes:

- Se sacrifica la encapsulación y la ocultación de datos, conceptos fundamentales en un lenguaje orientado a objetos, para facilitar el proceso de la prueba.
- Cuando un desarrollador pretende utilizar una clase desde otra definida en el mismo paquete (una situación muy común si por ejemplo el software de producción es una librería distribuable) mira en su interior, y si ve un método privado, automáticamente sabe que forma parte de la implementación de la clase y que, por tanto, no necesita saber nada de él. Con este procedimiento, por tanto, también se sacrifica la legibilidad del código fuente y se dificulta el desarrollo.
- Al situar las clases de prueba dentro del mismo paquete en el que está situado el código a probar, se están mezclando unas clases y otras, lo que dificulta claramente la gestión de configuraciones. A pesar de que existe un procedimiento para salvar este inconveniente (utilizando dos árboles de código fuente en paralelo, uno para el código de las pruebas y otro para el código a probar, y añadiéndolos a la variable de entorno CLASSPATH), el resultado es que el término paquete deja de ser “sinónimo” de directorio, lo cual no deja de ser una molestia añadida.

Por todos estos motivos se descarta el uso de esta técnica.

---

<sup>32</sup> En el Capítulo 7, Mock Objects, se explica en detalle el proceso de prueba de clases que hacen uso de clases colaboradoras en relaciones de asociación.

### 2.8.2.3. Utilizar clases anidadas

Este método se basa en explotar una de las características del modificador de acceso `private`, y es que aquellos métodos declarados como `private`, no sólo pueden ser accedidos desde el interior de su clase, sino también desde el interior de clases anidadas dentro de ella. El procedimiento consiste en lo siguiente:

1. Anidar la clase de pruebas dentro de la clase a probar. De esta forma es posible acceder a los métodos privados.
2. Cualificar la clase anidada con el modificador de acceso por defecto, es decir, la clase será accesible desde el paquete. Esto es fundamental ya que la clase necesita ser instanciada desde dentro del paquete cuando se construya el objeto `TestSuite` correspondiente.

Aunque desde el punto de vista de las posibilidades que el lenguaje Java ofrece, utilizar clases anidadas para probar métodos privados es una alternativa viable, no deja de presentar una serie de inconvenientes que desaconsejan totalmente su uso. Es más, su presencia en este libro no va más allá del mero estudio de las diferentes alternativas. Los inconvenientes son los siguientes:

- En general, y debido a motivos de mantenibilidad, no es una buena idea mezclar el código de las pruebas con el código a probar. Se trata sin duda de una técnica muy invasiva en este sentido.
- La clase de pruebas al ser compilada producirá un archivo `.class` que deberá ser eliminado del archivo `.jar` que contenga el código de producción.

### 2.8.2.4. Utilizar la API de Reflection de Java

Este método consiste en saltarse, durante la ejecución de las pruebas, el mecanismo de encapsulación que establece la máquina virtual de Java. Para ello se utiliza la API de Reflection de Java<sup>33</sup>. Esta API permite descubrir los atributos, métodos, constantes y otras propiedades de una clase en tiempo de ejecución. Típicamente se utiliza en situaciones extraordinarias, por ejemplo para construir depuradores del lenguaje o herramientas de desarrollo que muestran al desarrollador el contenido de las clases y sus propiedades por medio de una interfaz gráfica. Esta API no solo permite obtener la información de una clase en tiempo de ejecución sino que también permite obtener referencias a sus elementos, como métodos y atributos, y modificar sus propiedades. Esto último incluye la posibilidad de cambiar el modificador de acceso de un método de privado a público y en general de no público a público.

A continuación se lista, a modo de ejemplo, el código de un método que hace uso de la API de Reflection de Java para invocar un método privado. El método privado en cuestión se llama `procesarPeticiónHTTP` y simplemente recibe una petición HTTP en forma de cadena de caracteres y devuelve sus parámetros.

```
public void testProcesarPeticiónHTTP() {

    final String METODO = "procesarPeticiónHTTP";

    try {
```

---

<sup>33</sup> Para más información consultar el paquete `java.lang.reflect` de la `jdk`.

```

        HiloPeticion hiloPeticion = new
        HiloPeticion(null,1,null,null,null);
        Class clase = HiloPeticion.class;
        Method metodo = clase.getDeclaredMethod(METODO, new Class[]
        {String.class});
        metodo.setAccessible(true);
        String strPeticion = "GET /?peticion=\"clima\"&clima=\
        \"nublado\" HTTP/1.1";
        String strParametros = (String)metodo.invoke(hiloPeticion,
        new Object[]
        {strPeticion});
        assertEquals(strParametros,"?peticion=\"clima\"&clima=\
        \"nublado\"");
    } catch (Throwable e) {
        System.err.println(e);
        fail();
    }

    return;
}

```

El procedimiento es sencillo, simplemente se obtiene una referencia del método `procesarPeticionHTTP` que se utiliza para invocarlo. Nótese que al invocar dicho método utilizando la referencia, se pasa un array de objetos del tipo `Object`, que debe contener la lista de parámetros.

Alternativamente, existe una herramienta que permite realizar esta misma tarea de una forma mas sencilla y evitando al desarrollador conocer los detalles de utilización de la API de Reflection. La herramienta se llama JUnit-addons<sup>34</sup> y a continuación se muestra el ejemplo anterior, pero realizado a través de dicha herramienta.

```

public void testProcesarPeticionHTTP() {

    final String METODO = "procesarPeticionHTTP";

    try {

        HiloPeticion hiloPeticion = new HiloPeticion(null,1,null,null,null);
        String strPeticion = "GET /?peticion=\"clima\"&clima=\"nublado
        \" HTTP/1.1";
        String strParametros = (String)PrivateAccessor.invoke
        (hiloPeticion,METODO,new
        Class[] {String.class}, new Object[] {strPeticion});
        assertEquals(strParametros,"?peticion=\"clima\"&clima=\
        \"nublado\"");

    } catch (java.lang.Throwable e) {
        System.err.println(e);
    }
}

```

---

<sup>34</sup> JUnit-addons es una herramienta de código abierto y disponible para descarga desde SourceForge. El sitio Web del proyecto es: <http://sourceforge.net/projects/junit-addons>.

```

        fail();
    }

    return;
}

```

En este caso véase que el método `invoke` de la clase `PrivateAccessor` recibe el objeto, el nombre del método a invocar y sus parámetros, y se encarga de llevar a cabo todo el “trabajo sucio”.

Finalmente se van a enumerar las ventajas e inconvenientes del uso de la API de Reflection de Java para solucionar el problema de la prueba de métodos no públicos.

Ventajas:

- Permite una clara separación entre el código de producción, es decir, el código a probar y el código de la prueba.
- Respeta la API del código a probar respetando así mismo el mecanismo de encapsulación y ocultación de datos.

Inconvenientes:

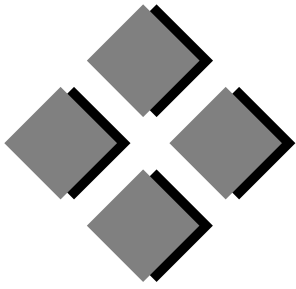
- Puesto que es necesario utilizar la API de Reflection, bien directa o indirectamente, a través de alguna librería específica como JUnit-addons, el desarrollador ha de escribir más líneas de código de las que serían necesarias para la prueba de un método público.
- Cuando se trabaja bajo entornos que facilitan la refactorización, como es el caso de Eclipse o IntelliJ, basta con cambiar el nombre de un método en la herramienta para que este cambio se propague a lo largo de todo el código fuente. Es decir, en todos aquellos lugares donde aparece el método, bien declarado o bien invocado, este cambia de nombre. Sin embargo, estas herramientas no son capaces de descubrir y cambiar nombres de métodos en el interior de cadenas de caracteres. Por este motivo el desarrollador deberá realizar estos cambios manualmente en el código de pruebas.

Como puede comprobarse, las ventajas de este procedimiento son sustanciales y los inconvenientes son mínimos y en parte inevitables. Por tanto, siempre que no quede más remedio que realizar la prueba de un método no público, se hará de esta forma.

## 2.9. Bibliografía

- [www.junit.org](http://www.junit.org)
- Goncalves, A.: *Get Acquainted with the New Advanced Features of JUnit 4*, 24 de julio de 2006.
- Matthew Young, J. T., Brown, K. y Glover, A.: *Java Testing Patterns*, Wiley, 1 de octubre de 2004.
- Harold, E.: *An early look at JUnit 4*, 13 de septiembre de 2005.
- Massol, V.: *JUnit in Action*, Manning Publications, 28 de octubre de 2003.
- Link, J.: *Unit Testing in Java: How Tests Drive the Code*, Morgan Kaufmann, abril de 2003.





# Capítulo 3

## Ant

### SUMARIO

3.1. Introducción	3.4. Creación de un proyecto básico
3.2. Instalación y configuración	3.5. Ejecución de los casos mediante Ant
3.3. Conceptos básicos	3.6. Bibliografía

## 3.1. Introducción

A la hora de desplegar el software asociado a un proyecto software cualquiera hay una serie de tareas que se repiten una y otra vez: compilación del código fuente, creación de directorios y archivos temporales, borrado de archivos resultantes de un proceso de compilación, ejecución de un binario, etc. Un modo de automatizar estas tareas es crear un fichero de *script*. Por ejemplo, en Windows mediante un fichero `.bat` o en Linux mediante un fichero `.sh`. Pero los *scripts* son dependientes del sistema operativo y además presentan limitaciones para realizar tareas específicas y operaciones complejas. Ant es una herramienta que ayuda a automatizar estas tareas de forma similar a como lo hace la herramienta Make, bien conocida por cualquier desarrollador, pero de forma independiente al sistema operativo. Es decir, Ant es una herramienta de despliegue de software de propósito general, pero que presenta una serie de características que la hacen especialmente adecuada para la compilación y creación de código Java en cualquier plataforma de desarrollo.

Un *script* Ant se escribe en forma de documento XML cuyo contenido va a determinar las operaciones o tareas disponibles a la hora de desplegar el software del proyecto. Aunque, normalmente, un único documento Ant puede utilizarse para desplegar todo el software de un pro-

yecto dado, normalmente, en proyectos grandes, se suelen escribir varios documentos que actúan de forma conjunta para ese propósito. Los documentos Ant pueden recibir cualquier nombre, si bien el nombre por omisión es `build.xml` (de la misma forma que para la utilidad Make el nombre por omisión es `makefile`). Estos documentos están estructurados en tareas de forma que cada tarea se encarga de realizar operaciones con entidad propia. Sin embargo, las tareas, como se verá más adelante, están interrelacionadas a través de dependencias.

Entre las ventajas de Ant se pueden mencionar las siguientes:

- Ant es software de código abierto y se puede descargar desde el sitio Web <http://ant.apache.org>.
- Ant está desarrollado en Java, por lo que es multiplataforma. Es decir, el mismo fichero `build.xml` se puede ejecutar en Windows o en Linux y los resultados serán equivalentes.
- Permite definir varias tareas a realizar y fijar las dependencias entre ellas. Ninguna tarea se ejecuta dos veces, ni antes de que se ejecuten las que la preceden según las dependencias indicadas.
- Aunque esta herramienta está especialmente diseñada para ofrecer soporte al lenguaje de programación Java, se puede utilizar con otros lenguajes de programación.

Si hacer un programa requiere la repetición de cierto número de tareas, como se ha mencionado al principio, esta situación se pone especialmente de manifiesto durante la fase de pruebas. En esta fase se ejecuta un juego de ensayo, se modifica el código, se recompila y se vuelve a ejecutar el juego de ensayo para comprobar si se han eliminado los errores encontrados. Y estas tareas se repiten sistemáticamente. Por tanto, en este capítulo se pretende dar una visión de cómo una herramienta como Ant puede ayudar en la fase de pruebas de un proyecto. Se describirán las tareas más habituales que se pueden realizar y se explicará cómo escribir un documento Ant para poder compilar, crear y probar programas en Java. Antes de empezar, se verá cómo instalar y configurar Ant.

## 3.2. Instalación y configuración

Para trabajar con Ant se necesita:

- Ant. El fichero con distribución se puede encontrar en <http://ant.apache.org>.
- JDK. Ya se ha mencionado que Ant está desarrollado en Java. Si no se dispone de esta herramienta se puede descargar desde la página [http://java.sun.com/javase/downloads/index\\_jdk5.jsp](http://java.sun.com/javase/downloads/index_jdk5.jsp).
- Un parser XML. La distribución de Ant incluye uno, así que no es necesario realizar instalaciones adicionales.

Una vez que Java esté instalado en la máquina, los pasos que se han de seguir para instalar Ant son los siguientes:

1. Descargar los ficheros binarios de Ant desde <http://ant.apache.org/bindownload.cgi> y extraerlos a un directorio creado para tal efecto. Por ejemplo, `C:\Archivos de programa\ant`.



2. Añadir la carpeta `C:\Archivos de programa\ant\bin` a la variable de entorno `PATH`.
3. Añadir los archivos `.jar` situados en la carpeta `C:\Archivos de programa\ant\lib` a la variable de entorno `CLASSPATH`.
4. Hacer que la variable de entorno `JAVA_HOME` apunte a la localización de la instalación de JDK.
5. Añadir la carpeta `directorio_de_jdk\lib\*` a la variable de entorno `CLASSPATH`.

Para comprobar que Ant ha sido instalado correctamente se ha de ejecutar la siguiente orden<sup>1</sup>, que mostrará la versión de Ant instalada:

```
c:\> ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
```

Si la instalación no se ha realizado correctamente, en Windows se obtiene el siguiente mensaje:

```
c:\> ant -version
"ant" no se reconoce como un comando interno o externo,
programa o archivo por lotes ejecutable.
```

### 3.3. Conceptos básicos

Como se ha mencionado en la introducción, las tareas para compilar y crear programas con la ayuda de Ant se especifican en un fichero con formato XML, que habitualmente se llama `build.xml`. Pero, ¿qué información incluye este fichero? En este apartado se definirá brevemente la estructura del fichero `build.xml` y los elementos básicos que lo componen: objetivos, propiedades y tareas.

El esquema general del fichero `build.xml` es el siguiente:

```
<?xml version="1.0"?>
  <!-- Al tratarse de un fichero en XML, debe comenzar con la de-
        claración de comienzo del documento, en la que se indica la
        versión de XML que se está utilizando ---->

  <project name="ejemplo" default="inicializar" basedir=".">
    <target name="inicializar">
      ....
    </target>

    <target name="limpiar" description="borrar ficheros temporales">
      ....
    </target>
```

---

<sup>1</sup> Se puede ejecutar desde cualquier directorio. Se usará el directorio raíz sólo como ejemplo.

```
<target name="compile" depends="init, limpiar">
    ....
</target>
</project>
```

La primera línea indica que se trata de un fichero XML. Después se define el elemento raíz del fichero XML mediante las etiquetas `<project>` y `</project>`. Puesto que es el elemento raíz, solo puede haber una etiqueta `<project>` en el fichero `build.xml`. Entre estas etiquetas se definen los objetivos necesarios para el proyecto, es decir, los *targets*, que habitualmente serán: compilar, crear directorios, borrar versiones antiguas de ciertos ficheros, etc. En cada uno de estos objetivos se escribirán las acciones o tareas (*tasks*) que se deberán llevar a cabo para cumplir las acciones asociadas al objetivo en el cual se han definido.

Los atributos de la etiqueta `<project>` que definen el proyecto son:

- **name:** da nombre al proyecto.
- **default:** indica cuál será el objetivo que se ejecutará si no se especifica ninguno. En el ejemplo este objetivo es el objetivo `inicializar`.
- **basedir:** las rutas especificadas dentro del documento Ant son relativas al directorio especificado en este atributo. En el ejemplo serán relativas al directorio actual “.”.
- **description:** se utiliza para describir brevemente el proyecto.

Un objetivo es un conjunto de acciones que se ejecutan de forma secuencial (más adelante se verá que esto no es siempre así y que cuando la ocasión lo requiere se pueden crear varios cauces de ejecución en paralelo). Cada objetivo se define entre las etiquetas `<target>` y `</target>`. Al igual que para el proyecto, se pueden especificar atributos que definan los objetivos. Algunos de ellos son los siguientes:

- **name:** da un nombre único al objetivo para identificarlo. Este nombre permite referenciar los objetivos a ejecutar cuando se invoca al documento Ant desde la línea de comandos.
- **description:** da una breve descripción de las acciones incluidas en el objetivo.
- **depends:** especifica las dependencias entre objetivos, es decir, se trata de una lista de los objetivos que deberán ejecutarse previamente a la ejecución del objetivo en el que se ha definido. En el ejemplo, si se ejecuta el objetivo `compile`, Ant ejecutará previamente los objetivos `init` y `limpiar`.

Los objetivos se definen por medio de tareas (o acciones). En Ant reciben el nombre de *tasks*. Una tarea puede realizar, como se verá más adelante, operaciones de muy diverso tipo, relacionadas con el despliegue del software. Una tarea puede contener distintas propiedades o atributos como los definidos para los objetivos. Ant incluye muchas tareas básicas, desde la compilación de código fuente hasta el manejo del sistema de ficheros pasando por la ejecución de peticiones HTTP. En el Apartado 3.3.3 se verán principalmente aquellas más interesantes desde el punto de vista del software Java y de las pruebas de software en general.

El objetivo de este libro no es presentar un manual de Ant sino describir las ventajas de esta tecnología aplicadas a las pruebas de software. Por tanto, no se describen de forma exhaustiva todas las propiedades, tareas, objetivos y atributos que dispone Ant. Solo se describen los necesarios para realizar las pruebas. La información completa sobre dichas tareas, propiedades y objetivos se puede encontrar en el manual en línea de Ant <http://ant.apache.org/manual/index.html>.

### 3.3.1. Propiedades

Las propiedades son análogas a las variables que existen en casi cualquier lenguaje de *script*, en el sentido en que establecen una correspondencia entre nombres y valores. Al margen de las propiedades que puede definir el desarrollador, Ant tiene incorporadas varias propiedades. Algunas de ellas son las siguientes:

- `ant.file`: ruta absoluta al fichero `build.xml`.
- `ant.project.name`: contiene el nombre del proyecto actualmente en ejecución, es decir, el valor del atributo `name` de `<project>`.
- `ant.version`: versión de Ant.

Pero lo interesante de las propiedades es que se pueden definir todas las que se desee en un proyecto Ant. Con las propiedades se puede parametrizar y reutilizar el fichero `build` con solo cambiar sus valores. Un uso muy común de las propiedades consiste en definir los nombres de los directorios que se van a usar. Por ejemplo, en qué directorio está el código fuente y en qué directorio se pondrá el código compilado. Para definir una propiedad se utiliza la etiqueta `<property>` con los atributos `name` y `value`. Por ejemplo:

```
<property name="src" value="." />
<property name="build" value="build" />
```

Las propiedades `src` y `build`, y en general todas las propiedades, se han de definir al principio del fichero `build.xml` para que se puedan usar después desde cualquier tarea simplemente poniendo el nombre de la propiedad entre `${ }`. En el ejemplo siguiente al ejecutarse la tarea `<javac>`, el atributo `srcdir` tomará el valor `"."` como directorio donde se aloja el código fuente y el atributo `destdir` tomará el valor `build`, como directorio donde dejar las clases compiladas:

```
<javac srcdir="${src}" destdir="${build}" />
```

#### 3.3.1.1. Estructuras *Path-Like*

A la hora de desplegar software es muy común manejar rutas a ficheros y directorios. Con Ant, las rutas se pueden manejar con un tipo llamado estructura *path-like*. Por ejemplo, la tarea `<javac>` tiene, entre otros, los atributos `srcdir`, `classpath`, `sourcepath`, `bootclasspath` y `extdirs`, y todos ellos toman como valor una ruta de directorio. Con las estructuras *path-like* se pueden definir *paths* o *classpaths* que serán válidos sólo durante la ejecución del do-

cumento Ant. Cuando se ejecute este documento se tomarán los *paths* y *classpaths* que se hayan definido en él y no los que estén definidos en la máquina sobre la que se esté ejecutando. Para especificar un valor del tipo *path-like* se necesita usar el elemento *pathelement*. Por ejemplo:

```
<classpath>
  <pathelement path="{classpath}"/>
  <pathelement location="lib/helper.jar"/>
</classpath>
```

En *pathelement*, el atributo *location* especifica un fichero o directorio relativo al directorio base del proyecto (o un nombre de fichero absoluto), mientras que el atributo *path* mantiene listas de ficheros o directorios separados por coma o punto y coma. El atributo *path* se usa con rutas predefinidas (véase más adelante en este apartado). En otro caso es mejor usar varios *pathelement* con el atributo *location*.

Para crear rutas que se quieren referenciar después, se utiliza el elemento *path* al mismo nivel que los objetivos, con elementos *pathelement* anidados. Para poder hacer esta referencia más adelante es necesario dar un nombre único a la ruta mediante el atributo *id*, que después será usado para referenciar esa ruta usando el atributo *refid*. Por ejemplo:

```
<path id="build.classpath">
  <pathelement path="{classes}"/>
  <pathelement path="{classes2}"/>
</path>

<target name="compile">
  <javac destdir="{build}" classpath="classes.jar" debug="on">
    <src path="{src}"/>
    <classpath refid="build.classpath"/>
  </javac>
</target>
```

### 3.3.1.2. Grupos de ficheros y directorios

Al desplegar software es habitual manejar grupos de ficheros y realizar algunas tareas con todos los ficheros del grupo. Por ejemplo, compilar todos los archivos con extensión *.java* que estén en determinado directorio. En Ant, existe el tipo *fileSet* para representar un grupo de ficheros. Este elemento puede contener anidados otros elementos. Algunos ejemplos de estos elementos son:

- *include*: especifica una lista de patrones de ficheros que se quieren incluir en la tarea donde se define el grupo de ficheros. Se separan por coma o espacio.
- *exclude*: especifica una lista de patrones de ficheros que se quieren excluir. Se separan por coma o espacio.
- *dir*: especifica la raíz del directorio usado para crear el *fileset*.

Si por ejemplo se quieren copiar los ficheros desde un directorio *src* a otro directorio *dest* excepto los que tienen extensión *.java*, la definición de la tarea *<copy>* sería la siguiente:

```

<copy todir="../dest">
  <fileset dir="src">
    <exclude name="**/*.java"/>
  </fileset>
</copy>

```

Algunas tareas, como `<javac>`, forman grupos de ficheros de forma explícita. Esto significa que tienen todos los atributos de `<fileset>` (consultar el manual de Ant [3] o [1]). En estos casos, no es necesario definir un elemento `<fileset>`, puesto que ya está implícito en la tarea. Para consultar las tareas que, como `<javac>`, incorporan estos atributos se puede consultar el manual de Ant [3] o [1].

De igual modo se pueden definir grupos de directorios. Un grupo de directorios se puede definir como un elemento anidado en una tarea, o como objetivo (`target`). Al igual que los grupos de ficheros, pueden contener elementos como `include`, `exclude`, etc. Por ejemplo:

```

<dirset dir="${build.dir}">
  <include name="apps/**/classes"/>
  <exclude name="apps/**/*Test*" />
</dirset>

```

También se pueden especificar listas de ficheros. Por ejemplo:

```

<filelist id="ficherosdoc" dir="${doc}" files="type.html,using.html"/>

```

donde el atributo `dir` especifica el directorio donde se encuentran los ficheros de la lista y `files` contiene la lista de ficheros propiamente dicha separados por comas.

Estas estructuras se pueden usar para definir las rutas en las estructuras *path-like*. Por ejemplo:

```

<classpath>
  <pathelement path="${classpath}"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
  </fileset>
  <pathelement location="classes"/>
  <dirset dir="${build.dir}">
    <include name="apps/**/classes"/>
    <exclude name="apps/**/*Test*" />
  </dirset>
  <filelist refid="ficherosdoc"/>
</classpath>

```

Este código, mediante el elemento `<classpath>`, define una ruta que mantiene el valor de `${classpath}`, seguido por todos los `.jar` del directorio `lib`, el directorio `classes`, todos los directorios llamados `classes` bajo el directorio `apps` subdirectorio de `${build.dir}`, excepto los que tengan el texto `Test` como parte de su nombre y los ficheros especificados en `FileList`.

### 3.3.2. Objetivos

Como se dijo en la introducción de este capítulo, al desplegar el software de un proyecto es necesario repetir una serie de operaciones bien diferenciadas. Esas operaciones se definen en los objetivos del documento Ant. Un objetivo es un conjunto de acciones que se ejecutan habitualmente de forma secuencial<sup>2</sup>. En este apartado se describirán algunos de los objetivos que se pueden definir en Ant y que posteriormente serán de utilidad para la realización de las pruebas. Entre las operaciones más habituales se pueden mencionar la compilación, las inicializaciones previas como creación de directorios o borrado de los resultados de compilaciones anteriores, creación de archivos, generación de la documentación, ejecución de los casos de prueba, generación de los resultados de las pruebas, etc.

La estructura para definir un objetivo es la siguiente:

```
<target name="..." description="..." depends="...">
    ...
</target>
```

Por ejemplo, para compilar las clases de un proyecto se puede definir el siguiente objetivo:

```
<target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}" />
</target>
```

Este objetivo (que depende del objetivo `init`) tiene una única tarea, la de compilación `<javac>`, a la que por medio de los atributos `srcdir` y `destdir` se le indica los directorios fuente y destino, que se recogen en las propiedades `${src}` y `${build}` que se habrán definido anteriormente (véase Apartado 3.3.1). El nombre del objetivo se puede utilizar para referenciar dicho objetivo desde otros puntos del documento Ant. Por ejemplo, en el atributo `depends` de otros objetivos que dependen de él.

### 3.3.3. Tareas

Las tareas son código que puede ser ejecutado para realizar operaciones de muy diverso tipo. Se definen en el cuerpo de los objetivos y por medio de su ejecución se logran los objetivos. La estructura para definir una tarea es la siguiente:

```
<nombre_tarea atributo1="valor1" atributo2="valor2" ... />
```

Ant dispone de cientos de tareas para hacer prácticamente cualquier cosa. Algunas de estas tareas no están completamente incluidas en la distribución de Ant y requieren de la instalación de ficheros `jar` adicionales. Este es el caso de las tareas que ejecutan las herramientas de pruebas, como JUnit. A continuación se verán algunas de las más usadas. Como ya se ha mencionado para obtener información detallada acerca de cada una de estas tareas, y de sus atributos, se puede consultar el manual en línea de Ant en <http://ant.apache.org/manual/index.html>.

---

<sup>2</sup> Existe una tarea, `<parallel>`, que permite la ejecución de varios hilos de forma concurrente. En el Apartado 3.5 se puede ver un ejemplo de utilización.

### 3.3.3.1. Tareas sobre ficheros

Existe un grupo de tareas relacionadas con el manejo de archivos. Algunas de ellas son:

- `zip`: crea un archivo `.zip`.
- `unzip`: descomprime un archivo `.zip`.
- `rpm`: crea un archivo de instalación en Linux, este comando es dependiente de la plataforma.
- `jar`: crea un archivo `.jar`.
- `manifest`: crea un archivo `manifest`.

Conviene detenerse en la tarea `<jar>` que sirve para crear ficheros `.jar` que contienen paquetes de clases y otra información asociada. Entre los atributos más comunes de esta tarea se pueden mencionar:

- `destfile` es el nombre que se da al fichero `.jar` que se va a crear.
- `basedir` es el directorio de referencia para hacer el fichero `.jar`. Es decir, el directorio del que se toman los ficheros a comprimir.
- Es posible refinar el conjunto de ficheros a incluir en dicho fichero usando los atributos `includes` o `excludes` entre otros.
- Esta tarea forma un conjunto de ficheros implícito por lo que soporta los atributos y elementos anidados de `fileset` (véase Apartado 3.3.1.2).

A continuación, se muestra un ejemplo para crear el fichero `.jar` con las clases de prueba del servidor de tráfico<sup>3</sup>:

```
<jar destfile="${build}/pruebasSistemaSoftware.jar" basedir="${build}">
  <manifest>
    <attribute name="Built-By" value="${user}" />
    <attribute name="Main-Class"
              value="pruebasSistemaSoftware.TramoTest.class" />
  </manifest>
</jar>
```

Además, como en el ejemplo, se puede anidar la tarea `<manifest>` para crear un fichero en el que se incluya información de interés. Por ejemplo, indicar cuál es la clase que contiene la función `main` dentro del `.jar`. Esta información se ordena en secciones y atributos. Una sección puede contener atributos. La estructura general de esta tarea es:

```
<manifest file="..."> <!-- nombre del fichero -->
  <attribute name="..." value="..." />
  ...
  <section name="...">
    <attribute name="Specification-Title" value="Example"/>
```

---

<sup>3</sup> Recordar que el servidor de tráfico está descrito en el Apéndice B.

```

    ...
  </section>
  ...
</manifest>

```

El contenido del fichero generado con la tarea `<manifest>` del ejemplo anterior sería el siguiente:

```

Manifest-Version: 1.0
Built-By: Daniel Bolaños

Main-Class: pruebasSistemaSoftware.TramoTest.class

```

### 3.3.3.2. Tareas de acceso al sistema de ficheros

Algunas de las tareas disponibles en Ant para acceder al sistema de archivos son:

- `chmod`: cambia los permisos de un archivo o un grupo de archivos (nótese que esta tarea es dependiente del sistema operativo ya que solo tiene sentido en Linux y similares).
- `copy`: copia un archivo o grupo de archivos desde una carpeta de origen a una de destino.
- `delete`: elimina un fichero, un directorio y todos sus ficheros y subdirectorios o un grupo de ficheros. En el ejemplo anterior, de uso de la tarea `<jar>`, se había definido la siguiente tarea `<delete>`:

```
<delete verbose="true" file="${build}/pruebasSistemaSoftware.jar"/>
```

Los atributos usados en este ejemplo indican:

- `verbose`: indica si se muestra el nombre de cada fichero borrado.
- `file`: especifica el nombre del fichero a borrar mediante un simple nombre, una ruta relativa o absoluta.
- `dir`: indica el directorio a borrar, incluyendo sus ficheros y subdirectorios.

También pueden anidarse grupos de ficheros o directorios. Por ejemplo:

```

<delete>
  <fileset dir="." includes="**/*.bak"/>
</delete>

```

que borra todos los ficheros con extensión `.bak` del directorio actual y cualquiera de sus subdirectorios.

- `mkdir`: crea un directorio. No hace nada si el directorio ya existe. Esta tarea sólo tiene un atributo, `dir`, que especifica el nombre del directorio que se crea.
- `move`: mueve un archivo o grupo de archivos desde una localización a otra.



### 3.3.3.3. Tareas de compilación

Para compilar el código fuente Java, Ant dispone de la tarea `<javac>`. Esta tarea tiene numerosos atributos que se pueden consultar en [1] y [3]. Algunos de los más habituales son los siguientes:

- `srcdir`: especifica dónde encontrar los ficheros que se van a compilar.
- `destdir`: especifica el directorio donde se dejarán las clases compiladas.
- `classpath`: indica el *classpath* a utilizar.
- `debug`: indica si el código fuente se debe compilar con información de depuración. Por omisión no se incluye esta información.
- `deprecation`: indica si el código fuente se debe compilar con información acerca de los elementos *deprecated* utilizados.

Además, tiene todos los atributos de `<fileset>`, pues como se comentó en el Apartado 3.3.1.2, esta tarea forma grupos de ficheros de forma explícita.

La tarea `<javac>` dispone de un elemento anidado: `<compilerarg>`. Con este elemento se especifican argumentos de la línea de comandos para el compilador. Uno de los atributos de este elemento es `value` que especifica el argumento de línea de comandos que se quiere pasar al compilador.

A continuación, se muestra un ejemplo, en el que las clases a compilar están en el directorio definido en la propiedad `${src}`, las clases compiladas se dejarán en el directorio `${build}`, el *classpath* a utilizar es el definido previamente en el fichero Ant con el nombre `project.class.path` y al compilador de Java se le pasa el argumento `Xlint`. Este argumento sirve para habilitar información sobre *warnings* no-críticos.

```
<javac srcdir="${src}" destdir="${build}">
  <classpath refid="project.class.path"/>
  <compilerarg value="-Xlint"/>
</javac>
```

### 3.3.3.4. Tareas de documentación

Es posible utilizar Ant para crear la documentación asociada a código fuente Java mediante la utilidad `javadoc` que esta embebida dentro de la tarea `<javadoc>`. Esta tarea simplemente encapsula la utilización de la herramienta de forma que es posible utilizarla dentro de un documento Ant. `<javadoc>` es posiblemente una de las tareas de Ant que tiene más opciones o atributos. Aquí solo se describirán algunos de ellos. Como se comentó al principio del Apartado 3.3.3 se puede encontrar la información detallada de los atributos de esta tarea en [1] y [3].

El siguiente ejemplo crea la documentación asociada al código fuente de las pruebas del sistema software descrito en el Apéndice B.

```
<javadoc packagenames="pruebasSistemaSoftware.*"
  sourcepath="${src}">
```

```

        destdir="${doc}">
        <classpath refid="project.junit42.class.path"/>
        <doctitle><![CDATA[<h1>Codigo de pruebas</h1>]]></doctitle>
</javadoc>

```

Los atributos usados en este ejemplo tienen el siguiente significado:

- **packagenames**: especifica la lista de paquetes a incluir.
- **sourcepath**: indica el directorio donde se encuentra el código fuente.
- **destdir**: indica el directorio de destino de los ficheros donde se almacenará la documentación generada.
- **classpath**: especifica dónde encontrar los ficheros de las clases de usuario.
- **doctitle**: indica el título que debe tener la página índice de la documentación generada. En el ejemplo se utiliza el segmento de código HTML `<![CDATA[ ... ]>` indicando que el título ha de ser “Código de pruebas”.

### 3.3.3.5. Tareas de ejecución

Estas tareas están relacionadas con la ejecución de programas. Existen tareas para:

- Invocar a la máquina virtual de Java (JVM), es decir, utilizar el comando `java` incluido en el JRE (Java Runtime Environment). El nombre de la tarea es `<java>` y algunos de sus atributos son:
  - **jar**: indica la localización del fichero `.jar` a ejecutar. Si se usa este atributo se debe usar también el atributo `fork`.
  - **fork**: si está a `true` se lanza la ejecución en el interior de otra máquina virtual. Por omisión está a `false`.
  - **classname**: indica una clase Java a ejecutar.

El siguiente ejemplo lanza la ejecución del servidor de tráfico:

```
<java jar="${build}/servidorEstadoTrafico.jar" fork="true"/>
```

- Ejecutar objetivos localizados en documentos Ant externos mediante la tarea `<ant>`. Con esta tarea se pueden crear subproyectos de forma que ficheros Ant demasiado grandes se puedan descomponer en varios subdocumentos Ant y de esta forma se hagan más manejables. Por ejemplo:

```
<ant antfile="build.xml" dir="../sistema software" target="makejar"/>
```

- **antfile**: especifica el fichero Ant donde está el objetivo que se quiere invocar. Si no se define, se toma por omisión el fichero `build.xml` del directorio especificado en **dir**.
- **dir**: especifica el directorio donde buscar el documento Ant referenciado.
- **target**: especifica el objetivo dentro del documento referenciado que se quiere ejecutar.

- Ejecutar comandos del sistema operativo: `exec`.
- Ejecutar varias tareas en paralelo: `parallel`. Se trata de una tarea que puede contener otras tareas que se ejecutarán en paralelo en un nuevo hilo de ejecución.
- Ejecutar tareas en secuencia: `Sequential`. Es una tarea que puede contener otras tareas Ant. Las tareas anidadas dentro de ella se ejecutan en secuencia. Se utiliza principalmente para permitir la ejecución secuencial de un subconjunto de tareas dentro de la tarea `<parallel>`. Esta tarea no tiene atributos y no permite el anidamiento de ningún elemento que no sea una tarea de Ant. En el siguiente ejemplo se muestra cómo la tarea `<sequential>` hace que se ejecuten en secuencia tres tareas mientras que otra se ejecuta en un hilo separado:

```
<parallel>
  <wlrn ... >
  <sequential>
    <sleep seconds="30"/>
    <junit ... >
    <wlstop/>
  </sequential>
</parallel>
```

- Suspender la ejecución de una tarea durante un tiempo: `sleep`. El tiempo de suspensión se especifica en sus atributos en horas, minutos, segundos o milisegundos. Por ejemplo:
- ```
<sleep milliseconds="100"/>
```
- Bloquear la ejecución hasta que se dan determinadas condiciones: `waitfor`. Su uso con la tarea `<parallel>` permite sincronizar varios procesos.

### 3.3.3.6. Tareas para la definición de propiedades

Estas tareas están relacionadas con la declaración y manejo de las propiedades. Una de ellas es `<property>`. Ejemplos de esta tarea se han utilizado en el Apartado 3.3.1.

### 3.3.3.7. Tareas relacionadas con las pruebas

También existen tareas relacionadas con las pruebas del software. Estas tareas requieren librerías externas. Es decir, no son parte de Ant. Entre ellas se pueden mencionar:

- `junit`: ejecuta pruebas de código Java mediante el *framework* JUnit. Esta tarea sólo funciona en las versiones de JUnit de la 3.0 en adelante. Nótese que es necesario instalar la versión 1.7 de Ant como mínimo para disponer de soporte para las versiones 4.x de JUnit.
- `junitreport`: genera un archivo XML combinando los archivos XML generados por la tarea `junit` en uno solo y aplica una hoja de estilo para construir un documento navegable con los resultados de la ejecución de las pruebas. En el Capítulo 6 se describe detalladamente cómo usar esta tarea.

Algunos de los atributos de `<junit>` son:

- `printsummary`: imprime estadísticas de cada prueba.

- `fork`: ejecuta las clases de prueba en una máquina virtual aparte.
- `haltonerror`: detiene el proceso si se detecta algún error durante la ejecución de las pruebas.
- `haltonfailure`: detiene el proceso si durante la ejecución de las clases de prueba se detecta algún fallo.
- `showoutput`: envía la salida generada por las pruebas al sistema desde el que se accede a Ant y a los *formatters*. Por omisión, sólo se envía a los *formatters*.

La tarea `<junit>` permite además el anidamiento de otros elementos. A continuación se enumeran algunos de ellos:

- `classpath`: establece dónde encontrar las clases necesarias para la ejecución de las pruebas.
- `formatter`: el resultado de las pruebas se puede mostrar en diferentes formatos. Los formatos predefinidos son: XML, `plain` (texto plano) y `brief` que solo muestra información detallada para los casos de prueba que fallan. El formato deseado se determina en el atributo `type`.
- `test`: indica el nombre de la clase de pruebas cuando solo existe una. Se puede utilizar junto con el elemento `<formatter>`.
- `batchtest`: define varias pruebas a ejecutar mediante un patrón sobre el nombre de las clases de prueba a ejecutar. Permite el anidamiento de elementos de definición de grupos de ficheros y directorios. En el atributo `todir` se almacenan los resultados de las pruebas. También puede usar el elemento `<formatter>`.
- `sysproperty`: especifica propiedades del sistema necesarias para las clases que se están probando. Estas propiedades están disponibles para la máquina virtual de Java mientras se ejecutan las pruebas. Entre los atributos disponibles están `key` y `value` que especifican propiedades y valores de propiedades.

### 3.3.3.8. Tareas definidas por el usuario

Una característica muy interesante de Ant es que permite al desarrollador definir nuevas tareas. Para ello se dispone de la tarea `<taskdef>`. La nueva tarea se puede utilizar en el resto del proyecto en el que ha sido definida. Para crear una nueva tarea es necesario definir el método `execute` en la clase que implementa la tarea. `<taskdef>` tiene numerosos atributos, entre ellos: `name` que da nombre a la tarea y `classname` que indica la clase Java que implementa dicha tarea. Para indicar dónde se encuentra esta clase se puede anidar el elemento `<classpath>`.

En el sistema software descrito en el Apéndice B, se ha definido una tarea Ant que ayuda en la automatización de las pruebas funcionales. Esta tarea se encarga de detener el servidor del estado del tráfico una vez que las pruebas funcionales han terminado. De esta forma, el arranque y detención del sistema se puede hacer desde dentro del propio documento Ant de forma completamente automatizada y sin intervención manual. La tarea definida recibe el nombre de `<detener-servidor>` y el código Ant utilizado para definirla es el siguiente:

```

<taskdef name="detener-servidor"
    classname="pruebasSistemaSoftware.anttasks.DetenerServidorXMLTask">
    <classpath>
        <pathelement location="${build}/pruebasSistemaSoftware.jar"/>
    </classpath>
</taskdef>

```

A continuación se lista el código de la clase que implementa esta tarea Ant.

pruebas sistema software/src/pruebasSistemaSoftware/anttasks/DetenerServidorXMLTask.java

```

package pruebasSistemaSoftware.anttasks;

import org.apache.tools.ant.BuildException;
import org.apache.tools.ant.Task;

import java.net.*;
import java.io.InputStream;

public class DetenerServidorXMLTask {

    private String m_strURL;
    private String m_strUsuario;
    private String m_strPassword;

    public DetenerServidorXMLTask() {
    }
    /**
     * Metodo responsable de la ejecucion de la tarea
     */
    public void execute() throws BuildException {

        try {

            String strURLCompleta = this.m_strURL + "?usuario=" +
                this.m_strUsuario + "&password=" + this.m_strPassword;
            java.net.URL url = new java.net.URL(strURLCompleta);
            HttpURLConnection con = (HttpURLConnection)
                url.openConnection();
            con.setRequestMethod("GET");
            con.setDoInput(true);
            //Lectura de la pagina html de respuesta
            byte respuesta[] = new byte[10000];
            int iBytes;
            InputStream input = con.getInputStream();
            do {
                iBytes = input.read(respuesta);
            } while(iBytes != -1);
            input.close();

```

```

        } catch (Exception e) {
            System.out.println("Error al ejecutar la tarea <detener-servidor>");
        }
    }

    /**
     * Recibe el parametro "url" de la tarea
     */
    public void setURL(String strURL) {

        this.m_strURL = strURL;
    }

    /**
     * Recibe el parametro "usuario" de la tarea
     */
    public void setUsuario(String strUsuario) {

        this.m_strUsuario = strUsuario;
    }

    /**
     * Recibe el parametro "password" de la tarea
     */
    public void setPassword(String strPassword) {

        this.m_strPassword = strPassword;
    }
}

```

El funcionamiento de esta clase es bien sencillo, simplemente se han de definir tres métodos (`setURL`, `setUsuario` y `setPassword`) que sirven para recibir desde el documento Ant los correspondientes parámetros asociados a la tarea (`url`, `usuario` y `password`). Posteriormente, se define un método llamado `execute` (debe llamarse así para que Ant pueda reconocerlo) que se encarga de realizar las operaciones propias de la tarea. En este caso, simplemente, abre una conexión HTTP, con el servidor y le envía una petición indicándole que ha de detenerse<sup>4</sup>. Adicionalmente se pueden realizar comprobaciones sobre la respuesta que el servidor proporciona a tal petición.

### 3.3.3.9. Otras tareas

Existen otras muchas tareas que se pueden agrupar en:

- Tareas de auditoría y cobertura, relacionadas con métricas de diseño de software Java.
- Tareas SCM, relacionadas con la gestión del código fuente, aunque no sólo.
  - `cvs`: permite ejecutar comandos sobre un repositorio Cvs.
  - `cvschangelog`: genera un informe en formato XML con los cambios almacenados en un repositorio Cvs.
  - `microsoft Visual SourceSafe`: permite realizar operaciones sobre un repositorio de este tipo.

---

<sup>4</sup> En el Apéndice B se describe el formato de las posibles peticiones que puede recibir el sistema.

- Tareas para acceso remoto mediante diferentes protocolos.
  - `ftp`: implementa la funcionalidad básica de un cliente FTP.
  - `telnet`: permite establecer sesiones remotas mediante el protocolo Telnet.
  - `sshexec`: permite la ejecución remota de comandos mediante SSH.
- Tareas que permiten el almacenamiento en ficheros de la salida de la ejecución de los documentos Ant (*logging tasks*).
- Tareas que permiten enviar correos electrónicos a través de un servidor SMTP.
- Tareas específicas para trabajar con el *framework* .NET.
- Tareas EJB relacionadas con el uso de Java Beans.

## 3.4. Creación de un proyecto básico

En este apartado se va a describir cómo crear un proyecto básico, utilizando los objetivos, tareas y propiedades definidos en los apartados anteriores. Es decir, como organizar el contenido de un documento Ant.

Para empezar se verá cómo compilar código fuente Java. El primer paso es preparar el directorio en el que se van a guardar las clases compiladas. Esto se realizará como un objetivo diferente para poder ejecutar los objetivos separadamente si fuera necesario. Por ejemplo, no conviene mezclar dentro de un mismo objetivo la compilación y la ejecución puesto que durante determinadas fases de la etapa de desarrollo solo interesa compilar y no ejecutar. Es decir, se trata de flexibilizar el uso del documento Ant.

```
<?xml version="1.0"?>.
<project name="prueba" default="compilar" basedir=".">

  <!-- propiedades globales del proyecto -->
  <property name="src" value="."/>
  <property name="build" value="build"/>

  <!-- Inicialización -->
  <target name="init">
    <mkdir dir="${build}"/>
  </target>

  <!-- Compilación del código java -->
  <target name="compilar" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

</project>
```

Una vez definido el documento Ant sólo queda ejecutar el objetivo “compilar” para compilar el código. Situados en el directorio donde se encuentra dicho fichero, se ha ejecutar la siguiente orden:

```
c:\>ant
```

Nótese que no se ha indicado explícitamente el objetivo a ejecutar, sin embargo todo funciona de la forma esperada porque se ha declarado `compile` como el objetivo por omisión (atributo `default` de `project`). Pero la regla general es:

```
c:\>ant nombre_objetivo
```

Al ejecutarse el objetivo `compile`, Ant consulta el atributo `depends`. En este caso tiene como valor `init`, lo que significa que antes de ejecutar el objetivo `compile` se ejecutará el objetivo `init` que crea el directorio donde se almacenarán las clases compiladas.

Se puede ampliar el ejemplo anterior para ver cómo eliminar las clases de versiones anteriores, cómo generar la documentación con `javadoc` y cómo se resuelve la dependencia entre los distintos objetivos. El siguiente documento Ant compila diversas clases Java y las coloca en un archivo `.jar`. Para lograrlo se han definido los siguientes objetivos:

- **limpiar.** Se eliminan los directorios que contienen las clases compiladas y la documentación que pudo ser generada anteriormente. Se eliminan ambos a la vez para que no exista una versión de documentación que no se corresponde con las clases compiladas y viceversa.
- **compile.** En la primera de sus tareas se crea, si no existe, el directorio donde se guardarán las clases compiladas. El directorio no existirá porque se ha eliminado con el objetivo `limpiar`, del cual depende `compile`. La segunda tarea compila las clases, situadas en `src` y almacena los resultados de la compilación en el directorio `build`. Para ello utiliza el `classpath` definido en la propiedad `classpath`.
- **jar.** Se genera un archivo `.jar` con todas las clases presentes en el directorio `destino` (`build`). Esta tarea depende del objetivo `compile` para garantizar que existan clases compiladas en este directorio antes de intentar generar el archivo `.jar`.
- **documentar.** En la primera tarea se crea, si no existe, el directorio donde se guardará la documentación generada. El directorio no existirá porque se ha eliminado con el objetivo `limpiar`, del cual depende `compile`. La segunda tarea genera la documentación del proyecto.
- **main.** Se ejecutan todos los objetivos para crear el programa y dejarlo listo para ser ejecutado.

Previamente, se han definido varias propiedades que definen los directorios que se usarán a lo largo del proyecto Ant.

```
<?xml version="1.0"?>

<project name="Proyecto_Basico" default="main" basedir=".">

    <!-- propiedades globales del proyecto -->
    <!-- definición de directorios que se van a usar -->
    <property name="src" value="." />
    <property name="build" value="build" />
    <property name="doc" value="doc" />

    <!-- características del proyecto -->
    <property name="classpath" value="lib/misclases.jar" />
```



```

<target name=limpiar>
  <delete dir="${build}" />
  <delete dir="${doc}" />
</target>

<target name=compilar depends="limpiar">
  <mkdir dir="${build}" />
  <javac srcdir="${src}"
        destdir="${build}"
        classpath="${classpath}" />
</target>

<target name=jar depends="compilar">
  <jar jarfile="clases.jar" basedir="${build}" includes="**"/>
</target>

<target name=documentar depends="limpiar">
  <mkdir dir="${doc}" />
  <javadoc packagenames="proyectobasico"
          sourcepath="${src}"
          destdir="${doc}" />
</target>

<target name=main depends="compilar, documentar">
</target>

</project>

```

## 3.5. Ejecución de los casos de prueba mediante Ant

Ahora es el momento de centrarse en cómo se pueden automatizar las pruebas de software mediante el uso de Ant y las tareas que proporciona para tal efecto. A partir de ahora para ilustrar todos los ejemplos se utilizará el sistema software descrito en el Apéndice B.

Antes de empezar a desarrollar el software de pruebas hay que establecer la estructura de directorios en la que se va a organizar el código de pruebas. En el caso de las pruebas del servidor de tráfico la estructura se mostró en la Figura 2.3 del Apartado 6 del Capítulo 2. El fichero Ant, en el ejemplo llamado `build.xml`, se suele colocar en el directorio principal del proyecto. En este caso “pruebas sistema software”. En este apartado se analiza el contenido de este fichero<sup>5</sup>.

Como ya se ha comentado anteriormente conviene definir en objetivos separados las diferentes operaciones que se quieren realizar, con el fin de hacer flexible el fichero Ant. De esta forma, se podrán ejecutar esas operaciones de forma aislada si así interesa, o varias de una vez según se hayan definido las dependencias entre objetivos. Algunos de los objetivos más habituales que se pueden definir en Ant para realizar las pruebas son:

- Inicialización del entorno para prepararlo para las pruebas. Suele incluir la creación de directorios necesarios para ejecutar las pruebas. Por ejemplo, para almacenar las clases compiladas, la documentación generada, etc.

---

<sup>5</sup> El documento completo se encuentra en la ruta `pruebas sistema software/build.xml`.

- Limpieza del entorno de las pruebas. Puede incluir tareas de borrado de directorios generalmente creados en otros objetivos. Por ejemplo, de los directorios de clases compiladas o de documentación generada. El fin de este objetivo es asegurarse de que, cuando se ejecutan nuevas pruebas o pruebas de regresión después de corregido el código, se hayan borrado versiones anteriores.
- Compilación del código de pruebas.
- Creación del fichero .jar del código de pruebas.
- Generación de la documentación.
- Ejecución del sistema que se quiere probar.
- Ejecución de las pruebas: unitarias, funcionales, etc. Este objetivo incluye la generación de informes de pruebas, tanto en formato HTML, como en formato PDF. Sin embargo los objetivos de documentación no tienen porqué estar ligados a los de ejecución de las pruebas y a menudo están separados. En grandes proyectos software no conviene generar la documentación hasta que vaya a ser utilizada puesto que es una tarea que puede consumir bastante tiempo.

Antes de la definición de los objetivos, el documento Ant suele contener un bloque de líneas de código para la definición de propiedades globales del proyecto, definición del *classpath*, etc. Es decir, de todo lo que va a ser usado en el fichero Ant para lograr los objetivos que permiten la construcción y prueba del sistema software. Por ejemplo, para las pruebas del servidor de tráfico las propiedades globales del proyecto definen los directorios que se van a usar y una propiedad llamada *user* que será referenciada en algunas de las tareas posteriores:

```
<property name="src" value="./src"/>
<property name="build" value="./build"/>
<property name="reports" value="./reports"/>
<property name="lib" value="./lib"/>
<property name="config" value="./config"/>
<property name="doc" value="./doc"/>
<property name="testData" value="./testData"/>
<property name="cobertura" value="cobertura" />
<property name="user" value="Daniel Bolanos"/>
```

Los directorios *src*, *lib*, *config* y *testdata* contienen, respectivamente, los ficheros con el código fuente de las pruebas, las librerías de las herramientas de pruebas que se van a usar, los datos de configuración y los ficheros con los casos de prueba. Los directorios *build*, *reports* y *doc* se crean después en la etapa de inicialización (objetivo *init*) para almacenar las clases ya compiladas y empaquetadas en un archivo .jar, los informes con los resultados de la ejecución de las pruebas y la documentación del código de pruebas. Estos tres directorios se pueden borrar invocando el objetivo de limpieza (*clean*) siempre que se desee o sea necesario. El directorio *cobertura* contendrá las clases instrumentalizadas sobre las que se analiza la cobertura alcanzada por las pruebas<sup>6</sup>.

---

<sup>6</sup> Consultar el Apartado 6.3.1.2.

El *classpath* para este ejemplo incluye todas las librerías necesarias para ejecutar las herramientas de pruebas que no forman parte del núcleo de Ant. La definición de dicho *classpath* es la siguiente:

```
<path id="project.common.class.path">
  <pathelement location="../sistema software/build/servidor
    EstadoTrafico.jar"/>
  <pathelement location="\${lib}/jfunc11/jfunc.jar"/>
  <pathelement location="\${lib}/junitaddons14/junit-addons-
    1.4.jar"/>
  <pathelement location="\${lib}/jtestcase40/jaxen-core.jar"/>
  <pathelement location="\${lib}/jtestcase40/jaxen-jdom.jar"/>
  <pathelement location="\${lib}/jtestcase40/org.jicengine-
    2.0.0.jar"/>
  <pathelement location="\${lib}/jtestcase40/jdom-1.0.jar"/>
  <pathelement location="\${lib}/jtestcase40/saxpath.jar"/>
  <pathelement location="\${lib}/jtestcase40/jtestcase_4.0.0.jar"/>
  <pathelement location="\${lib}/httpunit-1.6.2/httpunit.jar"/>
  <pathelement location="\${lib}/xmlunit1.0/xmlunit.jar"/>
  <pathelement location="\${lib}/jmock1.1/jmock-1.1.0.jar"/>
  <pathelement location="\${lib}/jmock1.1/jmock-cglib-1.1.0.jar"/>
  <pathelement location="\${lib}/cglib2.1.3/cglib-nodep-
    2.1_3.jar"/>
  <pathelement location="\${lib}/easymock2.2/easymock.jar"/>
  <pathelement location="\${lib}/easymockclassexension2.2.1/
    easymockclassexension.jar"/>
  <pathelement location="\${lib}/dbunit-2.2/dbunit-2.2.jar"/>
  <pathelement location="\${lib}/mysql/mysql-connector-java-
    5.0.5-bin.jar"/>
  <pathelement location="\${lib}/cobertura-1.8/cobertura.jar"/>
  <pathelement location="\${lib}/jwebunit-1.4/jwebunit-core-1.4.jar"/>
  <pathelement location="\${lib}/jwebunit-1.4/jwebunit-htmlunit-
    plugin-1.4.jar"/>
    <fileset dir="\${lib}/jwebunit-1.4/lib">
      <include name="*.jar"/>
    </fileset>
  <pathelement location="\${lib}/htmlunit-1.11/htmlunit-1.11.jar"/>
  <pathelement location="\${lib}/htmlunit-1.11/commons-httpclient-
    3.0.1.jar"/>
    <fileset dir="\${lib}/htmlunit-1.11">
      <include name="*.jar"/>
    </fileset>
  <pathelement location="\${build}"/>
  <pathelement location="\${build}/pruebasSistemaSoftware.jar"/>
  <pathelement path="\${java.class.path}"/>
</path>
```

Una vez preparado el entorno en el que se van a realizar las pruebas, se definen los objetivos. Para el ejemplo de las pruebas del servidor de tráfico se han definido, entre otros, los siguientes:

- Inicialización del entorno de pruebas. Se crean los directorios para almacenar las clases compiladas `\${build}` y los informes de pruebas `\${reports}/junitreport` y

`${reports}/pdfreport`. Además, se crea el fichero `.jar` del sistema software a probar para asegurar que al ejecutar las pruebas sobre él dicho fichero está disponible.

```
<target name="init">
  <mkdir dir="${build}"/>
  <mkdir dir="${reports}"/>
  <mkdir dir="${reports}/junitreport"/>
  <mkdir dir="${reports}/pdfreport"/>
  <!-- Para asegurar que el sistema está compilado y el jar
        disponible-->
  <ant antfile="build.xml" dir="../../sistema software" target=
    "makejar"/>
</target>
```

- Limpieza del entorno de las pruebas. Se borran los directorios de clases compiladas y de ficheros `.jar` (`${build}`) o de documentación generada (`${reports}`, `${doc}`, `${cobertura}`). El fin de este objetivo es asegurarse de que, cuando se ejecutan nuevas pruebas o pruebas de regresión después de corregido el código, se hayan borrado versiones anteriores.

```
<target name="clean">
  <delete verbose="true" dir="${build}"/>
  <delete verbose="true" dir="${reports}"/>
  <delete verbose="true" dir="${cobertura}"/>
  <delete verbose="true" dir="${doc}"/>
</target>
```

- Compilación del código de pruebas. Las dos primeras tareas de este objetivo compilan el código de pruebas unitarias con las dos versiones de JUnit que se tratan en este libro. La tercera compila el código de las pruebas funcionales y la última compila el código que implementa las nuevas tareas definidas para Ant<sup>7</sup>.

```
<target name="compile" depends="init">
  <javac srcdir="${src}/pruebasSistemaSoftware/junit381"
    destdir="${build}">
    <classpath refid="project.junit381.class.path"/>
    <compilerarg value="-Xlint"/>
  </javac>

  <javac srcdir="${src}/pruebasSistemaSoftware/junit42"
    destdir="${build}">
    <classpath refid="project.junit42.class.path"/>
    <compilerarg value="-Xlint"/>
  </javac>

  <javac srcdir="${src}/pruebasSistemaSoftware/funcionales"
    destdir="${build}">
    <classpath refid="project.junit42.class.path"/>
    <compilerarg value="-Xlint"/>
  </javac>
```

---

<sup>7</sup> Ver la tarea `<taskdef>` en el Apartado 3.3.3.8.

```

    <javac srcdir="${src}/pruebasSistemaSoftware/antTasks"
        destdir="${build}">
        <compilerarg value="-Xlint"/>
    </javac>
</target>

```

- Creación del fichero .jar del código de pruebas. Lo primero que se hace en este objetivo es borrar la versión anterior del fichero .jar. Después se crea dicho fichero según se explicó en el Apartado 3.3.3.1.

```

<target name="makejar" depends="compile">
    <delete verbose="true" file="${build}/pruebasSistema
        Software.jar"/>
    <jar destfile="${build}/pruebasSistemaSoftware.jar"
        basedir="${build}">
        <manifest>
            <attribute name="Built-By" value="${user}" />
            <attribute name="Main-Class"
                value="pruebasSistemaSoftware.TramoTest.class" />
        </manifest>
    </jar>
</target>

```

- Generación de la documentación. Primero se crea el directorio en el que se guardará la documentación generada. Después se crea dicha documentación según se vio en el Apartado 3.3.3.4.

```

<target name="generatedoc">
    <mkdir dir="${doc}" />
    <javadoc packagenames="pruebasSistemaSoftware.*"
        sourcepath="${src}"
        destdir="${doc}">
        <classpath refid="project.junit42.class.path"/>
        <doctitle><![CDATA[<h1>Codigo de pruebas</h1>]]></doctitle>
    </javadoc>
</target>

```

- Ejecución del sistema que se quiere probar. Para probar el sistema de tráfico hay que ejecutarlo. Esto se realiza llamando a la tarea run del fichero Ant que se definió para construir el programa objeto de las pruebas.

```

<target name="run">
    <ant antfile="build.xml" dir="../sistema software" target="run"/>
</target>

```

- Ejecución de las pruebas: unitarias, funcionales, etc. Este objetivo incluye la generación de informes de pruebas, tanto en formato HTML, como en formato PDF.

El objetivo que ejecuta las pruebas unitarias para el servidor de tráfico, se define mediante tres tareas:

- JUnit para ejecutar la batería de pruebas. Se han incluido en la batería de pruebas todas las clases con el sufijo `Test.class`. (`*Test.class`). Para cada una de estas clases se generará un informe en la carpeta `${reports}` con el resultado de su ejecución.

- JUnitReport y JUnitPDFReport para generar el informe con el resultado de la ejecución de la batería de pruebas en formato HTML y PDF respectivamente. En el Capítulo 6 se dan más detalles sobre el funcionamiento y utilización de estas herramientas.

```
<!-- Ejecucion de las pruebas unitarias-->
<target name="junit" depends="compile" description="Execute Unit Tests">
    <junit printsummary="yes" fork="true" haltonerror="no"
        haltonfailure="no" showoutput="yes">
        <formatter type="xml"/>
        <classpath location="${cobertura}/instrumented-
            classes/servidorEstadoTrafico.jar" />
        <sysproperty key="net.sourceforge.cobertura.datafile"
            file="${cobertura}/cobertura.ser" />
        <classpath refid="project.junit381.class.path"/>
        <batchtest todir="${reports}/junitreport">
            <fileset dir="${build}">
                <include
                    name="pruebasSistemaSoftware/junit381/*Test.class" />
            </fileset>
        </batchtest>
        <classpath refid="project.junit42.class.path"/>
        <batchtest todir="${reports}/junitreport">
            <fileset dir="${build}">
                <include
                    name="pruebasSistemaSoftware/junit42/*Test.class" />
            </fileset>
        </batchtest>
    </junit>

    <!-- Generacion de informes de resultados en formato html -->
    <junitreport todir="${reports}/junitreport">
        <fileset dir="${reports}/junitreport">
            <include name="TEST-*.xml" />
        </fileset>
        <report format="frames" todir="${reports}/junitreport"/>
    </junitreport>

    <!-- Generacion de informes de resultados en formato pdf -->
    <junitpdfreport todir="${reports}/pdfreport" styledir="structural">
        <fileset dir="${reports}/junitreport">
            <include name="TEST-*.xml" />
        </fileset>
    </junitpdfreport>
</target>
```

El objetivo que ejecuta las pruebas funcionales para el servidor de tráfico lanza la ejecución del sistema de tráfico y las pruebas en paralelo. Para ello se utiliza la tarea `<parallel>` con dos hilos de ejecución:

- Ejecución del sistema a probar. Se llama a la tarea `run` del fichero `Ant` del servidor de tráfico.

- Ejecución de las pruebas funcionales. Este hilo ejecuta varias tareas de forma secuencial. La secuencia queda indicada por la tarea `<sequential>`:
  - Espera a que el servidor de tráfico se haya puesto en marcha (tarea `<sleep>`).
  - Ejecuta las pruebas con la tarea `<junit>`. Los resultados de la prueba se guardarán en formato XML (tarea `<formatter>`). Después se define el *classpath* donde se encuentran las clases instrumentadas para analizar la cobertura de las pruebas<sup>8</sup>, y, mediante una referencia al *path* `project.junit42.class.path` definido al principio del documento Ant, las librerías necesarias para ejecutar las pruebas. Finalmente se ejecuta la batería de pruebas con la tarea `<batchtest>`, incluyendo todos los ficheros cuyo nombre termina en “Test.class”.
  - Cuando se han ejecutado todos los casos de prueba se detiene el servidor. Para detenerlo se invoca una tarea previamente incluida al comienzo de este mismo objetivo: `<detener-servidor>`.

```

<!-- Ejecución de las pruebas funcionales -->
<target name="functional" depends="makejar" description="Ejecucion de
las pruebas funcionales" >

    <!-- definición de la tarea para la detener el sistema a probar -->
    <taskdef name="detener-servidor"
        classname="pruebasSistemaSoftware.anttasks.Detener
        ServidorXMLTask">
        <classpath>
            <pathelement location="${build}/pruebasSistemaSoftware.jar"/>
        </classpath>
    </taskdef>

    <!-- Ejecución del sistema y pruebas funcionales en paralelo -->
    <parallel>
        <!-- ejecución del sistema a probar -->
        <ant antfile="build.xml" dir="../sistema software" target="run"/>
        <!-- ejecución de las pruebas funcionales -->
        <sequential>
            <!-- espera de inicialización del servidor -->
            <sleep milliseconds="100"/>
            <junit printsummary="yes" fork="true" haltonerror="no"
                haltonfailure="no" showoutput="yes">
                <formatter type="xml"/>
                <classpath location="${cobertura}/instrumented-classes/
                    servidorEstadoTráfico.jar" />
                <sysproperty key="net.sourceforge.cobertura.datafile"
                    file="${cobertura}/cobertura.ser" />
                <classpath refid="project.junit42.class.path"/>
                <batchtest todir="${reports}/junitreport">
                    <fileset dir="${build}">
                        <include name="pruebasSistemaSoftware/funcionales/
                            *Test.class"/>
                    </fileset>
                </batchtest>
            </junit>
        </sequential>
    </parallel>

```

---

<sup>8</sup> Para ver en detalle el análisis de la cobertura de las pruebas consultar el Apartado 6.3.1.2.

```

        </fileset>
    </batchtest>
</junit>
<detener-servidor url="http://localhost:1100/servidorEstado-
                    Trafico/" usuario="administrador"
                    password="12345"/>

</sequential>
</parallel>

<!-- Generación de informes de resultados en formato html -->
<junitreport todir="${reports}/junitreport">
    <fileset dir="${reports}/junitreport">
        <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames" todir="${reports}/junitreport"/>
</junitreport>

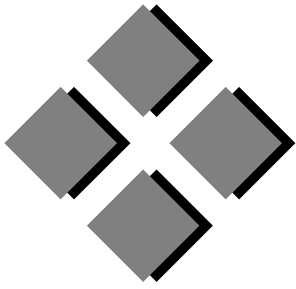
</target>

```

### 3.6. Bibliografía

- Holzner, S.: *Ant: The Definitive Guide*, segunda edición, O'Reilly Media, 2005.
- Hatcher, E. y Loughran, S.: *Java Development with Ant*, Manning Publications, 2003.
- *Apache Ant 1.7.0 Manual Online*, <http://ant.apache.org/manual/index.html>





# Capítulo 4

## Gestión de la configuración del Software

---

### SUMARIO

4.1. Introducción	4.6. Control de cambios
4.2. Principios básicos	4.7. Herramientas de GCS
4.3. Objetivos	4.8. Documentación
4.4. Líneas Base	4.9. Bibliografía
4.5. Actividades	

## 4.1. Introducción

Teniendo en cuenta que a todo proceso de pruebas le sigue comúnmente un proceso de corrección de errores y esto deriva en cambios en el software, se ha encontrado justificado introducir un breve capítulo de la gestión de configuración del software en este texto.

Se define Gestión de la Configuración del Software (GCS) como el proceso o conjunto de actividades que permite identificar y definir los objetos de un sistema, controlando sus cambios a lo largo de su ciclo de vida y en todas sus fases, registrando e informando de los estados de dichos objetos y de los cambios y verificando su corrección y su completitud.

La gestión de la Configuración del Software se define como la disciplina cuya misión es identificar, controlar y organizar la evolución de un sistema software, teniendo en cuenta que un sistema software está básicamente compuesto por código, documentación y datos.

En este capítulo se dará un repaso a los principios y objetivos principales de la GCS, continuando con una definición de conceptos básicos y útiles para la GCS de un proyecto software, tales como las líneas base. Seguidamente, se describen las actividades que se han de llevar a cabo durante el proceso de GCS, finalizando con las aportaciones que ofrecen las herramientas actuales de GCS y la documentación que es necesario generar durante este proceso.

## 4.2. Principios básicos

Se define Configuración del Software como el estado actual de un sistema software y las interrelaciones entre sus componentes constitutivos, que típicamente son, como ya se ha comentado, código, datos y documentación.

Dentro de la Ingeniería del Software, la GCS no es una actividad técnica, sino de gestión. Sin embargo, esta tarea de gestión está habitualmente asistida por una serie de herramientas, tal y como se verá en el Capítulo 5 “Herramientas de control de versiones”. Las actividades de la GCS permiten controlar formalmente la evolución de un sistema software o de un elemento software, garantizando la visibilidad en el desarrollo, y la trazabilidad en el producto. Así, permite conocer el estado del producto en la fase en la que se encuentra y en las etapas anteriores.

A continuación, se definirán algunos términos relevantes de GCS:

- **Repositorio:** el repositorio es una librería centralizada de ficheros y proporciona el control de versiones y la forma de acceder a dichas versiones. Los cambios en el repositorio no se hacen sobre el fichero almacenado como tal, sino generando una nueva versión del objeto que se vuelve a introducir en el repositorio. Dicha generación se hace a través de procesos que implica obtener objetos para modificar (*check out*) y que implica registrar la nueva versión (*check in*). Los usuarios pueden, por tanto, recuperar cualquier revisión del objeto. Existen múltiples herramientas automáticas, tanto comerciales como de libre distribución, que permiten realizar estas acciones incluyendo la propia creación del repositorio.
- **Solicitud de cambio:** el modelado de solicitud de cambio consiste en implementar un flujo de estados y una serie de roles y responsabilidades que actúan sobre un formulario de solicitud de cambio, que puede ser una solicitud de una evolución del sistema o bien la necesidad de una corrección. Según va pasando por determinados estados, los diferentes implicados actúan según sus roles. Para el ámbito de este texto, es suficiente únicamente el concepto de solicitud de cambio en términos del formulario de solicitud o petición del mismo por cualquier motivo justificado. Más adelante, se definirá su contenido básico.
- **Versión:** instancia de un sistema software que difiere en algo de otras instancias (nuevas funcionalidades, correcciones sustanciales, opera en hardware diferente, etc.). Nótese que el concepto versión es también aplicable a cada uno de los elementos software que componen dicho sistema software. De esta forma, una determinada versión de un sistema software se corresponderá con una determinada versión de cada uno de sus elementos software constituyentes.
- **Variante:** versiones entre las que existen pequeñas diferencias.

## 4.3. Objetivos

Muchos sistemas software tienen una vida larga desde que son implantados y comienza su operación hasta su retirada completa. Más aún, todos los sistemas software cambian a lo largo de su vida (diferente versión, distinta plataforma, mejoras, nuevas funcionalidades, nuevas normativas, etc.). Todos estos cambios se producen no solo en la fase de desarrollo sino también durante la fase de mantenimiento. Es necesario asegurar que los cambios producidos ocasionen el mínimo coste. Este es el propósito de la GCS.

Como puntos fundamentales, aunque no exhaustivos, la GCS asegura:

- Coherencia entre versiones.
- Seguridad ante pérdidas de software o de personal.
- Reutilización del software, tanto en el tiempo (versiones futuras) como en el espacio (otros proyectos).
- Recuperación de cualquier versión realizada por cualquier desarrollador en cualquier momento.
- Calidad. En este sentido se aceptan únicamente los elementos formalmente revisados y aprobados.

Una GCS debe ser capaz de contestar a preguntas tales como: ¿cuántas versiones existen del producto software o de un componente u objeto en particular?; ¿cómo se recupera una versión en particular?; ¿quién la hizo y cuándo?; ¿qué evolución ha sufrido?; ¿qué otras versiones o componentes se ven afectadas si se modifica un componente del sistema software?; ¿cómo y a quién se distribuyó una determinada versión?; ¿cómo se puede asegurar que realiza un cambio de manera consistente, sin solapamientos ni interferencias?; etc.

En definitiva, el objetivo de la disciplina de GCS es establecer y mantener la integridad de los productos generados durante un proyecto de desarrollo de software y a lo largo de todo el ciclo de vida.

Como objetivos secundarios y derivados de lo anterior se puede enumerar los siguientes:

- Identificar la configuración del software en las distintas fases, incluyendo la identificación de posibles cambios.
- Gestionar y controlar sistemáticamente los cambios que se efectúen en el software y que afecten a su configuración trazando los elementos y sus dependencias.
- Garantizar que los cambios se implantan adecuadamente.
- Garantizar que la documentación está actualizada de forma permanente.
- Informar de los cambios a todos aquellos a los que les afecte.
- Seguir la evolución del software a lo largo de su ciclo de vida manteniendo su integridad.

A continuación, se describen los conceptos de Elemento de Configuración y de Línea Base, ambos esenciales para la definición de una estrategia de GCS en un proyecto software.

## 4.4. Líneas base

### 4.4.1. Elementos de configuración

Se denomina elemento de configuración del software (ECS) a cada uno de los componentes básicos de un producto software sobre los que se realizará un control a lo largo de su ciclo de vida. Todos los ECS tienen un nombre asignado y pueden evolucionar. Los elementos de la configuración del software se organizan como objetos de configuración con su correspondiente nombre, descripción, atributos y relaciones.

Cuando un ECS se convierte en una línea base (este concepto se explica en la siguiente sección) se introduce en el repositorio que se denomina Repositorio del Proyecto o Base de Datos del Proyecto si se habla a nivel de proyecto. Otra posibilidad frecuente es trabajar con productos, con lo que en este caso, pasarían a ser Repositorio o Base de Datos de Producto.

Ejemplos de ECS son el plan de proyecto, el documento de especificación de requisitos software, los documentos de diseño, el plan de pruebas, el código fuente, el manual de usuario, etc.

### 4.4.2. Líneas base

Se define formalmente línea base como una configuración de referencia en el proceso de desarrollo del software a partir de la cual las revisiones se han de realizar de manera formal. El objetivo es, por tanto, controlar los cambios en el software, sin impedir llevar a cabo aquellos que sean justificados. Es decir, hay que hacer notar que, en ningún caso, el uso de líneas base impide realizar cambios; al contrario, los lleva a cabo de una manera metódica y procedimentada asegurando, además, que se hacen todos aquellos que deben realizarse y de una forma controlada.

Las líneas base no impiden realizar un cambio, sino ejecutar todos aquellos cambios justificados de forma controlada.

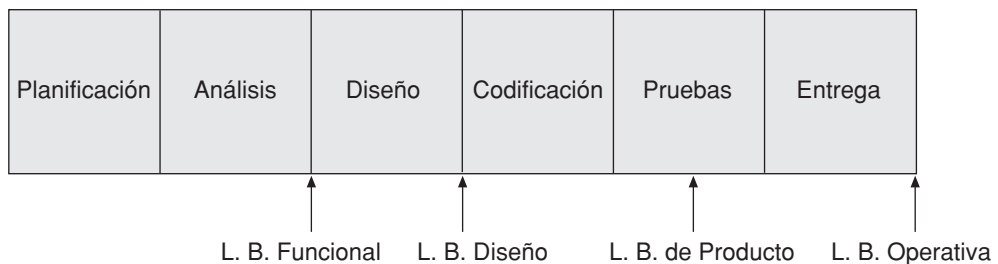
Como objetivos secundarios, derivados de la propia definición, se pueden enumerar los siguientes:

- Identificar los resultados de las tareas realizadas en cada fase del ciclo de desarrollo.
- Asegurar que se ha completado cada fase en el ciclo de desarrollo.
- Servir como punto de partida para los desarrollos y etapas posteriores.
- Servir como punto de partida para las peticiones de cambio.

Las líneas base se definen, junto con qué objetos o productos van a ser elementos de configuración, al inicio del proyecto, en la fase de planificación. Pueden coincidir o no con los hitos fijados para el proyecto o pueden corresponderse con el final de las distintas fases del ciclo de vida. En general, se definen de acuerdo con la estructura y naturaleza del proyecto. Las líneas base engloban todos los productos o elementos que se hayan generado entre la línea base anterior y la actual, que vayan a evolucionar en el tiempo y que interese tener un control sobre dicha evolución. Así, las líneas base más usuales y que, por defecto, se pueden definir como estándar son:

- Línea base Funcional: se suele definir al final de la etapa de análisis de requisitos y engloba todos los elementos de configuración que se hayan definido como tales y finalizado en su primera versión entre el inicio del proyecto y el final de la mencionada etapa. Engloba, por ejemplo, el plan de proyecto y el documento de especificación de requisitos.
- Línea base de Diseño: se define al final de la fase de diseño y contiene todos los documentos y productos generados en la fase de diseño (o, en cualquier caso, desde la última línea base) que vayan a evolucionar a lo largo del ciclo de vida del proyecto y de los que se quiera controlar su evolución.
- Línea base de Producto: se identifica en medio de la fase de pruebas cuando el producto final ya está integrado y probado pero aún no se han realizado las pruebas de aceptación. Engloba elementos tales como el plan de pruebas o el manual técnico.
- Línea base Operativa: se define al final del ciclo de vida de desarrollo, justo en la entrega del producto y contiene todos los elementos que sean susceptibles de ser modificados y se quiera controlar dichas modificaciones o se hayan definido como elementos de configuración que se hayan generado en su primera versión definitiva después de la línea base de producto, como, por ejemplo, el manual de usuario o el producto final entregable al usuario. Nótese que la línea base operativa va a servir de punto de partida para las tareas de mantenimiento que se realicen sobre el sistema ya entregado.

En la Figura 4.1 se muestra un esquema que indica el momento en el que se definen las líneas base estándar en un proyecto software.



**Figura 4.1.** Líneas base más comunes en el ciclo de vida de un desarrollo estándar.

De forma genérica, una vez que se ha desarrollado y revisado un objeto o elemento de configuración se convierte en línea base y pasa a formar parte de la siguiente línea base definida en el transcurso del proyecto. Una vez que un objeto pertenece a una línea base, todos los cambios se deben hacer según el procedimiento establecido en cada organización y cada cambio puede producir una nueva versión del objeto.

Un ECS pasa a ser línea base únicamente después de haber sido revisado y aprobado.

Las siguientes secciones recogen las actividades que son necesarias para llevar a cabo una correcta GCS y los motivos que hacen que estas actividades sean parte imprescindible de la GCS.

## 4.5. Actividades

Dentro de la GCS de un producto software, es necesario llevar a cabo las siguientes tareas:

- *Identificación de elementos.*

Consiste en reflejar la estructura de un producto software e identificar sus componentes y elementos de una forma única y accesible. Algunas de las cuestiones que debe abordar son de qué partes se compone el producto o cuáles son las relaciones entre los distintos componentes.

- *Control de versiones y releases.*

Consiste en gestionar las versiones de los objetos de configuración creados durante los procesos de Ingeniería del Software y controlar las versiones y releases de un producto a lo largo de su ciclo de vida. Las preguntas ¿cuáles son las versiones de cada elemento de la configuración? y ¿qué elementos componen la versión X? deben ser contestadas durante las tareas incluidas en esta actividad.

A cada componente se le asigna un número de versión único que se va incrementando a medida que el componente evoluciona. Es habitual comenzar con la versión número 1.0. Pasar a la 1.1, 1.2, etc., cuando se realicen modificaciones y saltar a la versión 2.0 cuando se realicen cambios significativos. Y así sucesivamente. Es conveniente establecer un protocolo para decidir cuando se cambia el primer número, lo que indica un cambio significativo, y seguir este procedimiento de forma estricta.

- *Control de cambios.*

Es un proceso que incluye proponer un cambio, evaluarlo, aprobarlo o rechazarlo, planificarlo, realizarlo y hacerle un seguimiento. Consiste, por tanto, en controlar los cambios que se producen a lo largo del ciclo de vida de un proyecto software estimando el impacto de dicho cambio y evaluando si es factible y rentable realizarlo.

Dado que el control de cambios es uno de los aspectos más relevantes de la GCS se ha considerado adecuado dedicarle una sección independiente, la Sección 4.6.

- *Auditoría y revisión de configuración.*

Consiste en validar que la configuración de un determinado proyecto software es correcta y completa. Esto conlleva mantener la consistencia entre sus elementos, así como asegurar que el cambio se ha implementado correctamente. También consiste en comprobar que el producto sea, en todo momento, conforme con los requisitos software. Las rutas de auditoría proporcionan información adicional sobre quién, cuándo y por qué se realizaron los cambios.

- *Registro de estados.*

Consiste en registrar e informar sobre los estados de los componentes con el propósito, entre otros, de poder tener estadísticas de los componentes del producto. También incluye la generación de informes de estado de la configuración de los componentes e informes sobre las solicitudes de cambio producidas.

## 4.6. Control de cambios

Como ya se ha mencionado, el propósito del control de cambios es controlar y gestionar los cambios que se producen en un sistema software a lo largo de toda su vida, tanto en desarrollo como en mantenimiento.

Los pasos que debe seguir el proceso completo de un cambio desde su solicitud hasta su finalización son:

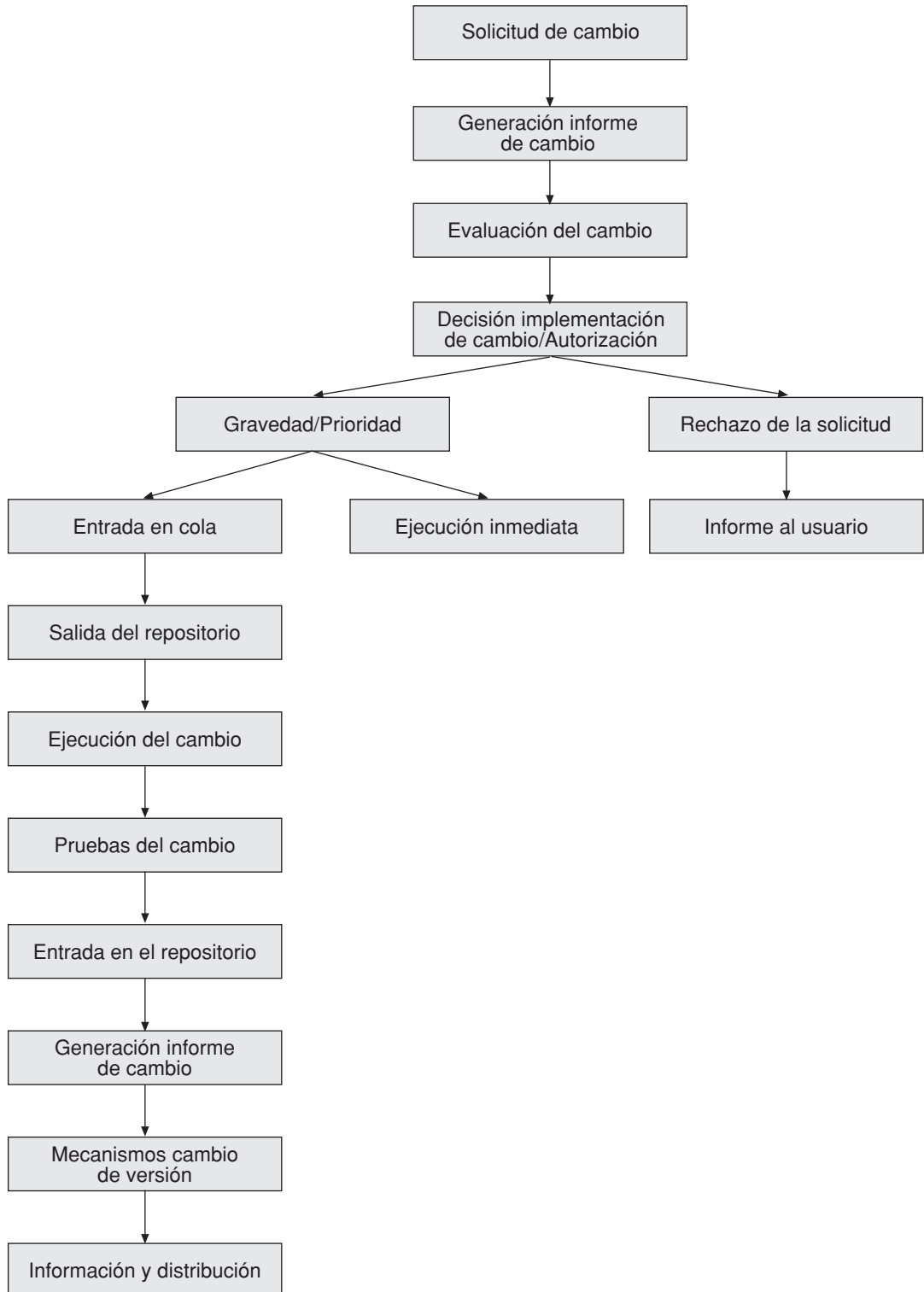
- Petición de cambio.
- Evaluación del esfuerzo, efectos secundarios, alcance, etc., del cambio.
- Descripción de posibles alternativas y su impacto (en el sistema de producción, en la producción, en el entorno).
- Evaluación de prioridades.
- Decisión sobre la conveniencia de la realización del cambio.
- Emisión de una orden de cambio.
- Autorización del cambio.
- Estimación y planificación de las actividades relacionadas con el cambio.
- Baja en la Base de Datos del objeto o componente a cambiar.
- Realización del cambio, que incluye revisión y, en su caso, modificación del diseño, modificación del código, modificación de la documentación e integración.
- Modificación de los objetos o componentes relacionados si es necesario.
- Pruebas, incluyendo pruebas regresivas.
- Alta del objeto o componente en la Base de Datos.
- Uso de mecanismos apropiados de cambio de versiones.
- Distribución de la nueva versión si es el caso.
- Generación de un informe de cambios.

Este procedimiento se muestra de forma gráfica en la Figura 4.2.

Siguiendo este procedimiento se asegura que el cambio se realiza de forma consistente y coherente sin posibles solapamientos. El objeto o componente es extraído de la base de datos una vez que se haya autorizado el cambio y vuelto a introducir una vez concluido éste.

Dependiendo del estado y etapa en la que se encuentre un ECS, hay distintos niveles de cambio. Así, un cambio informal es el que se aplica antes de que un ECS se convierta en una línea base y habitualmente no requiere de un procedimiento estricto, aunque es necesario llevar un cierto control.

El segundo nivel, que es el control de cambios a nivel de proyecto, se efectúa una vez que un ECS se convierte en una línea base, el ECS ya está introducido en la base de datos o repositorio y requiere de un procedimiento más estricto, como el descrito anteriormente.



**Figura 4.2.** Grafo de gestión de solicitudes de cambio.



El tercer y último nivel corresponde a un control de cambios totalmente formal que es el que se adopta cuando el producto software ya se ha distribuido a los clientes. En este caso, se realiza el cambio también siguiendo los pasos mencionados anteriormente y, posteriormente, se distribuye la nueva versión o release del producto software a los clientes.

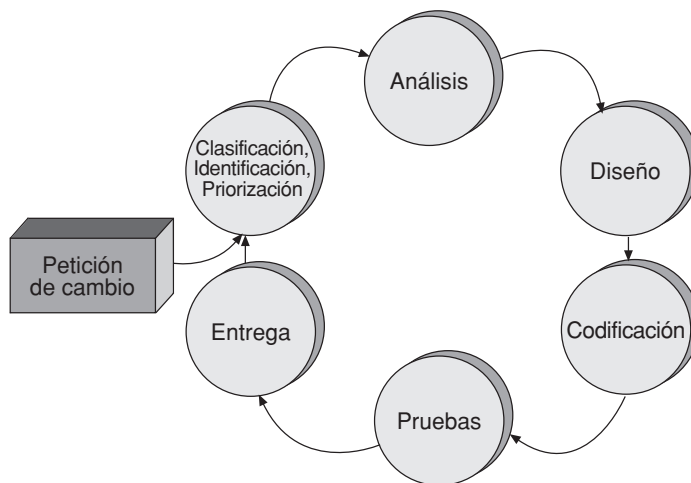
### 4.6.1. Motivos del cambio

Los cambios son continuos en un proyecto software, tanto en el proceso de desarrollo como en el proceso de mantenimiento. En las fases de desarrollo, los cambios son obvios ya que se está construyendo el producto. En el caso del mantenimiento los motivos pueden ser variados, como por ejemplo:

- Proveer continuidad de servicio: arreglar errores, recuperación de fallos, modificaciones debidas a cambios en SO o HW..., etc.
- Realizar cambios obligatorios debido a cambios en regulaciones gubernamentales o para mantener el nivel competitivo respecto a otros productos, dar soporte a peticiones de mejora, etc.
- Mejora de la funcionalidad o del rendimiento para cumplir nuevos requisitos del cliente, para adaptarlo a modos de trabajo particulares, etc.
- Reestructuración de código y de bases de datos, que incluye la mejora y puesta al día de documentación, mejora de diseño, etc.

En este sentido, cabe destacar que el proceso de mantenimiento, sea del tipo que sea, correctivo, perfectivo, preventivo o adaptativo, está muy ligado a la GCS ya que las actividades de mantenimiento provocan modificaciones en el sistema y cualquier modificación implica un cambio que tiene que estar sujeto a las estrategias y actividades de GCS, en especial el control de cambios y el control de versiones.

Especialmente en la etapa de mantenimiento, el ciclo de vida de un cambio es similar al ciclo de vida de desarrollo de un proyecto software, pero las fases tienen una duración mucho más corta. Esto se muestra en la Figura 4.3.



**Figura 4.3.** Ciclo de vida propuesto en el estándar IEEE 1219.

A continuación se detalla este ciclo de vida enumerando las tareas más relevantes en cada fase:

- Análisis
  - Entender el alcance del cambio a realizar.
  - Estudiar las restricciones de la realización del cambio.
  - Analizar los efectos del cambio.
- Diseño
  - Revisar el diseño (de arquitectura y detallado) para incorporar los cambios.
- Codificación
  - Localizar los puntos del código donde se han de introducir los cambios.
  - Programar los cambios.
  - Controlar los efectos secundarios.
  - Actualizar los comentarios del código.
- Pruebas
  - Generar nuevos casos de prueba.
  - Probar los cambios.
  - Realizar pruebas de regresión.
  - Realizar pruebas del sistema si son necesarias.
- Entrega
  - Reincorporar el software a producción.
  - Entregar la nueva documentación.
  - Actualizar la documentación externa.
- Mantenimiento: el mantenimiento del cambio también pasará a formar parte del mantenimiento de la aplicación, con lo que se considera incluido en el ciclo de vida.

Durante todo el ciclo y todas las actividades es imprescindible controlar la calidad del producto, tanto de los procesos como del producto (por ejemplo, una nueva versión) resultante.

Las últimas secciones de este capítulo recogen las funcionalidades principales que ofrecen las herramientas automáticas de GCS en la actualidad, sin entrar en ninguna en particular, y la documentación que se genera durante el proceso de GCS en un proyecto software.

## 4.7. Herramientas de GCS

Es más que recomendable llevar a cabo la GCS de un proyecto software mediante la utilización de una herramienta automática que ayude en la labor. De esta forma, se minimiza el riesgo de cometer inconsistencias, solapamientos, pérdida de versiones y demás efectos no deseables en términos de GCS durante un proyecto software.

Actualmente existe una importante oferta de herramientas de GCS. La gran mayoría aporta una serie de funcionalidades comunes que son, entre otras:

- Crear y gestionar un repositorio centralizado.
- Almacenar elementos y componentes.
- Gestionar y controlar versiones de objetos.
- Recuperar versiones antiguas.
- Gestionar el acceso de múltiples usuarios a las versiones alojadas en un determinado repositorio, es decir, facilitar un modelo de trabajo colaborativo.
- Gestionar conflictos entre usuarios concurrentes.
- Gestionar cambios y errores.
- Crear nuevas revisiones y asignación automática de la correspondiente numeración a la misma.
- Identificar diferentes líneas o ramas paralelas de desarrollo.
- Añadir, eliminar y renombrar archivos y directorios.
- Visualizar la información registrada de los cambios.
- Visualizar información sobre operaciones efectuadas.
- Gestionar informes.

Existen otras funcionalidades según la complejidad de la herramienta, pero las listadas anteriormente son ofrecidas por la gran mayoría de las herramientas existentes.

En cualquier caso, como se ha mencionado, es irresponsable llevar la GCS de un proyecto software (con la excepción de que fuera muy pequeño tanto en código como en tiempo como en equipo de trabajo) sin la utilización de una herramienta automática. Es obvio que manualmente es mucho más fácil cometer errores que lleven a sufrir pérdidas de software o situaciones similares. Adicionalmente al uso de la herramienta de GCS, se deberán realizar copias de seguridad periódicas del contenido del repositorio.

## 4.8. Documentación

Además del plan de GCS que define las actividades de GCS a realizar durante el ciclo de vida de un proyecto software y su planificación y organización, destacan también como documentación adicional, el Formulario de petición de cambio y el Informe de cambios del software, que se explicarán en esta sección.

El plan de GCS es único por proyecto. Sin embargo, para cada cambio realizado hay que rellenar un formulario de petición de cambio y un informe de cambios. El primero lo genera la persona que solicita el cambio y el segundo la persona que realiza el cambio o que es responsable de la ejecución del mismo.

### 4.8.1. Plan de GCS

El plan de gestión de configuraciones se realiza al principio del ciclo de vida del proyecto, a la vez que el plan de proyecto. Define las normas y procedimientos asociados a la implantación de un sistema de gestión de configuraciones en un proyecto software.

Se muestra a continuación un esquema de un posible Plan de Gestión de la Configuración según el estándar ANSI.

1. Introducción.
  - 1.1 Propósito.
  - 1.2. Alcance.
  - 1.3. Acrónimos.
  - 1.4. Referencias.
2. Gestión.
  - 2.1. Organización.
  - 2.2. Responsabilidades de la GCS.
  - 2.3. Control de Interfaces HW/SW.
  - 2.4. Implementación del plan de GCS y políticas aplicables.
  - 2.5. Directivas.
  - 2.6. Procedimientos.
3. Actividades de Gestión de Configuraciones Software.
  - 3.1. Identificación.
  - 3.2. Control.
  - 3.3. Informes de estado de la configuración.
  - 3.4. Auditorías y revisiones.
4. Herramientas, técnicas y metodologías.
5. Control de proveedores.
6. Recogida y retención de información.

### 4.8.2. Formulario de Petición de Cambios

Es el formulario que el cliente o usuario realizan para solicitar un cambio. También puede ser generado por un ingeniero de software que detecte un error o una posible mejora, por ejemplo, en el producto: la petición puede estar incluso motivada por la evolución del mercado.

Es necesario haber definido previamente un protocolo de comunicación en el contrato o en cualquier otro documento formal identificando, entre otras cosas, cuáles son los canales de comunicación (a quién se entrega esta petición) y cómo (en qué medio). También se ha definido previamente el tiempo máximo de respuesta después del envío de una petición de cambios y el procedimiento para realizar un seguimiento completo del mismo hasta su finalización satisfactoria.

Suele haber plantillas estándar para realizar esta petición pero, en cualquier caso, los campos básicos de los que debe constar son:

- Proyecto.
- Identificador de la solicitud.
- Solicitante de la petición.
- Fecha de la petición.
- Prioridad/Gravedad.
- Especificación del cambio requerido. Esto puede ser especificación del error o defecto encontrado, junto con sus condiciones de entorno y el soporte (datos de entrada, listados, etc.) o la especificación de los requisitos del cambio solicitado en el caso de que no sea un error.
- Motivo de la petición.
- Evaluación de la solicitud.
- Evaluador de la solicitud.
- Esfuerzo estimado.
- Módulos/Componentes afectados.
- Decisión de aceptación o rechazo.

### 4.8.3. Informes de Cambios

Una vez realizado el cambio solicitado de forma completa y correcta, se debe generar un informe de los cambios donde se detallen los cambios realizados y los efectos derivados de los mismos. Nuevamente, los datos de este informe y su estructura es variable en cada organización, pero de forma estándar suele contener, al menos, los siguientes campos:

- Objetivo de los cambios requeridos.
- Fecha.
- Esfuerzo (estimado) para realizar los cambios.
- Prioridad de la petición.
- Nombre de la persona encargada de realizar el cambio.
- Nombre de la persona que autorizó el cambio.
- Nombre de la persona que revisa el cambio.
- Programas o módulos/objetos/funciones modificados.
- Programas/periféricos/aplicaciones afectadas.
- Documentos u otros elementos afectados.
- Número de versión actual.

En ocasiones, el formulario de petición de cambios y el informe de cambios se aúnan y por cada cambio se genera un único formulario donde se describe el cambio solicitado y, una vez concluido, se informa de todos los aspectos contenidos en el informe de cambios. Un ejemplo de este tipo de formulario, bastante común, se muestra en la Figura 4.4.

<b>Formulario de cambio</b>		
Proyecto:		Identificador:
Solicitante de la petición:		Fecha de solicitud:
Cambio solicitado:		
Motivo de la petición:		
Evaluador del cambio:		Fecha de evaluación:
Módulos/Componentes afectados:	Módulos/Componentes relacionados:	
Evaluación del cambio:		
Esfuerzo estimado:	Prioridad/Gravedad:	Aceptado      Rechazado
Cambio realizado por:		Fecha de implementación:
Cambio autorizado por:		Fecha de autorización:
Comentarios/Observaciones:		
Módulos/Funciones/Componentes y documentos modificados:		
Nueva versión:	SÍ      NO	Revisado por:
Número de versión actual:	Fecha de revisión:	

Figura 4.4. Informe de cambios.

#### 4.8.4. Otros documentos

Cada organización define los documentos más convenientes para su situación en el caso de GCS. Algunos de los más frecuentes son:

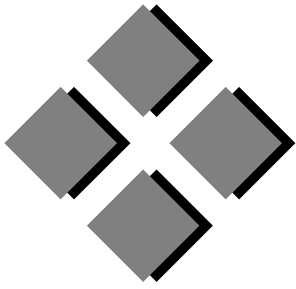
- *Dossier de gestión de la configuración.*  
Contiene los resultados de las actividades de gestión de configuraciones.
- *Dossier de gestión de cambios.*  
Contiene toda la información necesaria para asegurar el control de los cambios.

### 4.9. Bibliografía

- Sommerville, I.: *Ingeniería del Software*, sexta edición. Addison Wesley, 2001.
- Pressman, R.: *Ingeniería del Software: Un enfoque práctico*, sexta edición. McGrawHill, 2005.
- Dart, S.: *Concepts in Software Configuration Management*, Carnegie-Mellon University Technical Report, 1993.
- Dart, S.: *Spectrum of Functionality in Configuration Management Systems*. Software Engineering Institute. 2001. [http://www.sei.cmu.edu/legacy/scm/tech\\_rep/TR11\\_90/TOC\\_TR11\\_90.html](http://www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/TOC_TR11_90.html)
- Estublier, J.: *Software Configuration Management: A RoadMap*, Technical Report Grenoble University, 2000.
- Babich, W. A.: *Software Configuration Management*, Addison-Wesley, 1986.







# Capítulo 5

## Herramientas de control de versiones: Subversion (SVN)

---

### SUMARIO

- |                                               |                                                    |
|-----------------------------------------------|----------------------------------------------------|
| <b>5.1.</b> Introducción                      | <b>5.6.</b> Autenticación                          |
| <b>5.2.</b> ¿Por qué utilizar Subversión?     | <b>5.7.</b> Autorización                           |
| <b>5.3.</b> Descripción general de Subversión | <b>5.8.</b> Puesta en marcha                       |
| <b>5.4.</b> Instalación                       | <b>5.9.</b> Trabajando con Subversión: TortoiseSVN |
| <b>5.5.</b> Creación de repositorios          | <b>5.10.</b> Bibliografía                          |

## 5.1. Introducción

Como se vio en el Capítulo 4, durante la tarea de pruebas, el software está sometido a numerosos cambios: se crean procedimientos de prueba, se generan casos de prueba y documentación asociada, se modifica el código para corregir errores, etc. Normalmente esta tarea se realiza simultáneamente por los diferentes miembros del equipo de pruebas que se encargan de probar los distintos módulos del sistema ejecutando pruebas unitarias, de integración, funcionales, etc. (véase Capítulo 1). Es fácil que se den circunstancias en las que dos o más personas tengan que modificar el mismo código, o que el código modificado por terceros influya en sus pruebas. En estas situaciones, mantener un correcto control de esos cambios es vital para posibilitar un modelo de trabajo colaborativo con el cual garantizar el éxito final del proyecto.

Como también se ha comentado en el Capítulo 4, el control de versiones es fundamental para gestionar los cambios en el software. Por tanto, disponer de herramientas que automaticen y faciliten este trabajo es de vital importancia. Con este fin se incluye en este libro una descripción de las herramientas Subversion y TortoiseSVN, ambas de software libre. Subversion es una herramienta de control de versiones y TortoiseSVN es un cliente gráfico para acceder a Subversión desde plataformas basadas en Windows. Es decir, una interfaz gráfica para acceder remotamente a Subversion. Estas herramientas trabajan en red para que varios usuarios puedan trabajar sobre el mismo repositorio sin interferirse unos a otros.

El contenido de este capítulo no es una descripción completa de las herramientas mencionadas, puesto que no es el objetivo de este libro ser un manual detallado. En este capítulo sólo se hace referencia a las características y operaciones más útiles y habituales que se realizan durante la fase de pruebas.

## 5.2. ¿Por qué utilizar Subversion?

Son varias las razones para utilizar Subversion en este libro:

- Subversion es un sistema de control de versiones de software libre. Permite mantener un repositorio en red, de manera que puede ser accedido remotamente desde múltiples localizaciones.
- Proporciona herramientas que facilitan la realización de backups de los contenidos del repositorio.
- Subversión se encarga de mantener diferentes versiones de cada uno de los elementos software de un proyecto, sin limitarse únicamente al código fuente.
- Proporciona mecanismos para facilitar el trabajo colaborativo, que entre otras cosas ayudan a evitar la sobreescritura de versiones y la consecuente pérdida de información. Si bien es responsabilidad última de los usuarios controlar estas situaciones.
- Otra característica diferenciadora de Subversion es, que al contrario de otras herramientas de gestión de versiones como CVS, no sólo versiona ficheros, sino también carpetas e, incluso, las operaciones de mover y copiar se pueden realizar sobre unos y otras.
- Utiliza el mismo modelo de versionado que CVS, lo que puede facilitar la migración de una herramienta a otra y facilitar el uso de Subversion a aquellas personas que ya conozcan CVS.

## 5.3. Descripción general de Subversion

Subversion es un sistema centralizado para compartir información. En su núcleo hay un *repositorio*, que es un almacén central de datos. El repositorio almacena la información en forma de un *árbol de ficheros* - una jerarquía típica de ficheros y directorios. Esta estructura del repositorio se asemeja bastante a un sistema de ficheros habitual, pero tiene una característica añadida: este repositorio guarda y registra todos los cambios que se hayan hecho en él a lo largo del tiempo. Por tanto, se puede acceder a la última versión del árbol de ficheros, que en Subversion

se conoce como versión *HEAD*. Pero también se pueden ver estados previos del repositorio. De hecho, se puede recorrer toda la historia del contenido del repositorio.

En Subversion un repositorio es una secuencia de árboles de directorios. Cada árbol es una imagen del estado del software versionado en un momento del tiempo. Estas imágenes se llaman *revisiones* y son el producto de las operaciones de los participantes en el proyecto que tienen acceso al repositorio. Cada revisión tiene propiedades que no son versionadas, es decir, si se modifican se pierde su valor anterior. Algunos ejemplos de estas propiedades son: `svn:date`, que almacena la hora y la fecha de creación de la revisión; `svn:author`, que indica quién hizo la revisión; o `svn:log` que almacena el mensaje adjunto a la revisión. Al ser propiedades no versionadas se puede, por ejemplo, corregir un mensaje de log de una revisión anterior, o ampliarlo, sin que eso modifique la revisión.

Cuando se trabaja con ficheros compartidos, varios usuarios se conectan al repositorio, para acceder a esos ficheros, ya sea para leer o escribir en ellos. De este modo comparten la información. Como se ha comentado en el Capítulo 4, el problema que puede surgir es, que en un acceso simultáneo, mientras un cliente está actualizando una versión de un fichero, otro está leyendo esa misma versión, pero sin conocer esa nueva actualización. Es decir, distintas personas están trabajando con distintas versiones del software, lo que da lugar a problemas e inconsistencias en la información.

Existen varios modelos de versionado para resolver este problema. El utilizado por Subversion consiste en copiar-modificar-fusionar. Cada usuario crea una *copia de trabajo* personal de un fichero, una carpeta o de todo el proyecto en la máquina local. Los usuarios trabajan simultáneamente, modificando sus copias privadas. Cuando uno de ellos termina, incorpora sus modificaciones al repositorio. Cuando, después, otro usuario intenta incorporar sus cambios, Subversion le indica que su copia de trabajo está *desactualizada* (ha cambiado desde que lo copió). Este segundo usuario pide entonces a Subversion que fusione cualquier cambio del nuevo repositorio en su copia de trabajo. Si los cambios no se solapan la tarea es muy sencilla. Pero si los cambios sí se superponen aparece una situación de *conflicto*, entonces la fusión se realiza manualmente al poder observar los cambios existentes en el repositorio y los que el usuario había introducido en su copia de trabajo. Resuelto el conflicto se pueden incorporar los cambios en el repositorio. Normalmente estas situaciones son llamadas colisiones, y aunque manualmente es posible mezclar varias versiones de un mismo elemento software que fueron creadas en paralelo, es conveniente prevenir y evitar que esto ocurra. El motivo es que, frecuentemente, los cambios efectuados son incompatibles y, por tanto, no directamente fusionables, con la consiguiente pérdida de tiempo y de trabajo. Para prevenir estas situaciones es necesario contar con una correcta planificación y reparto de tareas.

Cuando se trabaja en red con Subversion, suele existir un repositorio común en algún servidor y varios clientes o usuarios que se conectan a través de la red. En estos casos el funcionamiento o modelo de red se basa en peticiones y respuestas. Cada usuario pasa la mayor parte del tiempo manejando copias de trabajo. Cuando necesita información de un repositorio, hace una petición mediante un acceso a una URL. El servidor le pide que se identifique y, una vez realizada la autenticación, el servidor le responde. Es decir, el cliente hace primero la petición y sólo se le requerirá la identificación si la petición realizada lo solicita. Si, por ejemplo, el repositorio está configurado para que todo el mundo pueda leer en él, nunca se pedirá autenticación para una solicitud de una copia de trabajo en un directorio local.

Si el cliente hace una petición que implica escribir en el repositorio, como, por ejemplo, confirmar los cambios realizados, se crea un nuevo árbol de revisión. Si la petición del cliente ya fue

autenticada, el nombre del usuario identificado se almacena en la variable `svn:author` de la nueva revisión. Si el servidor no pidió la identificación al cliente, la propiedad `svn:author` estará vacía.

## 5.4. Instalación

Como se ha mencionado en el apartado anterior, Subversion puede trabajar de modo que el repositorio o repositorios se encuentren en un servidor accesible por diversos clientes que trabajan sobre un mismo proyecto. Subversion puede trabajar sobre dos servidores: embebido dentro de Apache o bien con un servidor específico llamado `svnserve`. Apache puede acceder a un repositorio y hacerlo accesible para los clientes mediante el protocolo WebDAV que es una extensión de HTTP. `svnserve` es un servidor ligero que utiliza un protocolo propio sobre una conexión TCP/IP. En general, `svnserve` es más fácil de instalar y se ejecuta más rápido que el servidor basado en Apache, pero este último es más flexible. Por ejemplo, permite especificar diferentes permisos de acceso a cada carpeta o fichero para los diferentes usuarios, lo que no se puede hacer con `svnserve`. En los dos siguientes apartados se explica cómo instalar un servidor `svnserve` y Apache respectivamente.

### 5.4.1. Servidor basado en `svnserve`

Como ya se ha comentado, `svnserve` es un servidor ligero que permite que los clientes se conecten con él por medio de una conexión TCP/IP, usando una URL que comienza con `svn://` o `svn+ssh://`. Este servidor está incluido dentro de la distribución de Subversion. Para instalar el servidor se siguen los siguientes pasos:

1. Obtener la última versión de Subversion desde <http://subversion.tigris.org/> en su versión de instalación. En este libro se trabajará con la versión 1.4.2 por lo que el archivo binario descargado será: `svn-1.4.2-setup.exe`.
2. Ejecutar este fichero para instalar la aplicación y seguir las instrucciones del asistente de instalación.
3. Buscar los ficheros `svnserve.exe`, `libdb44.dll`, `libeay32.dll` y `ssleay32.dll` en el directorio de instalación de Subversion (generalmente `C:\Archivos de programa\Subversion\bin`)
4. Copiar dichos ficheros en un directorio del servidor, por ejemplo `c:\svnserve`. Si la instalación de Subversion se realiza en el servidor se puede omitir este paso.

Una vez instalado, se debe ejecutar `svnserve` en la máquina para poder utilizarlo. El modo más sencillo de ejecutar `svnserve` es introduciendo el siguiente comando en una ventana de comandos Windows:

```
csvnserve.exe --daemon
```

Con esta instrucción `svnserve` espera peticiones en el puerto 3690. La opción `--daemon` ejecuta `svnserve` como un servicio.

Para poder empezar a trabajar es preciso crear un repositorio, si aún no se dispone de ninguno. En el Apartado 5.5 se explica cómo hacerlo.

Suponiendo que se tiene un repositorio en `c:\repos\TestRepo`, se puede acceder al mismo mediante una dirección del tipo:

```
svn://MiServidor/repos/TestRepo
```

También se puede ejecutar `svnserve` como un servicio de Windows con la ventaja de que se arrancara automáticamente al iniciarse Windows por lo que si la maquina se cayera bastaría con reiniciarla para tener el repositorio operativo. Para instalar `svnserve` como un servicio de Windows nativo, se ejecuta la siguiente instrucción en una ventana de comandos:

```
sc create svnserve binpath= "c:\svnserve\svnserve.exe -service  
-root c:\repos" displayname= "Subversion" depend= tcpip  
start= auto
```

suponiendo que el repositorio está en `c:\repos`.

#### 5.4.1.1. Autenticación con `svnserve`

Por omisión, `svnserve` proporciona acceso anónimo de sólo-lectura. Esto significa que se puede acceder al repositorio, usando una URL de tipo `svn://`, para obtener información o actualizarla, pero no para confirmar ningún cambio.

Para permitir acceso de escritura en un repositorio, se debe modificar el fichero `conf/svnserve.conf` en el directorio del repositorio al que se quiera dar acceso. Este fichero controla la configuración del servicio `svnserve`, además de contener otra información útil. La forma más simple de dar acceso anónimo para escritura es la siguiente:

```
[general]  
anon-access = write
```

El problema es que no se sabrá quién ha hecho los cambios en el repositorio (la propiedad `svn:author` estará vacía). La solución a este problema es crear un fichero de contraseñas, llamado, por ejemplo, `ficherodeusuarios`, que debe tener la siguiente estructura:

```
[users]  
usuario = contraseña  
...
```

Una vez que se dispone de este fichero, la configuración de acceso al repositorio quedaría:

```
[general]  
anon-access = none  
auth-access = write  
password-db = ficherodeusuarios
```

Este ejemplo denegaría cualquier acceso a los usuarios no autenticados (anónimos), y daría acceso de lectura-escritura a los usuarios listados en `ficherodeusuarios`.

El fichero `fichero de usuarios` se puede incluir en el mismo directorio que `svnserve.conf` o en cualquier otro sitio de su sistema de ficheros. En este último caso se debe referenciar utilizando una ruta absoluta, o una ruta relativa del directorio `conf`. La ruta debe estar escrita según el formato de Unix. No funcionará si se utiliza el carácter “\” como separador o letras de unidades.

## 5.4.2. Servidor basado en Apache

La configuración basada en Apache es, como ya se ha comentado, algo más complicada de preparar, pero ofrece algunas ventajas:

- Se puede acceder al repositorio (únicamente en modo lectura) a través del propio navegador Web Apache utilizando el protocolo `http://`. Esto permite navegar por el repositorio sin tener un cliente de Subversión instalado. Si el repositorio es local, también se puede acceder utilizando `file://`.
- Al utilizar el protocolo WebDAV permite acceder al repositorio como a una carpeta del sistema de ficheros.
- Se puede utilizar cualquier mecanismo de autenticación que ofrezca Apache.
- Dada la estabilidad y seguridad de Apache, se obtiene automáticamente esa misma seguridad para el repositorio. Esto incluye la encriptación SSL (Secure Sockets Layer). Es decir, el acceso mediante `https://`.

Para preparar un servidor de Subversion sobre Apache es necesario disponer de los paquetes `httpd 2.0` y `mod_dav`, distribuidos con Apache, Subversion y `mod_dav_svn` distribuidos con Subversion. Una vez instalados será necesario configurar Apache modificando el fichero `httpd.conf`. La instalación ha de realizarse en una cuenta con permisos de administrador.

Para instalar Apache hace falta un ordenador con Windows 2000, WindowsXP con SP2, o Windows 2003. Utilizar Windows XP sin el Service Pack 2, podría corromper el repositorio.

Los pasos a seguir para instalar Apache y Subversion se describen a continuación:

1. Descargar Apache HTTP Server para Windows desde <http://httpd.apache.org> en su forma Win32 Binary (MSI Installer): `apache_2.0.59-win32-x86-no_ssl.msi`. Las versiones anteriores de Apache no funcionan con Subversion por cómo se compilaban dichas versiones para Windows. Tampoco funciona con Apache 2.2. Para la realización de este libro se ha decidido utilizar la versión más reciente disponible, es decir la 2.0.59.
3. Verificar la integridad de los ficheros descargados utilizando PGP o MD5. Para Windows, MD5 se puede descargar desde <http://www.fourmilab.ch/md5/>. Descargar el archivo `md5.zip` y descomprimir. Ejecutar desde la línea de comandos:

```
md5 apache_2.0.59-win32-x86-no_ssl.msi
```

El resultado debe ser igual al contenido del fichero `apache_2.0.59-win32-x86-no_ssl.msi.md5` que se puede encontrar en la URL <http://www.apache.org/dist/httpd/binaries/win32/>.

3. Instalar Apache ejecutando el fichero `apache_2.0.59-win32-x86-no_ssl.msi`.
4. El asistente de instalación pedirá los datos de información del servidor: *Network Domain* y *Server domain*. También solicita la dirección de correo electrónico del administrador. Si no se dispone de DNS para el servidor se puede poner la dirección IP. Se debe elegir la opción “for All Users, on Port 80, as a Service – Recommended”. Si hubiera problemas con la instalación porque este puerto está siendo utilizado por otro programa, se debe editar el fichero `httpd.conf` (normalmente en: `c:\Archivos de programa\Apache Group\Apache2\conf\`) y cambiar la línea `Listen 80` por un puerto libre. Para el uso de Apache como servidor de Subversion se recomienda el uso del puerto 8080.

Inicialmente, se pueden hacer las pruebas del servidor de modo local poniendo *localhost* como dominio de red y nombre de servidor. No olvidar en ese caso, al configurar más tarde el fichero `httpd.conf`, cambiar el nombre del servidor mediante la directiva `ServerName`.

5. Comprobar si la instalación de Apache ha sido correcta ejecutándolo. Para ello, abrir un navegador en el servidor y acceder a la URL <http://localhost>. Aparecerá una página web prediseñada indicando que todo ha funcionado correctamente. Comprobar que previamente se ha iniciado Apache como servicio (aparecerá el icono del monitor de Apache en la barra de tareas).
6. Descargar Subversion 1.4.2 desde <http://subversion.tigris.org/> en su versión de instalación: `svn-1.4.2-setup.exe`. Ejecutar este fichero para instalar la aplicación y seguir las instrucciones del asistente de instalación.

El siguiente paso es configurar Apache para su funcionamiento con Subversion. Pero antes es necesario crear el repositorio en el que se va a trabajar. En el Apartado 5.5 se explica cómo crear un repositorio. Una vez creado el repositorio se realizan las siguientes acciones:

1. Como el módulo `/bin/mod_dav_svn.so` (normalmente instalado en `c:\Archivos de programa\Subversion`) se distribuye con Subversion y no con Apache es necesario copiarlo en el directorio `modules` de Apache.
2. Copiar el fichero `/bin/libdb43.dll` desde el directorio de instalación de Subversion al directorio `modules` de Apache.
3. Editar el fichero `httpd.conf`. Esta tarea se puede realizar desde el grupo de programas de Apache en el menú de Inicio, Apache HTTP Server 2.0.59→Configure Apache Server→Edit the Apache..., o bien abriendo directamente dicho fichero con cualquier editor de texto. Añadir las siguientes líneas para cargar los módulos que debe conocer Apache para dar acceso al repositorio:

```
LoadModule dav_module      modules/mod_dav.so
LoadModule dav_svn_module  modules/mod_dav_svn.so
```

Es posible que la primera de estas líneas ya exista en el fichero `httpd.conf` y solo haya que quitarle el comentario (es decir, borrar el carácter #). Hay que asegurarse de que las líneas estén en este orden.

4. Ahora hay que indicar a Apache dónde estará el repositorio de trabajo. Esto se hace con la directiva<sup>1</sup> `SVNPath`, dentro del bloque `<Location>` que se añadirá al final del fichero `httpd.conf`:

```
<Location /svn>
    DAV svn
    SVNPath c:\SVN\repositorio
</Location>
```

Si se desea trabajar con varios repositorios se utilizará la directiva `SVNParentPath` en lugar de `SVNPath`. Con `SVNParentPath` se indica la ruta del directorio raíz de todos los repositorios con los que se vaya a trabajar. En este caso el bloque `<Location>` quedaría como se muestra a continuación:

```
<Location /svn>
    DAV svn
    SVNParentPath c:\SVN
</Location>
```

## 5.5. Creación de repositorios

Como se ha descrito en el Capítulo 4, el repositorio es el almacén central de la información relativa a uno o varios proyectos. El repositorio necesita poco mantenimiento, pero es importante su configuración inicial para evitar problemas y resolver los que puedan surgir.

En Subversion, el repositorio se puede crear bajo el sistema de ficheros FS (FSFS<sup>2</sup>) o bien con formato Berkeley Database (BDB). Si se quiere disponer de un repositorio compartido en red es necesario definirlo con formato FSFS.

Para crear un repositorio se utiliza la herramienta `svnadmin`, suministrada con Subversion. Un ejemplo de utilización es el siguiente:

```
c:\>svnadmin create c:/svn/repositorio
```

Esta instrucción crea un repositorio en el directorio `c:/svn/repositorio`. El directorio `c:/svn/repositorio` debe existir previamente. Por omisión, el repositorio se crea con formato FSFS. Para elegir el formato del repositorio es necesario utilizar la siguiente instrucción:

```
c:\>svnadmin create -fs-type tipo /ruta/del/repositorio
```

donde `tipo` es el formato deseado (`fsfs` o `bdb`) y `/ruta/del/repositorio` el directorio donde se desea crear el repositorio.

<sup>1</sup> Para más información sobre las directivas de Apache consultar la información contenida en el manual on-line de Apache: <http://httpd.apache.org/docs/2.2/es/mod/directives.html>.

<sup>2</sup> Sistema de ficheros propietario de Subversion que se utiliza como soporte de los repositorios. Se trata de una implementación versionada del sistema de ficheros nativo del sistema operativo.



Este nuevo repositorio comienza con la revisión 0 que consiste únicamente en un directorio raíz (/). A esta revisión se le asigna la propiedad `svn:date` indicando la fecha de creación del repositorio.

### 5.5.1. La estructura del repositorio

Para comenzar a trabajar con el repositorio, el primer paso es llevar el software del proyecto al repositorio (operación Importar, véase Apartado 5.9.4.1). Para ello es importante decidir cómo se va a organizar dicho software. Es decir, hay que organizar los ficheros en carpetas y subcarpetas. Aunque es posible renombrar y mover los ficheros o crear nuevas carpetas más tarde, es preferible tener una estructura lo más estable posible antes de empezar a utilizar el repositorio.

En la Figura 6 del Capítulo 2, se muestra la estructura de directorios utilizada en el repositorio definido para las pruebas del servidor de tráfico descrito en el Apéndice B. Dicho repositorio contiene dos directorios: uno para el código de producción (sistema software) y otro para el código de pruebas (pruebas sistema software). A su vez cada uno de estos directorios contiene los siguientes subdirectorios:

- `sistema software`:
  - `build`: contiene las clases compiladas y el fichero `.jar` del sistema del servidor de tráfico.
  - `config`: mantiene la configuración inicial del sistema para que la aplicación pueda funcionar.
  - `doc`: contiene la documentación del código fuente.
  - `log`: contiene un fichero que registra las conexiones al servidor de tráfico.
  - `src`: código fuente del servidor de tráfico.
- `pruebas sistema software`:
  - `build`: contiene las clases compiladas y el fichero `.jar` de las pruebas del sistema.
  - `documentFormat`: peticiones realizadas al servidor de tráfico.
  - `lib`: librerías necesarias para ejecutar las pruebas.
  - `reports`: en este directorio se guardan los informes resultados de las pruebas.
  - `src`: código de pruebas.
  - `testCases` y `testData`: casos de prueba, cuando se separan del código de pruebas.

### 5.5.2. Acceso al repositorio

El acceso a un repositorio alojado en un servidor se puede realizar usando un navegador Web y mediante direcciones URL. Es decir, con el protocolo `http://` o `svn://`, según el tipo de servidor

que se haya preparado. Por ejemplo, para acceder al repositorio que se acaba de crear y suponiendo que el nombre del servidor es *MiServidor*, se usará la dirección *http://MiServidor:8080/svn/repositorio/*. Cualquiera de estos dos protocolos de acceso también pueden ser encriptados. En este caso se utiliza *https://* o *svn+ssh://*.

Para acceder a su repositorio local, se necesita la ruta a esa carpeta. En este caso el acceso desde un navegador se realiza usando la ruta del repositorio con el formato *file://*.

### 5.5.3. Mantenimiento del repositorio

Para mantener el repositorio una vez creado y en uso, nunca se debe modificar “a mano”. Subversion proporciona las herramientas necesarias para gestionar un repositorio y realizar operaciones típicas de mantenimiento como inspeccionar el contenido y estructura, modificar dicha estructura o repararla, recuperar un repositorio o realizar copias de seguridad. Todas estas tareas se pueden realizar mediante la herramienta *svnadmin* que se distribuye con Subversion. Una de las acciones más interesantes es la que permite realizar copias de seguridad (o *backups*), lo que posibilita restaurar una imagen del repositorio si hay algún problema. La copia de seguridad puede ser incremental o completa. Una copia de seguridad incremental consiste en copiar solo los cambios realizados en el repositorio desde la última copia de seguridad. Para realizar una copia de seguridad incremental se utiliza la instrucción *svnadmin dump -incremental*.

Una copia de seguridad completa consiste en duplicar el directorio entero donde está el repositorio. Pero, a menos que se desactive cualquier acceso al repositorio, hacer simplemente una copia del repositorio corre el riesgo de generar una copia con fallos, puesto que alguien podría escribir a la vez en dicho repositorio. *svnadmin* dispone de la instrucción *hotcopy* que realiza los pasos necesarios para hacer una copia de seguridad completa del repositorio activo, sin necesidad de interrumpir el acceso al repositorio.

El método recomendado para crear una copia del repositorio de forma segura consiste en ejecutar la siguiente orden:

```
c:\>svnadmin hotcopy c:/svn/repositorio ruta/al/backup
```

El resultado final es una réplica del repositorio completamente funcional, que podría sustituir al actual si ocurriera algún problema.

## 5.6. Autenticación

Tras la configuración de Apache, se debe pensar en los permisos de acceso al repositorio. Tal y como se ha configurado la localización del repositorio (en el bloque *<Location>*), éste puede ser accedido por cualquier usuario de forma “anónima”:

- Cualquier usuario puede hacer una copia de trabajo de un repositorio URL (o cualquiera de sus subdirectorios) en su disco local.
- Cualquiera puede consultar la última revisión del repositorio simplemente con su navegador Web.
- Cualquiera puede modificar el repositorio.

La forma más sencilla de autenticar a un cliente es mediante el mecanismo de autenticación HTTP Basic, que usa un nombre de usuario y una contraseña para verificar que un usuario es quien dice ser. Apache proporciona una herramienta `htpasswd` para gestionar la lista de usuarios a los que se da acceso al repositorio y sus contraseñas.

Inicialmente hay que añadir los usuarios al fichero de claves. La primera vez que se añade un usuario es necesario utilizar el modificador `-c` que crea el fichero de contraseñas. Para cada usuario se añade el modificador `-m` para encriptar la contraseña. Por ejemplo:

```
c:\> htpasswd -cm C:/SVN/svn.pass daniel
New password: *****
Re-type new password: *****
Adding password for user daniel
c:\> htpasswd -m C:/SVN/svn.pass almudena
New password: *****
Re-type new password: *****
Adding password for user almudena
c:\>
```

Es posible que no se encuentre esta herramienta. Eso es porque hay que incluir el directorio `bin` de Apache en la variable de entorno `Path`<sup>3</sup>.

Después hay que indicar a Apache, en el fichero `httpd.conf` y dentro de la directiva `Location`, dónde encontrar el fichero de contraseñas:

- La directiva `AuthType` indica el tipo de autenticación que usa el sistema. En este caso la básica, es decir, usuario/contraseña.
- La directiva `AuthName` es un nombre que se da, de forma arbitraria, al dominio de autenticación. La mayoría de los navegadores mostrarán este nombre cuando aparezca un diálogo de autenticación preguntando el nombre y contraseña del usuario.
- La directiva `AuthUserFile` especifica la localización del fichero de contraseñas que se ha creado previamente usando `htpasswd`.

Tras añadir estas directivas, el bloque `<Location>` queda de la siguiente manera:

```
<Location /svn>
    DAV svn
    SVNParentPath c:\SVN

    # como autenticar un usuario
    AuthType Basic
    AuthName "Subversion repository"
    AuthUserFile c:\svn\svn.pass
</Location>
```

Con estas directivas solo se indica a Apache que debe pedir un usuario y contraseña si se requiere autorización. Lo que falta en este bloque es indicar a Apache qué clase de peticiones de los clientes requieren autorización. Cuando se requiere dicha autorización Apache reclamará

---

<sup>3</sup> Véase Apéndice A Variables de Entorno.

también la autenticación. Lo más simple es proteger todas las peticiones. Añadiendo `Require valid-user` todas las peticiones requieren la autenticación del usuario.

```
<Location /svn>
  DAV svn
  SVNParentPath c:\SVN

  # solo los usuarios autenticados tienen acceso al repositorio
  Require valid-user

  # como autenticar un usuario
  AuthType Basic
  AuthName "Subversion repository"
  AuthUserFile c:\svn\svn.pass
</Location>
```

Con esto, sólo los usuarios que se identifiquen como `daniel` o `almudena` y proporcionen correctamente la contraseña podrán hacer cualquier operación sobre el repositorio. Ningún otro usuario podrá acceder al repositorio.

Para que la nueva configuración tenga efecto hay que reiniciar el servicio de Apache. Si se intenta acceder al repositorio mediante la URL <http://MiServidor/svn/repositorio>, donde `Mi-Servidor` es el nombre del servidor donde se instaló Apache, se abrirá una ventana, como la de la Figura 5.1, solicitando usuario y contraseña.



**Figura 5.1.** Diálogo de autenticación para acceso al repositorio.

Como ya se ha comentado en el apartado de acceso al repositorio, se puede incrementar la seguridad del sistema de autenticación utilizando la encriptación SSL de las contraseñas para que los usuarios se conecten mediante *https://*.

## 5.7. Autorización

Con las modificaciones realizadas en el fichero `httpd.conf` se ha configurado la autenticación, pero no la autorización o control de acceso. Apache puede pedir identificación a los usuarios y confirmar su identidad, pero no puede permitir o restringir el acceso a cada usuario según quién sea.

Existen dos estrategias para controlar el acceso a los repositorios: dar acceso de forma general para acceder al repositorio o repositorios como una unidad, o dar diferentes permisos según los distintos directorios o carpetas del repositorio. En este apartado se describe brevemente cada una de ellas.

### 5.7.1. Control de acceso general

Como se ha visto en el apartado anterior, utilizando la directiva `Require valid-user` solo tendrán acceso al repositorio los usuarios dados de alta en el fichero de contraseñas. Pero no siempre se necesita un control tan férreo. Por ejemplo, se podrían restringir las operaciones de escritura en el repositorio solo a determinados usuarios, haciendo accesible para todo el mundo la lectura en el repositorio.

Para definir un control de acceso como el descrito en el párrafo anterior se usa la directiva `LimitExcept`. La modificación requerida en `httpd.conf` consiste en cambiar la línea:

```
Require valid-user
```

en el bloque `<Location>` definido previamente, por:

```
<LimitExcept GET PROPFIND OPTIONS REPORT>  
    Require valid-user  
</LimitExcept>
```

Con esto se indica que se requiere autenticación para todas las operaciones, salvo las incluidas en el bloque `<LimitExcept>`.

Este es solo un ejemplo de los tipos de acceso que se pueden definir de forma general para todos los usuarios. Hay otras posibilidades que se pueden explorar en la página web de documentación de Apache (<http://httpd.apache.org/docs/2.0/mod/directives.html>).

### 5.7.2. Control de acceso basado en directorios

Es posible establecer permisos de acceso más detallados, basados en nombres de usuarios y rutas dentro del repositorio, usando el módulo `mod_authz_svn`. Al igual que se hizo con otros módulos, hay que comprobar si el módulo `mod_authz_svn.so` se encuentra en el directorio `modules` de Apache. Si no es así hay que copiarlo en dicho directorio. Además se debe revisar si la línea:

```
LoadModule authz_svn_module modules/mod_authz_svn.so
```

se ha incluido en el fichero de configuración de Apache. Para activar este módulo también hay que modificar el bloque `<Location>` añadiendo la directiva `AuthzSVNAccessFile` que es-

pecifica dónde está el fichero que contiene los permisos de acceso al repositorio. En los ejemplos siguientes dicho fichero se llama `svn.authz`. Cómo definir el contenido de este fichero se describe en el Apartado 5.7.2.1.

Se puede considerar que existen tres patrones generales de acceso:

- Acceso anónimo. Apache nunca solicita la identificación del usuario. Todos los usuarios se consideran anónimos. El bloque `<Location>` contiene lo siguiente:

```
<Location /svn>
    DAV svn
    SVNParentPath c:\svn

    # política de control de acceso
    AuthzSVNAccessFile C:\svn\svn.authz
</Location>
```

- Acceso con autenticación. En este tipo de acceso todos los usuarios deben identificarse. Es necesario utilizar `Require valid-user` y establecer un medio de autenticación. El bloque `<Location>` se define de la siguiente manera:

```
<Location /svn>
    DAV svn
    SVNParentPath c:\svn

    # política de control de acceso
    AuthzSVNAccessFile C:\svn\svn.authz

    # solo los usuarios autenticados tienen acceso al repositorio
    Require valid-user

    # como autenticar un usuario
    AuthType Basic
    AuthName "Subversion repositories"
    AuthUserFile c:\svn\svn.pass
</Location>
```

- Acceso mixto. Se combina el acceso anónimo con el acceso con autenticación. Por ejemplo, se puede permitir acceso anónimo para operaciones de lectura sobre alguna parte del repositorio, pero restringido para todo tipo de operaciones en otras áreas del repositorio. En este tipo de acceso todos los usuarios acceden inicialmente como anónimos. Cuando un usuario solicita una petición que, según la política de acceso definida, requiere autenticación, Apache solicita al usuario que se identifique. Para definir este patrón de acceso son necesarias las directivas `Satisfy Any` y `Require valid-user`. En el caso de este tipo de acceso el bloque `<Location>` se define de la siguiente forma:

```
<Location /svn>
    DAV svn
    SVNParentPath c:\svn

    # política de control de acceso
    AuthzSVNAccessFile C:\svn\svn.authz
```

```
# se intenta acceso anónimo, si es necesario se solicita au-
tenticacion
Satisfy Any
Require valid-user

# como autenticar un usuario
AuthType Basic
AuthName "Subversion repository"
AuthUserFile c:\svn\svn.pass
</Location>
```

Apache asegurará que sólo los usuarios válidos puedan acceder al repositorio. En los ejemplos definidos, controla el acceso a /svn. Luego pasará el nombre de usuario al módulo de Subversion AuthzSVNAccessFile para que pueda controlar un acceso más detallado basado en las reglas que se especifican en el fichero svn.authz. Si en este fichero no se especifica un repositorio particular, la regla de acceso se aplicará a todos los repositorios bajo SVNParentPath.

#### 5.7.2.1. Definición del fichero de control de acceso

Después de configurar el bloque <Location>, se crea el fichero de control de acceso para definir la política de permisos, es decir, las reglas de autorización que permitirán ciertas acciones a ciertos usuarios sobre ciertas rutas o directorios del repositorio.

El fichero se divide en secciones, cada una de las cuales especifica un repositorio y una ruta hasta él. Las rutas se especifican como [nombreRepositorio:ruta] o simplemente [ruta].

En cada sección se especifican también los nombres de los usuarios autenticados y las reglas de acceso que cada uno tendrá para cada directorio: *r* para solo lectura y *rw* para lectura y escritura. Si un usuario no se menciona en una sección del fichero significa que no tiene permitido ningún tipo de acceso al directorio del repositorio representado en esa sección.

Si se está usando SVNParentPath y no se especifica un repositorio particular, la regla de acceso se aplicará a todos los repositorios bajo SVNParentPath. Si, por ejemplo, se tiene una sección llamada [ /fuentes ] sin especificar en qué repositorio, todas las reglas de acceso que se definan en dicha sección afectarán a todos los repositorios que tengan una ruta con ese nombre. Si se está usando la directiva SVNPath no hay ningún problema en usar solo la ruta al directorio puesto que solo hay un repositorio.

Por ejemplo, para el repositorio definido en el Apartado 5.5.1 se podría dar permiso de lectura y escritura para el directorio pruebas sistema software solo a los miembros del equipo de pruebas. Estos usuarios también tendrían acceso al directorio sistema software. Por tanto, se define el acceso para el directorio codigoFuente<sup>4</sup>. Por otra parte, los encargados de ejecutar las pruebas generadas por el equipo de pruebas y de depurar el código del sistema de tráfico, tendrían acceso de lectura, pero no de escritura para el directorio pruebas sistema software. Sin embargo, tendrían acceso total al directorio sistema software. Cual-

---

<sup>4</sup> Los directorios sistema software y pruebas sistema software son subdirectorios de otro superior llamado codigoFuente.

quier otro usuario tendría bloqueado el acceso a estos directorios del repositorio. Estos permisos se definirían con las siguientes secciones en el fichero de control de acceso:

```
[/codigoFuente]
daniel = rw
almudena = r
[/codigoFuente/sistema software]
almudena = rw
```

Los permisos son heredados por los directorios hijos. Así, `daniel` tendrá acceso de lectura y escritura para los directorios dentro de `pruebas sistemas software`. También se puede especificar en los subdirectorios accesos diferentes, por ejemplo, para `almudena`:

```
[/codigoFuente/pruebas sistema software]
daniel = rw
almudena = r
[/codigoFuente/sistema software]
almudena = rw
```

También se puede denegar el acceso a un directorio de forma explícita. Por ejemplo, para que `daniel` solo tenga acceso a los directorios `src` y `data` de `sistema software`:

```
[/codigoFuente]
daniel = rw
almudena = r
[/codigoFuente/sistema software]
almudena = rw
[/codigoFuente/sistema software/build]
daniel =
[/codigoFuente/sistema software/config]
daniel =
```

En este caso, `daniel` tendrá acceso total a todo el directorio `codigoFuente` pero no tendrá ningún permiso para `sistema software/build` y `sistema software/config`.

Cuando un usuario aparece en varias secciones del fichero de control de acceso, prevalece el permiso establecido para la ruta más específica. El módulo `mod_authz_svn` busca después el permiso para el directorio padre de esa ruta, etc. Es decir, el permiso establecido para una ruta prevalece sobre el permiso heredado de los directorios anteriores de nivel superior.

Por omisión, nadie tiene ningún permiso de acceso al repositorio. Si el fichero de control de acceso estuviera vacío nadie podría acceder al repositorio.

Para definir permisos a “todos los usuarios” se puede utilizar la variable `*`. Un ejemplo de uso sería el siguiente:

```
[/]
* = r
```

Esta regla de acceso significa que se da permiso de lectura a todos los usuarios en la raíz del repositorio, si se trabaja con `SVNPath`. Pero si se está usando `SVNParentPath`, ese mismo permiso se da a todos los usuarios en todos los repositorios, puesto que no se especifica el nom-



bre de ninguno de ellos. Después se pueden especificar permisos de *rw* a ciertos usuarios, en ciertos subdirectorios, en repositorios específicos.

La variable *\** es el único patrón que encaja con un usuario anónimo. Si se ha configurado el bloque `<Location>` como de acceso mixto, todos los usuarios empiezan accediendo a Apache de forma anónima. El módulo `mod_authz_svn` busca un valor *\** en las reglas definidas para la ruta que se quiere acceder. Si no puede encontrarlo, entonces es cuando se pide la autenticación al usuario.

Se pueden definir reglas de acceso para varios usuarios a la vez definiendo grupos de usuarios. Los grupos de usuarios se definen de la siguiente manera:

```
[groups]
pruebas = daniel, idoia
fuentes = daniel, almudena
todos = daniel, almudena, idoia
```

La definición de los grupos se coloca al principio del fichero de acceso. En las reglas de acceso, para distinguir un usuario de un grupo de usuarios, se antepone el carácter `@` al nombre del grupo. Por ejemplo:

```
[/codigoFuente]
@pruebas = rw

[/codigoFuente/sistema software]
@fuentes = rw
idoia = r
```

También se pueden definir grupos que contengan otros grupos. Por ejemplo:

```
[groups]
pruebas = daniel, idoia
fuentes = daniel, almudena
todos = @pruebas, @fuentes
```

## 5.8. Puesta en marcha

Una vez instalado el servidor, creados los usuarios y definidos los permisos de acceso, se pone en marcha el servidor para trabajar con el repositorio. Para ello, es necesario ejecutar Apache como un servicio.

Después, y una vez creada la estructura del repositorio en un ordenador local cualquiera (normalmente el del administrador), se procede a cargarla en el repositorio. Esta tarea se realiza mediante la operación Importar. Los detalles de la ejecución de esta operación se pueden ver en el Apartado 5.9.4.1.

## 5.9. Trabajando con Subversion: TortoiseSVN

TortoiseSVN es una shell, o interfaz de acceso, basada en Windows que permite acceder a los repositorios de Subversion con el Explorador de Windows. TortoiseSVN es un cliente para el sis-

tema de control de versiones Subversión. Se trata de software de código abierto para Windows que sustituye al cliente de la línea de comandos de Subversión facilitando enormemente la interacción con el repositorio. Cuando se trabaja con TortoiseSVN, realmente se trabaja con Subversion. En consecuencia, los conceptos de copia de trabajo, revisión, confirmación de cambios, actualización de la copia de trabajo, etc. se mantienen.

En este apartado se describe cómo instalar TortoiseSVN, y cómo utilizar esta herramienta dentro del ciclo de trabajo con el repositorio. Para ello, se describirán en detalle las operaciones básicas de interacción con Subversion necesarias para un adecuado control de versiones.

### 5.9.1. Instalación

Para poder trabajar con TortoiseSVN es necesario Windows 2000 SP2, Windows XP o superiores.

El software de TortoiseSVN se puede descargar de <http://tortoisesvn.tigris.org/>. Incluye un programa de instalación muy fácil de utilizar: se ejecuta dicho programa y se siguen las instrucciones. Si se desea instalar TortoiseSVN para todos los usuarios del equipo es necesario tener privilegios de administrador. Sin esos derechos el software sólo se instalará para el usuario actual. Una vez instalada la herramienta es necesario volver a arrancar el ordenador para poder usarla.

La interfaz de usuario está traducida a muchos idiomas. Entre otros, al castellano y al catalán. Los paquetes de idiomas están disponibles en forma de un instalador y se pueden descargar de la misma página que TortoiseSVN. Solo se tiene que ejecutar el programa de instalación y seguir las instrucciones. Si al ejecutar TortoiseSVN aún no está disponible la traducción, ejecutar TortoiseSVN→Configuración a través del menú contextual. Se abrirá una ventana en la que se puede seleccionar el idioma. Solo se podrán seleccionar los idiomas para los que se ha instalado el paquete correspondiente. Seleccionar un idioma actúa sobre el corrector de mensajes de registro y los textos de ventanas y menús.

### 5.9.2. Conexión con el repositorio

Ya se ha comentado al describir Subversion que normalmente los usuarios trabajan sobre una copia de trabajo local del repositorio. Por tanto, antes de conectarse al repositorio por primera vez hay que preparar el directorio o carpeta donde se va a almacenar esa copia de trabajo.

Realizada esta tarea, para conectarse a un repositorio, lo primero que se hará es obtener la información del proyecto a la que se tiene acceso, ejecutando la operación SVN Obtener... A esta operación se accede a través del menú contextual que aparece al presionar el botón derecho del ratón sobre la carpeta preparada para alojar la copia de trabajo. Creada la copia de trabajo (véase Apartado 5.9.4.2), se procede a trabajar sobre dicha copia.

### 5.9.3. Ciclo de trabajo

Después de crear la copia local del repositorio, todo está listo para comenzar el trabajo diario de desarrollo. Aunque TortoiseSVN permite realizar muchas operaciones, algunas de ellas no se uti-

lizan con frecuencia. En este apartado se pretende proporcionar una visión global de la forma de trabajar con el repositorio durante una jornada de trabajo típica. El ciclo de trabajo diario es el siguiente:

- Actualizar la copia de trabajo. La operación que permite esta tarea es:
  - SVN Actualizar.
- Realizar los cambios precisos en la copia de trabajo local. En este caso podemos:
  - Añadir nuevos elementos software (ficheros) a la configuración.
  - Borrar elementos software existentes.
  - Copiar elementos software.
  - Modificar el contenido de los ficheros.
- Revisar los cambios realizados y ver en qué estado está la copia de trabajo.
  - SVN diferenciar.
  - SVN revertir.
- Fusionar los cambios de la copia de trabajo con otros cambios que se hayan añadido al repositorio. De nuevo se utilizará SVN Actualizar y además:
  - Resuelto... que sobrescribe la versión del repositorio con la versión de la copia de trabajo cuando se ha producido un conflicto y se desea resolver el mismo con la versión de la copia de trabajo.
- Finalmente, se confirman los cambios realizados en la copia de trabajo, para llevarlos al repositorio:
  - SVN Confirmar...

En el apartado siguiente se describe cómo realizar estas operaciones, entre otras.

## 5.9.4. Operaciones básicas

### 5.9.4.1. Importar datos al repositorio: Importar...

Una vez establecida la estructura que va a tener el repositorio, se importa para llevarlo al servidor. Antes de realizar esta operación conviene quitar todos los ficheros que no se necesitan para construir la versión del proyecto. Por ejemplo: ficheros temporales, ficheros generados al compilar, etc. Esto se puede automatizar configurando Subversion de forma que ignore los ficheros que se ajusten a determinados patrones. Esta configuración se realiza a través de TortoiseSVN→Configuración.

Para importar el proyecto se selecciona la carpeta superior de la estructura de directorios del proyecto en el ordenador local. En el menú contextual se selecciona la operación TortoiseSVN→Importar... y TortoiseSVN solicita la URL del repositorio al que se quiere importar el proyecto (véase Figura 5.2). También se solicita un mensaje que se utiliza como un mensaje de registro.



**Figura 5.2.** Diálogo Importar.

Muchas de las operaciones de TortoiseSVN solicitan que se escriba un mensaje que detalle la operación realizada. Estos mensajes no son imprescindibles pero son muy importantes para recorrer la historia del repositorio y entender los cambios que se han realizado. De esta forma la existencia de cada nueva versión de un elemento software o de cada nueva modificación del repositorio tiene una justificación que puede ser consultada en cualquier momento.

Al aceptar, TortoiseSVN importa el árbol completo de directorios incluyendo todos los ficheros en el repositorio (salvo los que satisfacen los patrones de ignorar). Cuando se termina de importar el contenido del repositorio se muestra una ventana con la lista de ficheros y carpetas importados. El proyecto ahora está almacenado en el repositorio bajo el control de versiones.

Hay que tener en cuenta que la carpeta que se ha importado ¡NO está bajo el control de versiones! Para obtener una copia de trabajo local bajo el control de versiones se necesita obtener la versión que se acaba de importar.

Otra cosa a tener en cuenta es que no se deben importar al repositorio ficheros cuyo contenido sea temporal o se pueda obtener a partir de otros ficheros. Este es el caso, por ejemplo, de los ficheros `.class`. Por tanto, conviene limpiar estos archivos antes de hacer la ope-

ración de importación. Para hacer esto puede ayudar definir un objetivo de “limpieza” en el documento Ant<sup>5</sup>.

#### 5.9.4.2. Obtener una copia de trabajo: SVN Obtener...

La operación SVN Obtener... (en inglés, *SVN Checkout...*) hace una copia de trabajo en un directorio local, que se habrá creado previamente. Para ejecutar esta operación se selecciona el directorio local donde se desea poner la copia de trabajo. En el menú contextual se selecciona la operación TortoiseSVN→SVN Obtener..., que mostrará el cuadro de diálogo de la Figura 5.3. Si se introduce un nombre de carpeta que aún no exista, se creará un directorio con ese nombre. En el campo “URL de repositorio:”, se puede especificar una carpeta del repositorio, para obtener solo una parte del mismo.

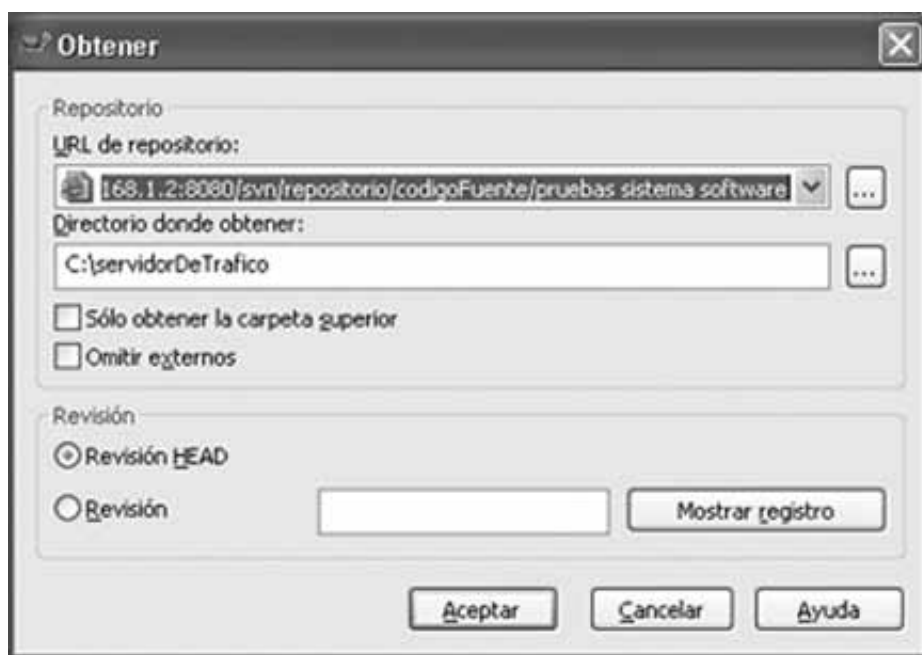


Figura 5.3. Diálogo Obtener.

Al terminar la operación de Obtener..., TortoiseSVN muestra una lista de los ficheros y carpetas recuperados desde el repositorio.

SVN Obtener... se debería hacer en un directorio vacío. Si se quiere obtener la revisión actual de repositorio en el mismo directorio desde el que se realizó la importación, Subversion dará un mensaje de error porque no puede sobrescribir los ficheros no versionados que ya existen, con los ficheros versionados. Se debe obtener la copia de trabajo en un directorio diferente o borrar antes la carpeta con el árbol existente.

<sup>5</sup> Véase el Capítulo 3 para consultar cómo trabajar con Ant y cómo definir objetivos que ayuden al desarrollador a construir un sistema software.








Marcando la opción “Sólo obtener la carpeta superior” se obtiene sólo la carpeta superior y se omiten todas las subcarpetas.

Si se marca esta opción, las actualizaciones en la copia de trabajo se deben realizar con la operación TortoiseSVN→Actualizar a la revisión... en vez de TortoiseSVN→Actualizar, puesto que la actualización estándar incluirá todas las subcarpetas y todos los vínculos externos.

Mientras se crea la copia de trabajo aparece una lista de los ficheros y carpetas que se están transfiriendo desde el repositorio.

Una vez obtenida la copia desde el repositorio se pueden ver los ficheros en el explorador de Windows con un icono sobreimpresionado sobre el icono original del fichero. Según el estado que cada fichero tenga en el repositorio y en la copia el icono es diferente. En la Tabla 5.1 se muestran los iconos existentes y su significado.

**Tabla 5.1.** Iconos de TortoiseSVN

Icono	Significado
	El fichero o carpeta permanece sin ser modificado respecto a la copia que se obtuvo del repositorio.
	Permite identificar los ficheros modificados desde la última vez que se actualizó la copia de trabajo, y que necesitan ser confirmados.
	Se ha producido un conflicto durante una actualización.
	Fichero de sólo lectura. Con este icono se indica que se debe obtener un bloqueo antes de editar ese fichero.
	Recuerda que se debe liberar el bloqueo si no se está utilizando el fichero para permitir a los demás que puedan confirmar sus cambios al fichero.
	Muestra que algunos ficheros o carpetas dentro de la carpeta actual se han marcado para ser borrados del control de versiones, o bien que falta un fichero bajo control de versiones dentro de una carpeta.
	El fichero o carpeta está programado para ser añadido al control de versiones.

#### 5.9.4.3. Enviar los cambios al repositorio: SVN Confirmar...

Una vez realizado el trabajo y los cambios necesarios hay que confirmar esos cambios y grabarlos en el repositorio para que estén al alcance de todos los participantes en el proyecto. Para confirmar los cambios al repositorio se utiliza la operación SVN Confirmar... (en inglés, *Commit*). Pero antes de confirmar hay que estar seguro de que la copia de trabajo está actualizada. Es decir, comprobar si otros usuarios han confirmado cambios desde la última vez que se recogió información del repositorio. El objetivo de esta actualización es determinar si hay conflictos, para resolverlos antes de confirmar los últimos cambios y evitar inconsistencias en el repositorio. Para actualizar la copia de trabajo se puede o bien ejecutar TortoiseSVN→SVN Actualizar

directamente, o bien ejecutar antes TortoiseSVN→Comprobar Modificaciones, para ver qué se ha cambiado localmente o en el servidor.

Si la copia de trabajo está actualizada y no hay conflictos, se pueden enviar los cambios al repositorio. Se seleccionan los ficheros y/o carpetas que se deseen confirmar y se selecciona la operación TortoiseSVN→SVN Confirmar... (véase Figura 5.4).



**Figura 5.4.** Diálogo *Confirmar*.

SVN Confirmar... permite publicar los cambios de cualquier número de ficheros y carpetas como una única transacción atómica. No es necesario hacerlo fichero a fichero. De hecho en la Figura 5.4 se observa la lista de cambios hechos en una copia de trabajo. Cuando el repositorio acepta una confirmación crea un nuevo estado del árbol de ficheros, es decir, crea una nueva revisión. A cada revisión se le asigna un número natural único, mayor que la revisión anterior. La revisión inicial de un repositorio se numera como 0 y consiste en un directorio raíz vacío. El número de revisión se aplica a un árbol completo, no a ficheros o carpetas particulares. Así, la revisión N representa el estado del repositorio tras la confirmación N-ésima. De este modo, las revisiones N y M de un fichero no tienen por qué ser diferentes.

Cuando se hace una confirmación, además de crearse una nueva revisión, si la operación ha requerido la autenticación del usuario, se graba el nombre del usuario en la propiedad `svn:author` de dicha revisión.

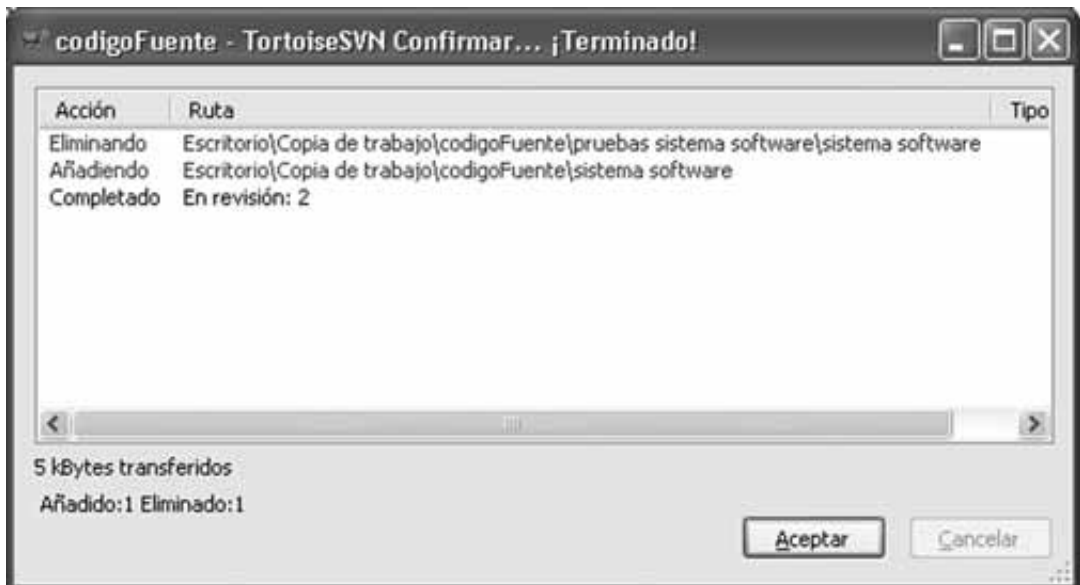
La parte inferior de esta ventana muestra una lista con todos los ficheros modificados, los ficheros añadidos, borrados o no versionados. Si no se desea que un fichero cambiado se confirme, se quita la marca de selección de ese fichero. Si se desea incluir un fichero no versionado, se marca para añadirlo a la confirmación. Las columnas se pueden personalizar, de forma que se puede seleccionar qué información mostrar y cambiar el ancho de las columnas. Haciendo doble clic en cualquier fichero modificado de esta lista, se ejecuta una herramienta externa de diferenciar para mostrar los cambios internos que se han producido en los ficheros a confirmar. También se pueden arrastrar ficheros desde aquí a otra aplicación.

Conviene escribir un mensaje de registro que describa los cambios que se están enviando al repositorio. De este modo es más fácil saber qué ocurrió cuando sea necesario revisar la historia del proyecto. TortoiseSVN incluye un corrector ortográfico para estos mensajes. Este corrector señalará las palabras mal escritas. Con el menú contextual se puede acceder a las correcciones sugeridas.

### ¿Confirmar ficheros o carpetas?

Cuando se confirman ficheros, solo se muestran los ficheros seleccionados. Cuando se confirma una carpeta se seleccionan, de forma automática, los ficheros que han cambiado. Si se olvida un fichero nuevo que se haya creado en esa carpeta, al confirmarla se encontrará y aparecerá en la lista.

Tras aceptar la confirmación de los cambios en el repositorio aparece una ventana de progreso (véase Figura 5.5). Los diferentes colores indican si se trata de una modificación, borrado o añadido.



**Figura 5.5.** El diálogo Progreso mostrando el progreso de una confirmación.

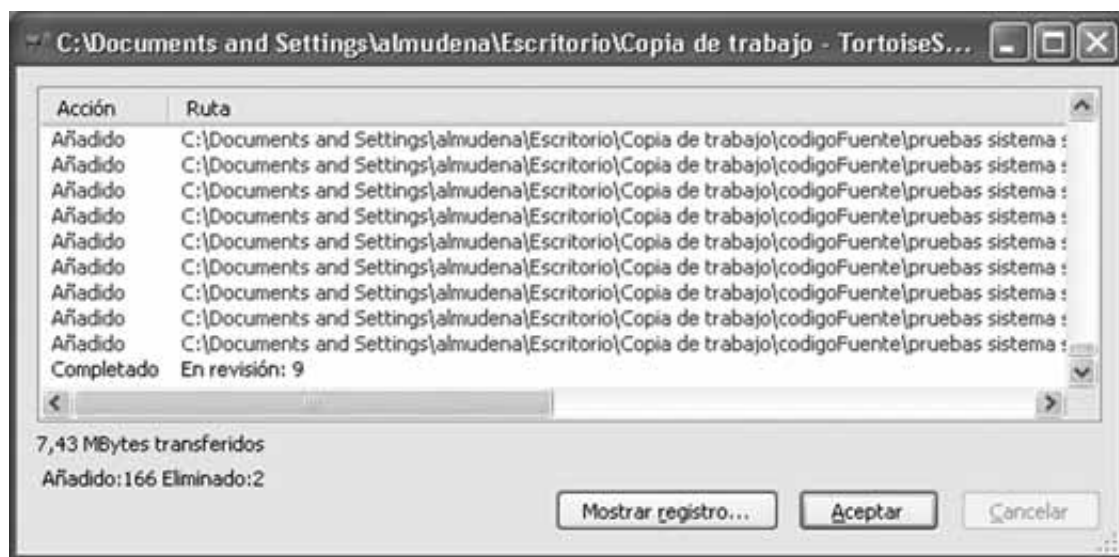


Al igual que para la operación de importación no se deben confirmar ficheros cuyo contenido sea temporal o se pueda obtener a partir de otros ficheros. Al igual que antes, puede ayudar en esta tarea previa a la confirmación definir un objetivo de “limpieza” en el documento Ant<sup>6</sup>.

#### 5.9.4.4. Actualizar una copia de trabajo: SVN Actualizar

Periódicamente, conviene asegurarse de que los cambios que hacen los demás se incorporen en la copia de trabajo local, para no trabajar en una copia desactualizada. Para obtener estos últimos cambios se utiliza la operación TortoiseSVN→SVN Actualizar (en inglés, *Update*). La actualización puede hacerse en ficheros sueltos, en un conjunto de ficheros o en directorios completos.

Cuando se ejecuta esta operación se abre una ventana como la de la Figura 5.6 que muestra el progreso de la actualización según se ejecuta. Los cambios que los demás hayan hecho se fusionarán con los ficheros de la copia local, guardando cualquier cambio de la copia de trabajo en los mismos ficheros. El repositorio no cambia cuando se hace una actualización.



**Figura 5.6.** Diálogo de progreso mostrando una actualización terminada.

Es posible que los cambios en la copia de trabajo se hayan realizado sobre las mismas líneas que han cambiado en el repositorio. En este caso se detecta un conflicto o colisión, que se muestra en rojo en la ventana. Haciendo doble clic en esas líneas se ejecuta una herramienta de fusión externa para resolver esos conflictos. Para ver cómo resolver los conflictos consultar el Apartado 5.9.4.5.

La operación SVN Actualizar actualiza la copia de trabajo a la revisión HEAD (la más reciente) del repositorio, que es el caso de uso más común. Si se desea actualizar a otra revisión, la operación adecuada es TortoiseSVN→?Actualizar a la revisión.... Actualizar a una revisión concreta que no tiene que ser la última, puede ser útil a veces para ver cómo estaba el reposito-

<sup>6</sup> Véase el Capítulo 3 para consultar cómo trabajar con Ant y cómo definir objetivos que ayuden al desarrollador a construir un sistema software.

rio en un momento anterior en su historia. Pero en general, actualizar ficheros individuales a una revisión anterior no es una buena idea, ya que deja la copia de trabajo en un estado inconsistente.

A veces, la actualización falla mostrando un mensaje donde se indica que ya existe un fichero local con el mismo nombre. Esto ocurre cuando Subversion<sup>7</sup> intenta obtener un fichero recién versionado, y se encuentra un fichero no versionado del mismo nombre en la copia de trabajo. Subversion nunca sobrescribirá un fichero no versionado —puede contener algo en lo que está trabajando, y que casualmente tiene el mismo nombre de fichero que otro desarrollador ha utilizado para su recién confirmado fichero. Si se obtiene este mensaje de error, la solución es renombrar el fichero local sin versionar. Tras completar la actualización, se puede comprobar si el fichero renombrado sigue siendo necesario. Si siguen saliendo mensajes de error, es mejor utilizar la operación TortoiseSVN→Comprobar modificaciones para mostrar todos los ficheros con problemas.

#### 5.9.4.5. Resolver conflictos: Editar conflictos y Resuelto...

Como se ha comentado en el apartado anterior, se pueden producir conflictos cuando se actualiza la copia de trabajo. Un conflicto ocurre cuando dos o más desarrolladores han hecho cambios en las mismas líneas de un fichero.

Para localizar el conflicto se abre el fichero en cuestión y se buscan líneas que empiecen con el texto <<<<<<. Subversion marca el área conflictiva de la siguiente forma:

```
<<<<<< nombre-del-fichero
cambios de la copia de trabajo
=====
código fusionado del repositorio
>>>>>> revisión
```

Además, para cada fichero en conflicto, Subversion crea tres ficheros adicionales junto al fichero que ha producido el conflicto:

- nombre-del-fichero.ext.mine

Este es el fichero de la copia de trabajo tal como estaba antes de actualizar (sin marcadores de conflicto). Este fichero tiene sus últimos cambios en él y nada más.

- nombre-del-fichero.ext.rREV-ANTIGUA

Este es el fichero que se obtuvo antes de empezar a hacer los últimos cambios.

- nombre-del-fichero.ext.rREV-NUEVA

Este es el fichero que se acaba de recibir desde el servidor. Este fichero corresponde a la revisión HEAD del repositorio.

Para resolver el conflicto se puede usar una herramienta externa de fusiones o editor de conflictos que se activa mediante TortoiseSVN→Editar Conflictos. También se puede utilizar cualquier otro editor para resolver el conflicto. Se hacen los cambios necesarios y se graba el fichero.

Después, se ejecuta el comando TortoiseSVN→Resuelto... y se confirman los cambios al repositorio. La operación Resolver realmente no resuelve el conflicto. Simplemente elimina los fi-

---

<sup>7</sup> Se recuerda que el control de versiones lo realiza realmente Subversion puesto que TortoiseSVN no es más que la herramienta de interacción con Subversion.

cheros `nombre-del-fichero.ext.mine` y `nombre-del-fichero.ext.r*`, dejándole confirmar los cambios.

Si el conflicto surge con ficheros binarios, Subversion nunca intentará mezclar dichos ficheros por sí mismo. El fichero local se mantendrá sin cambios y se crearán los ficheros `nombre-del-fichero.ext.r*`. Para descartar los cambios de la copia de trabajo y quedarse con la versión del repositorio se utiliza la operación Revertir... Si se desea mantener la versión de la copia de trabajo y sobrescribir la versión del repositorio, se utiliza el comando Resuelto... y se confirma.

#### 5.9.4.6. Registro de revisiones: Mostrar registro

En ocasiones es necesario consultar revisiones anteriores, para recuperar cierto elemento software y ver qué cosas cambiaron en una determinada revisión. Un modo de ver las revisiones que hay para un cierto fichero del repositorio es la operación TortoiseSVN→Mostrar registro (*Show log*). Para poder conocer qué se ha hecho en cada revisión y qué es lo que ha cambiado, es muy importante, como ya se ha mencionado, escribir un mensaje de registro cada vez que se confirme un cambio. Así se tendrá un registro detallado del proceso de desarrollo.

Al ejecutar esta operación se abre una ventana que recoge todos esos mensajes de registro y los muestra según se ve en la Figura 5.7.

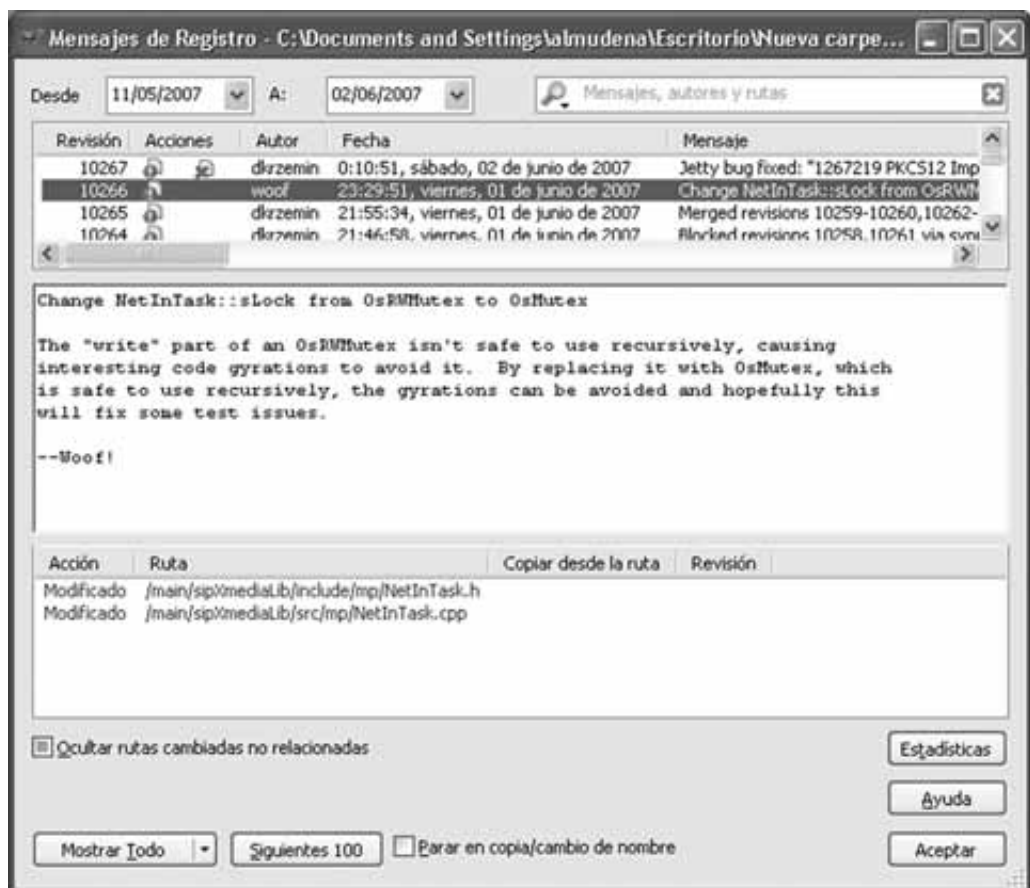


Figura 5.7. El diálogo de Registro de revisiones.

En la parte superior de esta ventana se muestra una lista de revisiones. Este resumen incluye el número de revisión, la fecha y la hora, la persona que confirmó la revisión y el inicio del mensaje de registro. La parte central muestra el mensaje de registro completo para la revisión seleccionada. Y la parte inferior presenta una lista de los ficheros y carpetas que se cambiaron como parte de la revisión seleccionada.

A través de esta ventana se puede obtener más información acerca de la historia del proyecto o acerca de una revisión determinada: si en la lista superior se selecciona una revisión cualquiera y se abre el menú contextual, aparecerá una lista de operaciones que se pueden realizar. Algunos ejemplos son:

- Comparar las revisiones seleccionadas con la copia de trabajo.
- Grabar la revisión seleccionada a un fichero para que pueda tener una versión más antigua de ese fichero.
- Actualizar la copia de trabajo a la revisión seleccionada.
- Revertir los cambios que se hicieron en la revisión seleccionada. Estos cambios se re- vierten en la copia de trabajo, por lo que esta operación no afecta al repositorio. Esto des- hará únicamente los cambios hechos en esa revisión. No reemplaza la copia de trabajo con el fichero entero en la revisión anterior. Es muy útil para deshacer un cambio anterior cuando se han hecho además otros cambios que no tienen que ver.
- Revertir a una revisión anterior. Sirve cuando se han hecho varios cambios, y luego se ne- cesita dejar las cosas como estaban en la revisión N. Los cambios se re- vierten en la copia de trabajo, por lo que esta operación no afecta al repositorio a menos que se confirmen los cambios. Esta operación deshacerá todos los cambios realizados tras la revisión selecciona- da, reemplazando el fichero o carpeta con la versión anterior.
- Editar el mensaje de registro o el autor adjunto a una confirmación anterior.

También se pueden seleccionar dos revisiones a la vez y compararlas.

Si se trabaja sobre la parte inferior de la ventana se puede trabajar con los ficheros y car- petas individuales de la revisión que esté seleccionada en la parte superior. Por ejemplo, se pue- den mostrar las diferencias hechas en la revisión seleccionada sobre el fichero seleccionado, abrir el fichero seleccionado, revertir los cambios hechos al fichero seleccionado en esa revisión o grabar la revisión seleccionada a un fichero para poder tener una versión antigua de ese fichero.

#### 5.9.4.7. Otras operaciones

En este apartado se enumeran y describen muy brevemente otras operaciones que pueden rea- lizarse con Subversion y TortoiseSVN. No se trata de una lista exhaustiva, sino de aquellas más comunes y que resultan más útiles para el control de versiones. Para mayor detalle consultar [2].

- Revertir... (*Revert...*):

Con esta operación se pueden deshacer los cambios que se hayan hecho en un fichero des- de la última actualización. Se selecciona el fichero y se ejecuta la operación Tortoi- seSVN→Revertir. Aparecerá una lista de los ficheros que han cambiado y que se pueden revertir.

Revertir sólo deshace los cambios locales. No deshace ningún cambio que ya haya sido confirmado. Para deshacer todos los cambios que se confirmaron en una revisión en particular, ver el Apartado 5.9.4.6.

- Comprobar modificaciones... (*Check for modifications...*)

A veces es muy útil saber qué ficheros se han modificado y qué ficheros han cambiado y confirmado los demás. Para averiguarlo se utiliza la operación TortoiseSVN→Comprobar Modificaciones.... Se mostrarán los ficheros modificados en la copia de trabajo y los ficheros no versionados que pueda haber. Con el botón Comprobar Repositorio también se pueden comprobar los cambios en el repositorio. Así, se puede comprobar la posibilidad de que aparezca un conflicto antes de actualizar una copia de trabajo. El estado de cada elemento se mostrará con distintos colores según un elemento modificado localmente, un elemento añadido, un elemento borrado, etc.

Para comprobar los cambios locales se puede ejecutar Comparar con Base (*Compare with Base*) en el menú contextual. Para comprobar los cambios en el repositorio hechos por los demás se utiliza la operación Mostrar Diferencias como Diff Unificado (*Show Differences as Unified Diff*).

Para ver dentro de los ficheros lo que ha cambiado se utiliza la operación Diferenciar (*Diff*).

- Añadir... (*Add...*)

Cuando se crean nuevos ficheros o directorios en el proceso de desarrollo, es necesario añadirlos al control de versiones. La operación para realizar esta tarea es TortoiseSVN→Añadir.... Una vez añadido el fichero es necesario confirmar la copia de trabajo para que esos ficheros y directorios estén disponibles para otros desarrolladores. Solo añadir un fichero o directorio en la copia de trabajo no afecta al repositorio.

- Eliminar... (*Delete...*)

Para borrar un fichero, o carpeta, bajo control de versiones, se debe utilizar TortoiseSVN→Eliminar... Como en el caso de Añadir... es necesario confirmar la copia de trabajo para que los ficheros borrados desaparezcan del repositorio.

Si el borrado se hace directamente desde el explorador de Windows no se podrá confirmar la copia de trabajo.

- Renombrar... (*Rename...*)

La operación de renombrar (TortoiseSVN→Renombrar...), se realiza como un borrado seguido de un añadir. Pero el fichero original (antes de ser renombrado) permanece en su lugar original en el repositorio. Por tanto, se debe borrar ese fichero del repositorio, confirmando dicho borrado. Si no se confirma la parte eliminada de los ficheros renombrados, se quedarán en el repositorio y cuando otros usuarios actualicen sus copias de trabajo no se eliminará el fichero antiguo.

Los cambios de nombre de carpeta se deben confirmar antes de cambiar cualquier fichero dentro de la carpeta; si no, la copia de trabajo puede quedar estropeada.

- Limpiar (*Clean up*)

La copia de trabajo se puede quedar en un estado inconsistente si una operación de Subversion no se puede completar correctamente. Por ejemplo, por problemas en el servidor. En ese caso se debe utilizar TortoiseSVN→Limpiar en la carpeta sobre la que no se ha podido completar la operación.

- Rama/Etiqueta... (*Branch/Tag...*)

Una de las características de los sistemas de control de versiones es la posibilidad de aislar cambios en una línea separada de desarrollo. Esto se conoce como una rama. Las ramas se utilizan a menudo para probar nuevas características sin molestar la línea principal del desarrollo. Cuando esa nueva característica es lo suficientemente estable, la rama de desarrollo se fusiona de nuevo en la rama principal.

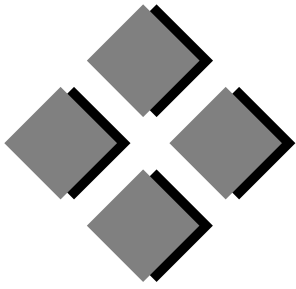
- Fusionar... (*Merge...*)

Mientras que las ramas se utilizan para mantener líneas de desarrollo separadas, en algún momento se tienen que fusionar los cambios hechos en una rama con la línea principal de desarrollo.

El proceso de fusión trabaja generando una lista de diferencias entre dos puntos del repositorio, y aplicando esas diferencias a la copia de trabajo. Conviene realizar las fusiones en una copia de trabajo sin modificar.

## 5.10. Bibliografía

- Collins-Sussman, B.; Fitzpatrick, ÇB. W. y Pilato, C. M.: *Version Control with Subversion*, 2006. <http://svnbook.red-bean.com/nightly/en/svn-book.pdf>
- Küng, S., Onken, L. y Large, S.: *TortoiseSVN. Un cliente de Subversion para Windows*, Versión 1.4.1. 2006.



# Capítulo 6

## Generación de informes sobre las pruebas

---

### SUMARIO

- |                                                                     |                                                   |
|---------------------------------------------------------------------|---------------------------------------------------|
| <b>6.1.</b> Introducción                                            | <b>6.3.</b> Informes sobre alcance de las pruebas |
| <b>6.2.</b> Informes con los resultados de ejecución de las pruebas | <b>6.4.</b> Bibliografía                          |

### 6.1. Introducción

A lo largo de este libro se cubre un amplio abanico de técnicas de prueba de software. Estas técnicas tienen como objetivo detectar el mayor número de defectos software que hayan sido introducidos durante etapas previas de la fase de desarrollo. Mediante el uso de las herramientas adecuadas es posible poner en práctica estas técnicas para automatizar de forma efectiva un alto porcentaje de los casos de prueba definidos. Estos mecanismos de automatización, que como se ha visto giran por término general en torno a la herramienta JUnit, tienen como último objetivo producir información acerca de los defectos encontrados. La calidad de esta información es de vital importancia ya que va a ser posteriormente utilizada por el desarrollador para encontrar en el código fuente la localización del defecto software y corregirlo. A lo largo de este capítulo se van a ver diferentes herramientas para presentar la información sobre el resultado de las pruebas al desarrollador de forma que se adapte a sus necesidades.

Otro tipo de información muy interesante para el desarrollador responsable de la definición de los casos de prueba y, en general para todo el equipo de pruebas e incluso el departamento de calidad, son los informes de cobertura de código. Este tipo de informes se pueden generar durante la ejecución de los casos de prueba y proporcionan información sobre la cobertura que dichos casos de prueba definidos efectúan sobre el código de producción o lo que es lo mismo, el

código objetivo de la prueba. En el Apartado 3 de este capítulo se presenta la herramienta Cobertura, que permite realizar esta tarea de una forma sencilla. Asimismo, se discutirá la forma correcta de interpretar la información, de naturaleza cuantitativa, contenida en esos informes para evaluar la calidad del código de pruebas generado, estimar recursos y decidir si se han alcanzado los objetivos fijados en el plan de pruebas.

## 6.2. Informes con los resultados de ejecución de las pruebas

En el Capítulo 2 se han visto diferentes mecanismos de ejecución del código de pruebas mediante JUnit. Básicamente, se presentaron dos formas de ejecutar los casos de prueba en las que la diferencia principal reside en la forma de presentar los resultados, a través de una interfaz gráfica (tipo AWT o JFC) o bien en modo texto. Adicionalmente, en el Capítulo 3 se introduce la herramienta Ant y cómo utilizarla para el despliegue y ejecución del código de pruebas. Ant utiliza la tarea `<junit>` para la ejecución de las clases de prueba y, por omisión, muestra el resultado de la ejecución de las pruebas en formato texto.

Estos mecanismos son perfectamente válidos y recomendables cuando el desarrollador está trabajando en su propia máquina y se trata de aplicaciones de escaso volumen en las que los resultados de las pruebas se pueden visualizar cómodamente sin necesidad de hacer scroll. Sin embargo, para aquellas aplicaciones en las que el volumen del código de pruebas es significativo, existen herramientas que permiten generar la información resultante de las pruebas en un formato navegable como es HTML. Estos documentos navegables pueden ser, por ejemplo, publicados en un sitio Web para estar accesibles a todo el equipo de pruebas.

### 6.2.1. Utilización de la tarea JUnitReport de Ant para generar informes con los resultados de ejecución de las pruebas

La herramienta Ant<sup>1</sup> (utilizada típicamente en cualquier proyecto de envergadura) presenta opcionalmente<sup>2</sup> una tarea llamada `<junitreport>` que se utiliza en combinación con la tarea `<junit>` (encargada de la ejecución de las clases de pruebas) para generar informes con los resultados de las pruebas en diferentes formatos. La mejor forma de conocer esta tarea es mediante un ejemplo:

A continuación se muestra el cuerpo del `<target>` de nombre `junit` definido en el documento Ant<sup>3</sup> responsable de la ejecución de las clases de prueba del sistema software presentado en el Apéndice B de este libro.

<sup>1</sup> Es necesario utilizar una versión de Ant no anterior a la 1.7 ya que solo a partir de esta versión existe soporte para utilizar la tarea `<junitreport>` para clases de prueba que utilizan las versiones 4.X de JUnit.

<sup>2</sup> La tarea `<junitreport>` es una tarea opcional de Ant por lo que necesita de librerías externas para funcionar. En particular necesita una librería para trabajar con documentos XML llamada Xalan XSLTC, que viene incorporada con la JDK 1.5 (necesaria para utilizar JUnit 4.x). Anteriores versiones de la JDK utilizan otras versiones de Xalan.

<sup>3</sup> El documento completo se encuentra en la ruta: `build.xml`...



pruebas sistema software/build.xml

```

<!-- Ejecucion de los casos de prueba -->
<target name="junit" depends="compile" description="Execute Unit Tests" >
  <junit printsummary="yes" fork="true" haltonerror="no" haltonfailure="no"
    showoutput="yes">
    <formatter type="xml"/>
    <classpath refid="project.junit381.class.path"/>
    <batchtest todir="${reports}/junitreport">
      <fileset dir="${build}">
        <include name="pruebasSistemaSoftware/junit381/*Test.class" />
      </fileset>
    </batchtest>
    <classpath refid="project.junit42.class.path"/>
    <batchtest todir="${reports}/junitreport">
      <fileset dir="${build}">
        <include name="pruebasSistemaSoftware/junit42/*Test.class" />
      </fileset>
    </batchtest>
  </junit>
  <!-- Generacion de informes de resultados en formato html -->
  <junitreport todir="${reports}/junitreport">
    <fileset dir="${reports}/junitreport">
      <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames" todir="${reports}/junitreport"/>
  </junitreport>
</target>

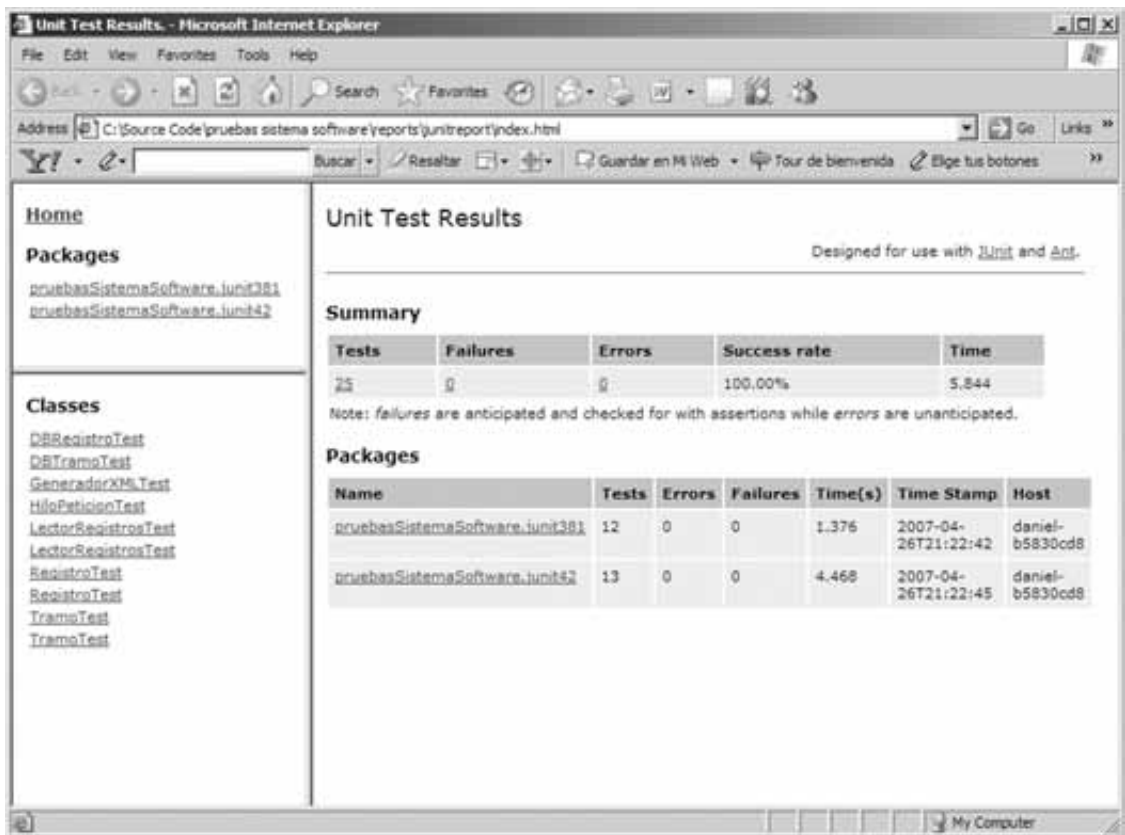
```

Como puede observarse, inicialmente se hace uso de la tarea `<junit>` para ejecutar todas las clases de prueba<sup>4</sup> como se vio en el Capítulo 3. Es necesario que el elemento `<formatter>`, que indica el formato de salida en el que se van a almacenar los resultados, tenga el atributo `type` con el valor `xml`. De esta forma la tarea `<junit>` generará documentos XML conteniendo los resultados de ejecución de cada una de las clases de prueba ejecutadas. Estos documentos reciben un nombre que se construye con el prefijo “TEST-” seguido del nombre de la clase sobre la cual contienen información. El segundo paso es definir la tarea `<junitreport>`, que va a tomar la información contenida en dichos documentos XML y la va a combinar produciendo un documento con formato HTML. El atributo `todir` indica el directorio de destino de los informes que se van a generar. Típicamente ha de ser una carpeta que se cree en el `<target>` de inicialización del documento Ant y se elimine en el `<target>` de limpieza. El elemento `<fileset>` se utiliza para indicar que documentos xml van a ser utilizados para generar el informe combinado. Por un lado se especifica el atributo `dir` que ha de contener el directorio en el que dichos documentos se encuentran. Por otro lado, se utilizan elementos `<include>` para añadir los ficheros que se desee. En este caso se indica el patrón `TEST-*.xml` que toma todos los archivos de resultados generados por la tarea `<junit>`. Finalmente, se utiliza el elemento `<report>` para seleccionar la forma en que se desea generar el documento HTML. Recibe los siguientes atributos:

<sup>4</sup> Nótese que las clases de prueba están divididas en dos paquetes dependiendo de la versión de JUnit que se haya utilizado en su construcción.

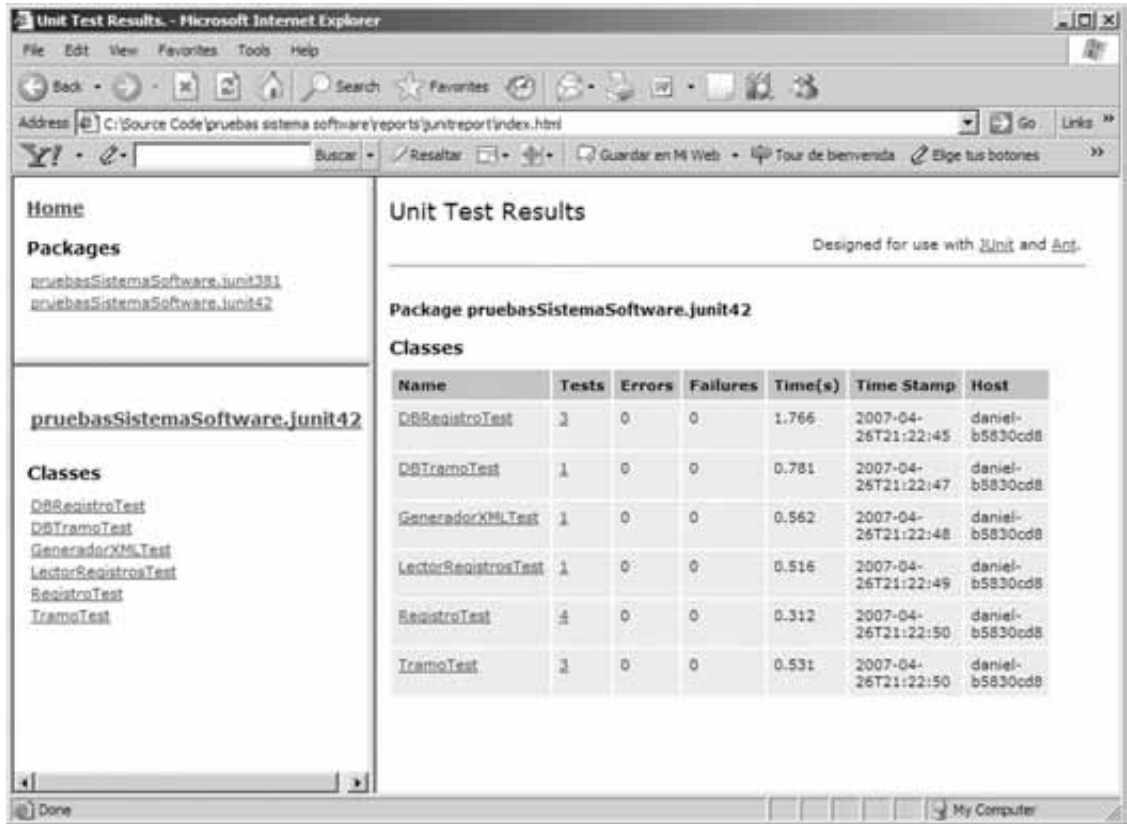
- **format**: este atributo puede tomar dos valores `frames` o `noframes` para construir documentos HTML que contengan frames (marcos) o no, respectivamente. Esta característica es interesante puesto que no todos los navegadores Web soportan el uso de frames.
- **styledir**: este atributo es especialmente interesante ya que permite al desarrollador indicar el directorio donde se encuentran el documento XSL que desea que se utilice para la transformación de documentos XML generados por `<junit>` en el documento HTML final. Ant cuenta por defecto con una hoja de estilo XSL que realiza esta tarea. Sin embargo, el desarrollador puede desear personalizar el formato del documento HTML definiendo su propia hoja de estilo.
- **todir**: contiene el directorio donde almacenar el documento HTML generado.

En la Figura 6.1 se muestra un ejemplo de documento HTML generado mediante JUnitReport. Para ello se ha utilizado el código Ant del ejemplo anterior.



**Figura 6.1.** Documento raíz del informe con los resultados de la ejecución de las clases de prueba.

Puesto que se ha utilizado el valor `frames` para el atributo `format` del elemento `report`, la página aparece dividida en tres secciones o *frames*. En la sección central se puede observar la información relativa a los paquetes de clases de prueba. Esta información es básicamente la misma que se ofrece siempre sólo que más estructurada y dentro de un documento navegable. En la Figura 6.2 aparece otro documento del mismo informe, esta vez conteniendo las clases de



**Figura 6.2.** Documento que muestra los resultados de la ejecución de las clases de prueba pertenecientes al paquete pruebasSistemaSoftware.junit42.

prueba pertenecientes al paquete pruebasSistemaSoftware.junit42. Finalmente, en la Figura 6.3 se muestra el documento con los resultados de ejecución de los métodos de la clase `RegistroTest`. Se ha introducido deliberadamente un defecto en el código fuente de la clase `RegistroTest` de forma que pueda verse cómo se presenta la información relativa a un fallo. Esta información es básicamente el mensaje producido por el método `assert`<sup>5</sup> que ha detectado el fallo, en la Figura “expected:<15> but was:<16>”. Sin embargo, como siempre, se sigue echando de menos información acerca del caso de prueba en el que se ha encontrado el fallo, lo que facilita enormemente su detección. En ocasiones los datos que aparecen en la condición que no se ha verificado sirven para determinar de forma única cuál es el caso de prueba que ha fallado, sin embargo esto no es cierto como norma general. Por último, comentar que las tareas `<junit>` y `<junitreport>` han sido definidas dentro del mismo `<target>`, por lo que se ejecutarán secuencialmente. En ocasiones puede interesar definir las en `<target>` separados de forma que sólo se invierta tiempo en generar documentación cuando esta realmente se va a utilizar.

<sup>5</sup> Entiéndase por método `assert` todo aquel que perteneciendo a la clase `Assert` de JUnit se utiliza para comprobar condiciones, por ejemplo `assertEquals`.

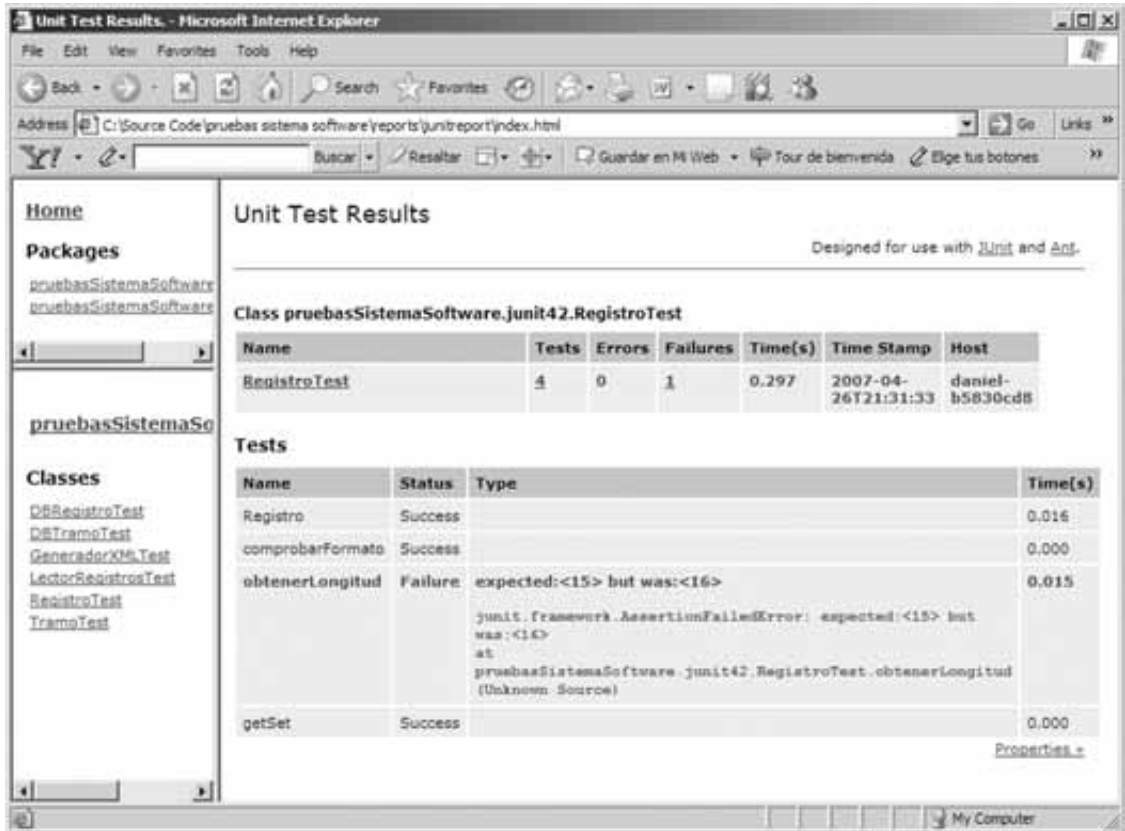


Figura 6.3. Documento que muestra los resultados de la ejecución de los métodos de prueba de la clase de prueba RegistroTest.

## 6.2.2. Otras librerías de interés: JUnit PDF Report

Otra librería de interés que puede ser utilizada para mejorar la usabilidad y en general la calidad de los informes de resultados de ejecución de pruebas es JUnitPDFReport. Se trata de una herramienta de código abierto que se encuentra disponible para libre descarga en el sitio Web <http://junitpdfreport.sourceforge.net/>. Como puede adivinarse, esta herramienta es capaz de generar informes en formato PDF que presenta las ventajas de ser multiplataforma y muy adecuado para lograr altas calidades de presentación e impresión.

Al igual que JUnitReport, JUnitPDFReport se utiliza en forma de tarea Ant. Sin embargo, en este caso la tarea no es una tarea opcional de Ant sino que debe ser definida explícitamente para ser utilizada<sup>6</sup>. A continuación se muestra el procedimiento de utilización de esta tarea dentro del documento Ant que se viene utilizando como ejemplo a lo largo de este capítulo. Nótese que la

<sup>6</sup> Un requerimiento adicional a considerar durante la instalación es el uso de la librería batik que debe ser descargada aparte. En particular el archivo .jar que representa esta librería debe ser situado en la carpeta lib de la instalación de JUnitPDFReport.

tarea `<junitpdfreport>` debe ser definida dentro del mismo `<target>` de nombre `junit` y a continuación del elemento `<junit>` que genera los documentos XML con los resultados de ejecución de cada una de las clases de prueba.

1. Definición de la tarea `<junitpdfreport>`: simplemente se ha de añadir la siguiente línea en el documento Ant:

```
<!-- necesario para la herramienta junitpdfreport -->
<import file="${lib}/junitpdfreport_essentials_1_0/build-junitpdfreport.xml"/>
```

esta línea referencia otro documento Ant donde se define la tarea. Este documento Ant forma parte de la distribución de JUnitPDFReport y debe ser almacenado en el disco en alguna carpeta del proyecto.

2. Utilización de la tarea `<junitpdfreport>` de forma similar a como se utiliza `<junitreport>`<sup>7</sup>. Basta con añadir el siguiente segmento de código XML al documento Ant.

```
<!-- Generacion de informes de resultados en formato pdf -->
<junitpdfreport todir="${reports}/pdfreport" styledir="default">
  <fileset dir="${reports}/junitreport">
    <include name="TEST-*.xml"/>
  </fileset>
</junitpdfreport>
```

Salvo el atributo `styledir` del elemento `<junitpdfreport>` todo es análogo a la tarea `<junitreport>` por lo que no se entrará en más detalles. Este atributo puede tomar cuatro valores diferentes que permiten elegir la estructura y contenido del documento PDF que se va a generar. Los posibles valores se listan a continuación.

styledir	descripción
default	El documento generado tiene un aspecto muy similar al del informe HTML generado por JUnitReport. Se incluye información detallada y también la salida estándar que es lo que habitualmente se observa cuando se ejecuta JUnit desde una ventana de comandos.
brief	Proporciona información muy somera, simplemente una lista de los métodos de prueba que presenta tres secciones: métodos que contienen fallos, métodos que contienen errores y métodos que han finalizado con éxito. Es útil para localizar los defectos de un simple vistazo.
structural	Muestra los resultados resumidos y organizados de forma jerárquica en una única tabla.
graphical	Proporciona información en forma de barras sobre los porcentajes de casos de prueba exitosos y fallidos.

En la Figura 6.4 se muestra un ejemplo de documento PDF generado con esta herramienta, el `styledir` utilizado es `structural`.

<sup>7</sup> De hecho JUnitReport y JUnitPDFReport pueden ser utilizadas simultáneamente tal y como se demuestra en el documento `pruebas sistema software/build.xml`.

The screenshot shows the Adobe Acrobat Professional interface with a PDF document titled 'Test results' dated 2007-4-26. The document contains a table with the following data:

Name	Successes	Errors	Failures	Time
Total	24	0	1	5.747s
pruebas Sistema Software junit01	12	0	0	1.297s
HttpFecionTest	5	0	0	0.445s
testProcesarFecionHTTP	1	0	0	0.215s
testProcesarFecionHTTP2	1	0	0	0.000s
testConstruirRespuestasXML	1	0	0	0.000s
testConstruirRespuesta	1	0	0	0.187s
testRun	1	0	0	0.247s
LectorRegistrosTest	1	0	0	0.455s
testLeerRegistros	1	0	0	0.205s
RegistroTest	4	0	0	0.250s
testRegistros	1	0	0	0.000s
testGetSet	1	0	0	0.000s
testComprobarFormato	1	0	0	0.000s
testObtenerLongitud	1	0	0	0.000s
TramoTest	2	0	0	0.125s
testObtenerLongitud	1	0	0	0.062s
testComprobarFormato	1	0	0	0.000s
pruebas Sistema Software junit02	12	0	1	4.500s
DBRegistroTest	5	0	0	1.828s
almacenarRegistro	1	0	0	0.360s
obtenerTramosTráfico	1	0	0	0.547s
obtenerTramosCamiónCortados	1	0	0	0.455s
DBTramoTest	1	0	0	0.828s
almacenarTramo	1	0	0	0.344s
GeneradorXMLTest	1	0	0	0.547s
construirXMLCamiónesAccidentes	1	0	0	0.141s
LectorRegistrosTest	1	0	0	0.000s
LeerRegistros	0	0	0	0.000s
RegistroTest	3	0	1	0.212s
Registro	1	0	0	0.015s
comprobarFormato	1	0	0	0.000s

**Figura 6.4.** Documento generado con la herramienta JUnitPDFReport que contiene los resultados de la ejecución de las pruebas.

### 6.3. Informes sobre alcance de las pruebas

Como se ha visto en el Capítulo 1, una parte importante del plan de pruebas consiste en acotar el alcance<sup>8</sup> de las pruebas. El alcance básicamente indica qué partes del sistema software

<sup>8</sup> El alcance de las pruebas es también conocido como cobertura de las pruebas. Sin embargo, a lo largo de este capítulo se utilizará en la medida de lo posible el término alcance para evitar la confusión con el término cobertura en el contexto de pruebas de caja blanca. Véase Capítulo 1 para obtener más información al respecto.

objetivo van a ser probadas y cuáles no. Se trata de un paso necesario y previo a la planificación de recursos para la fase de pruebas que sirve para establecer unos objetivos bien delimitados. Idealmente no debería ser necesario definir el alcance ya que el alcance debería ser el todo, desafortunadamente los recursos son limitados por lo que se ha de llegar a una solución razonable.

El alcance de las pruebas se refiere a todo tipo de pruebas, desde pruebas unitarias hasta pruebas de validación pasando por pruebas de integración, etc. A lo largo de este apartado se va a tratar la forma de generar información sobre el alcance de las pruebas realizadas. Es decir, informes que indiquen qué métodos de qué clases han sido probados y cuáles no. Tales informes sirven para verificar que, efectivamente, se ha alcanzado, valga la redundancia, el alcance establecido en el plan de pruebas.

Existen multitud de herramientas con las que generar informes sobre el alcance obtenido. A continuación se citan algunas de ellas: Clover, EMMA, GroboUtils, NoUnit, Jester, Hansel, Quilt, jcoverage, Cobertura, etc. Debido a su madurez y a las características que presenta, a lo largo de este capítulo se va a trabajar con la herramienta Cobertura. Esta herramienta es de código abierto y se encuentra disponible para libre descarga desde SourceForge en el sitio Web <http://cobertura.sourceforge.net/>.

### 6.3.1. Utilización de la herramienta Cobertura para generar informes de cobertura

Cobertura es una herramienta muy fácil de utilizar y que permite generar informes muy completos sobre la cobertura que las pruebas definidas proporcionan sobre el código a probar. A continuación, se listan sus principales características.

- Proporciona información sobre la cobertura alcanzada para cada paquete y cada clase del sistema. En particular, muestra información de cobertura a nivel de sentencia y cobertura a nivel de bifurcaciones. Es decir, qué porcentaje de sentencias y bifurcaciones presentes en el código fuente han sido probadas por al menos un caso de prueba.
- Calcula la complejidad ciclomática<sup>9</sup> del código de cada clase y cada paquete del sistema. Así como la complejidad global del sistema.
- Genera informes en formato HTML o XML. Mientras que los primeros pueden ser directamente visualizados para interpretar los resultados, los segundos pueden, por ejemplo, ser utilizados a la entrada de una transformación XSLT para generar informes personalizados a la medida del desarrollador.
- Proporciona varias tareas Ant de forma que la herramienta puede ser utilizada fácilmente desde un documento Ant XML.
- No necesita el código fuente de la aplicación sobre la cual se quiere obtener información de cobertura. Esto es debido a que Cobertura trabaja directamente sobre el *bytecode*, es decir, sobre clases ya compiladas. Sin embargo aunque Cobertura puede funcionar sin el

---

<sup>9</sup> La complejidad ciclomática de un método, en este caso, representa el número de caminos independientes en dicho método y es un indicador del número de casos de prueba necesarios para alcanzar una buena cobertura. En el Capítulo 1 se trata esta cuestión en profundidad.

código fuente, es interesante disponer del mismo para poder observar las zonas de código que han sido cubiertas o no por los casos de prueba diseñados.

El mecanismo de funcionamiento de esta herramienta es bien sencillo. Simplemente lee el *bytecode* del código a probar y lo modifica añadiendo sentencias de código en lugares estratégicos (es fácil de imaginar) de cada método. De esta forma cuando los métodos son invocados durante la ejecución de las pruebas estas sentencias también son ejecutadas. Cobertura puede conocer qué sentencias de código han sido probadas, simplemente viendo cuáles de esas sentencias añadidas han sido ejecutadas.

Una interesante observación es que esta herramienta puede utilizarse tanto en un contexto de pruebas de caja blanca como de caja negra. La razón es que para calcular la cobertura, puesto que Cobertura trabaja a nivel de *bytecode*, no necesita el código fuente de la aplicación, sino únicamente los archivos *.class*. Sin embargo, como se verá más adelante, el hecho de no disponer del código fuente, hace que la herramienta sea incapaz de mostrar qué sentencias de código han sido o no cubiertas por las pruebas. Por este motivo el desarrollador no puede saber qué casos de prueba necesita añadir a los ya existentes para mejorar la cobertura.

A continuación se va a ver una descripción de los pasos necesarios para utilizar esta herramienta. Aunque es posible utilizarla desde la línea de comandos, es mucho más cómodo hacerlo desde el documento Ant del código de pruebas<sup>10</sup>. Una vez que la herramienta ha sido descargada e instalada, el primer paso es indicar a Ant la localización de las nuevas tareas definidas en Cobertura. Posteriormente, dichas tareas van a ser utilizadas a lo largo del documento Ant.

### 6.3.1.1. Indicar a Ant la localización de las nuevas tareas

Durante el procesamiento del documento Ant, Ant necesita conocer la definición de todas las tareas presentes en dicho documento. Por este motivo es necesario indicar dónde puede encontrar la definición de las nuevas tareas introducidas por Cobertura. Algunas de estas tareas son `<cobertura-instrument>`, `<cobertura-report>`, `<cobertura-merge>` y `<cobertura-check>`, y se encuentran definidas dentro del archivo que acompaña a la distribución de Cobertura. Simplemente añadiendo las siguientes líneas al documento Ant es posible empezar a trabajar con Cobertura.

```
<!-- Clases de la herramienta Cobertura -->
<path id="cobertura.classpath">
  <fileset dir="${lib}/cobertura-1.8">
    <include name="cobertura.jar" />
    <include name="lib/**/*.jar" />
  </fileset>
</path>

<!-- Definicion de las tareas de Cobertura -->
<taskdef classpathref="cobertura.classpath" resource="tasks.properties" />
```

Estas líneas forman parte del documento Ant `build.xml` que sirve para desplegar el código de pruebas de la aplicación descrita en el Apéndice B de este libro. El primer bloque define un `classpath` en el que se incluyen las clases de Cobertura que acompañan a la distribución y que se

<sup>10</sup> Para aquellos no familiarizados con Ant, véase el Capítulo 3, “Ant”.



encuentran en las rutas indicadas por los elementos `<include>`. El segundo bloque simplemente define las tareas utilizando dicho classpath.

### 6.3.1.2. Instrumentalización de las clases que van a ser probadas

Este paso es fundamental y consiste en modificar el *bytecode* de las clases del sistema a probar de forma que Cobertura pueda conocer cuáles de las sentencias contenidas en dicho código son ejecutadas durante el proceso de pruebas. Por otra parte, el *bytecode* es utilizado para calcular la complejidad ciclomática. La entrada de este paso son las clases del sistema, que pueden ser indicadas una por una o bien mediante archivos .jar que las contengan, mientras que la salida es el conjunto de ficheros de entrada con el *bytecode* modificado (o instrumentalizado). Véase con un ejemplo:

```
<!-- Instrumentalizacion de las clases a probar (paso previo para
      calcular la cobertura)-->
<target name="instrument" depends="init">
  <mkdir dir="${instrumented-classes}"/>
  <cobertura-instrument todir="${instrumented-classes}">
    <fileset dir="sistema software/build">
      <include name="servidorEstadoTrafico.jar" />
    </fileset>
  </cobertura-instrument>
</target>
```

Se puede observar que este `<target>` depende del `<target>` `init`, es decir, Ant debe ejecutar el `<target>` de nombre `init` antes del `<target>` `instrument`. Esto es debido a que el `<target>` `init` se encarga de compilar el código fuente a probar y generar el archivo `servidorEstadoTrafico.jar` que es el que va a ser instrumentalizado. A este respecto, es importante que el código a instrumentalizar sea compilado con la tarea `<javac>` utilizando el atributo `debug="yes"`. Esto es debido a que los archivos .class deben contener información de depuración, como son los nombres de clases métodos y atributos, que Cobertura necesita para poder generar la documentación sobre cobertura.

Es posible utilizar varios elementos `<include>` y `<exclude>` para indicar qué clases de las contenidas en el directorio indicado en el atributo `todir` deben ser instrumentalizadas y cuáles no. Por ejemplo, el siguiente fragmento de código XML indicaría que una hipotética clase `Operacion.java` no debe ser instrumentalizada.

```
<exclude name="Operación.java">
```

El atributo `todir` de la tarea `<cobertura-instrument>` indica el directorio donde se quiere que se almacenen las clases instrumentalizadas. En caso de que no se especifique, Cobertura simplemente sobrescribe las clases originales con las clases instrumentalizadas.

### 6.3.1.3. Ejecución de las pruebas sobre las clases intrumentalizadas

Una vez que se dispone de clases instrumentalizadas, es necesario modificar la tarea `<junit>` del documento Ant de forma que ahora se realicen las pruebas sobre ellas en lugar de sobre las clases originales. A continuación se incluye la tarea `<junit>` modificada, dentro del mismo documento Ant del ejemplo anterior.

```

<!-- Ejecucion de los casos de prueba -->
<target name="junit" depends="compile" description="Execute Unit Tests">
  <junit printsummary="yes" fork="true" haltonerror="no" haltonfailu-
    re="no"
      showoutput="yes">
    <formatter type="xml"/>
    <classpath location="${cobertura}/instrumented-classes/
      servidorEstadoTrafico.jar"/>
    <sysproperty key="net.sourceforge.cobertura.datafile"
      file="${cobertura}/
        cobertura.ser" />
    <classpath refid="project.junit42.class.path"/>
  <batchtest todir="${reports}/junitreport">
    <fileset dir="${build}">
      <include name="pruebasSistemaSoftware/junit42/*Test.class"/>
    </fileset>
  </batchtest>
</junit>
<junitreport todir="${reports}/junitreport">
  <fileset dir="${reports}/junitreport">
    <include name="TEST-*.xml"/>
  </fileset>
  <report format="frames" todir="${reports}/junitreport"/>
</junitreport>
</target>

```

En realidad las modificaciones introducidas son mínimas. Obsérvese que se ha añadido al classpath de la tarea <junit> el archivo `servidorEstadoTrafico.jar` que contiene las clases instrumentalizadas. Es fundamental que dicho archivo aparezca en el classpath antes que las clases originales del sistema. Esto es debido a que la máquina virtual de Java cuando procesa el classpath para la búsqueda de clases, siempre toma la primera clase que encuentra cuyo nombre se corresponde con el de la clase buscada. La pregunta que surge es: ¿por qué es necesario incluir las clases originales en el classpath además de las instrumentalizadas<sup>11</sup>? Existen dos motivos:

- Puesto que no todas las clases a probar tienen por qué ser instrumentalizadas (anteriormente se habló del elemento <exclude>), no todas las clases necesarias para la prueba van a estar localizadas en el directorio de clases intrumentalizadas que ya ha sido añadido al classpath.
- Es posible que se pretenda ejecutar la tarea <junit> sin la intención de generar informes de cobertura, en este caso, las clases instrumentalizadas no existirán y, por tanto, las pruebas se realizarán sobre las originales ya que estas sí que están presentes en el classpath.

La segunda modificación consiste en indicar la localización física en disco del archivo `cobertura.ser` mediante una propiedad de sistema que recibe el nombre de `net.sourceforge.cobertura.datafile`. Este archivo es utilizado por el código añadido por Cobertura durante la instrumentalización para serializar la información sobre sentencias invocadas durante la ejecución del código de pruebas.

---

<sup>11</sup> El archivo `servidorEstadoTrafico.jar` con las clases originales está incluido como un elemento del classpath referenciado con el identificador `project.junit42.class.path`.

Adicionalmente, Cobertura proporciona una tarea que permite combinar la información contenida en varios archivos .ser en un único archivo que puede ser utilizado posteriormente para generar un informe de cobertura combinado. Esto es especialmente útil cuando, por ejemplo, se quiere conocer la cobertura proporcionada para varias aplicaciones de un mismo sistema distribuido. La tarea se llama `<cobertura-merge>` y a continuación se muestra un ejemplo de uso.

```
<cobertura-merge>
  <fileset dir="${cobertura}">
    <include name="aplicacionA/cobertura.ser" />
    <include name="aplicacionB/cobertura.ser" />
  </fileset>
</cobertura-merge>
```

Esta tarea recibe un atributo opcional de nombre `datafile` para indicar el nombre del fichero .ser que será creado como resultado de la combinación de los archivos .ser incluidos en el elemento `<fileset>` (su valor por defecto es `cobertura.ser` y es creado en el directorio actual).

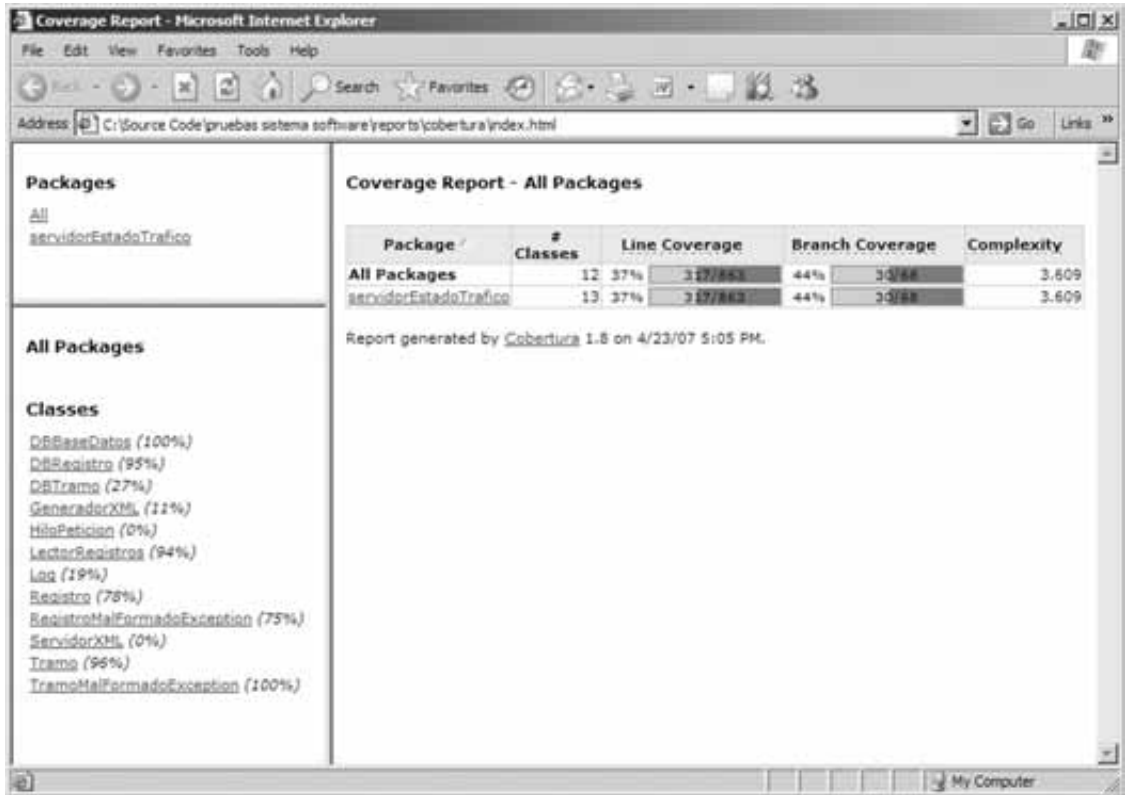
#### 6.3.1.4. Generación de los informes de cobertura

En este punto la ejecución de los casos de prueba mediante la tarea `<junit>` ha terminado, y toda la información sobre sentencias invocadas permanece almacenada en el archivo `cobertura.ser`. La generación de los informes con la información de cobertura se realiza a partir de la información contenida en este archivo mediante la tarea `<cobertura-report>`. A continuación se ve la forma de utilizar esta tarea:

```
<!-- Generacion del informe de cobertura -->
<target name="html-coverage-report" depends="instrument,junit">
  <cobertura-report destdir="${reports}/cobertura" srcdir="sistema
    software/src"/>
</target>
```

Simplemente se ha definido un `<target>` que se encarga de todo el proceso. Tiene como dependencias los `<target>` `instrument` y `junit` vistos anteriormente, es decir, lleva a cabo la instrumentalización de las clases y posterior ejecución de las pruebas sobre las clases instrumentalizadas. La tarea `<cobertura-report>` recibe dos atributos: `destdir` y `srcdir`. El primero de ellos representa el directorio donde se van a almacenar los ficheros que componen el informe de cobertura. El segundo representa el directorio donde se encuentra el código fuente de las clases a probar. Ambos atributos son obligatorios aunque es posible no asignar un valor válido al segundo de ellos en caso de que el código fuente no esté disponible. Lo que ocurre es que lógicamente el informe de cobertura no mostrará qué sentencias de código han sido cubiertas y cuáles no. Existen dos atributos opcionales para esta tarea, son `format` y `datafile`. El primero de ellos indica el formato en que se va a generar el informe, que puede ser HTML (valor por defecto) o bien XML. El segundo indica el nombre del archivo de serialización (el valor por defecto es `cobertura.ser`).

En la Figura 6.5 se muestra el documento raíz del informe generado (en formato HTML) para el sistema software del Apéndice B, que se corresponde con el código XML anterior. Como puede observarse, la página está dividida en tres secciones o *frames*. La sección superior izquierda permite seleccionar el paquete (*package*) de clases Java del sistema a probar cuya información de



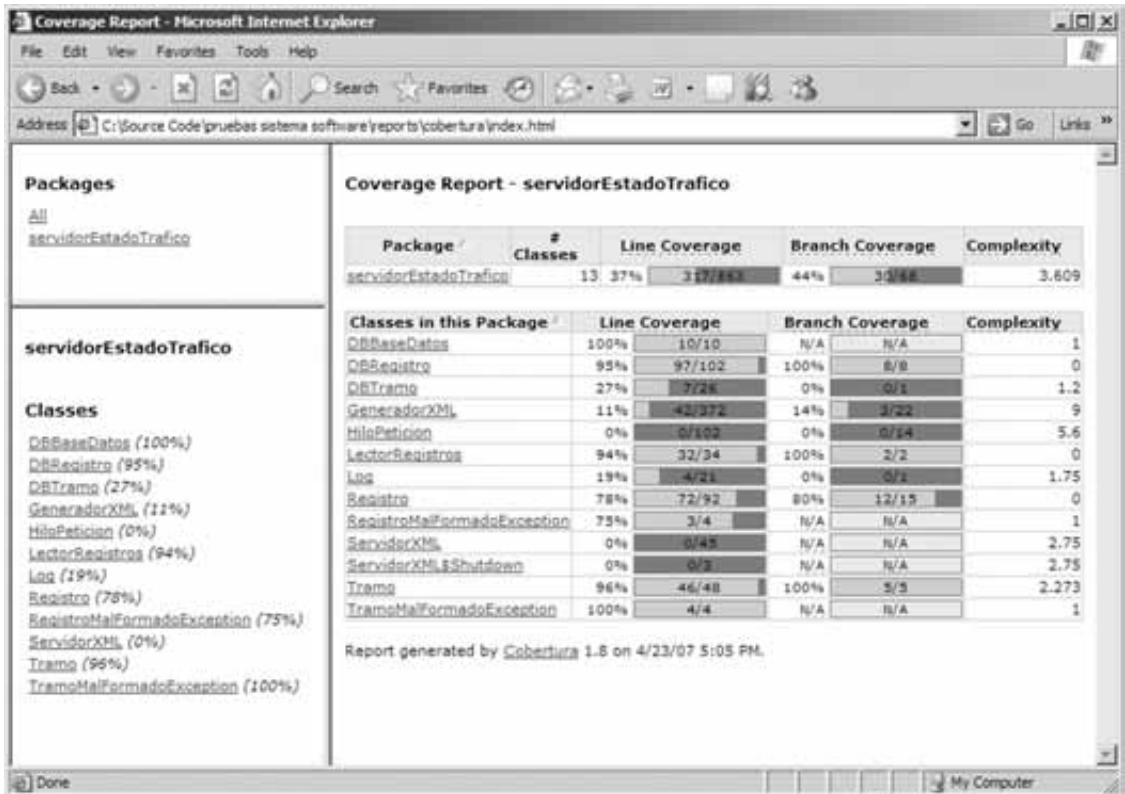
**Figura 6.5.** Documento raíz del informe de cobertura.

cobertura se desea visualizar. Pinchando sobre el enlace `servidorEstadoTráfico` aparecerá el documento mostrado a continuación. En la sección inferior izquierda aparecen todas las clases del sistema, pinchando sobre ellas se accede a los documentos de cobertura de clase.

En la sección de la derecha (la de mayor tamaño) aparece una tabla con la cobertura alcanzada para cada paquete del sistema y para el sistema completo (en este caso la información es idéntica en ambos casos ya que el sistema consta únicamente de un paquete). Las columnas de la tabla son el nombre del paquete, el número de clases que contiene, la cobertura de sentencia (“line coverage”), la cobertura de bifurcación (“branch coverage”) y en último lugar la complejidad ciclomática media. La cobertura de línea y de bifurcación está expresada tanto en valor porcentual (número de líneas o bifurcaciones cubiertas respecto al total) como en valor absoluto (número de líneas o bifurcaciones cubiertas del total presente en el código a probar).

En la Figura 6.6 se muestra el informe de cobertura de las clases contenidas en el paquete `servidorEstadoTráfico`. Se trata de un documento parecido al anterior en el que se muestra la información de cobertura particularizada para cada clase del paquete. Las barras de cobertura (dibujadas en dos colores) permiten visualizar rápidamente qué clases del paquete han recibido una buena cobertura y cuáles no. Para algunas clases la cobertura de bifurcación no tiene sentido ya que no existen bifurcaciones<sup>12</sup> sino que existe un único camino de ejecución.

<sup>12</sup> El lenguaje Java presenta numerosas sentencias de control que provocan bifurcaciones, algunas de ellas son los bucles `for` y `while` y las sentencias condicionales `if` y `else`.



**Figura 6.6.** Documento de cobertura del paquete `servidorEstadoTráfico`.

Finalmente, en la Figura 6.7 se muestra un informe de cobertura de clase. Este en particular corresponde a la clase `Registro`.

Los informes de cobertura de clase son muy interesantes ya que permiten, para cada método de la clase, ver cuales son las sentencias para las cuales se ha definido al menos un caso de prueba (aquellas marcadas con verde en el margen izquierdo) y las sentencias para las que no se ha definido ningún caso de prueba (aquellas sombreadas de color rojo). Esta información resulta de gran ayuda a la hora de diseñar los casos de prueba o verificar que los casos de prueba diseñados efectivamente están cumpliendo su objetivo (probando lo que deben). En la figura se ve el código del método `comprobarFormato` de la clase `Registro`, para el cual no se ha definido ningún caso de prueba que cubra las sentencias situadas en las líneas 104, 109 y 114. Rápidamente se observa que estas líneas corresponden a situaciones en los que el formato del objeto `Registro` es incorrecto y por lo tanto se lanza una excepción del tipo `RegistroMalFormadoException`. Si se pretende incrementar la cobertura del método bastaría diseñar tres casos de prueba adicionales que cubran tales sentencias.

### 6.3.1.5. Establecimiento y verificación de umbrales de cobertura

En este punto ya está claro el procedimiento para generar los informes de cobertura así como la correcta forma de interpretarlos. Sin embargo, la herramienta Cobertura aún tiene algunas sorpresas guardadas para el desarrollador. Se trata de un mecanismo para fijar umbrales de cober-

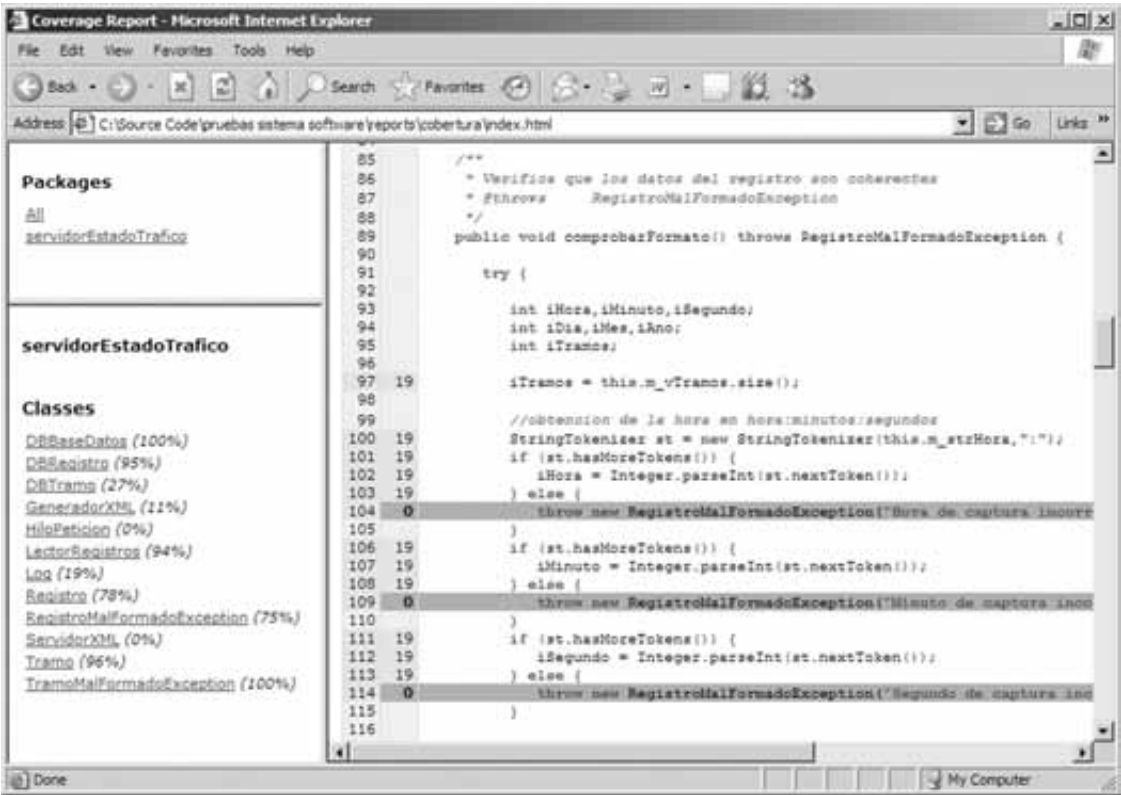


Figura 6.7. Documento de cobertura de la clase Registro.

tura, entendidos como valores mínimos de cobertura que el código de pruebas debe proporcionar. Este mecanismo se lleva a cabo por medio de la tarea <cobertura-check> que recibe una serie de atributos opcionales. En la siguiente tabla se comentan algunos de los más útiles (todos ellos reciben valor por defecto 0).

atributo	descripción
totallinerate	umbral mínimo para el porcentaje de cobertura de sentencia de todo el sistema.
totalbranchrate	umbral mínimo para el porcentaje de cobertura de bifurcación de todo el sistema.
packagelinerate	umbral mínimo para el porcentaje de cobertura de sentencia de todos los paquetes del sistema.
packagebranchrate	umbral mínimo para el porcentaje de cobertura de bifurcación de todos los paquetes del sistema.
linerate	umbral mínimo para el porcentaje de cobertura de sentencia de todas las clases del sistema.
branchrate	umbral mínimo para el porcentaje de cobertura de bifurcación de todas las clases del sistema.

Existe otro atributo llamado `haltonfailure` cuyo valor por defecto es `true` y que hace que la ejecución del documento Ant de despliegue del software falle en caso de no satisfacerse los umbrales de cobertura definidos. Esto es de gran utilidad ya que el desarrollador no puede pasar por alto esta circunstancia y, por tanto, es consciente de la necesidad de mejorar la cobertura.

Siguiendo con el mismo ejemplo se ha definido el `<target>` `check-coverage` que se encarga de verificar ciertos umbrales de cobertura. El segmento de código XML se muestra a continuación:

```
<!-- Verificacion de umbrales de cobertura -->
<target name="check-coverage" depends="instrument,junit,html-covera-
  ge-report">
  <cobertura-check totallinerate="30" totalbranchrate="50" linerate="20">
    <regex pattern="servidorEstadoTrafico.DBRegistro" branchrate="90"
      linerate="90"/>
    <regex pattern="servidorEstadoTrafico.DBTramo" branchrate="90"
      linerate="90"/>
  </cobertura-check>
</target>
```

En particular, el umbral de cobertura definido para el sistema completo es del 30% a nivel de sentencia y 50% a nivel de bifurcación. Resulta interesante ver cómo se ha utilizado el elemento `<regex>` para sobrescribir los umbrales de cobertura globales definidos en el elemento `<cobertura-check>`. En este caso se ha considerado que las clases `DBRegistro` y `DBTramo` son de una importancia crítica y por tanto se obliga a proporcionar una cobertura más alta que al resto. Una vez ejecutado este `<target>` el resultado que aparece en la consola de comandos es el siguiente:

```
check-coverage:
[cobertura-check] Cobertura 1.8 - GNU GPL License (NO WARRANTY) - See
  COPYRIGHT file
[cobertura-check] Cobertura: Loaded information on 13 classes.
[cobertura-check] servidorEstadoTrafico.Log failed check. Line covera-
  ge rate of 19.0% is below 20.0%
[cobertura-check] servidorEstadoTrafico.GeneradorXML failed check. Line
  coverage rate of 11.2% is below 20.0%
[cobertura-check] servidorEstadoTrafico.ServidorXML failed check. Line
  coverage rate of 0.0% is below 20.0%
[cobertura-check] servidorEstadoTrafico.ServidorXML$Shutdown failed
  check. Line coverage rate of 0.0% is below 20.0%
[cobertura-check] servidorEstadoTrafico.DBTramo failed check. Branch
  coverage rate of 0.0% is below 90.0%
[cobertura-check] servidorEstadoTrafico.DBTramo failed check. Line co-
  verage rate of 26.9% is below 90.0%
[cobertura-check] servidorEstadoTrafico.HiloPetition failed check. Line
  coverage rate of 0.0% is below 20.0%
[cobertura-check] Project failed check. Total branch coverage rate of
  44.1% is below 50.0%
```

BUILD FAILED

C:\Source Code\pruebas sistema software\build.xml:157: Coverage check failed. See messages above.

La ejecución del documento Ant ha fallado debido a que varios de los umbrales de cobertura no se han satisfecho. Nótese cómo los datos que aparecen se corresponden con los del documento de cobertura del paquete `servidorEstadoTrafico` de la Figura 6.

### 6.3.2. Interpretación de los informes de cobertura

No cabe duda de que los informes de cobertura de código son una fuente de información muy útil para evaluar la calidad del código de pruebas generado. Sin embargo, estos documentos solamente presentan información estadística. Por ejemplo, muestra que un 65% de las sentencias de código o un 57% de las bifurcaciones de una clase han sido ejercitadas en promedio durante el proceso de prueba. Sin embargo, estos números no expresan en sí mismos la forma en la que tales líneas de código han sido ejercitadas, o lo que es lo mismo, no hablan de la calidad con la que los casos de prueba han sido creados. A la hora de interpretar estos documentos se ha de tener siempre presente que una cobertura del 100% ya sea en sentencias, en bifurcaciones o en las dos cosas a la vez, que esté asociada a un informe de resultados libre de fallos, no garantiza ni mucho menos que se hayan encontrado todos los defectos del software que se está probando. A continuación se puede ver un ejemplo de esto mismo.

El método `conversion` recibe como parámetros un objeto `String strDato` y un valor boolean `bPrefijo` indicando si se ha de añadir el prefijo “dato.” Al comienzo del anterior `String`. Finalmente, se realiza una conversión a mayúsculas se añada o no el prefijo.

```
public String conversion(boolean bPrefijo, String strDato) {
    String strAuxiliar = null;

    if (bPrefijo) {
        strAuxiliar = "dato." + strDato;
    }

    return strAuxiliar.toUpperCase();
}
```

Para la prueba de este método se ha definido el siguiente método de prueba:

```
@Test public void conversion() {
    String strResultado = m_objeto.conversion(true, "prueba")
    assertEquals("DATO.PRUEBA", strResultado);
}
```

Una vez ejecutado el caso de prueba el informe de resultados dice que no ha habido ningún fallo y, lo que es más importante, el informe de cobertura dice que se ha alcanzado una cobertura del 100% en sentencias y del 100% en bifurcaciones. Todo esto es cierto y a primera vista hace pensar que el método `conversion` ha sido probado de una forma efectiva y suficiente. Nada más lejos de la realidad. Claramente, debido a un descuido, el desarrollador se ha olvidado de tratar adecuadamente el caso en que el parámetro `bPrefijo` vale `false`. En ese caso el método fallaría produciéndose la excepción `NullPointerException`. He aquí una situación engañosa, el informe de cobertura indica cobertura máxima y sin embargo existe un defecto fla-



grante oculto en el código fuente. Obviamente un correcto diseño de los casos de prueba <sup>13</sup> hubiese evitado esta circunstancia ya que al menos dos casos de prueba hubieran sido definidos, uno con el valor de `bPrefijo` a `true` y el otro a `false`.

En general, este patrón se repite con frecuencia en métodos que producen varios flujos de control debido al uso de condiciones. El siguiente ejemplo muestra un caso similar.

El método `informacionDatos` devuelve un mensaje indicando si hay datos disponibles. La forma de conocer si quedan datos es mediante el valor del parámetro `bDatosGuardados`, que indica si hay datos almacenados, y el objeto `Vector vNuevosDatos`, que contiene los nuevos datos (si los hay) que acaban de recibirse.

```
public String informacionDatos(boolean bDatosGuardados, Vector vNuevosDatos) {
    if ((bDatosGuardados) || (vNuevosDatos.size() > 0)) {
        return "quedan datos";
    }

    return "no quedan datos";
}
```

Para la prueba de este método se ha creado el siguiente método de prueba:

```
@Test public void informacionDatos() {
    String strResultado = m_objeto.informacionDatos(true,new Vector())
    assertEquals("quedan datos",strResultado);

    strResultado = m_objeto.informacionDatos(false,new Vector())
    assertEquals("no quedan datos",strResultado);
}
```

Como puede verse se han definido dos casos de prueba, uno negativo y uno positivo. El informe de resultados obtenido con `JUnitReport` indica que los casos de prueba se han ejecutado con éxito. Además, el informe HTML generado por `Cobertura` indica una cobertura del 100% en sentencias y del 100% en bifurcaciones. No obstante la situación se repite; existe un caso de prueba no definido (aquel para el cual el objeto `Vector` es `null`) que haría fallar al método. De nuevo, es necesario interpretar los informes de cobertura con mucha cautela.

Una vez visto esto, la cuestión es: ¿qué información útil y 100% veraz se puede extraer de un informe de cobertura? La respuesta es bien sencilla. El informe de cobertura permite hacerse una idea de qué es lo que falta por probar más que de qué partes del sistema se han probado de forma efectiva. Para obtener una respuesta a la efectividad de la prueba, es necesario revisar el trabajo realizado en el diseño de los casos de prueba. Esto se debe a la propia naturaleza cuantitativa de los datos presentes en el informe de cobertura, que nunca deben interpretarse de forma cualitativa.

En los dos siguientes apartados se van a tratar otras importantes aplicaciones de los informes de cobertura.

---

<sup>13</sup> El Capítulo 1 muestra información detallada acerca de cómo realizar un correcto procedimiento de diseño de casos de prueba para un método.

### 6.3.2.1. Estimación de recursos

Normalmente un equipo de desarrollo debe codificar clases al mismo tiempo que crea pruebas para las mismas. Se trata de un proceso de desarrollo en paralelo en el que muchas veces se fija como norma que ninguna clase debe ser subida al repositorio<sup>14</sup> sin su correspondiente clase de pruebas y habiendo pasado con éxito su ejecución. Sin embargo, a menudo debido a falta de tiempo (algo bastante común por desgracia en el mundo de desarrollo software) los desarrolladores empiezan a reducir esfuerzos en la tarea de codificación de clases de prueba para centrarse en el código de producción en sí. En estas situaciones, y aun existiendo una clase de prueba definida para cada clase creada, el ratio de métodos de prueba respecto a métodos a probar va decreciendo a la vez que el número de casos de prueba definidos para cada método decrece también. Una buena forma de que el responsable del equipo de desarrollo pueda advertir estas situaciones es mediante una correcta monitorización de los informes de cobertura.

Periódicamente se han de generar informes de cobertura sobre el código en el repositorio de forma que se pueda observar la evolución del grado de cobertura existente. Puesto que estos informes son muy útiles desde un punto de vista cuantitativo, es posible evaluar la calidad del proceso de desarrollo fijándose en el porcentaje de sentencias/bifurcaciones para las que no existe ningún caso de prueba definido y ejecutado. Si el responsable del equipo de desarrollo nota un descenso en la cobertura puede decidir solicitar más recursos o modificar la planificación de desarrollo en los términos que considere pertinentes con el fin de reconducir la situación.

### 6.3.2.2. Aseguramiento de la calidad de componentes software

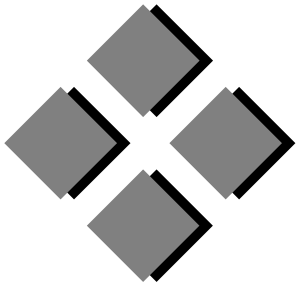
Gracias a la enorme utilidad de los informes de cobertura para detectar segmentos de código no probados, estos informes pueden ser utilizados como un mecanismo de alto nivel para la estimación de la calidad de un componente software desarrollado. De esta forma, componentes que no superen ciertos estándares de calidad para alguno de los datos que componen la estadística de cobertura serán devueltos al equipo de desarrollo.

## 6.4. Bibliografía

- Harold, E. R.: *Measure test coverage with Cobertura*, 3 de mayo de 2005, <http://www.ibm.com/developerworks/java/library/j-cobertura/>.
- Glover, A.: *In pursuit of code quality: Don't be fooled by the coverage report*, 31 de enero de 2006, <http://www.ibm.com/developerworks/java/library/j-cq01316/>.
- Hatcher, E.: *Automating the build and test process*, 14 de agosto de 2001, <http://www.ibm.com/developerworks/java/library/j-junitmail/>.

---

<sup>14</sup> Véase Capítulo 5, “Herramientas de control de versiones: Subversion (SVN)”.



# Capítulo 7

## Pruebas unitarias en aislamiento mediante Mock Objects: JMock y EasyMock

---

### SUMARIO

7.1. Introducción	7.5. Herramientas para la puesta en práctica de la técnica de Mock Objects: EasyMock y JMock
7.2. Diferencias entre Mock objects y Stubs	
7.3. Filosofía de funcionamiento de los Mock Objects	7.6. Comparativa entre EasyMock y JMock
7.4. Procedimiento general de utilización de Mock Objects	7.7. Bibliografía

## 7.1. Introducción

Es muy habitual encontrarse con diseños orientados a objetos en los que determinados objetos utilizan otros objetos para realizar ciertas tareas. Estos últimos son conocidos como objetos colaboradores y, habitualmente, son instanciados dentro del objeto original, o bien obtenidos desde alguna localización externa globalmente accesible. Por ejemplo, invocando a algún método de clase, accediendo a alguna variable global, etc.

Este tipo de relaciones entre objetos en las que un objeto contiene a uno o más llamados objetos colaboradores se llaman relaciones de asociación. Casos particulares de las relaciones de asociación son las relaciones de composición y agregación. Las relaciones de asociación, aunque a veces son un síntoma de escasa modularidad del diseño, en otras ocasiones son mecanismos completamente naturales y adecuados en el diseño de cierto componente software.

Un ejemplo de relación de asociación se puede observar entre las clases DBRegistro y DBTramo pertenecientes al sistema software presentado en el Apéndice B. En este caso, la clase DBTramo es clase colaboradora de la clase DBRegistro puesto que es utilizada por ésta para insertar en la base de datos los objetos de clase Tramo pertenecientes a un objeto de clase Registro. En realidad se trata de una relación de agregación ya que la clase DBTramo no puede contener a la clase DBRegistro y, por tanto, no se puede dar una relación cíclica. Por otro lado no puede ser una relación de composición puesto que DBRegistro no es responsable del ciclo de vida (creación y destrucción) del objeto de la clase DBTramo, sino que este objeto lo recibe como un parámetro en el constructor. En la Figura 7.1 se puede observar un diagrama UML que representa la relación entre estas dos clases. A ambos extremos de la línea de conexión se encuentra la multiplicidad de la asociación mientras que el rombo de color blanco indica que se trata de una agregación.



**Figura 7.1.** Relación de agregación entre las clases DBRegistro y DBTramo.

El problema que aparece cuando existen relaciones de asociación dentro del código es la dificultad de probar clases aisladamente. En particular, la dificultad estriba en probar la clase que hace uso de clases colaboradoras de forma aislada respecto a aquellas. El inconveniente es que si alguno de los casos de prueba creados y automatizados con JUnit falla no es posible conocer la procedencia de dicho fallo. ¿El fallo se ha producido debido a un defecto en el objeto que se está probando o bien el defecto se encuentra en alguno de los objetos colaboradores que el objeto a probar utiliza? En este contexto cabe plantearse la siguiente pregunta: ¿por qué no probar inicialmente aquellos objetos que no presentan relaciones de asociación? De esta forma, una vez verificado su correcto funcionamiento, el siguiente paso sería probar aquellos objetos que sí presentan relaciones de asociación. Esta estrategia es lo que comúnmente se conoce como desarrollo *bottom-up* (de abajo hacia arriba).

La estrategia *bottom-up* es especialmente útil en el desarrollo orientado a objetos dado que favorece la utilización de componentes software preexistentes. Es decir, favorece la reutilización.

Existe una corriente de desarrollo de software, *Test Driven Development* (desarrollo guiado por las pruebas) gran defensora de la estrategia *bottom-up*. El motivo es que facilita la construcción de software basada en pruebas. Este procedimiento consiste en lo siguiente<sup>1</sup>: se crea la clase de prueba de un objeto, se crea el objeto y cuando este supera las pruebas se continúa con el desarrollo del software de forma ascendente e incremental. Sin embargo, en la práctica, la estrategia *bottom-up* a menudo es utilizada en combinación con una estrategia *top-down* (de arriba hacia abajo) ya que permite hacerse una idea del sistema completo desde las fases iniciales.

Sin embargo, existen escenarios en los que la prueba de objetos aisladamente es de vital importancia. Esto es debido a que la implementación de determinados objetos que han de ser instanciados por el objeto a probar puede no estar disponible o bien estar sujeta a cambios. La ven-

<sup>1</sup> En realidad, según los principios de *Test Driven Development*, primero se crean los tests y luego los objetos realizándose todo ello de forma ascendente.

taja es que la interfaz con dichos objetos siempre está disponible por lo que puede resultar conveniente llevar a cabo las pruebas a la espera de que dichas implementaciones concluyan.

Por otro lado, es habitual encontrar objetos que presentan relaciones de asociación con otros objetos cuya instanciación y/o inicialización es muy costosa. Véase esto con un ejemplo. Supóngase que se pretende realizar pruebas sobre un objeto que utiliza otros objetos para obtener información de una base de datos. Nótese que en este caso el objetivo de la prueba no son los objetos que hacen de interfaz con la base de datos, sino el objeto que los utiliza. Más aún, supóngase que dichos objetos que interaccionan con la base de datos ya han sido desarrollados y probados siguiendo una estrategia ascendente. Es estas circunstancias, para probar el objeto va a ser necesario poner en marcha una base de datos que contenga determinada información para ser accedida durante las pruebas. Poner en marcha una base de datos puede ser una tarea relativamente costosa en tiempo, sobre todo cuando existen mecanismos que permiten prescindir de ella. Incluso cuando la base de datos esté en marcha, es mucho mas rápido simular las consultas a la base de datos mediante estos mecanismos que realizarlas de forma real <sup>2</sup>.

Estos mecanismos reciben el nombre de Mock Objects, cuya traducción literal al castellano sería algo así como “objetos simulados” y que, como se verá, representan un papel similar al de los resguardos comentados en el Capítulo 1 de este libro. La misión de estos objetos no es otra que la de permitir la prueba en aislamiento de una clase genérica, es decir, realizar la prueba sobre la clase separándola previamente del resto de clases con las que interactúa. A modo de resumen de lo anterior, esta necesidad aparece básicamente bajo las siguientes circunstancias:

- Una estrategia *bottom-up* no es recomendable o no es posible.
- La implementación de los objetos colaboradores del objeto a probar no está disponible.
- La inicialización de los objetos colaboradores es demasiado costosa.

Los Mock Objects son básicamente objetos que simulan a otros objetos a través de la adopción de su interfaz pública. Presentan las siguientes ventajas respecto a los objetos originales:

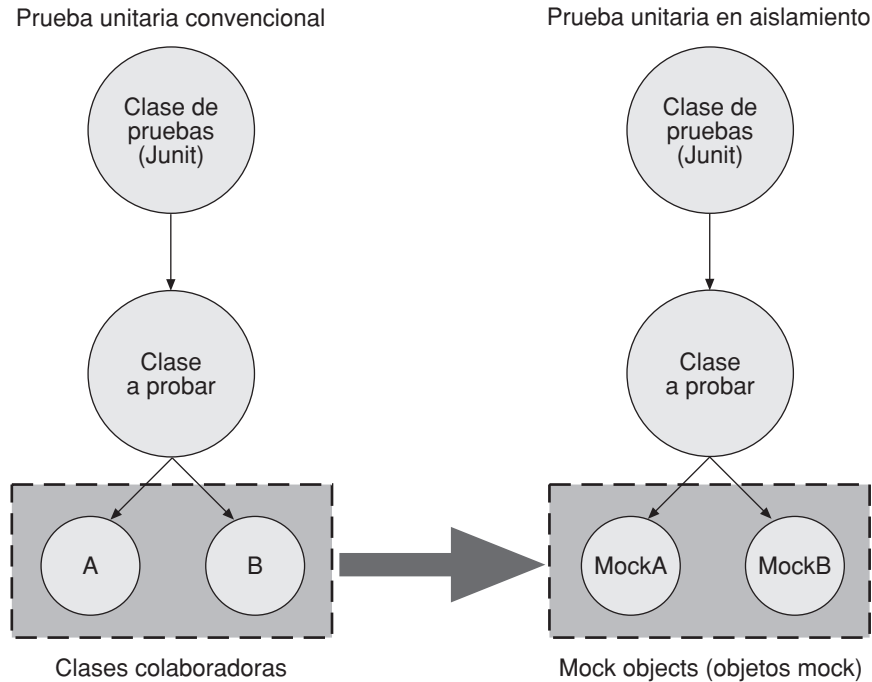
- Son fácilmente instanciables ya que no tienen otra misión que la de hacerse pasar por los objetos a los que simulan.
- Son irrompibles, es decir, no pueden fallar dado que no realizan ninguna tarea susceptible de fallo <sup>3</sup>.
- Pueden ser generados automáticamente mediante el uso de las herramientas adecuadas (<http://www.jmock.org/> o [www.easymock.org](http://www.easymock.org) ).
- Reducen el tiempo de ejecución de las pruebas. Esto es debido a que toda la funcionalidad asociada a terceros objetos contenidos en el objeto a probar es substituida por unas pocas líneas de código.

En la Figura 7.2 se muestra el esquema de pruebas unitarias convencional junto al esquema de pruebas unitarias en aislamiento utilizando la técnica de Mock Objects. Simplemente, se

---

<sup>2</sup> En el Capítulo 9 se realiza una discusión en profundidad sobre la utilidad de los Mock Objects para realizar dicha tarea.

<sup>3</sup> Ya que no pueden fallar, al probar una clase que hace uso de este tipo de objetos, si ocurre un fallo, automáticamente se sabe que dicho fallo pertenece a esa clase y no a alguna de las colaboradoras.



**Figura 7.2.** Comparación de esquemas de pruebas unitarias.

crean Mock Objects que sustituyen a las instancias de las clases colaboradoras. Claramente, siguiendo la nomenclatura utilizada en el Capítulo 1, la clase a probar juega el papel de conductor (haciendo el papel de una clase genérica del sistema real que utiliza los servicios de la clase a probar) mientras que los Mock Objects actúan de resguardos.

## 7.2. Diferencias entre Mock Objects y Stubs

La primera vez que se oye hablar del gran impacto que ha supuesto la aparición de la novedosa técnica para la prueba aislada de clases llamado Mock Objects, es inevitable pensar en los tradicionales *stubs*. La cuestión es: ¿cuáles son las aportaciones y ventajas de los Mock Objects en relación a los *stubs* que justifican su existencia y amplia difusión? Ambas técnicas son utilizadas comúnmente en las pruebas de software. Sin embargo, existe una diferencia fundamental entre ellas que reside en la forma de verificar que la prueba se ha realizado con éxito. Mientras que la técnica basada en *stubs* utiliza una verificación basada en el estado interno de los objetos colaboradores, la técnica basada en Mock Objects realiza una verificación basada en el comportamiento de dichos objetos, es decir, en la relación de comportamiento entre el objeto a probar y sus objetos colaboradores. Más adelante se explicarán en detalle los pormenores de este último procedimiento de verificación.

El gran inconveniente de una verificación basada en la comprobación del estado interno de un objeto es que han de crearse los mecanismos necesarios (traducido a las correspondientes líneas de código) en el interior del *stub* para mantener la información asociada al estado interno

del objeto al que representa. Esto último acaba convergiendo a reimplementar el objeto que se pretende sustituir. Más aún, puede darse el caso de que sea necesario crear métodos en el *stub*, no presentes en la interfaz original del objeto, que posibiliten la tarea de consulta al *stub* acerca de su estado interno. Como consecuencia de estos inconvenientes, el desarrollador puede llegar a realizar mayor trabajo que si, efectivamente, estuviera utilizando los objetos colaboradores originales en lugar de los *stubs*.

Un problema añadido de la técnica basada en *stubs*, es que éstos normalmente presentan una fuerte dependencia con la implementación de los objetos a los que sustituyen. Esto deriva en escasa mantenibilidad y refactorización<sup>4</sup> dificultosa.

## 7.3. Filosofía de funcionamiento de los Mock Objects

Una vez justificada la necesidad de utilizar Mock Objects en determinados escenarios, es interesante describir su modo de funcionamiento y, en particular, el aspecto que los ha hecho tan populares, es decir, la verificación basada en el comportamiento.

Los Mock Objects, una vez creados, han de ser inicializados con el comportamiento que se espera de ellos, es decir, con las expectativas. Las expectativas son básicamente las llamadas a métodos de la interfaz pública del Mock Object que se espera sean realizadas durante la prueba. Adicionalmente, se pueden establecer restricciones sobre los parámetros con los que se realizarán tales llamadas así como valores de retorno, etc. Una vez fijadas las expectativas, se procederá a realizar la prueba, en la cual determinados métodos del Mock Object serán invocados por el objeto a probar. Ahora queda determinar si tales llamadas a métodos se corresponden con las expectativas o no, lo que es equivalente a una verificación basada en el comportamiento. El resultado de esta verificación es lo que determinará el éxito o el fracaso de la prueba.

Una forma de entender esta filosofía de funcionamiento es mediante el paralelismo con una grabadora/reproductora de audio. Al igual que esta, los Mock Objects funcionan en modo grabación y en modo reproducción. Durante la fase de inicialización, es decir, cuando se fijan las expectativas, podría decirse que los Mock Objects están funcionando en modo grabación. Durante la prueba, los Mock Objects reciben una serie de llamadas. Si todo va bien, tales llamadas han de corresponderse con las expectativas grabadas anteriormente. En este último caso se puede decir que el Mock Object está funcionando en modo reproducción. El comportamiento previamente grabado está ocurriendo en tiempo real.

## 7.4. Procedimiento general de utilización de Mock Objects

A continuación se va a describir el procedimiento general de utilización de Mock Objects. Como se verá posteriormente, existen herramientas que asisten al desarrollador en la creación y utilización de Mock Objects facilitando muchas de estas tareas. Sin embargo, los conceptos subyacentes al procedimiento de utilización son comunes a todas ellas.

---

<sup>4</sup> El significado de este término y su influencia en el proceso de utilización de Mock Objects será explicado en detalle en el Apartado 7.4.

1. Crear las instancias de los Mock Objects: por cada objeto colaborador utilizado por la clase a probar, se deberá crear una instancia del Mock Object correspondiente, que lo sustituirá durante la prueba. Nótese que las clases a las que pertenecen dichas instancias deberán heredar de las clases de los objetos colaboradores a los que pretenden sustituir. Sólo de esta forma se evitarán los típicos errores de compilación causados por incompatibilidad de tipos, etc. Esta consideración es importante si los Mock Objects son creados desde cero. En caso contrario, es decir, utilizando herramientas como EasyMock o JMock, que serán comentadas posteriormente, estos detalles son transparentes al desarrollador.
2. Inicializar el comportamiento esperado de los Mock Objects, creación de las expectativas: existe un amplio rango de expectativas que el desarrollador puede especificar. Las más básicas son típicamente qué métodos van a ser llamados y opcionalmente con qué parámetros y con qué valor de retorno. Normalmente, más que los parámetros, lo que se fija en las expectativas son condiciones sobre los parámetros, es decir, qué condiciones han de satisfacer los parámetros con los que se llama a un determinado método. En el caso del valor de retorno, resulta de especial necesidad fijarlo cuando el método que se esté probando necesite que al invocar cierto método de una clase colaboradora, dicho método devuelva un valor en particular y no otro, para así continuar su ejecución normalmente. Dentro de las formas más complejas de fijar expectativas se podría incluir la especificación de secuencias de invocación de métodos, es decir, qué métodos han de ser llamados y en qué orden, o bien fijar el número exacto, mínimo o máximo de invocaciones, etc. Lógicamente cuanto más detalladas son las expectativas fijadas, más controlada está la prueba pero, por otra parte, es menos mantenible. Todo esto se verá más adelante con ejemplos.
3. Probar el objeto que utiliza los Mock Objects: este es el paso más importante, y en el que se llevará a cabo lo que es el principal objetivo para el cual se crearon los Mock Objects, conseguir probar aisladamente un objeto. En este punto las instancias de los Mock Objects han sido creadas y las expectativas han sido fijadas, la cuestión es: ¿cómo hacer que el objeto a probar utilice dichas instancias en lugar de los objetos colaboradores originales? La respuesta es, como casi siempre, depende. En este caso depende de la forma en la que los objetos colaboradores sean creados. Existen dos posibilidades:
  - Los objetos colaboradores son pasados al constructor de la clase a probar en forma de parámetros. En este caso, simplemente se pasarán como parámetros las instancias de los Mock Objects en lugar de los parámetros originales.
  - Los objetos colaboradores son creados en el interior de los objetos. Esta situación, que además se da con bastante frecuencia, sólo puede resolverse por medio de refactorización. Para los no familiarizados con el término, las técnicas de refactorización tienen como objetivo mejorar la legibilidad y la estructura del código fuente preservando su funcionalidad original. Durante una fase de desarrollo software convencional los procedimientos de refactorización y ampliación de la funcionalidad se suceden alternativamente. En este caso, el mecanismo de refactorización a aplicar tiene como objetivo encapsular todas las intanciaciones de objetos colaboradores en métodos accesibles desde fuera del objeto a probar<sup>5</sup>. De esta forma dichos métodos podrán ser sobrescritos por medio de herencia dentro del Mock Object que reemplaza al objeto colaborador.

---

<sup>5</sup> Más adelante, en el Apartado 7.5.1.2 Ejemplo de utilización de EasyMock, se puede ver la forma en la que se lleva a cabo este procedimiento de refactorización.



Un aspecto que, a pesar de que puede resultar obvio, conviene tener presente de cara a la planificación de las pruebas, es que la refactorización y, por tanto, la utilización de Mock Objects es solo posible en un contexto de pruebas de caja blanca en las que el código fuente está disponible.

4. Verificar la consistencia del comportamiento de los Mock Objects: tal y como se apuntaba anteriormente, este paso consiste en comprobar que al finalizar la prueba, los Mock Objects han satisfecho las expectativas. La única forma de realizar esta tarea es tomar la lista de expectativas e ir comprobando una a una que, en efecto, se han satisfecho. En la práctica, dicha tarea puede resultar realmente tediosa, sin embargo, afortunadamente existen herramientas que realizan esta comprobación simplemente invocando un método.

## 7.5. Herramientas para la puesta en práctica de la técnica de Mock Objects: EasyMock y JMock

Existen diferentes herramientas que asisten al desarrollador en la utilización de Mock Objects para la prueba aislada de clases. Antes de profundizar en las particularidades de estas herramientas, conviene tener presente que la utilización de Mock Objects no es una tecnología ni tampoco está ligada a ninguna herramienta en particular como algunos, por obvias razones de marketing, pretenden hacer creer. Mock Objects es una técnica alrededor de la cual existe una serie de herramientas que permiten ponerla en práctica con mayor facilidad y comodidad para el desarrollador. El objetivo principal de estas herramientas es, por un lado, eliminar las tareas repetitivas que implica el uso de Mock Objects y, por otro lado, proporcionar un mecanismo consistente de utilización de esta técnica de forma que el código generado sea fácilmente legible y comprensible. Véase a continuación una enumeración de las principales tareas que estas herramientas realizan.

- Creación de Mock Objects de forma transparente al desarrollador. Todo lo que el desarrollador necesita para crear un Mock Object es invocar un método que recibe como argumento la clase del objeto colaborador para el cual se desea crear un Mock Object. La herramienta de forma interna se encarga de definir la clase del Mock Object y de crear una instancia de dicha clase. Resulta interesante observar la capacidad que tiene la herramienta para crear un Mock Object de una clase colaboradora con tan solo conocer su nombre de clase (por ejemplo: `MiClaseColaboradora.class`) y sin conocer los detalles de implementación. Obviamente, el nombre de la clase permite a la herramienta utilizar la API de Reflection de Java para descubrir en tiempo de ejecución los métodos y propiedades de la clase colaboradora. Sin embargo, no le permite conocer la implementación de la clase colaboradora. Esto no constituye un problema puesto que la gran ventaja que presentan los Mock Objects en contraste con los *stubs* tradicionales es que los primeros son independientes de la implementación de la clase colaboradora y por tanto mucho más flexibles y mantenibles que los *stubs*.
- Mecanismo de definición de las expectativas. Estas herramientas presentan facilidades para expresar las expectativas asociadas a un Mock Object con tan solo escribir unas pocas líneas de código. Dichas expectativas son almacenadas internamente en el Mock Object de forma transparente al desarrollador y más adelante son utilizadas en el proceso de verificación del comportamiento.

- Mecanismo de almacenamiento de eventos de comportamiento. Una vez en tiempo de ejecución, el Mock Object creado por la herramienta es capaz de almacenar información acerca de qué métodos son invocados, con qué parámetros y valor de retorno, etc. Es lo que anteriormente se describió como “grabación de comportamiento”.
- Mecanismo automático de verificación. Con solo invocar un método es posible verificar que las expectativas definidas para el Mock Object se han satisfecho. Todo ello, una vez más, de forma transparente al desarrollador. Esto es así gracias a que los Mock Objects generados por la herramienta mantienen información sobre las expectativas definidas, así como información sobre los eventos de comportamiento que suceden durante la prueba. De este modo realizar dicha comparación es automático.

Una vez enumeradas las ventajas que el uso de estas herramientas proporciona al desarrollador, es el momento de presentar las dos herramientas más utilizadas en la actualidad: EasyMock y JMock. Aunque existen varias diferencias entre ellas, ambas se basan en el mecanismo de grabación de expectativas, ejecución y verificación comportamental anteriormente descrito.

### 7.5.1. EasyMock

EasyMock ha sido la primera herramienta de creación de Mock Objects que ha alcanzado una gran popularidad en el mundo de desarrollo Java. La primera versión fue desarrollada a mediados del año 2001 y desde entonces ha evolucionado dando lugar a nuevas versiones con funcionalidad mejorada. Este proyecto se encuentra actualmente alojado en [www.easymock.org](http://www.easymock.org). Se trata de un proyecto de código abierto que está disponible para descarga bajo la licencia MIT. La versión utilizada en este libro es la 2.2. Nótese que esta versión necesita tener instalado Java 2 en su versión 5.0 o superiores.

#### 7.5.1.1. Instalación

La instalación de esta herramienta es relativamente sencilla, se seguirán los siguientes pasos:

1. Descargar la versión 2.2 de EasyMock desde el sitio Web [www.easymock.org](http://www.easymock.org). Se trata de un archivo zip con el nombre `easymock2.2.zip`.
2. Descomprimir dicho archivo a una carpeta del ordenador. En su interior se encuentra el código fuente de la herramienta, la documentación, el *javadoc* así como la herramienta en sí, que es un archivo `.jar` con el nombre `easymock.jar`.
3. Añadir el archivo `easymock.jar` a la variable de entorno `CLASSPATH`<sup>6</sup>.

En este punto la instalación de la versión básica ha terminado. Sin embargo, esta versión de EasyMock sólo permite la creación de Mock Objects de interfaces y no de clases concretas. Nótese que, normalmente, cuando se presentan relaciones de asociación, crear interfaces suele ser una buena práctica diseño. De este modo una clase hace uso de sus clases colaboradoras por medio de interfaces en lugar de utilizar dichas clases directamente. Siempre que sea posible se ha de tener en cuenta este principio de diseño o bien crear las interfaces mediante un proceso de re-

---

<sup>6</sup> Esta tarea se puede realizar de diferentes formas, para obtener información en detalle véase Capítulo 3 Ant y Apéndice A Variables de entorno.

factorización. Sin embargo, existen situaciones, como por ejemplo cuando se hace uso de clases colaboradoras creadas por terceros, en las que no existen interfaces ni es posible crearlas. Para resolver estos casos existe una extensión de EasyMock que permite la creación de Mock Objects a partir de clases concretas.

El nombre de la extensión es EasyMock Class Extension y está disponible para descarga desde el sitio Web de EasyMock. La versión utilizada en este libro es la 2.2.1. A continuación se detalla el proceso de instalación:

1. Descargar la versión 2.2.1 de EasyMock Class Extensión desde el sitio Web [www.easymock.org](http://www.easymock.org). Se trata de un archivo .zip con el nombre `easymockclassex-tension2.2.1.zip`.
2. Descomprimir dicho archivo a una carpeta del ordenador. En su interior se encuentra el código fuente de la herramienta, la documentación asociada, el *javadoc* así como la herramienta en sí en forma de un archivo .jar con el nombre `easymockclassex-tension.jar`.
3. Añadir el archivo `easymockclassex-tension.jar` a la variable de entorno CLASSPATH.

Esta extensión ha sido construida alrededor de una librería externa llamada `cglib` (Librería de Generación de Código) en su versión 2.1. Dicha librería también necesita ser instalada. El proceso de instalación es el siguiente:

1. Descargar la versión 2.1 de la librería `cglib` desde el sitio Web <http://cglib.sourceforge.net/>. Para la realización de este libro se ha utilizado la última versión estable a fecha de escritura, que se corresponde con el nombre de archivo `cglib-nodep-2.1_3.jar`.
2. Añadir el archivo `cglib-nodep-2.1_3.jar` a la variable de entorno CLASSPATH.

Llegados a este punto se han realizado todas las tareas necesarias para comenzar a utilizar EasyMock.

### 7.5.1.2. Ejemplo de utilización de EasyMock

La mejor forma de dar a conocer las particularidades de esta herramienta es mediante un ejemplo práctico. El ejemplo va a estar centrado en la prueba aislamiento de la clase `LectorRegistros` perteneciente al sistema software presentado en el Apéndice B. Esta clase, básicamente, se encarga de leer información sobre el estado del tráfico desde un archivo en formato texto. Dicha información está organizada en forma de registros y tramos. Cada registro contiene información acerca del estado de una determinada carretera en un determinado momento en el tiempo. Asociado a cada registro existen una serie de tramos que amplían la información contenida en el registro, particularizada a cada uno de los tramos de la carretera en cuestión<sup>7</sup>.

En primer lugar merece la pena echar un vistazo al método `public Vector<Registro> leerRegistros()` de la clase `LectorRegistros`. Este método

---

<sup>7</sup> Para más detalles sobre el funcionamiento de esta clase y el contexto en el que se utiliza dentro del sistema software al que pertenece, véase Apéndice B.

hace uso de dos clases colaboradoras Registro y Tramo. Dichas clases permiten almacenar la información correspondiente a un registro o a un tramo respectivamente. Dicho de otra forma, existe una relación de asociación entre la clase LectorRegistros y Registro de cardinalidad 1 a n, es decir, la primera hace uso de la segunda. Asimismo existe una relación de asociación entre la clase Registro y Tramo de cardinalidad también 1 a n. Estas relaciones son fácilmente localizables en el código fuente del método que se lista a continuación:

sistema software/src/servidorEstadoTrafico/LectorRegistros(antes de refactorizar).java

```
public Vector<Registro> leerRegistros() throws IOException,
    RegistroMalFormadoException, TramoMalFormadoException {

    StringTokenizer strTokenizer;
    String strRegistro;
    String strToken;
    Registro registro;
    Tramo tramo;
    Vector<Registro> vRegistros;

    //Creacion del objeto para acceder al fichero
    File archivo = new File(this.m_strArchivo);

    //Creacion del objeto para la lectura desde el archivo
    FileInputStream entrada = new FileInputStream(archivo);

    //Creacion del objeto para leer lineas del dichero
    BufferedReader buffered = new BufferedReader(new InputStreamReader(
        entrada));

    //Creacion del vector para almacenar los registros
    vRegistros = new Vector<Registro>();

    do {

        //Lectura de un registro del archivo
        strRegistro = buffered.readLine();
        if (strRegistro == null)
            break;

        //Creacion de un objeto para extraer los elementos del String
        strTokenizer = new StringTokenizer(strRegistro,"|");
        //Se toman los elementos del String
        String strCarretera = strTokenizer.nextToken();
        String strFecha = strTokenizer.nextToken();
        String strHora = strTokenizer.nextToken();
        String strClima = strTokenizer.nextToken();
        String strObras = strTokenizer.nextToken();
        registro = crearRegistro(strCarretera,strFecha,strHora,strClima,
            strObras);
        registro.comprobarFormato();
    } while (true);
}
```

```

        //Se comprueba si hay tramos
        while (strTokenizer.hasMoreElements()) {
            //Se toman los elementos del String
            String strKMInicio = strTokenizer.nextToken();
            String strKMFin = strTokenizer.nextToken();
            String strCarriles = strTokenizer.nextToken();
            String strCarrilesCortados = strTokenizer.nextToken();
            String strEstado = strTokenizer.nextToken();
            String strAccidentes = strTokenizer.nextToken();

            //Creacion de un nuevo tramo
            tramo = crearTramo(strKMInicio, strKMFin, strCarriles,
                strCarrilesCortados, strEstado, strAccidentes);
            tramo.comprobarFormato();
            //Se guarda
            registro.anadirTramo(tramo);
        }

        //Se anade el registro al vector de registros
        vRegistros.add(registro);

    } while (true);

    return vRegistros;
}

```

El funcionamiento de este método es bastante sencillo, simplemente abre un archivo y carga la información contenida en él en objetos de la clase `Registro` y `Tramo`. Por cada línea del archivo se crea un objeto de la clase `Registro` y uno o varios de la clase `Tramo` dependiendo de su contenido. Debido a la existencia de estas dos clases colaboradoras, para probar el método `leerRegistros()` aisladamente va a ser necesario el uso de Mock Objects. Nótese que para tal efecto y debido a que las instancias de los objetos colaboradores se crean dentro del propio método, va a ser necesario llevar a cabo un proceso de refactorización tal y como se explicó en el Apartado 7.4 Procedimiento general de utilización de Mock Objects. Dicho proceso se realiza en dos pasos:

1. Creación de los métodos que instancian los objetos colaboradores. En este caso ya que se crean objetos colaboradores de dos clases, `Registro` y `Tramo`, va a ser necesario crear dos nuevos métodos en la clase `LectorRegistros`. Estos métodos son los siguientes:

```

protected Registro crearRegistro(String strCarretera, String strHora,
    String strFecha, String strClima, String strObras) {

    return new Registro(strCarretera, strFecha, strHora, strClima, strObras);
}

```

y

```

protected Tramo crearTramo(String strKMInicio, String strKMFin, String
    strCarriles, String strCarrilesCortados, String strEstado, String
    strAccidentes) {

```

```

        return new Tramo(strKMInicio, strKMFin, strCarriles,
            strCarrilesCortados, strEstado, strAccidentes);
    }

```

Nótese que estos métodos han sido definidos con el atributo de privacidad `protected` ya que, como se verá más adelante, es necesario permitir su sobrescritura en las sub-clases dentro del método de pruebas.

2. Localizar en el código del método aquellas sentencias en las que se instancien los objetos colaboradores y sustituirlas por llamadas a los métodos creados en el punto 1. En este caso las sentencias son:

```
registro = new Registro(strCarretera, strFecha, strHora, strClima, strObras);
```

que será reemplazada por:

```
registro = crearRegistro(strCarretera, strFecha, strHora, strClima, strObras);
```

y

```

tramo = new Tramo(strKMInicio, strKMFin, strCarriles,
    strCarrilesCortados, strEstado,
    strAccidentes);

```

que será reemplazada por:

```

tramo = crearTramo(strKMInicio, strKMFin, strCarriles,
    strCarrilesCortados, strEstado, strAccidentes);

```

Una vez finalizado el proceso de refactorización<sup>8</sup> el siguiente paso es crear el código de la prueba. A continuación se lista el código del método `testLeerRegistros()` perteneciente a la clase `LectorRegistrosTest`<sup>9</sup>. Dicho código será comentado en detalle posteriormente.

`pruebas sistema software/src/pruebasSistemasSoftware/junit381/LectorRegistrosTest.java`

```

import static org.easymock.classextension.EasyMock.*;

...

public class LectorRegistrosTest extends org.jmock.cglib.Mock
    ObjectTestCase {

    ...

```

<sup>8</sup> Nótese que según el procedimiento general de utilización de Mock Objects descrito en el Apartado 7.4, el procedimiento de refactorización ha de seguir a la instanciación de los Mock Objects y definición de las expectativas. Sin embargo, se ha optado por darle prioridad para presentar al lector el código definitivo del método a probar antes de comenzar la prueba.

<sup>9</sup> El código completo de la clase `LectorRegistrosTest` se encuentra en el archivo `src/junit42/LectorRegistrosTest.java`.

```
/*
 * Prueba del metodo leerRegistros utilizando la herramienta EasyMock
 */
public void testLeerRegistros() throws IOException {

    Vector<Registro> registros = null;
    final Vector<Registro> mocksRegistro = new Vector<Registro>();
    final Vector<Tramo> mocksTramo = new Vector<Tramo>();

    //Instanciacion del objeto a probar
    String strArchivoRegistros = "testData" + File.separator +
        "registros.txt";
    LectorRegistros lectorRegistros = new LectorRegistros(strArchiv
        Registros) {

        //Sobreescritura del metodo que instancia la clase colaboradora
        Registro protected Registro crearRegistro(String strCarretera,
            String strHora, String strFecha, String strClima, String strObras) {

            //Creacion del mock object para la clase colaboradora Registro
            Registro mockRegistro = createMock(Registro.class);

            //Definicion de la expectativas
            try {

                mockRegistro.comprobarFormato();
                mockRegistro.anadirTramo((Tramo)notNull());
                expectLastCall().times(4);
                replay(mockRegistro);

            } catch(RegistroMalFormadoException excepcion) {

                fail(excepcion.toString());
            }

            mocksRegistro.add(mockRegistro);

            return mockRegistro;
        }

        //Sobreescritura del metodo que instancia la clase colaboradora
        Tramo protected Tramo crearTramo(String strKMInicio, String
            strKMFin, String
                strCarriles, String strCarrilesCortados, String strEstado,
                String strAccidentes) {

            //Creacion del mock object para la clase colaboradora Registro
            Tramo mockTramo = createMock(Tramo.class);

            try {

                //Definicion de la expectativas
                mockTramo.comprobarFormato();
                replay(mockTramo);
```

```

        } catch (TramoMalFormadoException excepcion) {

            fail(excepcion.toString());
        }

        mocksTramo.add(mockTramo);

        return mockTramo;
    }
};

try {

    //Invocacion del metodo a probar
    registros = lectorRegistros.leerRegistros();

} catch (TramoMalFormadoException excepcion) {

    fail(excepcion.toString());

} catch (RegistroMalFormadoException excepcion) {

    fail(excepcion.toString());

}

//Verificacion de comportamiento de la clase Registro
for (Enumeration e = (Enumeration)mocksRegistro.elements();e.hasMoreElements();) {
    Registro mockRegistro = (Registro)e.nextElement();
    org.easymock.classextension.EasyMock.verify(mockRegistro);
}
//Verificacion de comportamiento de la clase Tramo
for (Enumeration e = (Enumeration)mocksTramo.elements();e.hasMoreElements();) {
    Tramo mockTramo = (Tramo)e.nextElement();
    org.easymock.classextension.EasyMock.verify(mockTramo);
}

assertEquals(registros.size(),1);

//Limpieza
mocksRegistro.removeAllElements();
mocksTramo.removeAllElements();
}

```

El primer punto a destacar es el `import` estático (`static`) de los miembros de la clase `org.easymock.classextension.EasyMock`. Dicha clase pertenece a la extensión `EasyMock Class Extension` y sus miembros van a ser utilizados para la creación de los `Mock Objects`. En caso de utilizarse únicamente la versión básica de `EasyMock`, las clases importadas serían `org.easymock.EasyMock.*`.

Puesto que se va a utilizar la extensión `EasyMock Class Extension` es necesario que la clase de prueba herede de la clase `org.jmock.cglib.MockObjectTestCase`.



Dentro del método `testLeerRegistros` lo primero es instanciar un objeto de la clase `LectorRegistros`, que es la clase a probar. Nótese que en el momento de la instanciación, y aprovechando la flexibilidad que permite la sintaxis del lenguaje Java, se han sobrescrito los métodos que crean instancias de las clases colaboradoras `Registro` y `Tramo`. Recuérdese que dichos métodos fueron creados anteriormente mediante refactorización justo con este fin. Estos métodos ahora van a crear instancias de Mock Objects en lugar de instancias de las clases colaboradoras correspondientes, con lo que se consigue la prueba en aislamiento de la clase `LectorRegistros`.

En la sobreescritura del método `crearRegistro` se realizan los siguientes pasos:

El método `createMock`<sup>10</sup> se utiliza para la creación de un Mock Object de la clase colaboradora `Registro`. Este objeto presenta la misma interfaz pública que un objeto cualquiera de la clase `Registro`, con las ventajas que se expusieron en el Apartado 7.1.

A la hora de fijar las expectativas, y echando un vistazo al código del método `leerRegistros`, cabe esperar que para cada objeto de la clase `Registro` creado, se invoquen los métodos `comprobarFormato` y `anadirTramo`. Concretamente, y mirando al archivo de registros del que este método lee<sup>11</sup>, se espera que para cada objeto `Registro` creado, el método `comprobarFormato` sea invocado una vez, y el método `anadirTramo` sea invocado exactamente 4 veces.

En EasyMock las expectativas se fijan de un modo completamente explícito<sup>12</sup>, que no es otro que llamar al método en cuestión tantas veces como se espere que sea llamado durante la ejecución. Esto puede resultar confuso inicialmente, pero nótese que con EasyMock, nada más crear un Mock Object, este se encuentra en modo grabación por omisión. Esto significa que invocar métodos sobre el Mock Object tiene como objetivo única y exclusivamente comunicar al Mock Object la secuencia exacta de métodos que se espera sean invocados sobre él cuando se encuentre en modo reproducción.

Al invocar un método para fijar una expectativa, si el método recibe parámetros, es posible fijar restricciones sobre dichos parámetros. En este caso la única restricción fijada es que el objeto de la clase `Tramo` no sea `null`, lo que, obviamente, es una restricción muy ligera. EasyMock, a través de la clase `EasyMock`, provee al desarrollador decenas de métodos para facilitar la creación de este tipo de restricciones. A continuación se listan algunas de las más utilizadas. Para obtener información más detallada se recomienda consultar la documentación de EasyMock ([www.easymock.org](http://www.easymock.org)).

- Restricciones generales: `anyObject()`, `anyBoolean()`, `anyInt()`, `anyChar()`, etc.
- Restricciones de comparación respecto a tipos básicos como `String`, `int`, `float`, `char`, `double`, etc: `eq(<TipoBasico>)`, `gr(<TipoBasico>)`, `lt(<TipoBasico>)`, etc.
- Expresiones de igualdad respecto a objetos: `same(T value)`.

---

<sup>10</sup> El método `createMock` es un método importado estáticamente que pertenece a la clase `org.easymock.classextension.EasyMock`.

<sup>11</sup> El archivo de registros se encuentra en la ruta `testData/registros.txt` y está formado por una única línea de texto con información acerca de un registro y cuatro tramos.

<sup>12</sup> Esta forma explícita de definir las expectativas presenta un inconveniente que puede observarse perfectamente en este ejemplo. Puesto que se invoca el método `comprobarFormato` y este puede lanzar una excepción que debe ser capturada, el compilador obliga a definir un bloque `try catch` que la capture. Sin embargo, esto es totalmente innecesario ya que el Mock Object no está realmente ejecutando el cuerpo del método `comprobarFormato`, y en la práctica dicha excepción jamás puede producirse.

La sentencia de código `expectLastCall().times(4)` es más una abreviatura que se utiliza para indicar que la última expectativa, es decir la llamada al método `anadirTramo`, se espera 4 veces. La expectativa definida sería equivalente a la obtenida al invocar 4 veces el método `anadirTramo` sobre el Mock Object.

El método `replay` simplemente cambia el estado del Mock Object de grabación a reproducción. Dicho método ha de invocarse siempre justo después de la definición de las expectativas, de modo que cuando el Mock Object sea utilizado dentro del método a probar, se encuentre en modo reproducción.

El Mock Object creado se almacena en un Vector de registros, que, como se verá más adelante, será utilizado en el proceso de verificación de comportamiento. Nótese que dicho vector ha sido declarado con el atributo `final`. Esto es así para que se pueda acceder desde fuera, dentro de los métodos sobrescritos.

El último paso es devolver el Mock Object creado como si realmente de un objeto de la clase `Registro` se tratara.

La sobreescritura del método `crearTramo` es casi completamente análoga a la de `crearRegistro`. Únicamente se ha de señalar que las expectativas para cada objeto `Tramo` instanciado son solamente la invocación del método `comprobarFormato`.

Este caso de prueba básicamente depende del contenido del archivo `registros.txt`. Ampliar el número de casos de prueba simplemente consistiría en crear nuevos archivos de registros con diferente contenido, así como adaptar las expectativas a tales contenidos.

Una consideración que ha de tenerse en cuenta a la hora de definir las expectativas es la mantenibilidad. Herramientas como EasyMock, permiten definir expectativas con un alto grado de detalle. Sin embargo, cuanto más detallada es una expectativa, más acoplada está al código de producción, por lo que cualquier mínimo cambio en este código tendrá repercusión en las pruebas creadas, que deberán ser modificadas. La definición de las expectativas siempre ha de ser un compromiso entre exhaustividad de la prueba y acoplamiento con el código de producción. A este respecto es muy importante tener en cuenta la buena práctica que supone definir interfaces para resolver las relaciones de asociación entre clases. La abstracción intrínseca de las interfaces permite evitar los típicos problemas derivados de la definición de expectativas acopladas a implementaciones de clases colaboradoras.

Nótese que en la sobreescritura de estos métodos los parámetros que el método sobrescrito recibe no son utilizados para nada, se pierden en la llamada. Esto es así porque el verdadero constructor de la clase colaboradora a la que el Mock Object substituye jamás es invocado y, por tanto, dichos parámetros no tienen utilidad.

El siguiente paso es la verificación del comportamiento de las clases colaboradoras a través de los Mock Objects que las han substituido durante la prueba. Simplemente se toma el vector de registros y el vector de tramos, que contienen los Mock Objects de los objetos `Registro` y `Tramo` respectivamente, y se invoca el método `verify` sobre ellos. Este método compara internamente las expectativas definidas para el Mock Object con las llamadas a métodos que ha recibido, en caso de detectarse alguna diferencia se produce un error. Este error permite al desarrollador conocer la anomalía y corregirla.

Por ejemplo, si las expectativas definidas para el Mock Object de la clase colaboradora `Registro` hubieran sido:

```
mockRegistro.comprobarFormato();
mockRegistro.anadirTramo((Tramo)notNull());
expectLastCall().times(3);
replay(mockRegistro);
```

se hubiera producido el siguiente error:

```
Unexpected method call anadirTramo(EasyMock for class servidorEstado
Trafico.Tramo): anadirTramo(notNull()): expected: 3, actual: 3 (+1)
```

que simplemente indica que se ha producido una llamada no esperada al método `anadirTramo` de la clase `Tramo`. Se esperaban tres llamadas y se han producido cuatro.

Finalmente, se realiza esta comprobación sobre el vector de registros obtenido.

## 7.5.2. JMock

JMock es la herramienta de creación de Mock Objects más utilizada actualmente en el mundo de desarrollo Java. Como se verá a continuación, a pesar de ser una herramienta muy similar a EasyMock, presenta una serie de ventajas sobre ella que la han hecho ganar muchos adeptos. Esto es, en gran medida, debido a la mayor madurez presente en la técnica de utilización de Mock Objects que había en el año 2004, fecha en la que esta herramienta fue desarrollada.

El proyecto JMock se encuentra actualmente alojado en el sitio Web [www.jmock.org](http://www.jmock.org). Se trata de un proyecto de código abierto. La versión utilizada en este libro es la 1.1.0.

### 7.5.2.1. Instalación

Para la instalación de esta herramienta, que es relativamente sencilla, se han de seguir los siguientes pasos:

1. Descargar la versión 1.1.0 de JMock desde el sitio Web [www.jmock.org](http://www.jmock.org). Se trata de un archivo `.jar` con el nombre `jmock-1.1.0.jar`.
2. Añadir el archivo `easymock.jar` a la variable de entorno `CLASSPATH`<sup>13</sup>.

En este punto la instalación básica ha terminado. Sin embargo, al igual que ocurre con EasyMock, la versión básica de JMock sólo permite la creación de Mock Objects de interfaces y no de clases concretas. Para resolver situaciones en las que es necesario la creación de Mock Objects para clases concretas, existe una extensión. Esta extensión existe en forma de archivo `.jar` y está disponible para descarga desde el sitio Web de JMock. La versión utilizada en este libro es la 1.1.0. A continuación se detalla el proceso de instalación:

1. Descargar la versión 1.1.0 de la extensión para prueba de clases concretas desde el sitio Web [www.jmock.org](http://www.jmock.org). Se trata de un archivo `.jar` con el nombre `easymock classexension2.2.1.zip`.

---

<sup>13</sup> Esta tarea se puede realizar de diferentes formas, para obtener información en detalle ver Capítulo 3 Ant y Apéndice I Variables de entorno.

2. Descomprimir dicho archivo a una carpeta del ordenador. En su interior se encuentra el código fuente de la herramienta, la documentación asociada, el *javadoc* así como la herramienta en sí en forma de un archivo `.jar` con el nombre `jmock-cglib-1.1.0.jar`.
3. Añadir dicho archivo `.jar jmock-cglib-1.1.0.jar` a la variable de entorno `CLASSPATH`.

Esta extensión, como se puede adivinar al ver su nombre, al igual que la correspondiente extensión de EasyMock, ha sido desarrollada alrededor de la librería `cglib` (Librería de Generación de Código) en su versión 2.1. Dicha librería también debe ser instalada. El proceso de instalación es el siguiente:

4. Descargar la versión 2.1 de la librería `cglib` desde el sitio Web <http://cglib.sourceforge.net/>. Para la realización de este libro se ha utilizado la última versión estable que se corresponde con el nombre de archivo `cglib-nodep-2.1_3.jar`.
5. Añadir el archivo `cglib-nodep-2.1_3.jar` a la variable de entorno `CLASSPATH`.

En este punto se han realizado todas las tareas necesarias para comenzar a utilizar la herramienta JMock.

### 7.5.2.2. Ejemplo de utilización de JMock

Al igual que como se hizo con EasyMock, la mejor forma de dar a conocer las particularidades de esta herramienta es mediante un ejemplo práctico. El ejemplo es el mismo que se utilizó para EasyMock, es decir, la prueba en aislamiento de la clase `LectorRegistros` perteneciente al sistema software presentado en el Apéndice B. De esta forma el lector podrá ver claramente las diferencias a la hora de utilizar ambas herramientas y elegir la que más le convenga de acuerdo a sus necesidades.

A continuación se lista el código del método `testLeerRegistro` perteneciente a la clase `LectorRegistros`<sup>14</sup>.

```
pruebas sistema software/src/pruebasSistemaSoftware/junit42/LectorRegistrosTest.java
```

```
//Imports necesarios para la herramienta JMock y para la extension CGLIB
import org.jmock.Mock;
import org.jmock.cglib.MockObjectTestCase;

...

public class LectorRegistrosTest extends org.jmock.cglib.MockObject
    TestCase {

    ...
```

<sup>14</sup> El código completo de la clase `LectorRegistrosTest` se encuentra en el archivo `src/junit381/LectorRegistrosTest.java`.

```
public void testLeerRegistros() throws IOException {

    Vector<Registro> registros = null;
    final Vector<Mock> mocksRegistro = new Vector<Mock>();
    final Vector<Mock> mocksTramo = new Vector<Mock>();

    //Instanciacion del objeto a probar
    String strArchivoRegistros = "testData" + File.separator +
    "registros.txt"; LectorRegistros lectorRegistros = new
    LectorRegistros(strArchivoRegistros) {

        //Sobreescritura del metodo que instancia la clase colaboradora
        Registro protected Registro crearRegistro(String strCarretera,
        String strHora, String
            strFecha, String strClima, String strObras) {

            //Creacion del mock object para la clase colaboradora Registro
            Mock mockRegistro = mock(Registro.class, new Class[]
            {String.class,String.class,String.class,String.class,
            String.class}, new Object[]
            {strCarretera,strFecha,strHora,strClima,strObras});
            //Definicion de las expectativas
            mockRegistro.expects(once())
                .method("comprobarFormato")
                .withNoArguments();
            mockRegistro.expects(atLeastOnce())
                .method("anadirTramo");
            mocksRegistro.add(mockRegistro);

            return (Registro)mockRegistro.proxy();
        }
    };

    //Sobreescritura del metodo que instancia la clase colaboradora
    Tramo protected Tramo crearTramo(String strKMInicio, String
    strKMFin, String strCarriles, String strCarrilesCortados, String
    strEstado, String
        strAccidentes) {

        //Creacion del mock object para la clase solaboradora Tramo
        Mock mockTramo = mock(Tramo.class,new Class[] {String.class,
        String.class,String.class,String.class, String.class,
        String.class}, new Object[] {strKMInicio,strKMFin,strCarriles,
        strCarrilesCortados,strEstado,strAccidentes});

        //Definicion de las expectativas
        mockTramo.expects(once())
            .method("comprobarFormato");
            .withNoArguments();
        mocksTramo.add(mockTramo);

        return (Tramo)mockTramo.proxy();
    }
};
```

```

try {

    //Invocacion del metodo a probar
    registros = lectorRegistros.leerRegistros();

} catch(TramoMalFormadoException excepcion) {

    fail(excepcion.toString());

} catch(RegistroMalFormadoException excepcion) {

    fail(excepcion.toString());

}

//Verificacion de comportamiento de la clase Registro
for (Enumeration e = (Enumeration)mocksRegistro.elements();e.hasMoreElements();) {
    Mock mockRegistro = (Mock)e.nextElement();
    mockRegistro.verify();
}
//Verificacion de comportamiento de la clase Tramo
for (Enumeration e = (Enumeration)mocksTramo.elements();e.hasMoreElements();) {
    Mock mockTramo = (Mock)e.nextElement();
    mockTramo.verify();
}

assertEquals(registros.size(),1);

//Limpieza
mocksRegistro.removeAllElements();
mocksTramo.removeAllElements();
}

```

El primer punto a destacar es el import de la clase `org.jmock.Mock`. Dicha clase se utiliza en JMock para encapsular los Mock Objects. Por otro lado, para utilizar la extensión basada en cglib y crear Mock Objects de clases concretas, es necesario realizar un import de la clase `org.jmock.cglib.MockObjectTestCase`.

Si se usa la extensión basada en cglib la clase de prueba deberá heredar de la clase `org.jmock.cglib.MockObjectTestCase`. En caso contrario heredará de la clase `org.jmock.MockObjectTestCase`.

De nuevo conviene prestar atención al cuerpo de los métodos sobrescritos `crearRegistro` y `crearTramo`, pues es ahí donde se realiza la instanciación de los Mock Objects y la definición de las expectativas.

La instanciación del Mock Object para la clase `Registro` se realiza a través del método `mock` de la clase `org.jmock.cglib.MockObjectTestCase`. En este caso es necesario pasar los argumentos del constructor al método que crea el Mock Object.

La definición de las expectativas se realiza a través de una serie de métodos que la clase `Mock` provee al desarrollador.

La sintaxis de definición de expectativas en JMock proporciona un mecanismo llamado *stubs* para la definición de expectativas sobre métodos que pueden ejecutarse un número indeterminado de veces o bien no ejecutarse. Es una forma flexible de fijar expectativas que resulta de gran utilidad cuando el desarrollador no conoce a ciencia cierta el número de veces que cierto método de la clase colaboradora va a ser llamado. No importa que cierto método se ejecute 0, 1 o 100 veces, lo que importa es que cuando se ejecute cumpla determinadas restricciones. Por ejemplo, en el caso de que no se considere relevante el número de veces que el método `anadirTramo` de la clase `Registro` es invocado (por ejemplo porque no se conoce ni importa el número de tramos contenido en el archivo de registros para cada registro) bastaría sustituir las expectativas definidas en el ejemplo por las siguientes líneas de código:

```
mockRegistro.expects(once())
               .method("comprobarFormato")
               .withNoArguments();
mockRegistro.stubs()
               .method("anadirTramo");
```

Una interesante característica que presenta JMock es la posibilidad de crear expectativas con excepciones asociadas. Es decir, se puede crear una expectativa que indique que cuando cierto método sea invocado, dicho método ha de lanzar una excepción<sup>15</sup>. Por ejemplo, existe la posibilidad de querer tratarse el caso de prueba en el que al instanciarse un objeto de la clase `Tramo` con determinada información incoherente en el archivo de registros, e invocarse el método de `comprobarFormato` sobre esa instancia, salte una excepción de la clase `TramoMalFormadoException`<sup>16</sup>. Si la clase `leerRegistros` ha sido implementada correctamente, debe dar respuesta a esta situación anómala. Sin embargo, como para la prueba del método `leerRegistros`, no se está utilizando un objeto de la clase `Tramo` real, sino un `Mock Object` que lo simula, el `Mock Object` nunca va a lanzar esa excepción a no ser que se le pida explícitamente. Esto es así porque obviamente el método `comprobarFormato` del `Mock Object` no tiene cuerpo, está vacío. Las siguientes líneas de código permiten hacer justo eso, pedirle al `Mock Object` que lance cierta excepción al ser invocado.

Las siguientes líneas de código permiten realizar el caso de prueba anteriormente descrito:

```
mockTramo.expects(once())
           .method("comprobarFormato");
           .withNoArguments();
           .will(throwException(new TramoMalFormadoException("Valor de
kilometro inicial
                               negativo.")));
```

Finalmente, merece la pena comentar la forma en que se realiza la verificación de comportamiento en JMock. Simplemente se invoca el método `verify` sobre cada uno de los `Mock Objects` que han intervenido en la prueba.

---

<sup>15</sup> Nótese que dicha excepción generada es una excepción artificial. Es decir, no responde a ninguna situación especial en la clase colaboradora, ya que de hecho esta jamás es instanciada. Se trata de una excepción que JMock lanza cuando el desarrollador se lo pide y de esta forma permite llevar a cabo cierto caso de prueba sin necesidad del objeto original.

<sup>16</sup> En realidad, este caso de prueba pertenece al caso general de prueba de excepciones esperadas, con la particularidad de que la excepción se produce en un objeto colaborador y no en el propio método de la prueba. Para obtener más información leer el Capítulo 2.

## 7.6. Comparativa entre EasyMock y JMock

A pesar de que durante un tiempo JMock ha sido una herramienta muy superior a EasyMock, actualmente, gracias a las nuevas versiones que han aparecido de EasyMock, la diferencia entre ambas herramientas no es tal. A continuación se va a realizar un repaso en forma de lista de las principales diferencias que aún persisten.

- Sintaxis de definición de las expectativas: EasyMock realiza una definición de expectativas basada en la invocación explícita de métodos mientras que JMock dispone de una sintaxis especial para definirlos. El mecanismo propuesto en JMock es mucho más intuitivo dado que las invocaciones explícitas a métodos se prestan a ser confundidas por el desarrollador con invocaciones al objeto real en lugar de al Mock Object. Y si al desarrollador le pueden confundir, al que seguro que confunden es al compilador de Java, ya que, como se ha señalado anteriormente, cuando esos métodos pueden lanzar excepciones que necesitan ser capturadas, el compilador obliga al desarrollador a rodear la definición de expectativas de un bloque `try catch`. Obviamente, desde un sentido práctico es totalmente innecesaria la captura de estas excepciones ya que jamás pueden ocurrir.
- Una interesante característica, aunque pueda resultar marginal, que posee únicamente JMock es el soporte para crear Mock Objects de objetos con métodos que realizan `callbacks`. Es decir, se pueden definir expectativas para la invocación de un método de forma que cierto `callback` sea invocado durante la prueba<sup>17</sup>. Esta posibilidad a pesar de no responder a una necesidad frecuente, permite una gran flexibilidad en la prueba de código “no convencional”.

Por estos motivos, aun siendo herramientas muy parecidas, se recomienda utilizar JMock como primera opción.

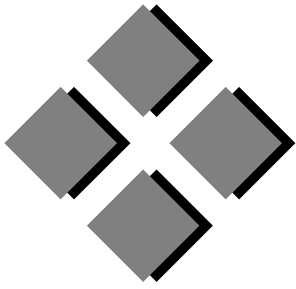
## 7.7. Bibliografía

- Fowler, M.: *Mocks Aren't Stubs*, 2 de enero de 2007, <http://www.martinflower.com/articles/mocksArentStubs.html>.
- Chaffee, A., y Pietri, W.: *Unit testing with mock objects*, 1 de noviembre de 2002, <http://www.ibm.com/developerworks/library/j-mocktest.html>.
- Rainsberger, J. B. (con contribuciones de Scott Stirling): *JUnit Recipes. Practical Methods for Programmer Testing*, Manning Publications, Greenwich, 2005.
- <http://www.easymock.org>.
- <http://www.jmock.org>
- Mackinnon, T.; Freemna, S. y Craig, P.: *Endo-Testing: Unit Testing with Mock Objects*, XP eXamined, Addison-Wesley, 2000.

---

<sup>17</sup> Para obtener más información acerca de esta característica de JMock véase el artículo “Cutom Stubs in JMock” que se encuentra en el sitio Web de JMock.





# Capítulo 8

## Mejora de la mantenibilidad mediante JTestCase

---

### SUMARIO

- |                                                         |                                                                                |
|---------------------------------------------------------|--------------------------------------------------------------------------------|
| <b>8.1.</b> Introducción                                | <b>8.4.</b> JTestCase como herramienta de documentación de los casos de prueba |
| <b>8.2.</b> Conceptos básicos                           |                                                                                |
| <b>8.3.</b> Definición de parámetros complejos con JICE | <b>8.5.</b> Bibliografía                                                       |

## 8.1. Introducción

Hasta ahora, y tal como ha sido descrito en el Capítulo 2, a la hora de escribir el código fuente correspondiente a los diferentes casos de prueba definidos para la prueba de un método, el procedimiento que se ha elegido es escribir cada caso de prueba a continuación del anterior. Siguiendo este procedimiento, a continuación se muestra un ejemplo del método de prueba para el cual se han definido tres casos de prueba. Se trata del método encargado de probar el método `obtenerLongitud`<sup>1</sup>, perteneciente a la clase `Tramo` del sistema software descrito en el Apéndice B de este libro.

---

<sup>1</sup> Este método se encarga de calcular la longitud correspondiente a un tramo de carretera haciendo uso de las propiedades `m_strKMInicio` y `m_strKMFin` de la clase `Tramo`.

`pruebas sistema software/src/pruebasSistemaSoftware/junit42/TramoTest.java`

```
@Test public void obtenerLongitud() {  
    // caso de prueba 1: normal  
    // (1) obtencion de los datos asociados al caso de prueba  
    String strKMInicio = "17";  
    String strKMFin = "23";  
    String strCarriles = "3";  
    String strCarrilesCortados = "1";  
    String strEstado = "Trafico Denso";  
    String strAccidentes = "Sin accidentes";  
    Tramo tramo = new Tramo(strKMInicio, strKMFin, strCarriles,  
        strCarrilesCortados, strEstado, strAccidentes);  
  
    // (2) invocacion del metodo a probar  
    int iLongitud = tramo.obtenerLongitud();  
  
    // (3) verificacion de las condiciones definidas  
    assertEquals(6, iLongitud);  
  
    // caso de prueba 2: formato incorrecto  
    // (1) obtencion de los datos asociados al caso de prueba  
    strKMInicio = "17";  
    strKMFin = "ab";  
    strCarriles = "3";  
    strCarrilesCortados = "1";  
    strEstado = "Trafico Denso";  
    strAccidentes = "Sin accidentes";  
    tramo = new Tramo(strKMInicio, strKMFin, strCarriles,  
        strCarrilesCortados, strEstado, strAccidentes);  
  
    // (2) invocacion del metodo a probar  
    iLongitud = tramo.obtenerLongitud();  
  
    // (3) verificacion de las condiciones definidas  
    assertEquals(-1, iLongitud);  
  
    // caso de prueba 3: longitud cero  
    // (1) obtencion de los datos asociados al caso de prueba  
    strKMInicio = "17";  
    strKMFin = "17";  
    strCarriles = "3";  
    strCarrilesCortados = "1";  
    strEstado = "Trafico Denso";  
    strAccidentes = "Sin accidentes";  
    tramo = new Tramo(strKMInicio, strKMFin, strCarriles,  
        strCarrilesCortados, strEstado, strAccidentes);  
  
    // (2) invocacion del metodo a probar  
    iLongitud = tramo.obtenerLongitud();  
  
    // (3) verificacion de las condiciones definidas  
    assertEquals(-1, iLongitud);  
}
```

Como puede observarse, a pesar de que este método cumple su objetivo, existe una clara redundancia dentro del mismo, ya que en realidad los tres casos de prueba son muy parecidos. En particular, puede observarse que la ejecución de los diferentes casos de prueba sólo se diferencia en la información asociada al caso de prueba y no en el esquema de ejecución, que permanece constante. Esto se puede considerar como general, ya que, salvo en contadas excepciones que más adelante se discutirán, este patrón se repite.

En general los siguientes elementos varían entre diferentes casos de prueba<sup>2</sup>:

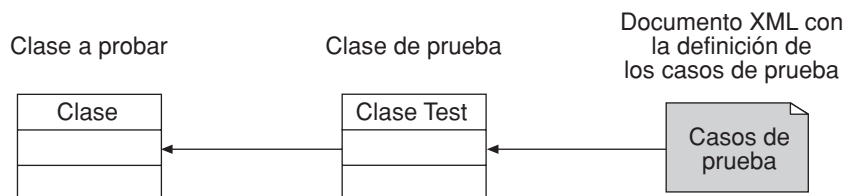
- Datos de entrada (en el ejemplo, la información con la que se instancia el objeto de la clase `Tramo`).
- Condiciones que se han de comprobar.

Sin embargo, los siguientes elementos permanecen invariantes a lo largo de todos ellos:

- Mecanismo de preparación de la prueba a partir de los datos de entrada del caso de prueba (en el ejemplo es simplemente la creación del objeto `Tramo`).
- Mecanismo de ejecución del método a probar (en el ejemplo, la invocación del método `obtenerLongitud` del objeto `Tramo`).
- Mecanismo de verificación de las condiciones definidas en el caso de prueba (en el ejemplo se hace a través del método `assertEquals`)

Una vez observado este patrón, cabe preguntarse si sería posible modificar el procedimiento de ejecución de los casos de prueba de forma que el desarrollador sólo tuviera que preocuparse de aquellos elementos que son diferentes para cada caso de prueba. La respuesta es sí, existe una herramienta llamada `JTestCase` que posibilita esta tarea y además proporciona otras características muy interesantes para el desarrollador. `JTestCase` es una herramienta de código libre que se encuentra disponible en el sitio Web <http://jtestcase.sourceforge.net/> bajo la licencia Common Public License versión 0.5. La principal novedad que aporta esta herramienta es una clara separación entre los datos definidos para los casos de prueba y el código de la prueba. Para ello, `JTestCase` proporciona una sintaxis de definición de casos de prueba en formato XML<sup>3</sup>. Posteriormente, los documentos XML creados pueden ser accedidos desde los métodos de prueba para así completarse el proceso. Como se puede observar, esta separación se corresponde perfectamente con la distinción que se ha hecho entre elementos variantes e invariantes.

En la Figura 8.1 se puede observar un diagrama de uso de esta herramienta. Como se describió en el Capítulo 2, para cada clase (en el caso de las pruebas unitarias) se define una clase



**Figura 8.1.** Diagrama de uso de `JTestCase`.

<sup>2</sup> En el Capítulo 1 se describen en detalle los diferentes elementos presentes en la definición de un caso de prueba.

<sup>3</sup> A lo largo de este capítulo se supone que el lector está familiarizado con la estructura de un documento XML. Información detallada acerca de este formato de marcado puede encontrarse en <http://www.w3.org/XML/>.

de pruebas, desde esta clase de prueba se accederá al documento XML con los casos de prueba definidos y se invocará a los métodos de la clase a probar con dicha información.

Las contribuciones de `JTestCase` en la ejecución de los diferentes casos de prueba se enumeran a continuación:

- Eliminación de redundancia a la hora de escribir el código de los métodos de prueba. En este sentido, evitar la redundancia permite reducir el tiempo de desarrollo, aumentar la legibilidad del código producido y disminuir la probabilidad global de defectos en el código de pruebas.
- Clara separación entre los datos asociados al caso de prueba y el código que ejecuta el caso de prueba. La definición de los casos de prueba para cada método está contenida en archivos XML. Esta separación mejora enormemente la mantenibilidad ya que añadir nuevos casos de prueba o modificar los casos de prueba existentes se puede hacer directamente sobre los archivos XML con lo que se evita la tarea de recompilar el código de pruebas. Es más, esta separación permite que el diseñador de los casos de prueba se mantenga al margen de la creación del código de prueba, que puede ser escrito por otra persona.

Por estas razones, `JTestCase` es una herramienta cuya utilización resulta muy útil en proyectos software de cierta envergadura y en el que el número de casos de prueba definidos puede ser inmanejable en otras circunstancias. Por otro lado, nótese que `JTestCase` no es una herramienta dependiente de JUnit y puede ser utilizada en combinación con otros frameworks de pruebas. A lo largo de este capítulo se expondrá en profundidad cómo sacar el máximo partido de esta herramienta y cómo resolver las particularidades que su uso supone en determinadas situaciones.

## 8.2. Conceptos básicos

La utilización de `JTestCase` básicamente se divide en dos fases: creación del documento XML con los casos de prueba y utilización de dicho documento dentro de los métodos de prueba.

### 8.2.1. Creación del documento XML con los casos de prueba

A la hora de utilizar `JTestCase`, el desarrollador ha de crear un documento XML para cada clase de prueba. Dicho documento contendrá la información asociada a los casos de prueba que hayan sido definidos para los métodos de esa clase. `JTestCase` proporciona una sintaxis<sup>4</sup> específica para realizar esta tarea, la mejor forma de empezar a conocerla es mediante un ejemplo. A continuación se lista un documento XML que contiene la información asociada a los casos de prueba del método `obtenerLongitud`. Estos casos de prueba son los correspondientes al ejemplo del Apartado 8.1.

---

<sup>4</sup> Dicha sintaxis está recogida formalmente en un documento XSD cuya URL es la siguiente: <http://jtestcase.sourceforge.net/dtd/jtestcase2.xsd>. `JTestCase` utiliza este documento para verificar que los documentos XML han sido escritos correctamente.

pruebas sistema software/testCases/TramoTest.xml

```

<?xml version ="1.0" encoding = "UTF-8"?>
<tests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://jtestcase.sourceforge.net/dtd/jtestcase2.xsd">
  <class name="Tramo">
    <method name="obtenerLongitud">
      <test-case name="Normal">
        <params>
          <param name="KMinicio" type="java.lang.String">17</param>
          <param name="KMFin" type="java.lang.String">23</param>
          <param name="Carriles" type="java.lang.String">3</param>
          <param name="CarrilesCortados" type="java.lang.String">1</param>
          <param name="Estado" type="java.lang.String">
            Trafico denso</param>
          <param name="Accidentes" type="java.lang.String">
            Sin accidentes</param>
        </params>
        <asserts>
          <assert name="result" type="int" action="EQUALS">6</assert>
        </asserts>
      </test-case>
      <test-case name="Formato incorrecto">
        <params>
          <param name="KMinicio" type="java.lang.String">17</param>
          <param name="KMFin" type="java.lang.String">ab</param>
          <param name="Carriles" type="java.lang.String">3</param>
          <param name="CarrilesCortados" type="java.lang.String">1</param>
          <param name="Estado" type="java.lang.String">Trafico denso</param>
          <param name="Accidentes" type="java.lang.String"> Sin accidentes</param>
        </params>
        <asserts>
          <assert name="result" type="int" action="EQUALS">-1</assert>
        </asserts>
      </test-case>
      <test-case name="Longitud cero">
        <params>
          <param name="KMinicio" type="java.lang.String">17</param>
          <param name="KMFin" type="java.lang.String">17</param>
          <param name="Carriles" type="java.lang.String">3</param>
          <param name="CarrilesCortados"
            type="java.lang.String">1</param>
          <param name="Estado" type="java.lang.String">
            Trafico denso</param>
          <param name="Accidentes" type="java.lang.String">
            Sin accidentes</param>
        </params>
        <asserts>
          <assert name="result" type="int" action="EQUALS">-1</assert>
        </asserts>
      </test-case>
    </method>
  </class>
</tests>

```

Como se puede observar, la sintaxis es muy intuitiva. Inicialmente, aparece la cabecera del documento XML con la versión y la codificación<sup>5</sup> y después una línea indicando la URL del documento XSD necesaria para la validación. La parte interesante viene después. Inicialmente se define una etiqueta `<class>` indicando que a continuación se va a incluir la información relativa a la clase cuyo nombre es pasado como parámetro de la etiqueta<sup>6</sup>. En el nivel inmediatamente inferior aparece la etiqueta `<method>` que se utiliza para indicar que a continuación se van a incluir los casos de prueba asociados a un método. Normalmente existirán tantas etiquetas `<method>` en este nivel como métodos tenga la clase. Para cada elemento `<method>` se definen tantos elementos `<test-case>` como casos de prueba se hayan definido para dicho método (tres en el ejemplo). Esta etiqueta tiene un parámetro llamado `name` que resulta de gran utilidad para incluir una descripción del caso de prueba o bien almacenar un identificador del mismo que pueda ser utilizado para referenciarlo dentro del Plan de Pruebas<sup>7</sup>. Es necesario que todas las etiquetas `<method>` tengan un atributo `name` con valor diferente, lo mismo ocurre con las etiquetas `<test-case>` correspondientes a un mismo método.

Como puede observarse en el ejemplo, para cada caso de prueba definido con la etiqueta `<test-case>`, existen dos bloques diferenciados:

1. **Parámetros de entrada al caso de prueba:** se trata de información necesaria para ejecutar la prueba, como por ejemplo datos de entrada al método de prueba, estado deseado del objeto de prueba, estado de entidades externas, etc. Cada uno de estos parámetros se representa con una etiqueta `<param>` mientras que todos ellos están englobados dentro de una etiqueta `<params>`. Cada etiqueta `<param>` consta de dos atributos. Por un lado el atributo `name` representa el nombre del parámetro, que se utiliza como forma de referenciarlo para obtener su valor dentro del método de prueba. El otro atributo es simplemente el tipo de dato al que pertenece el parámetro. Es importante tener en cuenta que JTestCase trabaja con tipos de datos basados en objetos y no tipos de datos primitivos. Esto significa que, por ejemplo, si un parámetro es de tipo `int`, el atributo `type` deberá recibir el valor `java.lang.Integer` en lugar de `int`. Los tipos de datos que pueden ser utilizados directamente son los siguientes: `Integer`, `Short`, `Long`, `Char`, `Byte`, `Float`, `Double`, `Boolean`, `String`, `Date`, `java.util.Hashtable`, `java.util.HashMap` y `java.util.Vector`, mientras que para tipos mas complejos o tipos definidos por el usuario, se deberá utilizar un mecanismo que será explicado en el Apartado 3 de este capítulo. Finalmente, el valor del elemento `<param>` representa el valor que dicho parámetro recibe para el caso de prueba en cuestión. En el ejemplo anterior, dentro de cada caso de prueba se han definido una serie de parámetros que están relacionados entre sí dado que en conjunto sirven para instanciar un objeto de tipo `Tramo`. Sin embargo, estos parámetros pueden ser completamente heterogéneos, y en general van a depender de las particularidades de cada caso de prueba.

---

<sup>5</sup> Nótese que en caso de que se quieran utilizar caracteres especiales se deberá sustituir la codificación UTF-8 por la que convenga según el caso.

<sup>6</sup> A pesar de que JTestCase permite definir múltiples elementos `<class>` dentro del mismo documento XML, se desaconseja su uso ya que por motivos de mantenibilidad es preferible no mezclar casos de prueba para varias clases diferentes dentro del mismo documento XML.

<sup>7</sup> En el Apartado 8.4 JTestCase como herramienta de documentación de los casos de prueba, se incluye información a este respecto.

2. Condiciones a verificar (asserts) en el caso de prueba: estas condiciones deben ser entendidas como valores o estados de variables que se deseen verificar una vez que el método de prueba ha sido ejecutado, como por ejemplo el código de retorno o el estado del objeto después de la ejecución de la prueba. Para cada caso de prueba se definirá una etiqueta `<asserts>` que contendrá las condiciones que se desea verificar. Cada una de ellas representada con una etiqueta `<assert>` que contiene toda la información necesaria para la ejecución de un método `assert`. Esta información consiste en el atributo `name`, que permite referenciar el `assert`, el atributo `type` que representa el tipo de dato de la variable sobre la cual se realizará el `assert` y el atributo `action` que representa la acción en la que está basada la verificación. Una lista completa de estas acciones se detalla a continuación, donde para cada acción aparece si tiene valor asociado o no y el método `assert` equivalente en JUnit, que es utilizado por JTestCase internamente.

action	parámetros	método assert equivalente en JUnit
ISNULL	no	<code>assertNull</code>
ISNOTNULL	no	<code>assertNotNull</code>
EQUALS	sí	<code>assertEquals</code>
NOTEQUALS	sí	
GT (>)	sí	
NOTGT (<=)	sí	
LT (<)	sí	
NOTLT (>=)	sí	
ISTYPEOF	sí	
ISTRUE	no	<code>assertTrue</code>

Obsérvese en el ejemplo que la acción utilizada es `EQUALS`, para la cual se ha definido el valor esperado dentro de la etiqueta `<assert>`. JTestCase proporciona un mecanismo extra para la verificación de condiciones que aumenta la flexibilidad proporcionada por la lista de acciones anterior. Este mecanismo permite definir datos para ser posteriormente utilizados por el desarrollador dentro del método de prueba para realizar verificación de condiciones *ad-hoc*. Un ejemplo de estos asserts a la medida podría ser la prueba de excepciones esperadas, comprobar que cierta variable presenta un valor dentro de un conjunto de valores dados o bien que su valor pertenece a un rango determinado. La sintaxis es la siguiente<sup>8</sup>:

```
<asserts>
  <assertparam name="variable" type="int"
    action="EQUALS">7</assert>
</asserts>
```

<sup>8</sup> Más adelante se incluirá un ejemplo de utilización de esta característica de JTestCase para la prueba de excepciones esperadas.

### 8.2.2. Acceso desde los métodos de prueba a los casos de prueba definidos en los documentos XML

Una vez que se ha definido un documento XML con los casos de prueba necesarios para la prueba unitaria de una clase, el siguiente paso consiste en escribir el código necesario en los métodos de prueba para acceder a esa información y así llevar a cabo la prueba. Para realizar esta tarea mediante JTestCase se deberá seguir el procedimiento descrito a continuación:

1. Carga del documento XML: se lleva a cabo mediante la creación de un objeto de la clase JTestCase utilizando el constructor de la clase, que recibe dos parámetros: por un lado el nombre del documento XML con los casos de prueba tal y como fue creado en disco y, por otro lado, el nombre de la clase definida en dicho documento tal y como se ha especificado en el atributo name de la etiqueta `<class>`<sup>9</sup>. Dicho constructor básicamente se encarga de cargar en memoria la información relativa a los casos de prueba necesarios para la prueba de la clase, por lo que deberá ser invocado una única vez y antes de la ejecución de los métodos de prueba. Por tanto, el lugar adecuado para realizar esta tarea va a ser dentro del constructor de la clase de prueba en el caso de que se esté utilizando JUnit en su versión 3.8.1 o bien en un método etiquetado como `@BeforeClass` en caso de utilizarse JUnit en su versión 4.x. De esta forma, el objeto JTestCase creado será almacenado en una variable de objeto que podrá ser posteriormente accedida dentro de cada uno de los métodos de prueba.
2. Obtención de los casos de prueba definidos para un método de prueba: una vez dentro de un método de prueba, es necesario obtener los casos de prueba definidos para dicho método, para ello se utiliza el método `getTestCasesInstancesInMethod` perteneciente a la clase JTestCase y que recibe como parámetro el nombre del método del cual se quieren obtener los casos de prueba tal y como aparece en el atributo name de la etiqueta `<param>` definida en el documento XML. Dicho método se invocará sobre la instancia creada en el punto anterior y devolverá un objeto perteneciente a la clase `Vector (java.util.Vector)`, cuyos elementos son objetos de la clase `TestCaseInstance`<sup>10</sup> que representa un caso de prueba.
3. Obtención de la información asociada a un caso de prueba: la información relativa a cada uno de los casos de prueba puede obtenerse iterando el `Vector` de objetos `TestCaseInstance` creado en el punto 2. Esta información básicamente se divide en tres categorías:
  - a. Parámetros del caso de prueba: para acceder a esta información, que no es otra que aquella contenida en el interior de la etiqueta `<params>` del documento XML, se utilizará el método `getTestCaseParams` perteneciente a la clase `TestCaseInstance`. Este método devuelve un objeto de la clase `HashMap`<sup>11</sup> con

---

<sup>9</sup> Nótese que JTestCase permite definir casos de prueba para más de una clase dentro del mismo documento, aunque dicha posibilidad ha sido desaconsejada.

<sup>10</sup> Como curiosidad indicar que esta clase `TestCaseInstance` posiblemente hubiera recibido el nombre más natural de `TestCase` en el caso de que no existiera ya una clase con dicho nombre en JUnit.

<sup>11</sup> La clase `HashMap` es básicamente una implementación de la clase `Map` de Java, es decir, un objeto que establece asociaciones clave-valor, que está construido sobre una tabla hash lo que permite tiempo de acceso constante a los elementos contenidos en él.



la información de los parámetros. Utilizando el método `get` de este objeto y pasándole como parámetro el valor del atributo `name` definido para ese parámetro en el documento XML, se podrá obtener el valor asociado al parámetro.

- b. Verificación de las condiciones: una vez obtenidos los parámetros y ejecutado el método a probar, el último paso consiste en la verificación de ciertas condiciones sobre los resultados obtenidos. Para ello se utiliza el método `assertTestVariable` perteneciente a la clase `TestCaseInstance`. Este método se encarga de verificar que cierta variable cumple cierta condición. Para ello recibe dos parámetros, el nombre de la condición, tal y como aparece en el parámetro `name` de la etiqueta `<assert>` definida en el documento XML, y el valor de la variable obtenido tras la ejecución del método de prueba (es decir, valor esperado y valor obtenido). El resultado es un valor de tipo `boolean` que indica si la condición se ha satisfecho o no. Típicamente este valor `boolean` se ha de utilizar como parámetro del método `assertTrue` para que JUnit reporte un fallo en caso de que su valor sea `false`. Este método es, además, el lugar propicio para indicar con un mensaje el tipo de fallo que se ha producido si este es el caso.
- c. Verificación de las condiciones *ad-hoc*: utilizando el objeto `TestCaseInstance` es posible obtener la información definida mediante etiquetas `<assertparam>` con el objetivo de realizar verificación de condiciones a medida. Esto se realiza mediante el método `getTestCaseAssertParams`, perteneciente a la clase `TestCaseInstance`. Este método devuelve un objeto de la clase `MultiKeyHashtable`<sup>12</sup> que contiene dicha información. Más adelante, en el Apartado 8.2.3 se mostrará un ejemplo de este tipo de verificación de condiciones.

A continuación se lista el código fuente del método `obtenerLongitud` encargado de la prueba del método del mismo nombre perteneciente a la clase `Tramo` del sistema software descrito en el Apéndice B de este libro. En la elaboración de este ejemplo se ha seguido paso por paso el procedimiento de acceso a la información de los casos de prueba descrito anteriormente. Nótese que este ejemplo es equivalente al primer ejemplo de este capítulo, salvo que esta vez se ha utilizado la herramienta JTestCase, por lo que el resultado es un método de prueba menos redundante y más mantenible. Nótese igualmente que el documento XML utilizado por este método es el utilizado en el ejemplo del Apartado 8.2.1.

Inicialmente se muestra el método `cargarCasosPrueba`, declarado con la etiqueta `@BeforeClass` y encargado de cargar la información de los casos de prueba en la variable de clase `m_jtestcase`. Posteriormente aparece el código del método que prueba el método `obtenerLongitud`.

`pruebas sistema software/src/pruebasSistemaSoftware/junit42/TramoTest.java`

```
@RunWith(TestClassRunner.class) public class TramoTest {  
  
    private static JTestCase m_jtestcase = null;
```

---

<sup>12</sup> La clase `MultiKeyHashtable` no pertenece a la API de Java sino a JTestCase, se trata de una tabla hash (utiliza internamente un objeto de la clase `java.util.Hashtable`) en la que cada valor está asociado a múltiples claves.

```

/**
 * Carga los casos de prueba al inicio de la prueba de la clase
 */
@BeforeClass static public void cargarCasosPrueba() {

    try {
        // Carga de los datos de los casos de prueba desde fichero
        String dataFile = "./testData/TramoTest.xml";
        m_jtestcase = new JTestCase(dataFile, "Tramo");

    } catch (Exception e) {
        e.printStackTrace();
    }
}

@Test public void obtenerLongitud() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases = m_jtestcase.getTestCasesInstancesInMethod("obtenerLongitud");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {

        // (1) obtencion de los datos asociados al caso de prueba
        TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);

        try {

            HashMap params = testCase.getTestCaseParams();
            String strKMInicio = (String)params.get("KMInicio");
            String strKMFin = (String)params.get("KMFin");
            String strCarriles = (String)params.get("Carriles");
            String strCarrilesCortados = (String)params.get("CarrilesCortados");
            String strEstado = (String)params.get("Estado");
            String strAccidentes = (String)params.get("Accidentes");
            Tramo tramo = new
Tramo(strKMInicio,strKMFin,strCarriles,strCarrilesCortados,
strEstado,strAccidentes);

            // (2) invocacion del metodo a probar
            Integer longitud = tramo.obtenerLongitud();

```

```

        // (3) verificación de las condiciones definidas
        boolean succeed =
testCase.assertTestVariable("result", longitud.intValue());
        assertTrue(testCase.getFailureReason(), succeed);

    } catch (JTestCaseException e) {

        // Ha ocurrido algun error durante el procesado de los datos
        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    }
}
}
}
}

```

Como puede observarse la ventaja de utilizar JTestCase es que, independientemente del número de casos de prueba que se definan para un determinado método, el código del método de prueba permanecerá inalterado, con la ganancia en mantenibilidad que ello conlleva.

### 8.2.3. Tratamiento de casos especiales

Existen situaciones en las que no todos los casos de prueba definidos para un método presentan las mismas características, por lo que se hace necesario un tratamiento especial que rompa con la uniformidad con que JTestCase suele tratar a todos los casos de prueba pertenecientes a un mismo método. Un ejemplo claro de caso de prueba especial es la prueba de una excepción esperada<sup>13</sup> para un método. Este caso de prueba típicamente representa un caso diferente del conjunto de casos que tratan de probar situaciones en las que no se espera que se produzca ninguna excepción. Es un caso distinto en cuanto a que los parámetros y condiciones con las que es expresado varían respecto del resto de casos de prueba.

Este tratamiento especial se lleva a cabo tanto en la definición del caso de prueba dentro del documento de prueba, como en su ejecución dentro del método de prueba. El modo de tratar estas situaciones se puede resumir en dos puntos:

- Definición del caso de prueba excepcional en el documento XML: en este caso la diferencia principal estriba en la utilización de condiciones *ad-hoc* mediante la definición de parámetros con la etiqueta `<assertparam>`. Estos parámetros son capaces de almacenar información adecuada para el caso de prueba a tratar de forma que sea posteriormente accesible de una forma flexible desde el método de prueba.
- Acceso a dicha información desde el método de prueba y realización de verificación de la condición. Esta verificación, por ser un caso especial, se suele realizar con los métodos `assert` que JUnit proporciona en lugar de con el método `assertTestVariable` de la clase `TestCaseInstance`.

A modo de ejemplo, se lista a continuación el código del método de prueba `comprobarFormato` perteneciente a la clase `TramoTest`, encargada de las pruebas unitarias sobre la cla-

<sup>13</sup> Para obtener información en detalle acerca de la prueba de excepciones esperadas ver el Apartado 2.8 conceptos avanzados en la prueba de clases Java.

se Tramo. Para la prueba de este método se han definido seis casos de prueba de los cuales cuatro de ellos realizan la prueba de la excepción esperada `TramoMalFormadoException`. La definición de estos casos de prueba se encuentra en el documento XML listado al final del ejemplo.

pruebas sistema software/src/pruebasSistemaSoftware/junit42/TramoTest.java

```
@Test public void comprobarFormato() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases = m_jtestcase.getTestCasesInstancesInMethod("comprobarFormato");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {

        // (1) obtencion de los datos asociados al caso de prueba
        TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);

        String strMensajeEsperado = null;

        try {

            HashMap params = testCase.getTestCaseParams();
            String strKMInicio = (String)params.get("KMInicio");
            String strKMFin = (String)params.get("KMFin");
            String strCarriles = (String)params.get("Carriles");
            String strCarrilesCortados = (String)params.get("CarrilesCortados");
            String strEstado = (String)params.get("Estado");
            String strAccidentes = (String)params.get("Accidentes");
            Tramo tramo = new
Tramo(strKMInicio,strKMFin,strCarriles,strCarrilesCortados,
strEstado,strAccidentes);

            // (2) obtencion de los parametros para la verificacion ad-hoc
            MultiKeyHashtable asserts = testCase.getTestCaseAssertParams();
            String[] strClaves = {"mensaje", "EQUALS"};
            strMensajeEsperado = ((String)asserts.get(strClaves));

            // (3) invocacion del metodo a probar
            tramo.comprobarFormato();
```

```

// (4) verificacion de las condiciones definidas
if (strMensajeEsperado != null) {
    fail("Una excepcion esperada no se ha producido");
}

} catch (JTestCaseException e) {

    // Ha ocurrido algun error durante el procesado de los datos
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (TramoMalFormadoException e) {

    if (strMensajeEsperado != null) {
        assertEquals("La excepcion contiene informacion inapropiada",
strMensajeEsperado,e.toString());
    } else {
        fail("Se ha producido una excepcion no esperada");
    }
}
}
}
}

```

La principal novedad en la prueba de este método respecto a lo visto anteriormente es la parte de obtención de los parámetros para la verificación *ad-hoc*. Esta tarea se realiza mediante el método `getTestCaseParams` perteneciente a la clase `TestCaseInstance`. Los parámetros están contenidos en una tabla hash que asocia cada parámetro a una secuencia de claves. Estas claves se corresponden con el valor de los atributos `name` y `action` definidos para la etiqueta `<assertparam>` en el documento XML, mientras que el parámetro es obviamente el valor de dicha etiqueta. La información contenida en el parámetro es el mensaje de texto asociado a la excepción esperada para el caso de prueba. Nótese que puesto que sólo interesa realizar la prueba de un tipo de excepción esperada, la clase de esa excepción, es decir, `TramoMalFormadoException`, no necesita incluirse en el documento XML como un parámetro más y puede estar directamente en el código del método de prueba.

El proceso de verificación de la condición depende del caso de prueba. Para los casos `Correcto1` y `Correcto2` la variable `strMensajeEsperado` toma el valor `null` por lo que en caso de producirse la excepción `TramoMalFormadoException`, se ejecutará la sentencia `fail` y el caso de prueba fallará. Para los casos de prueba `Incorrecto1`, `Incorrecto2`, `Incorrecto3` e `Incorrecto4` la variable `strMensajeEsperado` contiene el mensaje asociado a la excepción esperada por lo que el caso de prueba fallará si dicha excepción no se produce o bien si el mensaje asociado a la excepción no se corresponde con el valor de dicha variable.

La información correspondiente a los casos de prueba definidos para el método `comprobarFormato` se encuentra en el siguiente fragmento de documento XML.

pruebas sistema software/testCases/TramoTest.xml

```

<method name="comprobarFormato">
  <test-case name="Correcto1">
    <params>

```

```

    <param name="KMInicio" type="java.lang.String">17</param>
    <param name="KMFin" type="java.lang.String">23</param>
    <param name="Carriles" type="java.lang.String">3</param>
    <param name="CarrilesCortados" type="java.lang.String">1</param>
    <param name="Estado" type="java.lang.String">Trafico denso</param>
    <param name="Accidentes" type="java.lang.String">Sin accidentes</param>
  </params>
</test-case>
<test-case name="Correcto2">
  <params>
    <param name="KMInicio" type="java.lang.String">0</param>
    <param name="KMFin" type="java.lang.String">23</param>
    <param name="Carriles" type="java.lang.String">4</param>
    <param name="CarrilesCortados" type="java.lang.String">4</param>
    <param name="Estado" type="java.lang.String">Trafico fluido</param>
    <param name="Accidentes" type="java.lang.String">Camion volcado en el
arcen</param>
  </params>
</test-case>
<test-case name="Incorrecto1">
  <params>
    <param name="KMInicio" type="java.lang.String">-1</param>
    <param name="KMFin" type="java.lang.String">17</param>
    <param name="Carriles" type="java.lang.String">3</param>
    <param name="CarrilesCortados" type="java.lang.String">1</param>
    <param name="Estado" type="java.lang.String">Trafico denso</param>
    <param name="Accidentes" type="java.lang.String">Sin accidentes</param>
  </params>
  <asserts>
    <assertparam name="mensaje" type="java.lang.String"
action="EQUALS">Valor de kilometro inicial negativo.</assertparam>
  </asserts>
</test-case>
<test-case name="Incorrecto2">
  <params>
    <param name="KMInicio" type="java.lang.String">0</param>
    <param name="KMFin" type="java.lang.String">17</param>
    <param name="Carriles" type="java.lang.String">3</param>
    <param name="CarrilesCortados" type="java.lang.String">-1</param>
    <param name="Estado" type="java.lang.String">Trafico denso</param>
    <param name="Accidentes" type="java.lang.String">Sin accidentes</param>
  </params>
  <asserts>
    <assertparam name="mensaje" type="java.lang.String"
action="EQUALS">Valor de carriles cortados negativo.</assertparam>
  </asserts>
</test-case>
<test-case name="Incorrecto3">
  <params>
    <param name="KMInicio" type="java.lang.String">18</param>
    <param name="KMFin" type="java.lang.String">17</param>
    <param name="Carriles" type="java.lang.String">3</param>
    <param name="CarrilesCortados" type="java.lang.String">1</param>

```

```

        <param name="Estado" type="java.lang.String">Trafico denso</param>
        <param name="Accidentes" type="java.lang.String">Sin accidentes</param>
    </params>
    <asserts>
        <assertparam name="mensaje" type="java.lang.String"
            action="EQUALS">Valor de kilometro inicial superior a valor de
            kilometro final.</assertparam>
    </asserts>
</test-case>
<test-case name="Incorrecto4">
    <params>
        <param name="KMInicio" type="java.lang.String">0</param>
        <param name="KMFin" type="java.lang.String">17</param>
        <param name="Carriles" type="java.lang.String">3</param>
        <param name="CarrilesCortados" type="java.lang.String">4</param>
        <param name="Estado" type="java.lang.String">Trafico denso</param>
        <param name="Accidentes" type="java.lang.String">Sin accidentes</param>
    </params>
    <asserts>
        <assertparam name="mensaje" type="java.lang.String"
            action="EQUALS">Valor de carriles cortados superior a valor de
            carriles totales.</assertparam>
    </asserts>
</test-case>
</method>

```

Como puede observarse en el documento XML, no se han definido asserts para los casos de prueba `Correcto1` y `Correcto2`. En estos dos casos de prueba la verificación se basa en que la excepción `TramoMalFormadoException` no se produzca, lo que indica que el formato del objeto `Tramo` es el correcto.

## 8.3. Definición de parámetros complejos con JICE

En ocasiones, a la hora de definir un caso de prueba en un documento XML, es necesario definir parámetros cuya clase no está soportada por `JTestCase` directamente, es decir, no pertenece al siguiente grupo:

<code>java.lang.Integer</code>	<code>java.lang.Short</code>	<code>java.lang.Long</code>
<code>java.lang.Character</code>	<code>java.lang.Byte</code>	<code>java.lang.Float</code>
<code>java.lang.Double</code>	<code>java.lang.Boolean</code>	<code>java.lang.String</code>
<code>java.text.Date</code>	<code>java.util.Date</code>	<code>java.util.Hashtable</code>
<code>java.util.HashMap</code>	<code>java.util.Vector</code>	

Un caso frecuente es aquel en el que alguno de los parámetros definidos mediante la etiqueta `<param>` pertenece a una clase definida por el desarrollador y que por tanto no está soportada por `JTestCase`. En este tipo de situaciones existen dos alternativas:

1. Definición implícita del parámetro: consiste en expresar el parámetro (objeto) en función de los parámetros (objetos) que lo constituyen y que sí pertenecen a clases soportadas por JTestCase.
2. Definición explícita del parámetro: utilización de una extensión de JTestCase que usa la librería JICE<sup>14</sup> para definir objetos Java en documentos XML y posteriormente instanciarlos según esa información.

Mientras que la primera alternativa puede utilizarse en ocasiones en la que la instanciación de un objeto se puede realizar de forma simple, existen situaciones en las que el uso de JICE es absolutamente imprescindible. En los ejemplos anteriores puede comprobarse cómo objetos de la clase `Tramo` han sido definidos de forma implícita a través de los parámetros con los que se instancian objetos de dicha clase.

Sin embargo, existen muchos casos en los que determinados parámetros son objetos que a su vez están compuestos por otros objetos que también necesitan inicializarse. Se puede decir que, en ocasiones, es necesario crear un grafo de objetos como parámetro para determinada prueba. En estas situaciones, el uso de JICE como mecanismo de definición de estos grafos de objetos es absolutamente imprescindible.

JICE es una herramienta de código abierto que se encuentra disponible bajo la licencia GNU Lesser General Public License Versión 2.1. Esta herramienta está alojada en SourceForge en el sitio Web <http://jicengine.sourceforge.net/>. Sin embargo, JTestCase ha sido desarrollado con esta herramienta incorporada, de forma que no es necesario hacer ninguna descarga ni instalación adicional. JICE es una herramienta cuya utilización en el contexto de JTestCase es relativamente sencilla, por este motivo se recomienda su uso como regla general para la definición de objetos pertenecientes a clases no directamente soportadas por JTestCase. JICE, básicamente, presenta las siguientes características:

- Sintaxis de definición de objetos Java en formato XML. Esta sintaxis permite definir la forma en la que se va a instanciar un objeto indicando la clase del objeto y los parámetros con los que se va a invocar a su constructor.
- Sistema de instanciación de objetos Java basado en el procesamiento de archivos XML que contienen la definición de dichos objetos.

El modo en que JTestCase hace uso de JICE es fácil de imaginar: por un lado extiende la sintaxis de definición de parámetros añadiendo el atributo `use-jice` a la etiqueta `<param>`. De esta forma es posible definir objetos embebidos dentro de los documentos XML de definición de los casos de prueba. Por otro lado, el método `get` de la clase `TestCaseInstance`, cada vez que ha de instanciar un parámetro definido con el atributo `use-jice="yes"`, utiliza el sistema de instanciación de objetos provisto por JICE en lugar del sistema de instanciación por omisión. Esto hace que el modo en el que un objeto ha sido definido, ya sea mediante JTestCase básico o bien usando JICE, sea transparente para el desarrollador del método de prueba.

A continuación se muestra un ejemplo de utilización de JICE. Se trata de la prueba del método `construirXMLCarreterasAccidentes` perteneciente a la clase `GeneradorXML` del sistema software descrito en el Apéndice B. Este método recibe como parámetro de entrada un

---

<sup>14</sup> En realidad JICE es una herramienta cuya funcionalidad va mucho más allá de la utilizada en JTestCase, en particular esta herramienta es de gran utilidad para realizar inversión de control, o lo que es lo mismo IoC (Inversion of Control).



Vector (`java.util.vector`) de objetos Registro y crea a partir de él un documento XML (representado por un objeto que implementa la interfaz `org.w3c.dom.Document`) con la información relativa a tramos de carretera en los que se han detectado accidentes. Este es un caso típico en el que para definir los casos de prueba es necesario incluir información acerca del vector de objetos Registro a partir del cual se creará el documento XML. Existe un grafo de objetos consistente en un objeto Vector que contiene objetos Registro que a su vez contienen objetos Tramo, o lo que es lo mismo dos relaciones de agregación 1 -> N -> N. Como se puede comprobar en el siguiente documento XML, JICE proporciona una sintaxis ideal para describir este tipo de parámetros.

En primer lugar se muestra el documento XML con la definición del caso de prueba en el que se ha incluido un parámetro definido con JICE.

pruebas sistema software/testCases/GeneradorXMLTest.xml

```
<class name="GeneradorXML">
  <method name="construirXMLCarreterasAccidentes">
    <test-case name="Normal">
      <params>
        <param name="vRegistros" type="" use-jice="yes">
          <jice>
            <vector xmlns=http://www.jicengine.org/jic/2.0
              class="java.util.Vector">
              <entry action="parent.addElement(registro)">
                <registro class="servidorEstadoTrafico.Registro"
                  args="strCarretera,strHora,strFecha,strClima,strObras">
                  <strCarretera>M-40</strCarretera>
                  <strHora>12:23:45</strHora>
                  <strFecha>12/4/2007</strFecha>
                  <strClima>Nublado</strClima>
                  <strObras>No</strObras>
                </registro>
              </entry>
              <entry action="parent.addElement(registro)">
                <registro class="servidorEstadoTrafico.Registro"
                  args="strCarretera,strHora,strFecha,strClima,strObras">
                  <strCarretera>M-30</strCarretera>
                  <strHora>12:23:45</strHora>
                  <strFecha>12/4/2007</strFecha>
                  <strClima>Soleado</strClima>
                  <strObras>Si</strObras>
                <entry action="parent.anadirTramo(tramo)">
                  <tramo class="servidorEstadoTrafico.Tramo"
                    args="strKMInicio,strKMFin,strCarriles,
                      strCarrilesCortados,strEstado,strAccidentes">
                    <strKMInicio>1</strKMInicio>
                    <strKMFin>10</strKMFin>
                    <strCarriles>3</strCarriles>
                    <strCarrilesCortados>1</strCarrilesCortados>
                    <strEstado>retenciones</strEstado>
                    <strAccidentes>camion volcado en mitad de la
```

```

        calzada</strAccidentes>
    </tramo>
</entry>
<entry action="parent.anadirTramo(tramo)">
    <tramo class="servidorEstadoTrafico.Tramo"
        args="strKMInicio,strKMFin,strCarriles,
            strCarrilesCortados,strEstado,strAccidentes">
        <strKMInicio>11</strKMInicio>
        <strKMFin>20</strKMFin>
        <strCarriles>3</strCarriles>
        <strCarrilesCortados>1</strCarrilesCortados>
        <strEstado>retenciones</strEstado>
        <strAccidentes>turismo atravesado en el carril
            derecho</strAccidentes>
    </tramo>
</entry>
<entry action="parent.anadirTramo(tramo)">
    <tramo class="servidorEstadoTrafico.Tramo"
        args="strKMInicio,strKMFin,strCarriles,
            strCarrilesCortados,strEstado,strAccidentes">
        <strKMInicio>21</strKMInicio>
        <strKMFin>30</strKMFin>
        <strCarriles>3</strCarriles>
        <strCarrilesCortados>0</strCarrilesCortados>
        <strEstado>trafico fluido</strEstado>
        <strAccidentes>sin accidentes</strAccidentes>
    </tramo>
</entry>
</registro>
</entry>
</vector>
</jice>
</param>
</params>
<asserts>
    <assert name="result" type="boolean" action="ISTRUE"/>
</asserts>
</test-case>
</method>
</class>

```

Como puede observarse se ha definido un caso de prueba llamado Normal en el que existe un único parámetro de la clase Vector que contiene dos objetos Registro, el segundo de los cuales contiene tres objetos Tramo. Cada vez que se define un objeto, como se ha hecho en el ejemplo mediante las etiquetas <vector>, <registro> o <tramo>, se ha de indicar con el atributo class la clase Java a la que pertenece dicho objeto. Asimismo, es necesario que dicha clase se encuentre contenida en la variable de entorno CLASSPATH. El atributo args se utiliza para indicar los parámetros con los que se ha de invocar al constructor de la clase para crear el objeto. Dichos parámetros han de listarse a continuación, como es el caso en el ejemplo de los parámetros representados por las etiquetas <strCarretera>, <strHora>, <strFecha>, <strClima> y <strObras>. Por último cabe destacar la utilización de la etiqueta <entry>, que se utiliza para realizar operaciones que permiten la inicialización de los objetos definidos.

Estas operaciones consisten típicamente en invocar determinados métodos con determinados parámetros. La etiqueta posee un atributo llamado `action` en el cual se debe expresar el código Java de la operación a realizar. En el ejemplo se utiliza esta etiqueta para invocar los métodos `addElement` y `anadirTramo` de las clases `Vector` y `Registro` respectivamente. Los parámetros contenidos en el atributo `action` deben ser definidos a continuación en el interior de la etiqueta `<entry>`. Nótese que el valor de retorno correspondiente a la acción contenida en el atributo `action` se pierde, por lo que estas acciones no deben ser susceptibles de devolver ningún retorno de interés.

Una característica de la sintaxis de JICE es que no obliga al desarrollador a indicar el tipo al que pertenece un parámetro utilizado para la instanciación de un objeto cuando este parámetro pertenece a una clase como `String`, `int`, `double` o `boolean`. El mecanismo de instanciación de JICE es capaz de procesar el documento XML y averiguar la clase a la que pertenece el parámetro. Esta característica hace que describir los objetos sea una tarea menos tediosa y repetitiva. Otra característica interesante de la sintaxis que proporciona JICE, al contrario que otras herramientas como por ejemplo `JTestCase`, permite que el desarrollador defina etiquetas con cualquier nombre. Por ejemplo las etiquetas `<vector>`, `<registro>`, `<tramo>` o `<strCarretera>` son creadas por el desarrollador y JICE es capaz de conocer su significado y objetivo por su localización dentro del documento XML.

Seguidamente se lista el código fuente del método `construirXMLCarreterasAccidentes` que se encarga de ejecutar el caso de prueba anteriormente definido.

`pruebas sistema software/src/pruebasSistemaSoftware/junit42/GeneradorXMLTest.java`

```
@Test public void construirXMLCarreterasAccidentes() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases =

m_jtestcase.getTestCasesInstancesInMethod("construirXMLCarreterasAccidentes");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {

        // (1) obtencion de los datos asociados al caso de prueba
        TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);
```

```

try {

    // (2) invocacion del metodo a probar
    HashMap params = testCase.getTestCaseParams();
    Vector vRegistros = (Vector)params.get("vRegistros");
    System.out.println("size of the vector is: " + vRegistros.size());
    GeneradorXML generadorXML =
        new GeneradorXML(new Log("GeneradorXML", "./testData/prueba.log"));
    Document docObtenido =
        generadorXML.construirXMLCarreterasAccidentes(vRegistros);

    // (3) verificacion de las condiciones definidas
    DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
    Document docEsperado = docBuilder.parse("./testData/peticionAccidentes.xml");
    Diff diff = new Diff(docEsperado, docObtenido);
    boolean bResultado = testCase.assertTestVariable("result", diff.identical());
    assertTrue("Error en el formato del documento XML generado: " +
        diff, bResultado);

} catch (JTestCaseException e) {

    // Ha ocurrido algun error durante el procesamiento de los datos
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (javax.xml.parsers.ParserConfigurationException e) {

    e.printStackTrace();
    fail("Error al cargar el documento esperado");

} catch (org.xml.sax.SAXException e) {

    e.printStackTrace();
    fail("Error al parsear el documento esperado");

} catch (java.io.IOException e) {

    e.printStackTrace();
    fail("Error al cargar el documento esperado");

}

}
}

```

Como puede observarse la obtención del parámetro `vRegistros` mediante el método `get` de la clase `TestCaseInstance` es equivalente al caso en el que el parámetro no ha sido definido utilizando `JICE`. Por otro lado, y fuera de los objetivos de este capítulo, es interesante echar un vistazo a la forma en la que se ha realizado la verificación de la condición. Por un parte, se ha creado un documento XML utilizando el método `a probar`, es decir, `construirXMLCarreterasAccidentes` y por otra se ha cargado en memoria el documento que se espera se produzca al ejecutar dicho método. Finalmente, ambos documentos son comparados para

determinar si son idénticos y en dicho caso dar el caso de prueba como correctamente ejecutado. Dicha comparación se realiza mediante la clase `Diff`, perteneciente a la herramienta `XMLUnit`<sup>15</sup>. Con el objetivo de enriquecer el ejemplo, a continuación se muestra el documento XML que el método `construirXMLCarreterasAccidentes` genera a partir de la información definida en el caso de prueba.

```
<?xml version="1.0" encoding="UTF-8"?>
<ACCIDENTES>
  <Tramo>
    <Carretera>M-30</Carretera>
    <PuntoKilometricoInicio>1</PuntoKilometricoInicio>
    <PuntoKilometricoFin>10</PuntoKilometricoFin>
    <Carriles>3</Carriles><CarrilesCortados>1</CarrilesCortados>
    <Estado>retenciones</Estado>
    <Accidentes>camion volcado en mitad de la calzada</Accidentes>
  </Tramo>
  <Tramo>
    <Carretera>M-30</Carretera>
    <PuntoKilometricoInicio>11</PuntoKilometricoInicio>
    <PuntoKilometricoFin>20</PuntoKilometricoFin>
    <Carriles>3</Carriles>
    <CarrilesCortados>1</CarrilesCortados>
    <Estado>retenciones</Estado>
    <Accidentes>turismo atravesado en el carril
derecho</Accidentes></Tramo>
</ACCIDENTES>
```

A lo largo de este apartado se ha presentado JICE como una herramienta que permite la definición de objetos pertenecientes a clases no directamente soportadas por `JTestCase`. Es indudable su utilidad a la hora de complementar las posibilidades de `JTestCase`, sin embargo, la utilización de esta herramienta también presenta algunos inconvenientes, que son los siguientes:

- JICE no es capaz de manejar excepciones por lo que si el constructor de una clase está declarado de forma que pueda lanzar excepciones, no es una buena idea utilizar esta herramienta para crear instancias de esa clase.
- Dado que los documentos XML no se validan en tiempo de compilación y sin embargo dependen fuertemente de la implementación del código Java al que están asociados, en caso de que se produzca un cambio en dicho código y no se actualice el documento XML, se producirá un error que en lugar de ser en tiempo de compilación será en tiempo de ejecución.

Estas restricciones y, en particular, el hecho de que los constructores que pueden lanzar excepciones no sean adecuados para trabajar con JICE, deben ser tenidas en cuenta durante la fase de diseño de forma que se pueda simplificar el proceso de pruebas.

---

<sup>15</sup> Esta herramienta será explicada en profundidad en el Capítulo 10 Pruebas de documentos XML: `XMLUnit`.

## 8.4. JTestCase como herramienta de documentación de los casos de prueba

Al principio de la fase de pruebas los casos de prueba son definidos en unas plantillas especiales que pasan a formar parte del Plan de Pruebas <sup>16</sup>. Por otro lado, como se ha visto, JTestCase proporciona una sintaxis que permite realizar esta definición en documentos XML. La pregunta que surge es: ¿es necesario realizar este doble trabajo de definición de los casos de prueba? La respuesta a esta pregunta es no, los documentos XML deben servir como complemento de la información recogida en el plan de pruebas, pero no deben contener información redundante. Esto se debe a que mantener información redundante va en perjuicio de la mantenibilidad de la documentación. Es deseable que cada cambio que se realice se realice de forma única y no implique realizar modificaciones en paralelo en otras partes del software con lo que se evitan las inconsistencias.

La forma de documentación de un caso de prueba es la siguiente: cada caso de prueba ha de identificarse con un nombre único que se corresponderá con el valor del atributo `name` en la etiqueta `test-case` del documento XML. Acompañando al plan de pruebas debe existir un documento que contenga una descripción de cada caso de prueba, que será referenciado por ese identificador único, e indicando el archivo XML en el que se especifica la información de bajo nivel del caso de prueba. Por otro lado, se definirá el documento XML como se ha explicado en este apartado.

En la Figura 8.2 se muestra el esquema resultante de lo anterior:

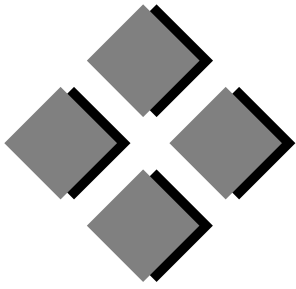


**Figura 8.2**

## 8.5. Bibliografía

- <http://jtestcase.sourceforge.net/>
- <http://jicengine.sourceforge.net/>

<sup>16</sup> Información detallada acerca de este proceso se puede encontrar en el Capítulo 1.



# Capítulo 9

## Prueba de aplicaciones que acceden a bases de datos: DBUnit

---

### SUMARIO

9.1. Introducción

9.2. Técnicas de prueba

9.3. Prueba del código perteneciente  
a la interfaz de acceso a la base de datos:  
DBUnit

9.4. Bibliografía

## 9.1. Introducción

Hoy en día es difícil encontrarse con aplicaciones reales de cierto volumen que no utilicen bases de datos como sistema de almacenamiento o intercambio de información. De hecho, debido al rápido desarrollo de las tecnologías de la información y al consiguiente aumento del volumen de información disponible, las bases de datos representan el núcleo de muchas aplicaciones informáticas. En estas aplicaciones todo gira en torno a la base de datos de forma que el código aplicación básicamente se divide en dos categorías:

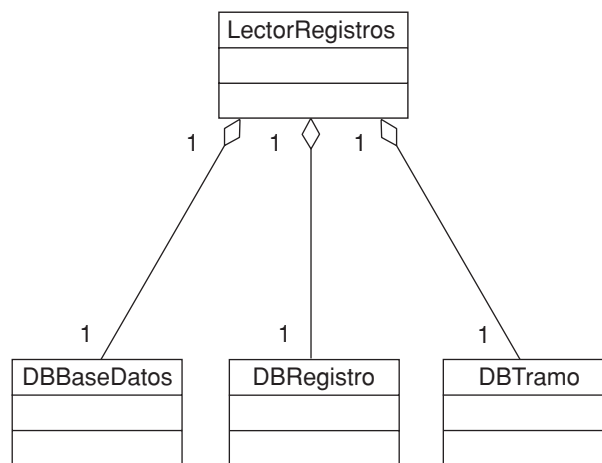
- Lógica de negocio: parte de la aplicación encargada del acceso a la información y posterior procesado. Incluye, por tanto, el código responsable del acceso a bases de datos.

- Lógica de presentación: parte de la aplicación cuyo objetivo es mostrar la información ya procesada al usuario de una forma adecuada. Engloba el código subyacente a la base de datos, rutinas de impresión y presentación, etc.

Durante la fase de diseño de software es fundamental que se establezca una clara separación entre las clases responsables de la lógica de negocio y la lógica de presentación. Desgraciadamente, esto no ocurre siempre, lo que deriva en un alto acoplamiento que dificulta enormemente el proceso de la prueba.

A la hora de diseñar una aplicación que accede a bases de datos, normalmente, se definen una serie de clases que actúan de interfaz entre la base de datos y el resto del sistema. Dichas clases se encargan de almacenar y cargar información desde la base de datos, escondiendo los detalles de acceso a la misma al resto del sistema. La información obtenida es posteriormente procesada y transformada por otras clases para realizar una tarea en particular. Por ejemplo, las clases de la interfaz con la base de datos del sistema software descrito en el Apéndice B<sup>1</sup> se encargan de obtener información sobre el estado del tráfico de vehículos desde una base de datos, posteriormente otras clases del sistema se encargan de convertir dicha información en documentos XML que son servidos bajo demanda mediante el protocolo http. Como se puede ver en la Figura 9.1, existe una relación de composición entre la clase `LectorRegistros` y sus colaboradoras `DBBaseDatos`, `DBRegistro` y `DBTramo`, encargadas del acceso a la base de datos. La multiplicidad de estas relaciones es 1 a 1 y se trata de composiciones, puesto que la clase `LectorRegistros` es responsable del ciclo de vida (creación y destrucción) de las otras tres clases.

Parece claro que el código que interacciona con la base de datos es una parte fundamental de todo sistema software. Dicho de otra forma, es la base de la pirámide, ya que si hay problemas en el acceso a la base de datos todo lo que ha sido construido por encima se derrumba. Por lo tanto, es fundamental realizar una serie de pruebas que garanticen la calidad de esta parte del software. A lo largo de este capítulo se va a tratar de dar una visión global del procedimiento a seguir y de cómo agilizarlo mediante el uso de una herramienta excepcional: DBUnit.



**Figura 9.1.** Relación de composición entre las clases `LectorRegistros` y sus colaboradoras `DBBaseDatos`, `DBRegistro` y `DBTramo`.

<sup>1</sup> Estas clases son `DBBaseDatos`, `DBRegistro` y `DBTramo`, para obtener más detalles consultar el Apéndice B.



## 9.2. Técnicas de prueba

A la hora de probar el código que accede a una base de datos básicamente existen dos técnicas bien diferenciadas. La primera de ellas está basada en Mock Objects, que son utilizados para emular el comportamiento de la base de datos. La otra técnica consiste en realizar las pruebas contra una base de datos real.

### 9.2.1. Utilización de Mock Objects

Esta técnica simplemente consiste en probar el código que accede a la base de datos pero sin la base de datos, es decir, cada uno de los objetos que componen la interfaz con la base de datos es substituido por un objeto Mock<sup>2</sup>. Esta técnica presenta una serie de ventajas inherentes a la ausencia de interacción con una base de datos real, que son las siguientes:

- El tiempo de ejecución de las pruebas es significativamente menor ya que no existe comunicación con el driver de acceso a la base de datos y, por consiguiente, no hay comunicación con la base de datos. En aplicaciones de uso intensivo de bases de datos esta ventaja resulta crucial, y representa uno de los mayores atractivos de los objetos Mock.
- No es necesario poner en marcha y configurar una base de datos de prueba, con el ahorro en tiempo que ello conlleva.
- Permite probar de forma sencilla situaciones excepcionales que se pueden dar al acceder a una base de datos, como son las excepciones de Java. La prueba de estas excepciones se realiza simplemente indicando al método correspondiente del Mock Object que ha de lanzar la excepción. Esta tarea se realiza durante el proceso de grabación de comportamiento del método.
- El código del sistema se está probando de forma independiente al driver de acceso a la base de datos por lo que presumiblemente se está verificando su correcto funcionamiento para cualquier driver.

Sin embargo también existen importantes inconvenientes, como los mencionados a continuación:

- El mecanismo de prueba utilizando objetos Mock, a pesar de estar asistido por herramientas como EasyMock o JMock<sup>3</sup> suele consumir bastante tiempo.
- En caso de que se produzcan cambios en la estructura de la base de datos el código de las pruebas seguirá funcionando sin notificar el problema. En realidad este inconveniente no es más que fruto de uno mucho mayor que se describe a continuación.
- La utilización de Mock Objects no es realmente una técnica de prueba del código que accede de acceso a la base de datos, ya que este código no es probado sino substituido por objetos Mock.

---

<sup>2</sup> Los objetos Mock son una técnica que permiten la realización de pruebas unitarias aislando la clase a probar de las clases colaboradoras, que son substituidas por objetos (objetos Mock) que no pueden fallar y cuyo comportamiento se determina al inicio de la prueba. Para más información se recomienda consultar el Capítulo 7.

<sup>3</sup> Estas herramientas son explicadas en profundidad y con ejemplos en el Capítulo 7.

Una vez visto esto, está claro que la técnica basada en Mock Objects aunque útil, por sí misma es claramente insuficiente ya que parte del sistema permanece sin ser probado. La recomendación general es utilizar esta técnica para la prueba de condiciones excepcionales que serían de otra forma muy difícil de reproducir al interaccionar con una base de datos real. Por otro lado se hace necesario el uso de una técnica complementaria enfocada a la prueba del código de acceso a la base de datos, que se describe en el siguiente apartado.

### 9.2.2. Utilización de una base de datos real

Esta técnica consiste en probar el código que accede a la base de datos utilizando una base de datos real. Dicha base de datos ha de ser equivalente a la que será la base de datos de producción. Puesto que trabajar directamente con la base de datos de producción es impensable, ya que sería muy fácil deteriorar su integridad, es necesario trabajar con bases de datos auxiliares. Típicamente existen las siguientes bases de datos:

- Base de datos de producción: se trata de la base de datos sobre la cual está trabajando la aplicación, en caso de que las pruebas formen parte de una actividad de mantenimiento, o bien la base de datos que será utilizada una vez que la aplicación esté terminada. En cualquier caso es una base de datos con información útil que no puede perderse.
- Base de datos de prueba: se trata de una base de datos idéntica en estructura a la base de datos real en cuanto al tipo de gestor de bases de datos, el driver con el que se accede al mismo y la estructura de tablas que contiene. Sin embargo, su contenido no tiene por qué ser equivalente al de la base de datos de producción<sup>4</sup>, y como se verá más adelante normalmente variara dependiendo del caso de prueba. La base de datos de prueba puede existir de forma local a cada desarrollador o bien existir de forma única y compartida para todos ellos. Este último caso suele ser el más frecuente en aplicaciones de gran envergadura.
- Base de datos de integración: esta base de datos se utiliza durante la fase de integración y sirve para verificar que los cambios que se han realizado sobre la base de datos de prueba han sido también aplicados a la base de datos de producción. Estos cambios pueden ser la creación de nuevos procedimientos almacenados o la modificación de la estructura de tablas. Por tanto, es necesario crear una replica en estructura y contenido de la base de datos de producción y ejecutar los tests sobre ella. Una vez superados los tests, el código de la aplicación ya puede pasarse a producción con suficiente confianza.

La ganancia de realismo de la prueba es evidente. El hecho de trabajar con una base de datos real significa que condiciones como la temporización en el acceso o particularidades del SGBD (Sistema Gestor de Bases de Datos) van a ser completamente reales y equivalentes a los que la aplicación se va a encontrar una vez en producción. Sin embargo, también presenta una serie de problemas añadidos que resultan fáciles de imaginar y son los siguientes:

- Es necesario crear, configurar y mantener varias bases de datos. En ocasiones esto puede ser un problema mayor por la necesidad de múltiples licencias.

---

<sup>4</sup> No obstante, en ocasiones, para realizar pruebas de rendimiento en volumen es necesario realizar una copia del contenido de la base de datos de producción en la base de datos de prueba.

- El tiempo de ejecución de los casos de prueba puede ser considerable.
- Las situaciones excepcionales en el acceso a la base de datos son difíciles de reproducir y en ocasiones simplemente no hay forma.

### 9.2.3. Procedimiento y recomendaciones

A la hora de probar una aplicación que trabaja con bases de datos, la principal cuestión que se ha de tener en cuenta es que al estar ejecutando el código contra una entidad externa (la propia base de datos) el proceso de prueba es mucho más lento. Por un lado, las clases de prueba de las clases que componen la interfaz con la base de datos van a tardar más tiempo en ejecutarse. Por otro lado, las clases de prueba de clases que utilicen como clases colaboradoras aquellas que componen la interfaz con la base de datos, también van a tardar mucho más tiempo en ejecutarse. Lógicamente, esto puede evitarse mediante una prueba unitaria propiamente dicha, es decir, en aislamiento utilizando Mock Objects. Está claro que las clases de acceso a la base de datos necesitan probarse mediante la estrategia descrita en el apartado anterior, la cuestión es: ¿cómo se ha de probar el resto del código de la aplicación? Típicamente se da uno de estos escenarios:

1. La aplicación a probar contiene un gran volumen de métodos que utilizan información de la base de datos. De esta forma un gran porcentaje de los casos de prueba van a acceder finalmente, a través de clases colaboradoras, a la base de datos<sup>5</sup>. En este caso la única forma de garantizar un tiempo razonable de ejecución de los casos de prueba es utilizando Mock Objects. Se creará un objeto Mock para cada una de las clases de la interfaz de acceso a la base de datos que actúen de clases colaboradoras de otras clases ajenas a dicha interfaz.
2. La aplicación a probar utiliza la base de datos, pero únicamente de forma puntual, es decir, para obtener información de configuración o bien almacenar periódicamente cierta información del estado, etc. En estos casos solo unos pocos casos de prueba accederán a la base de datos directa o indirectamente por lo que no se recomienda la utilización de Mock Objects.

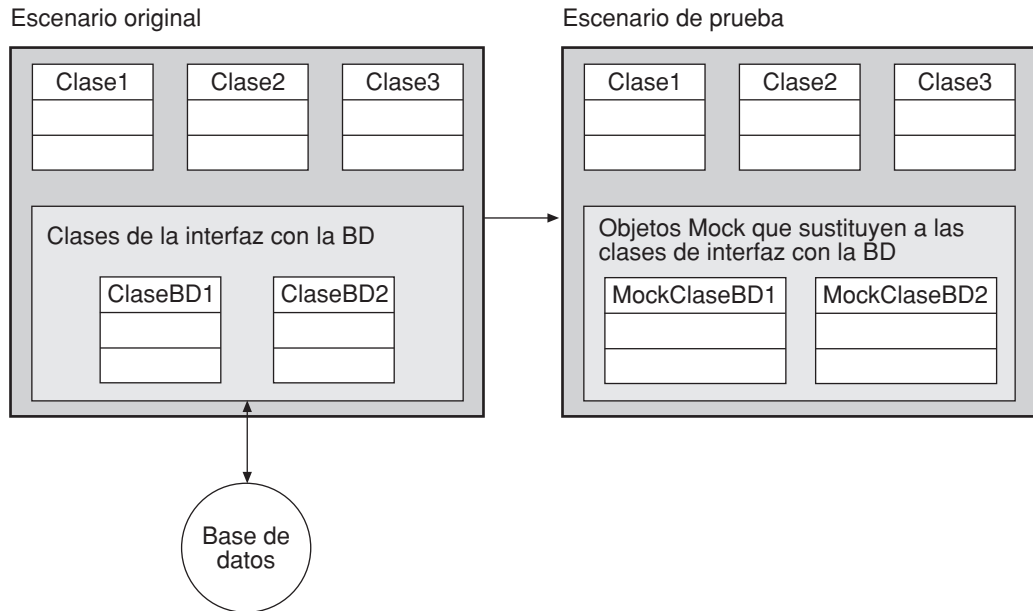
En cualquier caso y, como norma general, para aplicaciones de grandes dimensiones siempre se recomienda la utilización de Mock Objects ya que de otra forma, y en equipos de desarrollo con varios ingenieros de pruebas trabajando de forma concurrente, la dependencia de las pruebas con la base de datos puede ser muy compleja e incluso llegar a ser inmanejable. Esto es debido a que, normalmente, no es posible disponer de una base de datos local a cada desarrollador y la base de datos de prueba ha de estar compartida entre todos ellos.

Típicamente, y como se muestra en la Figura 9.2, en aplicaciones reales se suelen realizar dos pasos bien diferenciados. El primer paso consiste en realizar pruebas sobre el sistema (pruebas unitarias y de integración) utilizando Mock Objects para sustituir las clases originales de acceso a la base de datos (por tanto no existe interacción real con la base de datos y las pruebas son mas eficientes en tiempo y recursos). El siguiente paso, que también se puede hacer en paralelo, consiste en probar por separado las clases correspondientes a la interfaz con la base de datos en presencia de la base de datos de prueba. Como último paso se deberá hacer una prueba

---

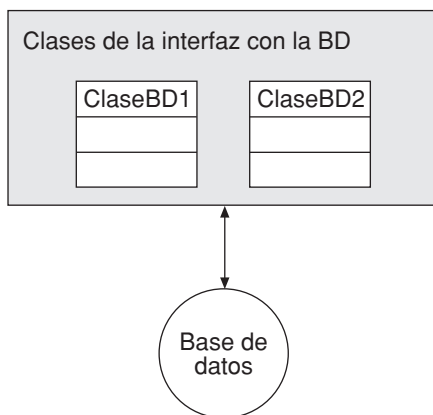
<sup>5</sup> Este es el caso de la aplicación descrita en el Apéndice B, dicha aplicación gira en torno a una base de datos ya que su cometido es transformar información obtenida desde la base de datos a documentos XML que pueden ser utilizados por un cliente Web.

1. Prueba de las clases del sistema sustentada por Mock Objects (sin BD real)



2. Prueba de las clases de interfaz con la BD en presencia de una BD real

Escenario original



**Figura 9.2.** Escenarios de prueba en aplicaciones con acceso a bases de datos.

de integración para comprobar que, efectivamente, las clases de la interfaz con la BD interactúa correctamente con el resto de clases del sistema. Asimismo para las pruebas de rendimiento normalmente no se utilizan Mock Objects ya que las mediciones de rendimiento han de ser lo mas realistas posibles, y el acceso a la base de datos es un factor crítico.

En los siguientes apartados de este capítulo se va a describir justamente cómo realizar el segundo paso, es decir, la prueba de las clases de acceso a la base de datos en presencia de la base de datos. Nótese que para llevar a cabo el primer paso basta conocer el funcionamiento de los Mock Objects, que fue explicado en el Capítulo 7 de este libro.

## 9.3. Prueba del código perteneciente a la interfaz de acceso a la base de datos: DBUnit

### 9.3.1. Introducción

La prueba de código que accede a bases de datos no es una tarea sencilla ya que al estar trabajando con una entidad externa, todo se complica bastante. Sin embargo, existe una extensión de JUnit que permite facilitar enormemente este trabajo, su nombre es DBUnit. Se trata de una herramienta de código abierto que está disponible para libre descarga desde el sitio Web de SourceForge en la dirección <http://dbunit.sourceforge.net/>. La primera versión de DBUnit data de principios del año 2002, desde entonces ha evolucionado enormemente hasta la última versión publicada a finales del año 2006. Dicha versión es la 2.2 y es la que ha sido utilizada para desarrollar los ejemplos de este libro. A diferencia de otras extensiones de JUnit, DBUnit es una herramienta bastante compleja, que ofrece diferentes posibilidades de utilización. A lo largo de este capítulo se van a presentar diferentes modos de utilización para resolver diferentes problemas.

La técnica general de prueba de código que accede a una base de datos se puede resumir en los siguientes pasos.

1. Se inicializa el contenido de la base de datos de acuerdo al caso de prueba.
2. Se ejecuta el método a probar. Que normalmente ejecutara una sentencia SQL sobre la base de datos para modificar su contenido.
3. Se compara el contenido de la base de datos con el contenido esperado de acuerdo al caso de prueba.

DBUnit proporciona una serie de mecanismos que facilitan enormemente dichas tareas al desarrollador y que son los siguientes:

- Proporciona mecanismos para almacenar y representar conjuntos de datos procedentes de la base de datos. Dichos conjuntos de datos reciben el nombre<sup>6</sup> de datasets y pueden representar por ejemplo el contenido de la base de datos en un momento dado o por ejemplo el conjunto de registros obtenidos tras realizar una consulta SQL. Normalmente, los datasets son almacenados en disco en formato XML aunque como se verá, pueden ser contruidos dinámicamente según la situación lo requiera.
- Es capaz de realizar consultas sobre la base de datos y plasmar los resultados en forma de datasets que pueden ser utilizados durante la prueba.
- Proporciona mecanismos de comparación de datasets. Esto es fundamental ya que habitualmente el objetivo de la prueba es verificar que el estado de la base de datos (representado en forma de dataset) se corresponde con el estado esperado de acuerdo al caso de prueba (también representado mediante un dataset).

---

<sup>6</sup> Nótese que “dataset” es completamente equivalente a “conjunto de datos”, sin embargo conviene tener siempre presente este término, ya que así aparece en la documentación que acompaña a DBUnit.

### 9.3.2. Creación de una clase de pruebas con DBUnit

A la hora de utilizar DBUnit se deberá crear, como hasta ahora, una clase de pruebas para cada una de las clases presentes en la interfaz de acceso a la base de datos.

#### 9.3.2.1. Definición de la clase de prueba

En este apartado se describe paso a paso el proceso de definición de una de clase de prueba con DBUnit. En subsiguientes apartados se describirá la forma de realizar la prueba del código en sí.

1. Definir la clase de prueba de modo que herede de la clase `DBTestCase`. Dicha clase hereda internamente de la clase `DatabaseTestCase` que a su vez hereda de la clase `TestCase` de JUnit, lo que es necesario cuando se está trabajando con versiones de JUnit anteriores a la 4.0.
2. La clase de prueba necesita tener acceso a la base de datos de pruebas para realizar su cometido. Para ello hace uso internamente de un objeto que implementa la interfaz `IDatabaseTester`. Dicho objeto, en su configuración por defecto obtiene la información necesaria para conectar con la base de datos a través de las propiedades del sistema, localizadas en la clase `System`. La forma más sencilla de proporcionar dichas propiedades es desde el constructor de la clase de prueba. Más adelante se verá con un ejemplo.
3. La clase `DBTestCase` es una clase abstracta (`abstract`) que obliga a redefinir al menos el método `getTestCase`. Dicho método debe devolver un objeto que implemente la interfaz `IDataset`<sup>7</sup>. Este objeto tiene que contener un dataset representando el contenido con el que la base de datos ha de ser inicializada al comienzo de la ejecución de la clase de pruebas. La forma más sencilla de construir este objeto es mediante la clase `FlatXmlDataSet` que permite construir un dataset a partir de un documento XML almacenado en disco. La clase `DatabaseTestCase` de la cual hereda `DBTestCase` tiene redefinidos los métodos `setUp` y `tearDown` de la clase `TestCase` de JUnit. De este modo, dentro del método `setUp` inicializa el contenido de la base de datos utilizando el dataset proporcionado por el método `getTestCase` que ha sido redefinido por el desarrollador. Este comportamiento resulta de gran utilidad, y puesto que las versiones 4.x de JUnit no utilizan la clase `TestCase` como clase base de la clase de pruebas, es necesario realizar la inicialización de otra manera, se verá más adelante. A continuación se lista el código del método `setUp` de la clase `DatabaseTestCase`, en el que se puede observar perfectamente el comportamiento descrito.

```
pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java
```

```
/**
 * Returns the database operation executed in test setup.
 */
protected DatabaseOperation getSetUpOperation() throws Exception
```

<sup>7</sup> La interfaz `IDataset` juega un papel fundamental en DBUnit, de hecho toda clase que represente un dataset ha de implementar dicha interfaz, como por ejemplo las clases `FlatXmlDataSet` o `QueryDataSet`.

```

{
    return DatabaseOperation.CLEAN_INSERT;
}

protected void setUp() throws Exception
{
    super.setUp();
    final IDatabaseTester databaseTester = getDatabaseTester();
    assertNotNull( "DatabaseTester is not set", databaseTester );
    databaseTester.setSetUpOperation( getSetUpOperation() );
    databaseTester.setDataSet( getDataSet() );
    databaseTester.onSetup();
}

```

Seguidamente, se presenta un ejemplo de clase de prueba utilizando DBUnit. Se trata de la clase de prueba DBRegistroTest encargada de probar la clase DBRegistro<sup>8</sup> perteneciente al sistema software descrito en el Apéndice B de este libro.

pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java

```

//DBUnit
import org.dbunit.*;
import org.dbunit.dataset.*;
import org.dbunit.dataset.xml.*;
import org.dbunit.database.IDatabaseConnection;
import org.dbunit.operation.DatabaseOperation;
import org.dbunit.dataset.filter.DefaultColumnFilter;
import org.dbunit.database.QueryDataSet;
import org.dbunit.dataset.datatype.*;
import org.dbunit.dataset.DataSetException;

@RunWith(TestClassRunner.class) public class DBRegistroTest extends
    DBTestCase {

    private static DBBaseDatos m_dbBaseDatos;
    private static DBRegistro m_dbRegistro;
    private static DBTramo m_dbTramo;

    /**
     * Constructor de la clase
     */
    public DBRegistroTest() {

        super();
        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_DRIVER_CLASS, "com.mysql.jdbc.Driver" );
    }

```

<sup>8</sup> La clase DBRegistro representa la interfaz entre el sistema software descrito en el Apéndice B y la tabla registro de la base de datos. Básicamente ejecuta sentencias SQL para obtener y almacenar información en esta tabla. Se recomienda asimismo revisar el Apéndice B para conocer la estructura de la base de datos.

```

        System.setProperty( PropertiesBasedJdbcDatabaseTester.
            DBUNIT_CONNECTION_URL, "jdbc:mysql://localhost:3306/trafico" );
        System.setProperty(
            PropertiesBasedJdbcDatabaseTester.DBUNIT_USERNAME, "root" );
        System.setProperty(
            PropertiesBasedJdbcDatabaseTester.DBUNIT_PASSWORD, "root" );
    }

    /**
     * Devuelve el conjunto de datos para la inicializacion
     */
    protected IDataset getDataSet() throws Exception
    {
        return null;//new FlatXmlDataSet(new
        FileInputStream("dataset.xml"));
    }

    ...
}

```

Inicialmente, se incluyen las sentencias `import` necesarias para `DBUnit`<sup>9</sup> y a continuación se declara la clase `DBRegistroTest`, heredando de `DBTestCase` y se define el constructor de la clase. En el constructor se ha utilizado el método `setProperty` de la clase `System` para pasarle a esta clase los datos de conexión con la base de datos, es decir:

- Nombre de la clase que representa al driver de acceso a la base de datos: `"com.mysql.jdbc.Driver"`. Se trata del driver JDBC<sup>10</sup> para acceder a un SGBD MySQL. Se ha elegido MySQL ya que es un sistema disponible de forma gratuita desde el sitio Web (<http://www.mysql.com/>) y que ha demostrado un extraordinario rendimiento en todo tipo de aplicaciones.
- URL de la conexión con la base de datos: `"jdbc:mysql://localhost:3306/trafico"`. Esta URL contiene información de la localización en la red del SGBD (dirección IP y puerto), así como el nombre de la base de datos, en este caso "trafico".
- Nombre de usuario y password de acceso a la base de datos: estos datos, por motivos de seguridad, normalmente deberán ser obtenidos desde un fichero encriptado en lugar de aparecer directamente en el código fuente.

Finalmente, se define el método `getDataSet` que en este caso simplemente devuelve `null` ya que se va a utilizar una versión 4.x de JUnit para ejecutar la clase de prueba. Idealmente sería deseable no tener que definir este método que realmente no sirve para nada. Sin embargo, debido a que `DBUnit` no está preparado para trabajar con las versiones 4.x de JUnit, es

---

<sup>9</sup> Como siempre, por motivos de espacio se han omitido sentencias `import` ajenas a la herramienta `DBUnit` así como información adicional presente en la clase. El archivo de código fuente al que pertenece esta clase contiene la información al completo y puede encontrarse...

<sup>10</sup> JDBC es el estándar de comunicación entre una aplicación Java y un sistema gestor de bases de datos (SGBD). Permite desarrollar aplicaciones Java que utilizan SQL para acceder a bases de datos de forma que las particularidades de comunicación con diferentes SGBD quedan ocultas al desarrollador.



necesario definirlo para que la clase de pruebas compile. En caso de que por ejemplo se fuera a utilizar la versión 3.8.1, el método tendría el siguiente aspecto.

```
/**
 * Devuelve el conjunto de datos para la inicializacion
 */
protected IDataSet getDataSet() throws Exception
{
    return new FlatXmlDataSet(new FileInputStream("dataset.xml"));
}
```

### 9.3.2.2. Definición de los métodos de prueba

En este apartado se va a describir la forma en que DBUnit ha de utilizarse para definir los métodos de prueba, que serán aquellos que ejecutan sentencias SQL sobre la base de datos. Una buena forma de clasificar estos métodos es en base al tipo de sentencias SQL que ejecutan sobre la base de datos. Se dividen en dos grandes grupos cuyo mecanismo de prueba es significativamente diferente por lo que se comentarán por separado.

#### 9.3.2.2.1. MÉTODOS QUE CAMBIAN EL ESTADO DE LA BASE DE DATOS

Como su propio nombre indica, estos métodos ejecutan sentencias SQL que alteran el contenido de la base de datos pero no obtienen información contenida en ella. Ejemplos claros son las sentencias SQL INSERT, DELETE, ALTER, etc. El objetivo de la prueba de estos métodos es, por tanto, verificar que el estado de la base de datos se ha modificado tal y como se espera. Es decir, si por ejemplo se ejecuta un comando INSERT que inserta tres registros en una determinada tabla, el objetivo de la prueba es verificar que dichos registros no estaban en la base de datos inicialmente y que una vez ejecutado el método, dichos registros aparecen en la base de datos y además el resto de la base de datos permanece inalterada. En el fondo lo que se debe probar es que las consultas SQL definidas son correctas.

El procedimiento general para la prueba de estos métodos es el siguiente:

1. Se inicializa la base de datos de forma que contenga información conocida.
2. Se ejecuta el método a probar conforme a los datos definidos para el caso de prueba.
3. Se obtiene la información contenida en la base de datos y la información que se espera contenga la base de datos tras la ejecución del método a probar. Esta última forma parte del caso de pruebas definido.
4. Se comparan ambas informaciones y si existe alguna diferencia el caso de prueba se da como no superado.

Véase con un ejemplo la forma en que DBUnit facilita llevar a cabo dicho procedimiento. Se trata del método de prueba `almacenarRegistro` perteneciente a la clase de prueba `DBRegistroTest` definida anteriormente. Este método se encarga de ejecutar los casos de prueba (solo uno en el ejemplo) del método homónimo perteneciente a la clase `DBRegistro`. A continuación, se lista el código de dicho método para que pueda verse la forma en la que interactúa con la base de datos:

pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java

```

/**
 * Almacena un objeto Registro en la tabla registro
 */
public void almacenarRegistro(Registro registro) throws
SQLException, RegistroMalFormadoException {

    // conversion de la fecha
    String strFecha = convertirFechaFormatoBD(registro.obtenerFecha());

    String strSentencia;

    strSentencia = "INSERT INTO
        registro(carretera,hora,fecha,clima,obras) VALUES (\\"
        + registro.obtenerCarretera() + "\",\\"
        + registro.obtenerHora() + "\",\\"
        + strFecha + "\",\\"
        + registro.obtenerClima() + "\",\\"
        + registro.obtenerObras() + "\")";

    // ejecucion de la sentencia
    this.m_statement.executeUpdate(strSentencia);

    // obtencion de la clave primaria
    ResultSet resultset = this.m_statement.getGeneratedKeys();
    int columns = resultset.getMetaData().getColumnCount();
    resultset.next();

    // obtencion de la clave primaria asociada al registro insertado
    (valor de la primera columna)
    int iId = resultset.getInt(1);
    registro.modificarId(iId);

    // almacenamiento de los tramos en la base de datos
    Vector vTramos = registro.obtenerTramos();
    for(Enumeration e = vTramos.elements() ; e.hasMoreElements() ; ) {
        Tramo tramo = (Tramo)e.nextElement();
        m_dbTramo.almacenarTramo(registro,tramo);
    }

    resultset.close();
}

```

El método simplemente ejecuta la sentencia SQL y obtiene la clave primaria del registro insertado. Nótese que dicho valor de la clave primaria no necesita ser verificado ya que se asume que el gestor de base de datos y el driver JDBC funcionan correctamente, por lo que dicha prueba entraría dentro de las denominadas “probar la plataforma”.

A continuación se lista el código del método de prueba almacenarRegistro:

pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java

```
/**
 * Metodo de prueba del metodo almacenarRegistro
 */
@Test public void almacenarRegistro() {

    // inicializacion del caso de prueba
    String strCarretera = "M-40";
    String strHora = "12:23:45";
    String strFecha = "1/3/2007";
    String strClima = "Nublado";
    String strObras = "No";
    Registro registro = new Registro(strCarretera, strHora, strFecha,
    strClima, strObras); Tramo tramo1 = new Tramo("1", "16", "3", "1",
    "Trafico Denso", "Sin accidentes");
    registro.anadirTramo(tramo1);
    Tramo tramo2 = new Tramo("17", "23", "3", "1", "Trafico Denso",
    "Sin accidentes");
    registro.anadirTramo(tramo2);

    try {

        // (1) inicializacion del contenido de la base de datos
        IDatabaseConnection dbConnection = this.getConnection();
        IDataset dsInicializacion = new FlatXmlDataSet(new
        File("../testData/dataSets/
        inicializacion/DBRegistro.almacenarRegistro.xml"));
        DatabaseOperation.TRUNCATE_TABLE.execute(dbConnection,
        ds Inicializacion);

        // (2) ejecucion del metodo a probar
        m_dbRegistro.almacenarRegistro(registro);

        // (3) obtencion del contenido de la tabla tramo tal y como esta
        en la BD IDataset dsObtenido = getConnection(). createDataSet();

        // (4) obtencion del contenido esperado
        IDataset dsEsperado = new FlatXmlDataSet(new
        File("../testData/dataSets/
        esperado/DBRegistro.almacenarRegistro.xml"));

        // (5) verificacion
        Assertion.assertEquals(dsEsperado, dsObtenido);

    } catch (SQLException e) {

        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");

    } catch (Exception e) {

        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");

    }
}
```

Inicialmente, se crea un objeto de la clase `Registro` que contiene dos objetos de la clase `Tramo`. La información contenida en ese objeto es la que se almacenará en la base de datos por medio del método a probar. Posteriormente, se inicializa el contenido de la base de datos de forma que se corresponda con el dataset contenido en el archivo `DBRegistro.almacenarRegistro.xml`. Para ello se utiliza la clase abstracta `DatabaseOperation`. Esta clase juega un papel muy importante en `DBUnit` ya que representa una operación realizada sobre la base de datos. Se utiliza típicamente justo antes y justo después de la ejecución del método a probar con el objetivo de determinar el contenido de la base de datos. Esta clase abstracta contiene variables de tipo `static` con objetos que permiten realizar diferentes operaciones. Todos ellos reciben dos parámetros, el primero es un objeto que implementa la interfaz `IDatabaseConnection`, necesario para acceder a la base de datos y el segundo es un dataset (objeto que implementa la interfaz `IDataSet`) que tendrá un significado u otro dependiendo de la operación. En la siguiente tabla se recogen las diferentes operaciones disponibles:

Objeto que realiza la operación	Operación
<code>DatabaseOperation.UPDATE</code> (pertenece a la clase <code>UpdateOperation</code> )	Actualiza la base de datos con el contenido del dataset.
<code>DatabaseOperation.INSERT</code> (pertenece a la clase <code>InsertOperation</code> )	Inserta en la base de datos la información contenida en el dataset.
<code>DatabaseOperation.DELETE</code> (pertenece a la clase <code>DeleteOperation</code> )	Elimina de la base de datos la información presente en el dataset.
<code>DatabaseOperation.DELETE_ALL</code> (pertenece a la clase <code>DeleteAllOperation</code> )	Elimina de la base de datos todas las filas (registros) de las tablas presentes en el dataset. A diferencia de <code>DELETE</code> , se utiliza para borrar el contenido de tablas enteras en lugar de una eliminación selectiva.
<code>DatabaseOperation.TRUNCATE_TABLE</code> (pertenece a la clase <code>TruncateTableOperation</code> )	Lleva a cabo la misma acción que <code>DELETE_ALL</code> , es decir, realiza una sentencia SQL de tipo <code>TRUNCATE</code> sobre las tablas presentes en el dataset. Nótese que no todos los SGBD soportan la sentencia SQL <code>TRUNCATE</code> . Sin embargo, cuando esta está disponible, es la forma más eficiente de eliminar todos los registros de una tabla.
<code>DatabaseOperation.REFRESH</code> (pertenece a la clase <code>RefreshOperation</code> )	Actualiza los registros contenidos en la base de datos con la información del dataset. Las claves primarias juegan, por tanto, un papel fundamental en esta operación.
<code>DatabaseOperation.CLEAN_INSERT</code> (pertenece a la clase <code>CompositeOperation</code> )	Realiza secuencialmente una operación <code>DELETE_ALL</code> seguida de una operación <code>INSERT</code> . La información contenida en el dataset se utiliza para ambas operaciones.
<code>DatabaseOperation.NONE</code> (pertenece a la clase <code>DummyOperation</code> )	No realiza ninguna operación.

Obsérvese la enorme potencia de la clase `DatabaseOperation`, con solo definir un dataset e invocar un método es posible hacer que la base de datos contenga exactamente los datos necesarios para la prueba. Asimismo cabe destacar las operaciones `REFRESH` y `CLEAN_INSERT` ya que representan dos estrategias de pruebas bien diferenciadas e igual de eficaces dependiendo del contexto. Mientras que `REFRESH` es utilizada habitualmente cuando se trabaja con una base de datos en la que existen otros datos que no interesa modificar (por ejemplo en una base de datos de prueba compartida entre varios desarrolladores), la operación `CLEAN_INSERT` es de naturaleza destructiva y se utiliza típicamente cuando la base de datos es local al desarrollador.

En el ejemplo la operación utilizada es `TRUNCATE_TABLE`, que elimina el contenido de las tablas presentes en el dataset definido en el documento `DBRegistro.almacenarRegistro.xml` situado en la carpeta de nombre inicialización. Dicho documento se lista a continuación:

```
pruebas sistema software/testData/dataSets/inicializacion/DBRegistro.almacenarRegistro.xml
```

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <registro/>
  <tramo/>
</dataset>
```

Las tablas cuyo contenido se va a eliminar son por tanto registro y tramo. Puesto que el método de prueba lo que hace es almacenar en la base de datos un registro con sus correspondientes tramos, tiene sentido que ambas tablas estén vacías inicialmente, de forma que sea más fácil comprobar posteriormente que la inserción se ha realizado con éxito.

Definir un dataset en forma de documento XML es muy sencillo, simplemente se ha de escribir la cabecera del documento XML indicando versión y codificación, y definir un elemento raíz `<dataset>` que englobará a todos los registros que se desee definir. Para cada registro se definirá un elemento que descienda de `<dataset>` cuyo nombre ha de ser el nombre de la tabla tal y como fue creada en la base de datos y cuyos atributos se corresponderán con los campos de dicha tabla. Finalmente los valores de dichos atributos representarán la información del registro. Este procedimiento se llevará a cabo típicamente de forma manual durante el proceso de definición de los casos de prueba.

Una vez inicializado el contenido de la base de datos, se ejecuta el método de prueba `almacenarRegistro` de la clase `DBRegistro`<sup>11</sup> que inserta el objeto `Registro` creado. El siguiente paso es obtener el contenido de la base de datos una vez realizada la inserción. Para ello se utiliza el método `getDataSet` de la interfaz `IDatabaseConnection`. Este método accede a la base de datos y devuelve un dataset reflejando su contenido<sup>12</sup> en forma de un objeto (`dsObtenido`) que implementa la interfaz `IDataSet`.

<sup>11</sup> El objeto `m_dbRegistro` de la clase `DBRegistro` se ha instanciado en el constructor de la clase de prueba de modo que se comparte entre los diferentes métodos de prueba. Esto debe considerarse una recomendación general.

<sup>12</sup> Nótese que este dataset contiene todos los registros de todas las tablas que existan en la base de datos en ese momento, por lo que si el número es elevado, puede tardar cierto tiempo en ejecutarse. Este tipo de consideraciones han de tenerse muy en cuenta a la hora de diseñar los casos de prueba.

En este caso el dataset que representa el contenido de la base de datos únicamente contiene registros pertenecientes a las tablas registro y tramo, ya que la base de datos solo contiene estas dos tablas. De esta forma, este dataset puede compararse directamente con el dataset esperado que, obviamente, sólo contiene registros de las tablas tramo y registro. Sin embargo, es muy común encontrarse con que el dataset esperado contiene registros de un subconjunto del conjunto total de tablas de la base de datos, por lo que no es posible compararlo directamente con un dataset que refleje el contenido total de la base de datos. En estos casos se ha de seguir el siguiente procedimiento:

1. Obtener un dataset que refleje el contenido de la base de datos (dataset obtenido) mediante el método `getDataSet` de la interfaz `IDatabaseConnection`.
2. Extraer del anterior dataset los datos de las tablas que se vean afectadas por la consulta. Para ello se ha de utilizar el método `getTable` de la interfaz `IDataSet` que devuelve un objeto implementando la interfaz `ITable`.
3. Obtener de forma análoga al punto anterior las tablas contenidas en el dataset esperado.
4. Comparar una por una las tablas procedentes del dataset esperado con las tablas procedentes del dataset obtenido. Para ello se ha de utilizar el método `assertEquals` de la clase `Assertion`, que recibe dos objetos que implementan la interfaz `ITable` y comprueba que son idénticos notificando un fallo a JUnit en caso contrario.

El siguiente paso en el ejemplo consiste en obtener el dataset con la información que se espera exista en la base de datos tras ejecutar la sentencia de inserción (dataset esperado). Para ello se instancia un objeto de la clase `FlatXmlDataSet` al que se le pasa la ruta en disco del documento XML que representa el contenido esperado. Dicho archivo se lista a continuación:

pruebas sistema software/testData/dataSets/esperado/DBRegistro.almacenarRegistro.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <registro id="1" carretera="M-40" hora="12:23:45" fecha="2007-03-01"
    clima="Nublado" obras="No"/>
  <tramo id="1" kmInicio="1" kmFin="16" carriles="3" carrilesCortados="1"
    estado="Trafico Denso" accidentes="Sin accidentes" registro="1"/>
  <tramo id="2" kmInicio="17" kmFin="23" carriles="3" carrilesCortados="1"
    estado="Trafico Denso" accidentes="Sin accidentes" registro="1"/>
</dataset>
```

Como puede observarse el dataset contiene un registro de la tabla registro y dos registros de la tabla tramo, todos ellos con idénticos datos a los que se utilizaron para construir el objeto de la clase `Registro` y los dos objetos de la clase `Tramo` al principio del método de prueba.

Finalmente se utiliza el método `assertEquals` de la clase `Assertion` de `DBUnit` (ojo no confundir con la clase `Assert` de JUnit), que se encarga de verificar que dos datasets son exactamente iguales. La clase `Assertion` utiliza el método `fail` así como otros métodos de la clase `Assert` para comunicar a JUnit que se ha producido un fallo en la ejecución de un caso de prueba. Esta clase además genera mensajes de error muy descriptivos que son pasados a JUnit y permiten al desarrollador hacerse una idea clara de lo que está ocurriendo. Por ejemplo, su-

póngase que debido a un defecto software en el método `almacenarRegistro`, solo el primer Tramo de cada Registro se almacena en la base de datos. En este caso al realizar la comparación con el dataset esperado visto anteriormente, se produciría un fallo que JUnit reportaría de la siguiente forma:

```
junit.framework.AssertionFailedError: row count (table=tramo)
  expected:<2> but was:<1>
at org.dbunit.Assertion.assertEquals(Assertion.java:128)
at org.dbunit.Assertion.assertEquals(Assertion.java:80)
at pruebasSistemaSoftware.junit42.DBRegistroTest.almacenarRegistro
  (Unknown Source)
```

Este mensaje indica que al contar las filas (`row count`) de la tabla `tramo` (`table=tramo`) se ha encontrado con una única fila cuando se esperaban dos (`expected:<2> but was:<1>`).

Obviamente, el método `assertEquals` de la clase `Assertion` no sólo realiza una comparación contando el número de registros, sino mirando en su interior. Si, por ejemplo, al realizar la inserción el campo `clima` del objeto `Registro` se hubiera intercambiado con el campo `obras`, el mensaje de error sería el siguiente:

```
junit.framework.AssertionFailedError: value (table=registro, row=0,
  col=clima): expected: but was:
at org.dbunit.Assertion.assertEquals(Assertion.java:147)
at org.dbunit.Assertion.assertEquals(Assertion.java:80)
at pruebasSistemaSoftware.junit42.DBRegistroTest.almacenarRegistro
  (Unknown Source)
```

Nótese que únicamente se reporta que el campo `clima` de la tabla `registro` en la fila numero 0, tiene el valor `No` cuando el valor esperado era `Nublado`. Sin embargo el otro fallo, es decir, el valor `Nublado` en el campo `obras` cuando se esperaba `No`, no es reportado. Esto es así porque JUnit siempre detiene la ejecución de un caso de prueba después de que se produzca el primer fallo.

Un detalle que merece especial atención son las claves primarias. Estas claves se utilizan para diferenciar de forma única los registros pertenecientes a una tabla en la base de datos. Para cada tabla se define normalmente un tipo de dato entero que el gestor de base de datos se encarga de incrementar automáticamente cada vez que se produce una nueva inserción. Se trata por tanto de un campo cuyo valor no se indica de forma explícita en la sentencia SQL de inserción, sino que es calculado en cada inserción por el SGBD en base a un valor que almacena internamente. Este valor es siempre el valor del último registro insertado incrementado en una unidad, de forma que si después de crearse la tabla 10 registros fueron insertados, en la siguiente inserción el SGBD asignará el valor 11 al campo de auto-incremento. Hasta aquí todo debería resultar completamente familiar para cualquiera que tenga unos conocimientos mínimos de bases de datos. El problema aparece cuando se realizan comparaciones de datasets que involucren claves primarias. En estos casos es imposible conocer el valor que va a tomar un campo de auto-incremento antes de realizar una inserción, por lo tanto no parece posible fijar un valor esperado para dicho campo a la hora de definir el dataset esperado asociado al caso de prueba. Por suerte DBUnit ha previsto esta situación, por ejemplo, la sentencia

```
DatabaseOperation.TRUNCATE_TABLE.execute(dbConnection,dsInicializacion);
```

del método `almacenarRegistro` no sólo elimina el contenido<sup>13</sup> de las tablas presentes en el dataset sino que además indica al SGBD que ha de resetear todos los valores de auto-incremento presentes en las tablas del dataset. Si, por ejemplo, dicha sentencia no se hubiera utilizado y antes de realizar la inserción en la tabla se tuviera que realizar una inserción (a pesar de estar la tabla vacía), el valor del campo `id` del nuevo registro insertado sería 2 por lo que al comparar el dataset obtenido con el dataset esperado visto anteriormente, se produciría un fallo que JUnit reportaría de la siguiente forma:

```
junit.framework.AssertionFailedError: value (table=registro, row=0,
  col=id): expected:<1> but was:<2>
at org.dbunit.Assertion.assertEquals(Assertion.java:147)
at org.dbunit.Assertion.assertEquals(Assertion.java:80)
at pruebasSistemaSoftware.junit42.DBRegistroTest.almacenarRegistro
(Unknown Source)
```

No obstante, al final del siguiente apartado se verán otras alternativas a la hora de trabajar con campos de tipo auto-incremento.

#### 9.3.2.2.2. MÉTODOS QUE OBTIENEN INFORMACIÓN CONTENIDA EN LA BASE DE DATOS MEDIANTE CONSULTAS

Estos métodos, como su propio nombre indica, son aquellos encargados de ejecutar sentencias SQL sobre la base de datos que no alteran su contenido sino que devuelven cierta información. Son las denominadas consultas SQL, que se realizan mediante la sentencia `SELECT`. La prueba de estos métodos, a diferencia de la prueba de los métodos del apartado anterior, resulta más compleja y tiene dos objetivos bien diferenciados:

- Verificar la corrección de las sentencias SQL definidas: consiste en determinar si la consulta SQL ha sido bien escrita, es decir, si los datos que obtiene dicha consulta son los que se espera se obtengan de la base de datos. Por supuesto, no se trata de probar si el lenguaje SQL funciona correctamente, sino si se ha hecho un correcto uso del mismo. El procedimiento para realizar dicha comprobación es el siguiente:
  1. Se inicializa la base de datos con un conjunto de datos que permita realizar la consulta que se quiere probar. Dichos datos se han de incluir en la definición del caso de prueba.
  2. Se ejecuta la consulta SQL embebida asociada al método a probar pero sin ejecutar el método a probar, sólo la consulta SQL.
  3. Se comparan los datos obtenidos en el punto anterior con los datos esperados según el caso de prueba. En caso de que sean diferentes se considera que el caso de prueba ha fallado.
- Asegurar que el código que guarda los datos obtenidos al ejecutar la consulta para que sean posteriormente procesados funciona correctamente: se trata de verificar que los datos son almacenados correctamente. Realmente no es una prueba de código que accede a la base de datos, pero este código también necesita ser probado y de una forma muy sencilla.

---

<sup>13</sup> Nótese que eliminar el contenido de una tabla no implica resetear los campos de auto-incremento.



Para ello es necesario tener en cuenta el contenido de la base de datos. El procedimiento general es el siguiente.

1. Se inicializa la base de datos con un conjunto de datos que permita realizar la consulta.
2. Se ejecuta el método a probar que ejecuta la consulta. Este método típicamente devuelve los datos obtenidos en la consulta.
3. Se comparan los datos obtenidos en el punto anterior con los datos esperados según el caso de prueba. En caso de que sean diferentes se considera que el caso de prueba ha fallado.

A continuación, se va a mostrar un ejemplo que demuestra como integrar ambos procedimientos en un método de prueba utilizando DBUnit. Se trata del método de prueba `obtenerTramosTrafico` cuya misión es probar el método del mismo nombre perteneciente a la clase `DBRegistro`. Este método ejecuta una consulta SQL sobre la base de datos para obtener los registros cuyos tramos tienen un estado del tráfico determinado. Finalmente, retorna un Vector de objetos `Registro` en el que cada objeto únicamente contiene los objetos `Tramo` cuyo estado del tráfico coincide con el requerido. El siguiente listado de código fuente corresponde a dicho método.

`pruebas sistema software/testData/dataSets/esperado/DBRegistro.almacenarRegistro.xml`

```
public Vector<Registro> obtenerRegistrosTramosTrafico(String
strEstado) throws SQLException {

    String strSentencia = obtenerConsultaTramosTrafico(strEstado);

    // ejecucion de la sentencia
    ResultSet resultset = this.m_statement.executeQuery(strSentencia);

    Vector<Registro> vRegistros = new Vector<Registro>();
    int iLastId = -1;
    Registro lastRegistro = null;
    while(resultset.next()) {

        int iId = resultset.getInt("id");
        String strKMInicio = resultset.getString("kmInicio");
        String strKMFin = resultset.getString("kmFin");
        String strCarriles = resultset.getString("carriles");
        String strCarrilesCortados =
resultset.getString("carrilesCortados");
        String strAccidentes = resultset.getString("accidentes");
        int iRegistro = resultset.getInt("tramo.registro");
        Tramo tramo = new Tramo(iId, strKMInicio, strKMFin, strCarriles,
            strCarrilesCortados, strEstado, strAccidentes);
        // comprobar si se trata de otro tramo del anterior registro o de
        uno diferente
        if (iRegistro == iLastId) {
            lastRegistro.anadirTramo(tramo);
```

```

    } else {
        String strCarretera = resultSet.getString("registro.carretera");
        String strHora = resultSet.getString("registro.hora");
        String strFecha = resultSet.getString("registro.fecha");
        String strClima = resultSet.getString("registro.clima");
        String strObras = resultSet.getString("registro.obras");
        Registro registro = new Registro(iRegistro, strCarretera, strHora,
            strFecha, strClima, strObras);
        registro.anadirTramo(tramo);
        vRegistros.add(registro);
        lastRegistro = registro;
        iLastId = iRegistro;
    }
}

resultSet.close();

return vRegistros;
}

```

El funcionamiento del método es bastante sencillo, simplemente obtiene la consulta SQL a ejecutar, en forma de un objeto de la clase `String`, mediante el método `obtenerConsultaTramosTrafico`, ejecuta la consulta y guarda la información obtenida tomándola del objeto `ResultSet` que la contiene. Lo primero que llama la atención es que la consulta SQL no haya sido definida dentro del propio método sino que se obtiene llamando a un método auxiliar. En realidad esta separación es fruto de un proceso de refactorización que tiene como objetivo facilitar el procedimiento de la prueba. El método `obtenerConsultaTramosTrafico`, como puede verse a continuación, construye la consulta SQL a ejecutar en función de un parámetro. De esta forma es posible probar que la sentencia SQL es correcta sin necesidad de invocar al método que la utiliza para obtener los datos.

sistema software/src/servidorEstadoTrafico/DBRegistro.java

```

public String obtenerConsultaTramosTrafico(String strEstado) {

    String strSentencia;

    strSentencia = "SELECT registro.carretera,
        registro.hora,
        registro.fecha,
        registro.clima,
        registro.obras,
        tramo.id, tramo.kmInicio,
        tramo.kmFin, tramo.carriles,
        tramo.carrilesCortados,
        tramo.estado, tramo.accidentes,
        tramo.registro
    FROM registro, tramo

```

```

        WHERE tramo.estado = \"\" + strEstado + "\"" and registro.id =
            tramo.registro
        ORDER BY tramo.registro";
    return strSentencia;
}

```

A continuación se lista el código fuente del método de prueba:

pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java

```

@Test public void obtenerTramosTrafico() {
    try {
        // (1) inicializacion del contenido de la base de datos
        IDatabaseConnection dbConnection = this.getConnection();
        IDataset dsInicializacion = new FlatXmlDataSet(new
            File("./testData/dataSets/
                inicializacion/DBRegistro.obtenerTramosTrafico.xml"));
        DatabaseOperation.CLEAN_INSERT.execute(dbConnection,dsInicializacion);

        // (2) obtencion del dataset esperado
        IDataset dsEsperado = new FlatXmlDataSet(new
            File("./testData/dataSets/esperado/
                DBRegistro.obtenerTramosTrafico.xml"));

        // (3) prueba de correccion de la sentencia SQL
        // ejecucion de la consulta
        QueryDataSet dsObtenido1 = new QueryDataSet(getConnection());
        dsObtenido1.addTable("consulta",
            m_dbRegistro.obtenerConsultaTramosTrafico("Trafico denso"));
        // comparacion con el dataset esperado
        Assertion.assertEquals(dsEsperado,dsObtenido1);

        // (4) prueba del codigo que guarda los datos obtenidos en la consulta
        // ejecucion del metodo a probar
        Vector<Registro> vRegistros =
            m_dbRegistro.obtenerRegistrosTramosTrafico("Trafico Denso");
        IDataset dsObtenido2 =
            this.crearDataSetConsulta(vRegistros,"consulta");
        // comparacion con el dataset esperado
        Assertion.assertEquals(dsEsperado,dsObtenido2);
    } catch (SQLException e) {

        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    } catch (DataSetException e) {

        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    }
}

```

```

    } catch (Exception e) {

        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    }
}

```

Este método realiza dos tareas bien diferenciadas. Primero se inicializa el contenido de la base de datos mediante la operación `CLEAN_INSERT`. Esta operación es muy poderosa ya que, por un lado, limpia el contenido de las tablas que aparecen en el dataset y, por otro lado, inserta en la base de datos el contenido del dataset. Dicho contenido necesita ser conocido para poder establecer cuál es el resultado esperado al ejecutar la consulta. En el ejemplo el contenido del dataset de inicialización es el siguiente:

pruebas sistema software/testData/dataSets/inicializacion/DBRegistro.obtenerTramosTráfico.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<dataset>
  <registro id="1" carretera="M-40" hora="12:23:45" fecha="2007-03-01"
    clima="Nublado" obras="No"/>
  <registro id="2" carretera="N-II" hora="12:23:45" fecha="2007-03-01"
    clima="Soleado" obras="No"/>
  <tramo id="1" kmInicio="1" kmFin="16" carriles="3"
    carrilesCortados="1" estado="Tráfico Denso" accidentes=
      "Sin accidentes" registro="1"/>
  <tramo id="2" kmInicio="17" kmFin="23" carriles="3"
    carrilesCortados="1" estado="Tráfico Denso" accidentes=
      "Sin accidentes" registro="1"/>
  <tramo id="3" kmInicio="1" kmFin="14" carriles="3"
    carrilesCortados="0" estado="Tráfico Fluído" accidentes=
      "Sin accidentes" registro="2"/>
  <tramo id="4" kmInicio="15" kmFin="28" carriles="3"
    carrilesCortados="1" estado="Tráfico Denso" accidentes=
      "Sin accidentes" registro="2"/>
</dataset>

```

Este dataset contiene 2 elementos en la tabla registro y 4 elementos en la tabla tramo. De esos 4 dos pertenecen al primer registro y los otros dos al segundo, mientras que el estado del tráfico es "Tráfico denso" solo para 3 de ellos.

El siguiente paso en la prueba es la obtención del dataset esperado que se realiza, como se vio anteriormente, mediante la clase `FlatXmlDataSet`. El dataset esperado está almacenado en disco en forma de documento XML y tiene el siguiente aspecto:

pruebas sistema software/testData/dataSets/esperado/DBRegistro.obtenerTramosTráfico.xml

```

<?xml version='1.0' encoding='UTF-8'?>
<dataset>

```

```

<consulta carretera="M-40" hora="12:23:45" fecha="2007-03-01"
  clima="Nublado" obras="No" id="1" kmInicio="1" kmFin="16"
  carriles="3" carrilesCortados="1" estado="Tráfico Denso"
  accidentes="Sin accidentes" registro="1"/>
<consulta carretera="M-40" hora="12:23:45" fecha="2007-03-01"
  clima="Nublado" obras="No" id="2" kmInicio="17" kmFin="23"
  carriles="3" carrilesCortados="1" estado="Tráfico Denso"
  accidentes="Sin accidentes" registro="1"/>
<consulta carretera="N-II" hora="12:23:45" fecha="2007-03-01"
  clima="Soleado" obras="No" id="4" kmInicio="15" kmFin="28"
  carriles="3" carrilesCortados="1" estado="Tráfico Denso"
  accidentes="Sin accidentes" registro="2"/>
</dataset>

```

Se esperan tres filas de datos, cada una de ellas conteniendo la información de uno de los tramos con “tráfico denso” junto con la información del registro al que pertenece el tramo. El nombre de los elementos es “consulta” ya que no se trata de elementos de una tabla en la base de datos sino de elementos fruto de la ejecución de una consulta SQL. A la hora de definir un dataset esperado tras la ejecución de una consulta es muy importante seleccionar qué columnas deben aparecer en dicho dataset y cuáles no. Idealmente el planteamiento es sencillo, deben aparecer todas las columnas de todas las tablas que intervengan en la consulta. En el ejemplo, estas columnas serían todas las de las tablas registro y tramo. Sin embargo, la realidad es bastante más complicada que eso. Como puede observarse la columna de nombre `id` de la tabla registro<sup>14</sup> no aparece en el dataset (los elementos `consulta` tienen un atributo `id`, pero este es el identificador del tramo en la tabla tramo, no del registro al que pertenece el tramo<sup>15</sup>). Esto es debido a que, por las propias restricciones de la sintaxis XML no es posible que dos atributos de un mismo elemento tengan el mismo nombre o, lo que es lo mismo, no es posible que un elemento `consulta` tenga dos atributos con el nombre `id`. Más adelante se volverá sobre esta cuestión..

A continuación, se verifica que la sentencia SQL a ejecutar es la correcta, es aquí cuando se hace uso del método refactorizado `obtenerConsultaTramosTráfico` para obtener la sentencia SQL en forma de `String`. En este punto resulta de gran utilidad la clase `QueryDataSet`, que implemente la interfaz `IDataset` y es capaz de ejecutar una sentencia SQL en forma de `String` sobre la base de datos y devolver un dataset. Este dataset no es posteriormente comparado con el dataset esperado visto anteriormente. En caso de que sean iguales se puede concluir que la sentencia SQL está bien escrita, por lo que el primer objetivo del método de prueba se ha completado.

El siguiente objetivo es la prueba del código que almacena los datos obtenidos al ejecutar la consulta, para lo cual, obviamente, es necesario invocar el método a probar. En este caso se obtiene un `Vector` de objetos `Registro` que debe contener exactamente la misma información presente en el dataset esperado. La cuestión es ¿cómo comparar el contenido del `Vector` con el contenido del dataset? La primera posibilidad que viene a la mente es iterar el `Vector` e ir comparando sus elementos (objetos `Registro` y `Tramo`) con el contenido del dataset. Acceder a la información en el dataset es posible utilizando los métodos de la interfaz `IDataset`. Sin embargo, existe una forma mucho mejor de realizar esta comparación. La idea es construir un objeto que implemente la interfaz `IDataset` a partir de la información contenida en el `Vector` de

<sup>14</sup> En el Apéndice B se describe en detalle la estructura de cada una de las tablas de la base de datos.

<sup>15</sup> Nótese que los atributos de los elementos `consulta` del dataset son los campos ordenados de la tabla registro y tramo consecutivamente (aunque algunos de ellos no aparezcan).

objetos. DUnit proporciona una clase llamada `DefaultTable` que es ideal para la construcción de datasets dinámicamente<sup>16</sup>. De esta forma, el objeto construido podrá posteriormente ser comparado con el dataset esperado mediante el método `assertEquals` de la clase `Assertion`, y así probar que el código de almacenamiento de la información de la consulta funciona correctamente.

DUnit proporciona dos clases muy útiles que se utilizan en combinación para generar datasets a partir de información contenida en objetos. Estas clases son `DefaultTable` y `DefaultDataSet`. Normalmente, se utiliza `DefaultTable` para crear una tabla de información, que puede ser una tabla en la base de datos o simplemente el resultado de una consulta. `DefaultDataSet` permite combinar múltiples tablas de las creadas anteriormente para la construcción de un dataset.

Nótese que la clase `DefaultTable` implementa la interfaz `ITable` mientras que la clase `DefaultDataSet` implementa la interfaz `IDataset`. Por este motivo ambas pueden utilizarse para realizar verificación de condiciones mediante los métodos de la clase `Assertion`.

A continuación se muestra el código fuente del método `crearDataSetConsulta`, encargado de construir un dataset a partir de los datos obtenidos en una consulta.

```
pruebas sistema software/src/pruebasSistemaSoftware/junit42/DBRegistroTest.java
```

```
IDataset crearDataSetConsulta(Vector<Registro> vRegistros, String
strNombre) throws DataSetException {
```

```
    Column columnas[] = new Column[13];
    columnas[0] = new Column("carretera",DataType.VARCHAR);
    columnas[1] = new Column("hora",DataType.VARCHAR);
    columnas[2] = new Column("fecha",DataType.VARCHAR);
    columnas[3] = new Column("clima",DataType.VARCHAR);
    columnas[4] = new Column("obras",DataType.VARCHAR);
    columnas[5] = new Column("id",DataType.INTEGER);
    columnas[6] = new Column("kmInicio",DataType.VARCHAR);
    columnas[7] = new Column("kmFin",DataType.VARCHAR);
    columnas[8] = new Column("carriles",DataType.VARCHAR);
    columnas[9] = new Column("carrilesCortados",DataType.VARCHAR);
    columnas[10] = new Column("estado",DataType.VARCHAR);
    columnas[11] = new Column("accidentes",DataType.VARCHAR);
    columnas[12] = new Column("registro",DataType.INTEGER);
```

```
    DefaultTable tabla = new DefaultTable(strNombre,columnas);
```

```
    for(Enumeration e = vRegistros.elements() ; e.hasMoreElements() ; ) {
        Registro registro = (Registro)e.nextElement();
```

<sup>16</sup> Hasta ahora los datasets se han construido a partir de documentos XML o bien a partir de información contenida en la base de datos, sin embargo nunca se ha construido un dataset desde cero como es este caso.

```

Vector<Tramo> vTramos = registro.obtenerTramos();
for(Enumeration ee = vTramos.elements() ; ee.hasMoreElements() ; ) {
    Tramo tramo = (Tramo)ee.nextElement();
    tabla.addRow(new Object[] {
        registro.obtenerCarretera(),
        registro.obtenerHora(),
        registro.obtenerFecha(),
        registro.obtenerClima(),
        registro.obtenerObras(),
        tramo.obtenerId(),
        tramo.obtenerKMInicio(),
        tramo.obtenerKMFin(),
        tramo.obtenerCarriles(),
        tramo.obtenerCarrilesCortados(),
        tramo.obtenerEstado(),
        tramo.obtenerAccidentes(),
        registro.obtenerId()
    });
}

return new DefaultDataSet(tabla);
}

```

La gran ventaja de este método es que es válido para construir datasets a partir de cualquier información contenida en un Vector de objetos Registro. Por tanto es muy general y puede reutilizarse para construir “datasets obtenidos” durante el proceso de prueba de cualquier método que ejecute una consulta sobre las tablas registro y tramo base de datos. Siempre se ha de buscar soluciones generales y no a medida de cada método de prueba, ya que de otra forma el proceso de prueba se convierte en una tarea tediosa y muy costosa en tiempo. La ganancia en generalidad de este procedimiento repercute directamente en la mantenibilidad del código de pruebas.

El último paso en el ejemplo es realizar una comparación entre el dataset esperado y el dataset construido mediante `crearDataSetConsulta`. Como se vio anteriormente el dataset esperado no contiene el campo `id` de la tabla registro ya que colisionaría con el campo `id` de la tabla tramo. Esto no supone ningún problema ya que los dos objetos Registro en la base de datos se diferencian más allá del valor de ese campo. Sin embargo el dataset esperado tampoco debe contener dicho campo `id`, lo que se ha tenido en cuenta a la hora de crearlo. Una reflexión razonable sería la siguiente ¿por qué no omitir los campos de auto-incremento en el proceso de comparación? Puesto que el valor de estos campos es asignado por el SGBD no hay de qué preocuparse ya que no pueden fallar. Sin embargo, esto no es del todo cierto, y solo es válido para el caso en el que las filas de una determinada tabla puedan diferenciarse más allá del valor de sus campos auto-incremento (normalmente claves primarias). Afortunadamente, es posible definir casos de prueba de modo que esta condición se cumpla siempre (en el ejemplo puede verse como todas las filas de la tabla tramo y de la tabla registro son diferentes más allá del valor del campo `id`). Esto último se recomienda como norma general. A pesar de todo existe un pequeño inconveniente, cuando se obtiene un dataset a partir de información contenida en la base de datos, este dataset puede contener columnas que no interesan para el proceso de comparación con el dataset esperado. Por ejemplo, es posible que no interese verificar el valor de la columna `id`. Para resolver esta situación, DBUnit proporciona un mecanismo para la eliminación selectiva de

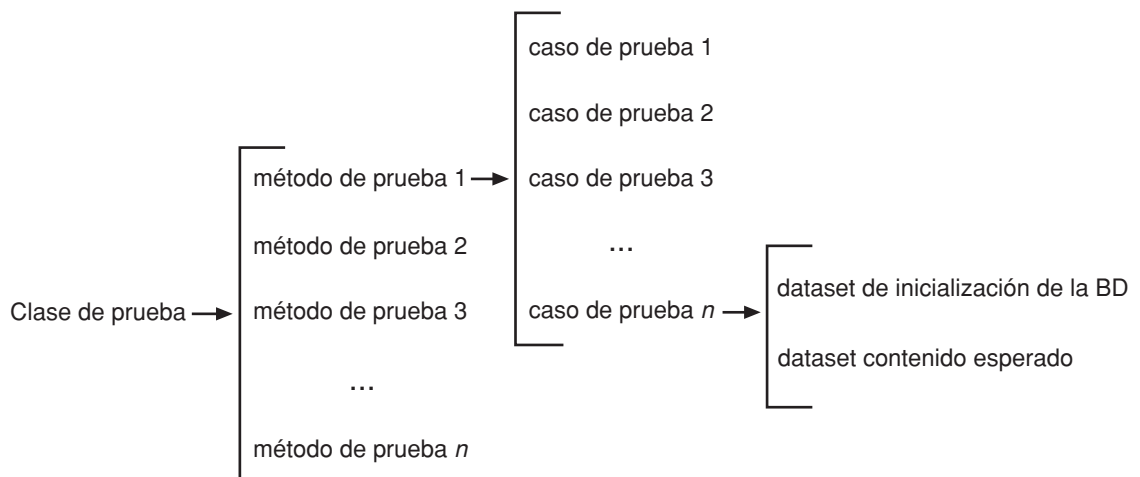
columnas en tablas<sup>17</sup>. El siguiente segmento de código muestra como eliminar de una tabla todas las columnas excepto aquellas presentes en la tabla esperada. Posteriormente el contenido de ambas tablas podrá ser comparado sin mayores problemas.

```
ITable tablaFiltrada =
DefaultColumnFilter.includedColumnsTable(tablaObtenida,
tablaEsperada.getTableMetaData().getColumns());
```

Este procedimiento es también útil en casos en los que se trabaja con campos de tipo fecha y hora en los que el código a probar asigna valores que no pueden ser conocidos *a priori* y que por tanto no interesa verificar.

### 9.3.3. Definición de los casos de prueba

Como se ha visto en los anteriores apartados, la definición de un caso de prueba sobre código que accede a una base de datos es una tarea compleja. En este apartado se van a ofrecer una serie de recomendaciones que harán la vida del desarrollador más llevadera. En los ejemplos anteriores, por razones de espacio, solo se ha ejecutado un caso de prueba en cada método de prueba, sin embargo en una prueba real normalmente se definen múltiples casos de prueba. Como puede observarse en el esquema de la Figura 9.3, para cada clase de prueba se definen una serie de casos de prueba para cada uno de sus métodos. Por otro lado para cada caso de prueba ha de existir (típicamente en forma de documento XML en disco) un dataset con el contenido con el que inicializar la base de datos antes de la ejecución del caso de prueba y otro con el contenido que se espera que haya en la base de datos una vez ejecutado el caso de prueba. Sin embargo, idealmente un único dataset podría ser utilizado para inicializar el estado de la base de datos para diferentes casos de prueba e incluso para diferentes métodos pertenecientes a una clase de prueba.



**Figura 9.3.** Esquema de una clase de prueba sobre una clase de acceso a la base de datos.

<sup>17</sup> Es posible obtener tablas (objetos que implementan la interfaz `ITable`) a partir de cualquier dataset, ya que los datasets heredan de la clase `AbstractDataSet` que presenta el método `getTable`.





**Figura 9.4.** Árbol de directorios con los datasets definidos para las clases de prueba de las clases `DBRegistro` y `DBTramo`.

En cuanto a cuestiones de nomenclatura y organización en disco, en la Figura 9.4 se muestra el árbol de directorios y correspondientes datasets asociados al código de pruebas del sistema presentado en el Apéndice B. Cada archivo XML con un dataset es nombrado siguiendo el siguiente patrón: “clasePrueba.metodoPrueba.xml” aunque en caso de que se definieran más de un caso de prueba para cada método de prueba el patrón sería: “clasePrueba.metodoPrueba.id-CasoPrueba.xml”

### 9.3.4. Recomendaciones

A continuación, se listan algunas recomendaciones a la hora de trabajar con DBUnit.

- Es fundamental realizar previamente un buen diseño, de forma que el código de acceso a la base de datos esté claramente diferenciado del resto del código del sistema. Solo así es posible utilizar herramientas como DBUnit de una forma efectiva. Recuérdese que las pruebas son una parte fundamental de la fase de desarrollo por lo que se ha de facilitar su realización en todo lo posible.
- Antes de realizar la prueba del código de acceso a la base de datos, verificar que el diagrama relacional de la base de datos ha sido creado correctamente. En particular se han de comprobar los siguientes puntos:

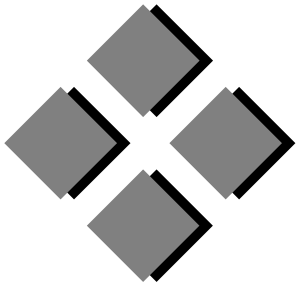
- Todas las tablas existen y han sido creadas con el nombre correcto.
  - Cada campo en una tabla dada tiene el nombre y tipo de dato asociado adecuado.
  - Se han definido valores por defecto en los casos necesarios.
  - Las claves primarias han sido seleccionadas para cada tabla.
  - Las claves externas<sup>18</sup> han sido definidas junto con las correspondientes restricciones y mecanismos de cascada.
  - Se han creado los procedimientos almacenados.
  - Se han creado los triggers necesarios junto con los procedimientos almacenados asociados.
- Utilizar una base de datos local para cada desarrollador en lugar de una base de datos compartida entre todos ellos. De esta forma, es posible realizar pruebas conociendo el estado exacto de cada tabla en la base de datos y sin correr el riesgo de producir efectos laterales que entorpezcan la tarea de otros desarrolladores. Otra ventaja de que no haya varios desarrolladores trabajando sobre la misma base de datos es que construir datasets reflejando el contenido de la base de datos es mucho más rápido cuando la base de datos sólo contiene unos pocos registros en ciertas tablas, insertados *ad-hoc* en el proceso de inicialización del caso de prueba.
  - Nunca crear casos de prueba que dependan del estado en que el caso de prueba anterior ha dejado la base de datos. Un principio fundamental de mantenibilidad es inicializar el contenido de la base de datos al principio de cada caso de prueba.
  - Los datasets definidos en los casos de prueba deberán ser lo más pequeños posibles, de forma que cumplan su objetivo y sean fácilmente mantenibles. Además, durante la fase de diseño de los mismos es muy importante tratar de reutilizarlos de forma que por ejemplo un dataset de inicialización del contenido de la base de datos pueda compartirse entre diferentes casos de prueba e incluso entre diferentes métodos de prueba de una misma clase de prueba.
  - Aunque anteriormente se ha hablado de procedimientos almacenados, estos dificultan enormemente el proceso de prueba por lo que se ha de evitar su uso en la medida de lo posible. Un procedimiento almacenado es un conjunto de sentencias SQL que reside en el SGBD y por tanto no es visible. Para probarlos es necesario utilizar un enfoque de caja negra.

## 9.4. Bibliografía

- Siggelkow, B.: *DBUnit Made Easy*, 13 de octubre de 2005, <http://www.oreillynnet.com/lpt/wlg/8096>
- <http://dbunit.sourceforge.net/>
- Ambler, S.: *Agile Database Techniques: Effective Strategies for the Agile Software Developer*, John Wiley & Sons, 2003.

---

<sup>18</sup> También conocidas como claves extranjeras o foráneas.



# Capítulo 10

## Pruebas de documentos XML: XMLUnit

---

### SUMARIO

<b>10.1.</b> Introducción	<b>10.5.</b> Cómo salvar diferencias superficiales
<b>10.2.</b> Configuración de XMLUnit	<b>10.6.</b> Prueba de transformaciones XSL
<b>10.3.</b> Entradas para los métodos de XMLUnit	<b>10.7.</b> Validación de documentos XML durante el proceso de pruebas
<b>10.4.</b> Comparación de documentos XML	<b>10.8.</b> Bibliografía

## 10.1. Introducción

En los últimos años el formato XML se ha convertido en uno de los estándares mas utilizados para el intercambio y representación de la información. Durante este tiempo, han surgido multitud de especificaciones basadas en el lenguaje XML para dar respuesta a las necesidades más diversas, desde la construcción de diálogos vocales (VXML) hasta la descripción de expresiones matemáticas (MathML), etc. A consecuencia de todo esto, cada día es mayor el volumen de aplicaciones software que hacen uso de este tipo de documentos y, por tanto, mayor importancia cobra el garantizar que estos documentos son construidos de una forma correcta.

Los documentos XML son utilizados por las aplicaciones normalmente para el intercambio de datos o para la representación de información<sup>1</sup>. Un ejemplo por todos conocido de esto último, es el formato XHTML, utilizado típicamente para la representación de información en apli-

---

<sup>1</sup> Este es el caso de la aplicación del servidor de tráfico (véase Apéndice B) que genera un documento XML como respuesta a una petición de información sobre el estado de las carreteras.

caciones Web. En todos estos casos existe la necesidad de verificar que el código XML generado sea el adecuado, acorde con la especificación de requisitos software, tanto en contenido como en estructura.

La primera idea que viene a la mente para probar que el código XML generado es el que se tiene que generar, es comparar visualmente el código esperado y el creado por el software en fase de pruebas. Lógicamente, las comprobaciones visuales nos remontan a la prehistoria de las pruebas de software, son muy lentas y es muy fácil cometer errores. XMLUnit es una herramienta que automatiza este trabajo. Se trata de un *framework* de código abierto disponible en el sitio Web <http://xmlunit.sourceforge.net/>.

XMLUnit extiende las clases `Assert` y `TestCase` de JUnit<sup>2</sup> con las clases `XMLAssert` y `XMLTestCase` respectivamente, incluidas en el paquete `org.custommonkey.xmlunit`. Utilizando estas clases se puede:

- Comparar código XML perteneciente a varios documentos.
- Validar el resultado de transformar código XML usando XSLT.
- Evaluar expresiones XPath en un código XML.
- Validar un documento XML.

XMLUnit también permite examinar código HTML (incluso mal formado) como XML válido, de forma que sea posible hacer comprobaciones sobre páginas Web.

Para hacer pruebas de software que maneja y genera documentos XML la estrategia consiste en hacer aserciones sobre los documentos usando XPath<sup>3</sup> para recuperar el contenido de ambos documentos y compararlos para comprobar si son iguales. XMLUnit encapsula el uso de XPath y facilita las pruebas del software que maneja documentos XML. Con XMLUnit se crea una clase de prueba que extiende a `XMLTestCase` y en su interior se definen los metodos de prueba, inicializacion, etc tal y como se hace con JUnit. Habitualmente, en XMLUnit, el código XML que se obtiene como resultado de ejecutar el software en pruebas se denomina código de prueba o de test (código obtenido). El código esperado se denomina código de control. Esta es la terminología que se usará en adelante.

En este capítulo se describe cómo configurar XMLUnit y los diferentes formatos aceptados por los métodos de las clases de XMLUnit. Después se explicará cómo escribir pruebas para software que maneja código XML, para comprobar que las transformaciones XSL se han realizado correctamente y para validar los documentos XML usando un DTD o un esquema.

## 10.2. Configuración de XMLUnit

Para poder ejecutar XMLUnit hace falta:

- JUnit 3.8.1<sup>4</sup>.

---

<sup>2</sup> Véase Capítulo 2. Pruebas Unitarias: JUnit.

<sup>3</sup> XPath (XML Path Language) es un lenguaje que permite construir expresiones que recorren y procesan un documento XML, para seleccionar partes del documento. XPath permite buscar y seleccionar teniendo en cuenta la estructura jerárquica del XML. XPath fue creado para su uso en el estándar XSLT, en el que se usa para seleccionar y examinar la estructura del documento de entrada de la transformación.

<sup>4</sup> La distribución binaria de XMLUnit 1.0 no es compatible con versiones anteriores a JDK1.4.1, JUnit 3.8.1 y Ant 1.5.

- Un parser para XML. Hay muchos disponibles y se puede utilizar cualquier librería compatible con JAXP. Por ejemplo Xerces-J<sup>5</sup>.
- Un motor de transformación compatible con JAXP/Trax para transformaciones XSLT y expresiones XPath. Por ejemplo Xalan-J<sup>6</sup>.

Si se usa la versión 1.4 de JDK o posteriores, la librería de clases de Java ya contiene los parsers de XML y los motores de transformación XSLT necesarios. De todos modos se pueden utilizar otros, para lo que es necesario configurar XMLUnit. Esta configuración se puede realizar mediante las propiedades de `System` o mediante los métodos estáticos de la clase `XMLUnit`.

Para configurar XMLUnit mediante las propiedades de `System` se debe dar valor a estas propiedades antes de ejecutar ninguna prueba. Por ejemplo:

```
System.setProperty("javax.xml.parsers.DocumentBuilderFactory",
"org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
System.setProperty("javax.xml.parsers.SAXParserFactory",
"org.apache.xerces.jaxp.SAXParserFactoryImpl");
System.setProperty("javax.xml.transform.TransformerFactory",
"org.apache.xalan.processor.TransformerFactoryImpl");
```

Utilizando los métodos estáticos de la clase `XMLUnit` del paquete `org.custommonkey.xmlunit`, la configuración se realiza estableciendo el parser y el motor de transformación en el método `setUp()` de la clase de pruebas, si se desea utilizar el mismo para todas las pruebas. Por ejemplo:

```
public void setUp() throws Exception {
    XMLUnit.setControlParser("org.apache.xerces.jaxp.DocumentBuilder
FactoryImpl");
    // La siguiente línea no es imprescindible. Si no se especifica un
    // parser para el código de test, se usa el mismo parser que en el
    // código de control
    XMLUnit.setTestParser("org.apache.xerces.jaxp.DocumentBuilder
FactoryImpl");
    XMLUnit.setSAXParserFactory("org.apache.xerces.jaxp.SAXParser
FactoryImpl");
    XMLUnit.setTransformerFactory("org.apache.xalan.processor.
TransformerFactoryImpl");
}
```

La ventaja de la primera alternativa es que afecta a todo el sistema JAXP se use o no XMLUnit. La ventaja de la segunda opción es que se puede especificar un parser diferente para el código de pruebas y el de control, y cambiarlo en cualquier momento de las pruebas<sup>7</sup>. Solo hace falta comprobar la compatibilidad de los diferentes parsers.

<sup>5</sup> <http://xerces.apache.org>

<sup>6</sup> <http://xalan.apache.org/xalan-j>

<sup>7</sup> En este caso, la especificación del parser y del motor de transformación XSLT no se realiza en el método `setUp()`, sino en la prueba en la que se deseen utilizar.

### 10.3. Entradas para los métodos de XMLUnit

Los métodos de XMLUnit que esperan código XML como parámetro de entrada, lo pueden recibir de diferentes fuentes. Las más habituales son:

- `DOM Document`. Se tiene todo el control sobre la creación del documento XML, que puede ser el resultado de una transformación XSLT mediante la clase `Transform`.
- `SAX InputSource`. Es la forma más general puesto que `InputSource` permite leer de un `InputStream` o un `Reader` indistintamente. Conviene usar un `InputStream` encapsulado por un `InputSource` si se quiere que el parser de XML elija la codificación adecuada de XML.
- Objeto de tipo `String`. El `DOM Document` se construye a partir del `String` usando el parser especificado. Como se ha visto en el apartado anterior este puede ser diferente para el código XML de control y el de prueba.
- `Reader`. El `DOM Document` se construye a partir de la entrada `Reader` usando el parser especificado. No se debe usar esta opción si la codificación del XML a comparar es distinta de la codificación por omisión de la plataforma, puesto que el sistema IO de Java no leerá el XML de entrada. En este caso es mejor usar otra versión del método utilizado para la comparación que asegure que la codificación se tratará de forma adecuada.

### 10.4. Comparación de documentos XML

Una de las pruebas más habituales que hay que realizar consiste en determinar si dos trozos de código XML son iguales. De esta forma se comprueba si el código XML producido por el software que se está probando es igual al código XML esperado, acorde con las especificaciones software prefijadas. En XMLUnit dos trozos de código XML pueden ser iguales, similares o diferentes. Se consideran iguales si no existe ninguna diferencia entre ellos, es decir, son idénticos. Si se encuentran diferencias, estas pueden ser recuperables, en cuyo caso los códigos se consideran similares. Por ejemplo, el código:

```
<Tramo>
  <Carretera>N-401</Carretera>
  <Carriles>3</Carriles>
  <CarrilesCortados>1</CarrilesCortados>
  <Estado>Trafico fluido</Estado>
</Tramo>
```

es similar al código:

```
<Tramo>
  <Carretera>N-401</Carretera>
  <Estado>Trafico fluido</Estado>
  <Carriles>3</Carriles>
  <CarrilesCortados>1</CarrilesCortados>
</Tramo>
```

ya que contienen los mismos elementos en diferente orden, y esos elementos son hermanos. En el primer fragmento de código, el elemento `<Estado>` aparece como cuarto nodo de `<Tramo>`, sin embargo, en el segundo fragmento aparece como segundo elemento.

Si las diferencias son irre recuperables se concluye que los trozos de código XML son diferentes.

La clase más importante para la comparación de código XML es `DifferenceEngine`, pero la mayoría de las veces se usa de manera indirecta a través de las clases `Diff` o `DetailedDiff`. `DifferenceEngine` analiza y compara el documento esperado y el obtenido durante la ejecución de la prueba. Cuando encuentra una diferencia, envía un mensaje a un `DifferenceListener` que decide cómo tratar esa diferencia (véase Apartado 10.5.3) y pregunta a un `ComparisonController` si la comparación debe ser interrumpida. A veces el orden de los elementos en los dos trozos de código XML no es significativo. En este caso, `DifferenceEngine` necesita ayuda para determinar qué elementos comparar. Esta ayuda se la presta un `ElementQualifier`.

Los tipos de diferencias descritos hasta ahora que se pueden encontrar con XMLUnit y se pueden consultar en la guía del usuario de XMLUnit (<http://xmlunit.sourceforge.net/user-guide/html/index.html>). Estas diferencias se enumeran en la interfaz `DifferenceConstants` y se representan mediante instancias de la clase `Difference`. Además del tipo de diferencia, esta clase mantiene información de los nodos que se han detectado como diferentes. `DifferenceEngine` pasa las diferencias encontradas como instancias de la clase `Difference` al `DifferenceListener`, una interfaz cuya implementación por omisión viene dada por la clase `Diff`. Algunas de estas diferencias se pueden ignorar implementando dicha interfaz<sup>8</sup>.

Para comparar dos trozos de código XML y comprobar si son iguales se recomienda definir la clase de prueba extendiendo la clase `XMLTestCase`. `XMLTestCase` proporciona muchos métodos `assert...` que simplifican el uso de XMLUnit. La clase `XMLAssert` proporciona los mismos `assert...` como métodos estáticos. Se debe usar `XMLAssert` en lugar de `XMLTestCase` cuando no se pueda heredar de `XMLTestCase`.

Un ejemplo de prueba que hereda de `XMLTestCase` es el siguiente:

```
public class claseXMLTest extends XMLTestCase {
    public claseXMLTest (String nombre) {
        super(nombre);
    }

    public void testDeIgualdad() throws Exception {
        String XMLdeControl = "<Tramo><Carretera>M-30</Carretera>
                                </Tramo>";
        String XMLdePrueba = "<Tramo><Carriles>3</Carriles></Tramo>";
        assertXMLEqual("Comparación de documentos XML", XMLdeControl,
                        XMLdePrueba);
    }
}
```

Con el método `assertXMLEqual()`, XMLUnit determina si dos piezas de XML son idénticas o similares. En el ejemplo anterior, los dos trozos de código XML son diferentes y la

---

<sup>8</sup> Véase Apartado 10.5.3.

prueba fallará. El mensaje de error indica la diferencia encontrada y la localización de los nodos que se están comparando.

Cuando se comparan dos códigos XML, se crea una instancia de la clase `Diff`. Esta clase almacena el resultado de la comparación y lo muestra con los métodos `identical()` y `similar()`. El método `assertXMLEqual()` comprueba el valor de `Diff.similar()`. También se dispone del método `assertXMLIdentical()` que comprueba el valor de `Diff.identical()`.

Otra forma de definir la prueba es crear una instancia de la clase `Diff` directamente, sin usar la clase `XMLTestCase`. Por ejemplo, para comparar si la información generada por el servidor de tráfico<sup>9</sup> cuando se realiza una petición sobre el estado de las carreteras es la que se desea obtener se escribe el siguiente código:

```
pruebas sistema software\src\pruebasSistemaSoftware\funcionales\PeticionesTest.java
```

```
Diff diff = new Diff(docEsperado,docObtenido);
boolean bResultado = diff.identical();
assertTrue("Error en el formato del documento XML generado: " + diff,
bResultado);
```

Después de ejecutar la petición, almacenarla en la variable `docObtenido` y de cargar el documento esperado en la variable `docEsperado`, se buscan las diferencias entre los documentos. Cuando se encuentre la primera diferencia, se almacena en `diff` y se muestra el resultado de la comparación mediante la aserción `assertTrue()`.

Por razones de eficiencia `Diff` detiene la comparación cuando encuentra la primera diferencia. Para obtener todas las diferencias entre dos códigos XML hay que utilizar una instancia de la clase `DetailedDiff` que se construye usando una instancia previa de `Diff` que lista todas las diferencias entre los dos documentos. Por ejemplo:

```
Diff diff = new Diff(docEsperado,docObtenido);
DetailedDiff myDiff = new DetailedDiff(diff);
assertTrue(myDiff.toString(), diff.identical());
```

### 10.4.1. ¿Qué métodos de aserción utilizar para comparar código XML?

Las clases `XMLAssert` y `XMLTestCase` contienen varios métodos para comparar dos trozos de código XML. Los nombres de los métodos usan la palabra *Equal*, para indicar que son similares, es decir, que entre ellos existen diferencias recuperables. Ya se ha visto que `assertXMLEqual()` indica si los trozos de código son similares. Para ver si son iguales, es decir, no hay ninguna diferencia, se usa `assertXMLIdentical()`. Si se quiere comprobar si hay diferencias se usa el método `assertXMLNotIdentical`. Para saber si alguna de las diferencias no es recuperable se usa `assertXMLNotEqual()`.

---

<sup>9</sup> Véase el Apéndice B.



`assertXMLEqual()` es un método sobrecargado. Cada una de las versiones proporciona diferentes formas de especificar el código a comparar: `String`, `InputStream`, `Reader` o `Document` (véase Apartado 10.3). Y para cada método hay una versión que añade el parámetro `err` para crear el mensaje si la aserción falla.

Si no se necesita usar un *DifferenceListener* diferente al usado por omisión por `Diff`, no es necesario capturar una diferencia en una instancia de esta clase, ya que

```
Diff d = new Diff(XMLdeControl, XMLdePrueba);
assertTrue("los codigos comparados son similares, " + d.toString(),
    d.similar());
```

es equivalente a escribir:

```
assertXMLEqual("los codigos comparados son similares", XMLdeControl,
    XMLdePrueba);
```

Si se utiliza la clase `DetailedDiff`, porque interese conocer todas las diferencias entre los códigos XML comparados, no es posible usar los métodos de las clases `XMLAssert` y `XMLTestCase`.

## 10.5. Cómo salvar diferencias superficiales

A veces, las diferencias que se encuentran entre dos documentos XML se pueden ignorar si no afectan al objetivo del software que se está probando. Este es el caso, por ejemplo, de los espacios en blanco, los comentarios o el valor de los atributos y el orden en el que estos aparecen. Sin embargo, nótese que estas salvedades van a estar supeditadas a la información al respecto contenida en el documento de especificaciones software.

### 10.5.1. Ignorar los espacios en blanco

Un parser de XML trata un espacio en blanco como un nodo de texto con contenido vacío, de forma que en el código XML:

```
<Tramo>
  <Carretera>N-401</Carretera>
  <Estado>Trafico fluido</Estado>
</Tramo>
```

se contarían cinco nodos en el elemento `Tramo`: un elemento de texto vacío, el elemento `Carretera`, otro elemento vacío, el elemento `Estado` y otro elemento vacío. Muchas veces puede interesar que no se tengan en cuenta estos espacios. XMLUnit proporciona el método `setIgnoreWhitespace()` que ignora los elementos con contenido vacío cuando se compara código XML. Sin embargo, no ignora los espacios en blanco dentro del valor de un nodo de texto.

Si los espacios en blanco se van a ignorar para todas las pruebas se puede especificar esta característica en el método `setUp()`, usando:

```
XMLUnit.setIgnoreWhitespace(true);
```

## 10.5.2. Ignorar los comentarios

Al igual que para los espacios en blanco, XMLUnit proporciona un método que ignora los comentarios en los documentos XML. Este método pertenece a la clase XMLUnit y para activar esta característica se invoca el método:

```
XMLUnit.setIgnoreComments (true);
```

Al igual que con los espacios en blanco, si los comentarios se quieren ignorar para todas las pruebas, esta instrucción se incluiría en el método `setUp()`.

## 10.5.3. La interfaz *DifferenceListener*

De forma general, XMLUnit proporciona la forma de ignorar ciertas diferencias mediante la interfaz *DifferenceListener*. Implementando esta interfaz se puede definir qué significa “diferente” para cada prueba. Si todas las diferencias encontradas se ignoran con la nueva implementación de *DifferenceListener*, los trozos de código XML se considerarán similares.

La interfaz *DifferenceListener* contiene dos métodos: `differenceFound()` y `skippedComparison()`. Cuando XMLUnit encuentra una diferencia, se invoca el método `differenceFound()` pasando como parámetro un objeto *Difference*. Como las instancias de *Difference* contienen los detalles de las diferencias encontradas (véase Apartado 10.4), se puede determinar si se trata de la diferencia que se pretende ignorar o no. Este método devuelve una de las siguientes constantes:

- `RETURN_ACCEPT_DIFFERENCE`: indica que la diferencia encontrada se acepta como se ha definido en *DifferenceConstants*. Es decir, no se ignora.
- `RETURN_IGNORE_DIFFERENCE_NODES_IDENTICAL`: indica que los nodos identificados como diferentes se deben interpretar para esta prueba como idénticos.
- `RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR`: indica que los nodos identificados como diferentes se deben interpretar para esta prueba como similares.

Si XMLUnit encuentra dos nodos que no puede comparar se invoca el método `skippedComparison()`. Este método es invocado si la *Difference* encontrada por `differenceFound()` es del tipo `NODE_TYPE`.

El ejemplo siguiente implementa la interfaz *DifferenceListener* de forma que se ignoren las diferencias en los valores de texto, haciendo similares los documentos comparados si solo se encuentra dicha diferencia.

```
public class IgnorarValoresTextoDifferenceListener implements
    DifferenceListener {

    private boolean diferenciaIgnorada(Difference diferencia) {
        int IdDiferencia = diferencia.getId();
        if (IdDiferencia == DifferenceConstants.TEXT_VALUE.getId()) {
            return true;
        }
    }
}
```

```

    }
    return false;
}

public int differenceFound(Difference diferencia) {
    if (diferenciaIgnorada(diferencia)) {
        return RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR;
    } else {
        return RETURN_ACCEPT_DIFFERENCE;
    }
}

public void skippedComparison(Node control, Node test) {
}
}

```

Para utilizar la nueva implementación de la interfaz *DifferenceListener* se invoca el método `overrideDifferenceListener()` de la clase *Diff* y después se realiza la aserción deseada. En el ejemplo siguiente, después de capturar la primera diferencia encontrada en la variable `diferencia`, si los documentos no son similares a pesar de las diferencias ignoradas, `assertTrue()` fallará, indicando que hay diferencias irre recuperables.

```

diferencia.overrideDifferenceListener(new
IgnorarValoresTextoDifferenceListener());
assertTrue(diferencia.toString(), diferencia.similar());

```

Un ejemplo de implementación de *DifferenceListener* es la clase *IgnoreTextAndAttributeValuesDifferenceListener*. Cuando solo interesa comprobar la igualdad en la estructura de los dos códigos XML a comparar, sin tener en cuenta las diferencias en el valor de los atributos o el orden de los mismos, esta clase hace que la comparación ignore las diferencias en los valores de texto y en los atributos. Un ejemplo de utilización de esta clase es la comparación del esqueleto de dos trozos de código XML:

```

public void CompararEstructurasXML() throws Exception {
    String XMLdeControl = "...";
    String XMLdePrueba= "...";
    DifferenceListener miDifferenceListener = new
        IgnoreTextAndAttributeValuesDifferenceListener();
    Diff diferencia = new Diff(XMLdeControl, XMLdePrueba);
    diferencia.overrideDifferenceListener(miDifferenceListener);
    assertTrue("las estructuras del XML de prueba y de control
coinciden",
        deiferencia.similar());
}

```

## 10.6. Prueba de transformaciones XSL

Con XMLUnit se puede probar si una transformación XSL se ha realizado correctamente. Esta comprobación se realiza usando la clase *Transform*. Conociendo el código XML que se

quiere transformar, la hoja de estilo y el código XML esperado, se puede comprobar si el resultado de la transformación coincide con la salida esperada de la siguiente manera:

```
public void testTransformacionXSL() throws Exception {
    String entradaXML = "...";
    File hojaEstilo = new File("...");
    Transform miTransformacion = new Transform(entradaXML,
        hojaEstilo);
    String salidaEsperadaXML = "...";
    Diff miDiferencia = new Diff(salidaEsperadaXML, miTransformacion);
    assertTrue("La transformación fue correcta", miDiferencia.
        similar());
}
```

La instancia de la clase `Transform` se crea con el documento que se quiere transformar y una hoja de estilo, dando como resultado la transformación del documento de entrada. Después se compara dicha instancia con la salida esperada mediante una instancia de la clase `Diff` y una aserción como se explicó en el Apartado 10.4.

El resultado de la transformación se puede recuperar como un `String` o un `DOM Document` mediante los métodos `getResultString()` y `getResultDocument()` de la clase `Transform` respectivamente. Por ejemplo, si la salida esperada está disponible como `Document`, se comprueba la corrección de la transformación de la siguiente manera:

```
public void testTransformacionXSL() throws Exception {
    File entradaXML = new File("...");
    File hojaEstilo = new File("...");
    Transform miTransformacion = new Transform
        (new StreamSource(entradaXML),
        new StreamSource(hojaEstilo));
    Document salidaEsperadaXML = XMLUnit.buildDocument
        (XMLUnit.getControlParser(),
        new FileReader("..."));
    Diff miDiferencia = new
    Diff(salidaEsperadaXML, miTransformacion.getResultDocument());
    assertTrue("La transformación fue correcta", miDiferencia.
        similar());
}
```

La salida de `Transform` se puede usar como entrada para una comparación, una validación o cualquier otro tipo de prueba. Una transformación es un modo diferente de crear la entrada para cualquier otra utilidad de `XMLUnit`.

## 10.7. Validación de documentos XML durante el proceso de pruebas

Muchas veces es conveniente comprobar que el documento XML que sirve como entrada de algún método de la aplicación es válido. El objetivo es asegurarse de que, al ejecutarse las pruebas, los fallos detectados se deban realmente a defectos en el software y no a que los datos de entrada, en este caso el documento XML, no sean válidos. El tipo de documentos que hay que

validar son generados por el sistema en tiempo de ejecución, según una estructura predecible. Entonces, es aconsejable añadir pruebas que verifiquen que esos documentos se ajustan a la estructura esperada. Para ello se puede usar un DTD o un esquema XML.

XMLUnit da soporte para la validación de código XML con la clase `Validator`, que encapsula las operaciones necesarias para dicha validación. En los apartados siguientes se introduce el uso de esta clase para la validación de código XML frente a un DTD o frente a un esquema.

### 10.7.1. Validación frente a un DTD

Validar frente a un DTD es sencillo si el código XML contiene una declaración del tipo de documento, `DOCTYPE`, con un identificador `SYSTEM`. En este caso el parser localizará el documento usando el identificador dado. Es necesario crear un objeto `Validator` con un constructor de un solo argumento. Por ejemplo:

```
InputStream codXML = new InputStream(new FileInputStream(miDocXML));
Validator v = new Validator(codXML);
boolean esValido = v.isValid();
```

Puede suceder que el código XML no contenga la declaración del tipo de documento, `DOCTYPE`, o, aunque la contenga, se quiera validar frente a otro DTD. En estos casos se puede localizar el DTD en tiempo de ejecución. Ahora, para crear la instancia de `Validator` se debe usar una versión del constructor con dos argumentos especificando el argumento `systemID` como la URL alternativa donde encontrar el DTD.

```
InputStream codXML = new InputStream(new FileInputStream(miDocXML));
Validator v = new Validator(codXML, (new File(miDTD)).toURI().
    toURL().toString());
assertTrue(v.toString(), v.isValid());
```

Otra forma de especificar la localización del DTD es usar un `EntityResolver` mediante el método `XMLUnit.setControlEntityResolver`. Esta solución permite usar un catálogo OASIS<sup>10</sup> con la librería de Apache XML Resolver<sup>11</sup> para resolver la localización del DTD. Por ejemplo:

```
InputStream codXML = new InputStream(new FileInputStream(miDocXML));
XMLUnit.setControlEntityResolver(new CatalogResolver());
Validator v = new Validator(codXML);
boolean esValido = v.isValid();
```

con el catálogo:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
<public publicId="-//Some//DTD V 1.1//EN" uri="midtd.dtd"/>
</catalog>
```

<sup>10</sup> <http://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html>

<sup>11</sup> <http://xml.apache.org/commons/components/resolver/index.html>

## 10.7.2. Validación frente a un esquema XML

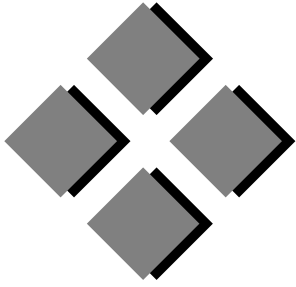
Por omisión, la validación de código XML se realiza frente a un DTD. Para validar frente a un esquema XML hay que usar el método `useXMLSchema` de la clase `Validator` que habilita esta opción.

El documento que se pretende validar debe declarar un espacio de nombres usando la URI del esquema y debe tener el atributo `schemaLocation` que indica al parser de XML dónde encontrar la definición del esquema. Si no existe el atributo `schemaLocation`, el parser de XML tratará de usar la URI del espacio de nombres como una URL en la que leer la definición del esquema. A veces no es posible disponer de un `schemaLocation`, ni usar una URL válida. Para estos casos, JAXP hace posible dar la localización del esquema mediante programación con el método `setJAXP12SchemaSource` de la clase `Validator`. Dicha localización se puede especificar como un `String` que contiene una URI o un `InputStream`, un `InputSource` o un `File` del que se puede leer el esquema. En el ejemplo siguiente la localización del esquema se define mediante un fichero:

```
String ejemplo = "";
Validator v = new Validator(ejemplo);
v.useXMLSchema(true);
v.setJAXP12SchemaSource(new File("ejemplo.xsd"));
assertTrue(v.toString(), v.isValid());
```

## 10.8. Bibliografía

- Rainsberger, J. B. (con contribuciones de Stirling, S.): *JUnit Recipes. Practical Methods for Programmer Testing*, Manning Publications, Greenwich, 2005.
- Bacon, T.; Bodewing, S.: *XMLUnit Java User's Guide*, abril 2007 (<http://xmlunit.sourceforge.net/userguide/html/index.html>).



# Capítulo 11

## Prueba de aplicaciones Web

### SUMARIO

**11.1.** Introducción

**11.3.** Prueba de un sitio Web

**11.2.** Herramientas para la automatización de la prueba

**11.4.** Bibliografía

### 11.1. Introducción

A lo largo de este capítulo se van a describir técnicas de prueba de aplicaciones Web enfocadas a la realización de las pruebas de sistema y de validación (también conocidas como pruebas funcionales). El objetivo de estas técnicas de prueba va a ser, por tanto, verificar que el sistema Web en desarrollo satisface los requisitos contenidos en el Documento de Especificación de Requisitos Software elaborado al inicio de la fase de desarrollo del producto <sup>1</sup>. No obstante, como se verá, muchas de estas técnicas pueden aplicarse en la prueba aislada de componentes Web, es decir, aisladamente respecto a la aplicación Web a la que pertenecen. Se ha utilizado la expresión “prueba aislada de componentes” en lugar de “realización de pruebas unitarias” porque la tarea de pruebas se va a centrar más en los efectos producidos por estos componentes Web (páginas HTML, conexiones HTTP, flujo de navegación, etc.) que en las propias clases que constituyen dichos componentes.

---

<sup>1</sup> Información en detalle acerca de este tipo de pruebas y su contexto en el proceso de pruebas puede encontrarse en el Capítulo 1.

El procedimiento básico de pruebas de aplicaciones Web se basa en definir unas entradas (información a partir de la cual la aplicación generara una serie de contenidos Web) y realizar comprobaciones sobre las salidas (contenidos Web generados: documentos HTML, JavaScript, documentos XML, etc.). Se trata, por tanto, de pruebas funcionales que se realizan mediante un enfoque de caja negra, es decir, los casos de prueba se construyen sin tener en cuenta la estructura interna de la aplicación sino únicamente sus entradas y salidas esperadas. Estas entradas y salidas son típicamente peticiones HTTP y documentos Web respectivamente. En contraposición, las pruebas de componentes Web se pueden realizar mediante un enfoque de caja negra o caja blanca.

La automatización de las pruebas funcionales en general y de las pruebas funcionales sobre aplicaciones Web en particular, permite disminuir notablemente el tiempo empleado para la validación de la aplicación y, lo que es incluso más importante<sup>2</sup>, reducir considerablemente el tiempo empleado en el mantenimiento de la misma gracias a la facilidad para ejecutar las pruebas de regresión.

Antes de continuar leyendo, puesto que la mayoría de los ejemplos con los que se va a ilustrar este capítulo giran en torno al sistema software descrito en el Apéndice B de este libro, se recomienda encarecidamente la lectura de dicho Apéndice. Puesto que este capítulo está dedicado fundamentalmente a las pruebas funcionales, conviene al menos tener una idea general de la funcionalidad de dicho sistema. Visto desde el exterior y, a grandes rasgos, este sistema se puede considerar como un pequeño servidor Web que genera contenidos sobre el estado del tráfico en función de información que obtiene de un fichero de registros o bien de una base de datos. Como todo servidor Web, los contenidos generados son documentos Web (formatos XML y HTML) y están accesibles a través del protocolo HTTP. La prueba de este sistema presenta una gran diferencia con respecto a la prueba de una aplicación Web convencional alojada en un servidor Web como pueda ser Apache. La diferencia es que mientras que en este último caso la aplicación está contenida en el servidor Web que lógicamente no entra dentro del alcance de las pruebas, en el primer caso el servidor Web sí forma parte del objetivo de las pruebas ya que está embebido en el propio sistema a probar.

## 11.2. Herramientas para la automatización de la prueba

A la hora de poner en práctica las técnicas de prueba de aplicaciones Web, existe una serie de herramientas que facilitan el trabajo del desarrollador gracias a que permiten automatizar la mayor parte de las tareas y ocultar al desarrollador los detalles de manejo de protocolos y formatos de representación Web. Se trata de herramientas relativamente recientes y que, sin embargo, han alcanzado una gran popularidad y madurez. Actualmente las más utilizadas son JWebUnit, HtmlUnit y HttpUnit<sup>3</sup> y todas ellas se basan en la emulación de un navegador Web para interactuar con el servidor Web en el que está alojada la aplicación a probar. Puesto que el objetivo es verificar la navegación y el contenido de los documentos Web que son mostrados al usuario a través de la ventana del navegador, la única forma de realizar la prueba es emulando la interacción del usuario con la aplicación Web, es decir, emulando el navegador Web. En la

---

<sup>2</sup> Normalmente, la fase de desarrollo de una aplicación es de menor duración que la fase de mantenimiento.

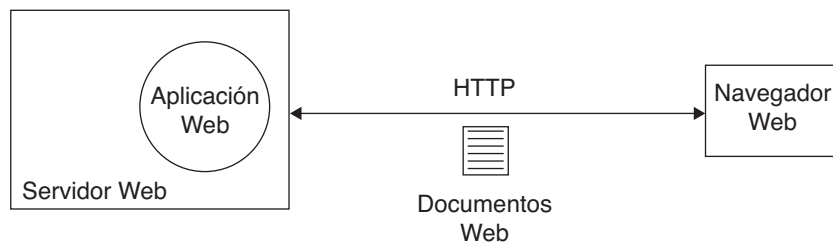
<sup>3</sup> Llama la atención que estas tres herramientas tienen el sufijo Unit como parte del nombre, cuando no están orientadas a pruebas unitarias sino a pruebas funcionales. Este fenómeno se puede explicar atendiendo a la enorme popularidad de la herramienta JUnit, que ha conseguido que un gran porcentaje de las herramientas dedicadas a complementarla utilicen el sufijo Unit como forma de ser inmediatamente reconocidas como tales.



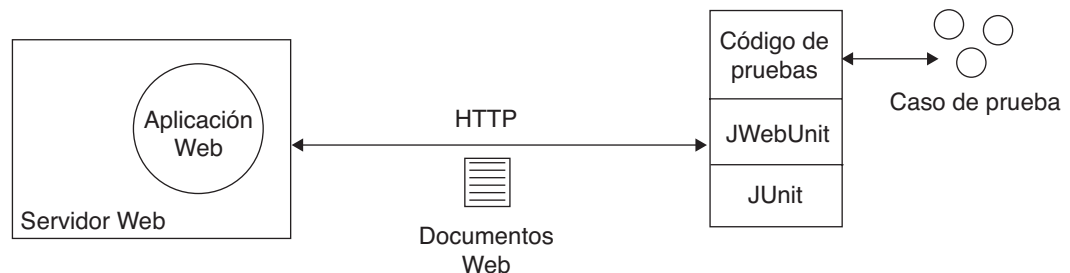
Figura 11.1 se muestran los dos esquemas de interacción Web, en el esquema tradicional el usuario utiliza acceder a la aplicación Web mediante el navegador. En el esquema de pruebas funcionales, el código de pruebas desarrollado hace uso de la herramienta JWebUnit (para la figura se ha escogido JWebUnit, pero podría ser cualquiera de las tres mencionadas) para ejecutar los casos de prueba sobre la aplicación Web.

Diferentes navegadores Web (FireFox, Internet Explorer, etc.) e incluso diferentes versiones del mismo navegador presentan particularidades (manejo de las cookies, visualización de ciertos elementos como los frames, etc.) que incrementan la complejidad de la prueba. Estas herramientas tienen en cuenta muchos de estos detalles permitiendo, por ejemplo, probar la navegación sobre un sitio Web con las características comunes a todos los navegadores o bien teniendo en cuenta las particularidades de cualquiera de ellos. A continuación, se listan las principales características que dichas herramientas presentan para asistir al desarrollador en el proceso de pruebas:

- Comunicación mediante el protocolo HTTP (e incluso HTTPS) con el servidor Web para el envío de peticiones y recepción de respuestas.
- Emulación de la interacción con el navegador Web, es posible realizar clics sobre enlaces y rellenar formularios HTML y enviarlos.
- Mecanismos de procesamiento de documentos HTML y métodos assert especializados para la verificación de la estructura de estos documentos. Asimismo posibilitan verificar código JavaScript embebido dentro de estos documentos.
- Mecanismos de captura de documentos XML que pueden ser procesados posteriormente desde niveles superiores.



Esquema de interacción Web durante las pruebas funcionales



**Figura 11.1.** Comparación de los esquemas de interacción Web.

En los siguientes subapartados se va a realizar una breve descripción de las tres herramientas que se utilizarán a lo largo de este capítulo. Nótese que dado que todas ellas presentan características similares, se han enumerado por orden cronológico.

Una característica común a todas estas herramientas y que las hace realmente atractivas y recomendables es que, dado que la interacción que realizan sobre la aplicación Web es únicamente a través del protocolo HTTP, permiten realizar pruebas sobre todo tipo de aplicaciones Web independientemente del lenguaje en el que hayan sido desarrolladas.

### 11.2.1. HttpUnit

Esta herramienta es la más antigua de todas y probablemente la primera herramienta de automatización de pruebas sobre aplicaciones Web que ha alcanzado una gran difusión. Se trata de una herramienta de código abierto que está alojada en el sitio Web <http://httpunit.sourceforge.net/>. Es una herramienta muy completa en lo que concierne a las pruebas de documentos Web. Es capaz de procesar documentos HTML y verificar sus enlaces, tablas, formularios e incluso código JavaScript. Por otro lado, es capaz de convertir documentos XML obtenidos vía HTTP en objetos que pueden ser verificados por el código de pruebas.

### 11.2.2. HtmlUnit

Esta herramienta es muy similar a la anterior, en teoría, debido a la forma en que ha sido implementada y a la interfaz que ofrece al desarrollador, está más orientada al manejo de documentos Web obtenidos al realizar peticiones sobre el servidor Web que al protocolo HTTP en sí. Sin embargo, en la práctica, las posibilidades que presenta al desarrollador, así como la forma de uso, son muy similares a las de HttpUnit. A pesar de que HtmlUnit sea realmente similar a HttpUnit, existe un punto a su favor, y es que se trata de un proyecto, también de código libre, más activo. Lo que quizás sea debido a una implementación mucho más clara y por tanto mantenible. Esta herramienta está disponible en el sitio Web <http://htmlunit.sourceforge.net/>.

### 11.2.3. JWebUnit

JWebUnit es una herramienta que presenta una naturaleza diferente a las dos anteriores y, aunque para el desarrollador este detalle puede resultar más o menos transparente, en realidad se trata de una herramienta de nivel superior. JWebUnit en sí misma no es más que una capa de software que se apoya en herramientas como HttpUnit o HtmlUnit para realizar la tarea de emulación del servidor Web así como el procesamiento y verificación de los documentos Web. Por este motivo no puede esperarse ninguna ventaja en ambos sentidos respecto a las herramientas anteriores. Sin embargo, presenta dos grandes ventajas respecto a aquellas:

- Presenta una interfaz de programación muy sencilla por lo que es muy fácil de utilizar y posibilita realizar tareas relativamente complejas en muy pocas líneas de código.
- Consta de una serie de métodos assert muy especializados que la hacen especialmente adecuada para la prueba de alto nivel de aplicaciones Web. La ventaja de estos métodos assert

es que en caso de encontrarse fallos, son capaces de proporcionar a JUnit (y por tanto finalmente al desarrollador) mensajes con información muy precisa sobre el fallo detectado.

JWebUnit es una herramienta de código abierto y se encuentra alojada en la página Web <http://jwebunit.sourceforge.net/> desde donde se puede descargar y obtener documentación así como algunos ejemplos de uso. JWebUnit fue inicialmente construida sobre HttpUnit aunque posteriormente se abandonó tal implementación y se implementó alrededor de HtmlUnit<sup>4</sup>.

## 11.3. Prueba de un sitio Web

### 11.3.1. Pruebas de navegación

Una vez se ha finalizado el desarrollo de una aplicación Web, por ejemplo utilizando JSP (Java Server Pages), lo primero que se debe verificar es que dicha aplicación permite al usuario navegar a través de sus páginas de idéntica forma a como se definió en el documento de especificación de requisitos. Si por ejemplo se trata de una aplicación de venta electrónica, existirá una secuencia de acciones que el usuario puede llevar o no a cabo dependiendo de la página de la aplicación en la que se encuentre. En el caso de un usuario no registrado en el sistema, este podrá acceder al catálogo de productos en venta o incluso seleccionar productos y visualizar el contenido de su “cesta de la compra” pero, sin embargo, no podrá acceder a la sección “ver pedidos realizados” o a la sección “ver el estado del pedido” ya que dichas secciones contienen información privada para la cual se requiere registro previo.

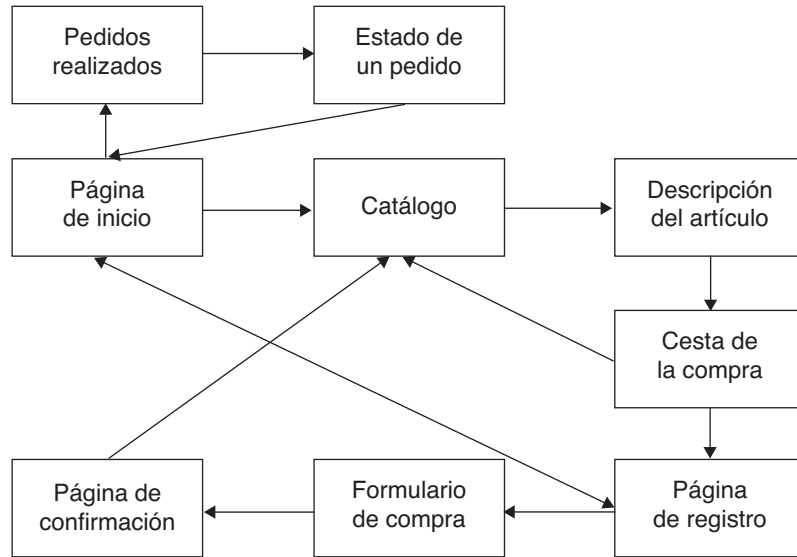
Todas estas restricciones van a depender de cómo se hayan definido los requisitos de la aplicación pero, en general, se pueden representar en forma de diagrama de estados en el que cada estado representa una página Web y las acciones para cambiar de estado son los enlaces y los formularios presentes en los documentos HTML. De esta forma todos los estados que salen de un estado dado representan las páginas Web accesibles mediante enlaces o formularios desde la página Web asociada a dicho estado.

En la Figura 11.2 se muestra el diagrama de estados de una aplicación Web de venta electrónica en el que las flechas representan enlaces y los recuadros son las páginas Web. Nótese que estos diagramas de estados siempre toman la forma de grafos orientados y típicamente con ciclos. En estos grafos los nodos son los páginas Web y las aristas son los formularios o enlaces.

Como puede observarse en la Figura 11.2, se trata de una versión simplificada de lo que sería un grafo de navegación real, en particular, no se ha tenido en cuenta lo siguiente:

- Normalmente desde cualquier página se debe poder volver a la página de inicio o a la página anterior, etc.
- Las aplicaciones Web hacen uso de variables de sesión que almacenan el estado de la navegación así como operaciones que el usuario haya realizado hasta el momento, los valores que tomen de estas variables se traducen, por tanto, en restricciones sobre la navegación. Por ejemplo, típicamente existirá una variable de sesión que indique si el usuario se ha registrado o no en el sistema, dependiendo del valor de esta variable el enlace hacia la página de registro deberá estar disponible o no. Este tipo de consideraciones deben tenerse muy en cuenta a la hora de construir el grafo de navegación.

<sup>4</sup> La utilización de HttpUnit y HtmlUnit se comentará a lo largo de los siguientes apartados de este capítulo.



**Figura 11.2.** Diagrama de navegación de una aplicación de venta electrónica.

### 11.3.1.1. Procedimiento general de prueba

Una vez se ha construido el grafo de navegación, el procedimiento consiste en recorrer el grafo de forma que en cada nodo (página HTML<sup>5</sup>) se verifique que solo determinadas aristas (enlaces y formularios) están disponibles. El procedimiento se detalla a continuación.

1. Se crea una lista con todos los enlaces de la aplicación. Se trata de una lista de pares identificador de enlace (atributo `id` de la etiqueta `<a>` de HTML) y título de la página asociada. Ambos valores deben ser únicos dentro de la aplicación. Asimismo se crea una lista con todos los formularios presentes en páginas HTML de la aplicación. Análogamente al caso anterior se trata de una lista de pares nombre de formulario (atributo `name` de la etiqueta `<form>` de HTML) y título de la página Web a la que conducen.
2. Se crea una lista con los nodos de forma que cada nodo contenga la siguiente información:
  - a. Título de la página Web asociada al nodo.
  - b. Lista con los enlaces que deben estar presentes en dicha página Web así como el título de las páginas de destino para cada uno de esos enlaces.
  - c. Lista de formularios<sup>6</sup> que deben estar presentes en dicha página Web así como la información necesaria para completar los campos y controles de tales formularios de

<sup>5</sup> Aquellos nodos que no sean páginas Web pero que estén accesibles desde páginas Web de la aplicación y por tanto formen parte del grafo (como por ejemplo documentos XML, documentos PDF, etc.) van a constituir hojas del grafo que no necesitan ser procesadas, únicamente verificadas.

<sup>6</sup> Nótese que dado un formulario dependiendo de los datos con los que se rellene, la página Web obtenida al enviar el formulario puede variar. Por tanto, en estos casos es necesario definir varios conjuntos de datos para cada formulario, en particular, tantos conjuntos como páginas Web de destino puedan darse.

forma que puedan ser enviados. Asimismo debe incluirse el título de la página Web que se espera se obtenga al enviar cada uno de los formularios.

- d. Se recorre la lista de nodos de forma que para cada nodo se realiza lo siguiente:
  3. Se comprueba que la página Web asociada al nodo contiene todos los enlaces que debe y no contiene ninguno de los otros enlaces presentes en la lista de enlaces creada en el punto 1. Una vez verificado lo anterior, se navega a cada uno de los enlaces y se comprueba que el título de la página Web a la que conducen es la esperada.
  4. Se comprueba que la página Web asociada al nodo contiene todos los formularios que debe y no contiene ninguno de los otros formularios presentes en la lista de formularios creada en el punto 1. Para cada uno de los formularios presentes, se rellenan los campos y se envía de modo que pueda verificarse que la página a la que conducen es la página esperada.

A pesar de que, *a priori*, poner en práctica este procedimiento puede parecer complejo, es bien sencillo si se dispone de las herramientas adecuadas. En el siguiente apartado se habla de ello.

### 11.3.1.2. Utilización de JWebUnit y JtestCase para realizar las pruebas de navegación

La idea es definir un método de pruebas de navegación y utilizar una de las herramientas anteriormente descritas para realizar la navegación propiamente dicha, es decir, simular el navegador Web de forma que sea posible seguir enlaces entre documentos y ejecutar formularios de forma totalmente automatizada. La herramienta más adecuada en este caso es JWebUnit. Esto es debido a que no va a ser necesario hacer un procesado a bajo nivel de los documentos Web (principal característica de HttpUnit y HtmlUnit) y, sin embargo, sí que resulta muy ventajoso utilizar una API sencilla como la de JWebUnit.

El otro aspecto importante a considerar es la forma en la que se va a representar la información asociada al grafo de navegación, es decir, aquella construida en los puntos 1 y 2 del procedimiento visto en el apartado anterior. La primera idea que surge consiste en definir e inicializar las correspondientes estructuras de datos al comienzo del método de prueba. No obstante esta solución presenta un gran inconveniente y es que en caso de que se desee modificar el flujo de navegación, por ejemplo añadiendo una página más a la aplicación entre otras páginas que ya existan, será necesario modificar el código de las pruebas de forma que se adapte al nuevo diagrama de estados obtenido. Esto dificulta enormemente la mantenibilidad del código de pruebas ya que las aplicaciones Web suelen ser aplicaciones de naturaleza muy dinámica por lo que los cambios en el flujo de navegación y por tanto en el grafo de navegación suceden constantemente durante la fase de mantenimiento. Sin embargo, es posible solventar este problema extrayendo la información asociada al grafo de navegación del método de pruebas. Es decir, por un lado se creará un método de pruebas encargado de verificar un grafo de navegación genérico siguiendo el procedimiento general y por otro lado se utilizara un formato genérico de representación de grafos de navegación para representar el grafo de la aplicación a probar. Las ventajas que se obtienen son dos:

- El flujo de navegación de la aplicación Web puede ser modificado al mismo tiempo que el grafo de navegación, por lo que no es necesario modificar el código fuente de las pruebas.
- El método de verificación de grafos de navegación creado puede ser inmediatamente reutilizado para verificar grafos de navegación asociados a otras aplicaciones.

Está claro que la ganancia en mantenibilidad es sustancial, la cuestión es ¿cómo realizar esta separación? En el Capítulo 8 se presentó la herramienta JTestCase como un medio de mejorar la mantenibilidad del código de pruebas<sup>7</sup>. Esta herramienta básicamente permite separar la definición de los casos de prueba del código de las pruebas. La separación se realiza mediante documentos XML dentro de los cuales se definen los casos de prueba utilizando una sintaxis especial. Posteriormente es posible cargar esta información para ejecutar los casos de prueba dentro de un método de ejecución genérico.

Visto esto, parece que una forma muy natural de representar los grafos de navegación es en forma de documentos XML (al fin y al cabo cada uno de los nodos del grafo de navegación se puede considerar como un caso de prueba) mediante la sintaxis que proporciona JTestCase. Por otro lado, se ha de crear un método que sea capaz de leer uno de estos documentos (mediante la API de JTestCase) y realizar una verificación del grafo de navegación por medio de la herramienta JWebUnit.

A continuación, se lista, a modo de ejemplo el código fuente de una clase genérica de verificación de la navegación. Se trata de la clase *NavegacionTest*, que ha sido definida como parte del código de pruebas funcionales del sistema software del Apéndice B (paquete `pruebasSistemaSoftware.funcionales`). Toda la clase gira en torno al método `pruebaNavegacion` encargado de verificar un grafo de navegación genérico.

`pruebas sistema software/src/pruebasSistemaSoftware/funcionales/NavegacionTest.java`

```
package pruebasSistemaSoftware.funcionales;

//Java
import java.io.*;
import java.lang.*;
import java.util.*;

//JUnit
import org.junit.Test;
import org.junit.BeforeClass;
import static org.junit.Assert.*;
import org.junit.internal.runners.TestClassRunner;
import org.junit.runner.RunWith;

//JWebUnit
import net.sourceforge.jwebunit.junit.WebTestCase;
import net.sourceforge.jwebunit.junit.WebTester;
import net.sourceforge.jwebunit.util.TestingEngineRegistry;

//JTestCase
import org.jtestcase.JTestCase;
import org.jtestcase.JTestCaseException;
import org.jtestcase.TestCaseInstance;

@RunWith(TestClassRunner.class) public class NavegacionTest extends
    WebTestCase {
```

<sup>7</sup> Se recomienda revisar los contenidos de este capítulo para una mejor comprensión de los ejemplos que vienen a continuación.

```

/**
 * Objeto para almacenar los casos de prueba
 */
private static JTestCase m_jtestcase = null;

/**
 * Constructor de la clase
 */
public NavegacionTest() {

    super();
    //Instanciacion del objeto tester
    tester = new WebTester();
    //Seleccion del plugin a utilizar por JWebUnit
    setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
    //Se indica la url base de la aplicacion a probar
    getTestContext().setBaseUrl("http://localhost:1100/servidorEstadoTrafico");
}

/**
 * Carga los casos de prueba al inicio de la prueba de la clase
 */
@BeforeClass static public void cargarCasosPrueba() {

    try {
        // Carga de los datos de los casos de prueba desde fichero
        String dataFile = "./testCases/NavegacionTest.xml";
        m_jtestcase = new JTestCase(dataFile, "Navegacion");

    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * Prueba de navegacion
 */
@Test public void pruebaNavegacion() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases = m_jtestcase.getTestCasesInstancesInMethod("pruebaNavegacion");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {

```

```

// obtencion de los datos asociados al caso de prueba
TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);

try {

    HashMap params = testCase.getTestCaseParams();
    String strURL = (String)params.get("url");
    Vector vEnlaces = (Vector)params.get("vEnlaces");
    Vector vFormularios = (Vector)params.get("vFormularios");

    // comprobacion de enlaces
    for(Enumeration e = vEnlaces.elements() ; e.hasMoreElements() ; ) {
        // se carga la pagina
        beginAt(strURL);
        // se verifica el enlace
        Enlace enlace = (Enlace)e.nextElement();
        assertLinkPresent(enlace.m_strId);
        clickLink(enlace.m_strId);
        if (enlace.m_strTitulo.compareTo("sin título") != 0)
            assertEquals(enlace.m_strTitulo);
    }
    // comprobacion de formularios
    for(Enumeration e = vFormularios.elements() ; e.hasMoreElements() ; ) {
        // se carga la pagina
        beginAt(strURL);
        // se verifica el formulario
        Formulario formulario = (Formulario)e.nextElement();
        assertFormPresent(formulario.m_strNombre);
        Iterator it = formulario.m_mapCampos.keySet().iterator();
        while(it.hasNext()) {
            //System.out.println("This is a form:" + it.next());
            String strKey = (String)it.next();
            assertFormElementPresent(strKey);
            setTextField(strKey, (String)formulario.m_mapCampos.get(strKey));
        }
        assertSubmitButtonPresent(formulario.m_strBotonEnvio);
        submit(formulario.m_strBotonEnvio);
        assertEquals(formulario.m_strTitulo);
    }

} catch (JTestCaseException e) {

    // Ha ocurrido un error en el procesamiento de los datos del caso de prueba
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

}

}

}

```

En primer lugar, señalar que la clase `NavegacionTest` hereda de la clase `WebTestCase`. Esta última pertenece a `JWebUnit` y representa la interfaz con la herramienta, además esta clase hereda de `JTestCase` y sobrescribe los métodos `setUp` y `tearDown` para realizar inicialización y liberación de recursos respectivamente. En realidad esta clase no es más que un encapsu-



sulado de la clase `WebTester` y, de hecho, una forma alternativa de utilizar `JWebUnit` sin necesidad de heredar de `WebTestCase` es instanciando un objeto de la clase `WebTester` y usarlo directamente. Lógicamente es más cómodo heredar de `WebTestCase` y así tener disponibles los métodos desde la propia clase de pruebas, sin embargo, esto tiene un inconveniente y es que, a fecha de escritura de este libro, la herramienta `JWebUnit` no está preparada para trabajar con las versiones 4.x de `JUnit` y, por tanto, el objeto `WebTester` (objeto colaborador de la clase `WebTestCase`) nunca es inicializado ya que su inicialización se realiza en el método `setUp` de `WebTestCase` y este nunca es invocado por la herramienta `JUnit` en las versiones 4.x<sup>8</sup>. Sin embargo, como puede verse en el ejemplo, este objeto llamado `tester` puede inicializarse en el constructor de `NavegacionTest` y de esta forma se soluciona el problema.

Dentro del constructor de `NavegacionTest` se elige el plugin<sup>9</sup> con el que se desea que trabaje `JWebUnit` (a día de hoy solamente está disponible el plugin de `HtmlUnit`) y se selecciona la que va a ser la url base de la aplicación<sup>10</sup>. El último paso de inicialización es la carga de los casos de prueba desde el método `cargarCasosPrueba` que, etiquetado con `@BeforeClass`, se ejecuta una única vez antes de la ejecución de los métodos de prueba.

El método de prueba `pruebaNavegacion` es el responsable de la prueba de navegación. Inicialmente carga los casos de prueba, que son la url del documento (nodo del grafo de navegación a verificar), un `Vector` de objetos `Enlace` con los enlaces que deben aparecer en el documento y un `Vector` de objetos `Formulario`<sup>11</sup> con los formularios que se espera aparezcan en el documento. Para comprobar que un enlace existe dentro del documento, se carga el documento mediante el método `beginAt` (cuyo parámetro es una ruta relativa respecto a la url base de la aplicación fijada anteriormente con el método `setBaseUrl`) y se utiliza el método `assertLinkPresent` que comunica un fallo a `JUnit` en caso de que el enlace no exista. Posteriormente se sigue el enlace mediante el método `clickLink`, que se comunica con el servidor `Web` para cargar un nuevo documento `Web` correspondiente al enlace (igual que hace un navegador `Web`). A continuación con el método `assertTitleEquals` se comprueba que el título de la nueva página cargada se corresponde con el título de la página esperada. Se ha de tener en cuenta que es necesario ejecutar el método `beginAt` dentro del bucle de comprobación de enlaces puesto que el método `clickLink` cambia la página en cada iteración. Para verificar la presencia de formularios, el procedimiento es muy similar, se carga la página con `beginAt` y se itera el objeto `HashMap` para obtener los campos del formulario y realizar comprobaciones sobre los mismos. En cada iteración se extrae un campo y se comprueba que existe en el formulario mediante el método `assertFormElement`, asimismo se asigna valor al campo con el método `setTextField` para así poder enviar el formulario posteriormente. Una vez todos los campos han sido comprobados y rellenados con la información adecuada, se comprueba mediante el método `assertSubmitButton` que el botón de envío existe y se envía el formulario mediante el método `submit`. El último paso es comprobar que la página devuelta por el formulario tiene el mismo título que la página esperada.

A continuación, se lista el documento XML que contiene la información del grafo de navegación. Como se vio anteriormente el método `pruebaNavegacion` es un método genérico,

<sup>8</sup> Recuérdese que en `JUnit` 4.x el método `setUp` se ha substituido por el uso de la anotación `@Before`.

<sup>9</sup> Recuérdese que `JWebUnit` no tiene funcionalidad de emulación de navegadores `Web`, sino que la toma de otras herramientas.

<sup>10</sup> Por motivos de generalidad, esta url debería cargarse desde el documento XML de casos de prueba que contiene la información del grafo de navegación, sin embargo se ha realizado de esta forma para no oscurecer el ejemplo.

<sup>11</sup> El código completo de la clase `Enlace` así como de la clase `Formulario` se encuentra disponible en el directorio pruebas Sistema software/src/pruebasSistemaSoftware/funcionales.

y este documento no es más que un ejemplo de grafo de navegación que puede ser verificado desde dicho método.

```
pruebas sistema software/testCases/NavegacionTest.xml
```

```
<?xml version ="1.0" encoding = "UTF-8"?>
<tests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="http://jtestcase.sourceforge.net/dtd/jtestcase2.
xsd">
  <class name="Navegacion">
    <method name="pruebaNavegacion">
      <!-- caso de prueba de la pagina Informacion -->
      <test-case name="Informacion">
        <params>
          <param name="url" type="java.lang.String">/?peticion=informacion
        </param>
          <param name="vEnlaces" type="" use-jice="yes">
            <jice>
              <vectorEnlaces xmlns="http://www.jicengine.org/jic/2.0"
                class="java.util.Vector">
                <entry action="parent.addElement(enlace)">
                  <enlace class="pruebasSistemaSoftware.funcionales.
                    Enlace"
                    args="strId,strTítulo">
                      <strId>tramosMasTrafico</strId>
                      <strTítulo>sin título</strTítulo>
                    </enlace>
                </entry>
                <entry action="parent.addElement(enlace)">
                  <enlace class="pruebasSistemaSoftware.funcionales.
                    Enlace"
                    args="strId,strTítulo">
                      <strId>tramosMenosTrafico</strId>
                      <strTítulo>sin título</strTítulo>
                    </enlace>
                </entry>
                <entry action="parent.addElement(enlace)">
                  <enlace class="pruebasSistemaSoftware.funcionales.
                    Enlace"
                    args="strId,strTítulo">
                      <strId>carrilesCortados</strId>
                      <strTítulo>sin título</strTítulo>
                    </enlace>
                </entry>
                <entry action="parent.addElement(enlace)">
                  <enlace class="pruebasSistemaSoftware.funcionales.
                    Enlace"
                    args="strId,strTítulo">
                      <strId>obras</strId>
                      <strTítulo>sin título</strTítulo>
                    </enlace>
                </entry>
              </vectorEnlaces>
            </jice>
          </param>
        </params>
      </test-case>
    </method>
  </class>
</tests>
```

```

        <enlace class="pruebasSistemaSoftware.funcionales.
            Enlace"
            args="strId,strTítulo">
            <strId>accidentes</strId>
            <strTítulo>sin título</strTítulo>
        </enlace>
    </entry>
    <entry action="parent.addElement(enlace)">
        <enlace class="pruebasSistemaSoftware.funcionales.
            Enlace"
            args="strId,strTítulo">
            <strId>carretera</strId>
            <strTítulo>sin título</strTítulo>
        </enlace>
    </entry>
    <entry action="parent.addElement(enlace)">
        <enlace class="pruebasSistemaSoftware.funcionales.
            Enlace"
            args="strId,strTítulo">
            <strId>clima</strId>
            <strTítulo>sin título</strTítulo>
        </enlace>
    </entry>
    <entry action="parent.addElement(enlace)">
        <enlace class="pruebasSistemaSoftware.funcionales.
            Enlace"
            args="strId,strTítulo">
            <strId>detenerServidor</strId>
            <strTítulo>sin título</strTítulo>
        </enlace>
    </entry>
    <entry action="parent.addElement(enlace)">
        <enlace class="pruebasSistemaSoftware.funcionales.
            Enlace"
            args="strId,strTítulo">
            <strId>informacion</strId>
            <strTítulo>sin título</strTítulo>
        </enlace>
    </entry>
</vectorEnlaces>
</jice>
</param>
<param name="vFormularios" type="" use-jice="yes">
    <jice>
        <vectorFormularios xmlns="http://www.jicengine.org/jic/2.0"
            class="java.util.Vector"/>
    </jice>
</param>
</params>
</test-case>
<!-- caso de prueba de la pagina Formulario de autenticacion -->
<test-case name="Formulario de autenticacion">
    <params>
        <param name="url" type="java.lang.String">/?peticion=detener
            Servidor
        </param>

```

```

<param name="vEnlaces" type="" use-jice="yes">
  <jice>
    <vectorEnlaces xmlns="http://www.jicengine.org/jic/2.0"
      class="java.util.Vector"/>
  </jice>
</param>
<param name="vFormularios" type="" use-jice="yes">
  <jice>
    <vectorFormularios xmlns="http://www.jicengine.org/jic/2.0"
      class="java.util.Vector">
      <entry action="parent.addElement(formulario)">
        <formulario
          class="pruebasSistemaSoftware.funcionales.
            Formulario"
          args="strNombre,strTítulo,mapCampos,strBotonEnvio">
          <strNombre>autenticacion</strNombre>
          <strTítulo>Formulario de autenticacion</strTítulo>
          <mapCampos xmlns="http://www.jicengine.org/jic/2.0"
            class="java.util.HashMap">
            <entry action="parent.put(strCampo,strValor)">
              <strCampo>usuario</strCampo>
              <strValor>administrador</strValor>
            </entry>
            <entry action="parent.put(strCampo,strValor)">
              <strCampo>password</strCampo>
              <strValor>123456</strValor>
            </entry>
          </mapCampos>
          <strBotonEnvio>enviar datos</strBotonEnvio>
        </formulario>
      </entry>
    </vectorFormularios>
  </jice>
</param>
</params>
</test-case>
</method>
</class>
</tests>

```

La estructura de este documento es más sencilla de lo que parece. En primer lugar, cabe indicar que el documento contiene únicamente información sobre dos nodos (dos casos de prueba por tanto) del grafo de navegación, que es el correspondiente a la aplicación Web del Apéndice B. El primer caso de prueba está asociado al documento de información del servidor y el segundo es el relativo al documento con el formulario de autenticación que aparece al ejecutarse una petición de detención del servidor<sup>12</sup>. No se han incluido más nodos del grafo para no complicar el ejemplo. Para cada documento se ha incluido información<sup>13</sup> sobre cómo construir un objeto de la clase `Vector` con todos los enlaces que se espera que dicho documento contenga. Se trata de un `Vector` de objetos `Enlace` que contienen el identificador del enlace así

<sup>12</sup> La estructura completa de estos documentos así como información sobre el flujo de navegación de esta aplicación se encuentra en el Apéndice B.

<sup>13</sup> Se ha utilizado `JTestCase` combinado con `Jice` ya que las clases `Elemento` y `Enlace`, al igual que cualquier otra clase definida por el desarrollador, no están directamente soportadas por `JTestCase`.

como el título de la página al que conducen. En el caso de enlaces que no conducen a páginas HTML, dado que estas páginas no podrán contener una etiqueta HTML `<title>`, el valor del título será “sin título”. De igual forma, para cada documento se ha incluido información sobre cómo construir un objeto de la clase `Vector` con todos los formularios que se espera estén disponibles en dicho documento. Se trata de un `Vector` de objetos `Formulario`, que contienen el nombre del formulario, un objeto de la clase `HashMap` con pares “nombre de campo en el formulario” y “valor de dicho campo”, el nombre del botón que se ha de utilizar para enviar el formulario así como el título de la página a la que se espera conduzca el formulario una vez que ha sido rellenado con la información contenida en el objeto `HashMap`.

Al echar la vista atrás sobre el procedimiento general y comparar con la forma de ponerlo en práctica mediante el método `pruebaNavegacion`, se observa que si bien se ha comprobado que todos los enlaces que debían estar en los documentos, efectivamente estaban allí y además conducían a las páginas adecuadas, no se ha comprobado que en esos documentos no existen enlaces a documentos restringidos de la aplicación (es decir, no accesibles según el grafo de navegación). Aunque en el ejemplo no se ha realizado dicha verificación por motivos de espacio, a continuación se indican los pasos necesarios para realizarla:

- Se ha de definir en el documento XML de los casos de prueba un `Vector` de objetos `Enlace` y un `Vector` de objetos `Formulario` que contengan todos los enlaces y formularios que aparecen en la aplicación Web. Puesto que esta información es global a todos los casos de prueba, las definiciones han de hacerse en forma de parámetros globales a la clase en lugar de parámetros asociados a casos de prueba<sup>14</sup>.
- Se ha de cargar la información anterior dentro del método `PruebaNavegacion` y se han de utilizar los métodos `assertLinkNotPresent` y `assertFormNotPresent` para verificar que determinados enlaces y formularios, respectivamente, no pertenecen al documento.

### 11.3.2. Prueba de enlaces rotos

Uno de los fenómenos más frustrantes para un usuario de una aplicación Web es encontrarse con un enlace roto, es decir, un enlace a una página o recurso Web que ha dejado de estar disponible. En este tipo de situaciones el usuario se ve obligado a esperar unos segundos frente a la pantalla del ordenador antes de que aparezca en el navegador el correspondiente mensaje de error “página no encontrada”. Una página con enlaces rotos transmite una sensación de dejadez y abandono, como si la página no hubiera sido actualizada o revisada recientemente. Detectar enlaces rotos tiene por tanto un claro objetivo, que es mejorar la usabilidad de la aplicación Web a través de la mejora de la experiencia del usuario. A continuación, se describe el procedimiento general para la detección de enlaces rotos, que no es ni más ni menos que una búsqueda en anchura de este tipo de enlaces en el grafo de navegación.

1. Se crea una lista vacía que va a ser utilizada para almacenar los enlaces de la aplicación que ya han sido comprobados. Esta lista es necesaria ya que prácticamente todo grafo de navegación es un grafo con ciclos y, por tanto, es necesario establecer una condición de parada para que la búsqueda acabe.

---

<sup>14</sup> En el sitio Web de JTestCase (<http://jtestcase.sourceforge.net/>) se explica con ejemplos la forma de definir este tipo de parámetros globales.

2. Se crea una lista vacía que va a ser utilizada para almacenar los enlaces rotos detectados.
3. Se crea una lista que va a contener los enlaces pendientes por explorar y se introduce la URL de la página de inicio de la aplicación en dicha lista.
4. Mientras queden elementos en la lista de enlaces pendientes, se extrae el primer enlace de la lista, y se llevan a cabo los siguientes pasos:
  - a. Si el enlace está incluido dentro de la lista de enlaces explorados se vuelve al inicio del punto 4.
  - b. Se comprueba si la URL del enlace va más allá de los dominios de la aplicación a probar (enlaces a otros sitios Web), si es así, se vuelve al inicio del punto 4. Obviamente no interesa detectar enlaces rotos en otros sitios Web.
  - c. Se carga la página a la que conduce dicho enlace. Si al cargar la página se produce un error, se introduce el enlace en la lista de enlaces incorrectos y se vuelve al punto 4.
  - d. Se procesa la página y se obtiene la lista de enlaces que contiene. Se añaden dichos enlaces al final de la lista de enlaces pendientes por explorar.
  - e. Se añade el enlace a la lista de enlaces explorados.
5. Una vez que la lista de enlaces por explorar esté vacía, todos los enlaces rotos detectados están contenidos en la lista de enlaces rotos. Se debe probar a mano que tales enlaces están realmente rotos y solucionar el problema. La revisión manual es con frecuencia necesaria ya que, en ocasiones, enlaces correctos pueden resultar inaccesibles en un momento dado debido a las condiciones de tráfico de la red.

Tras una breve reflexión se llega a la conclusión de que la prueba de enlaces rotos es realmente que una prueba de navegación relajada en la que no existe un grafo de navegación explícito y, por tanto, no existen las restricciones asociadas al mismo. De esta forma si se realizan pruebas de navegación se están realizando al mismo tiempo pruebas de enlaces rotos. La ventaja de las pruebas de enlaces rotos es que no están guiadas por datos, es decir, no necesitan de la definición de casos de prueba. En aplicaciones Web muy dinámicas en las que continuamente se esté cambiando el flujo de navegación y, por tanto, sea demasiado costoso mantener un grado de navegación actualizado, las pruebas de enlaces rotos permiten mantener la usabilidad del sitio Web en unos niveles saludables.

Aun no se ha dicho una sola palabra acerca de la herramienta a utilizar para poner en práctica este procedimiento, lógicamente se necesita un emulador de navegador Web para la carga y procesamiento de los documentos HTML, por lo que JWebUnit, HtmlUnit o HttpUnit son todas ellas, *a priori*, candidatas. Sin embargo, después de echar un vistazo a la API de estas herramientas se observa que JWebUnit no permite obtener la lista de enlaces presentes en un documento HTML. Por este motivo se recomienda utilizar cualquiera de las otras dos herramientas, cuya utilización se va a tratar en detalle en los siguientes apartados.

### 11.3.3. Pruebas de estructura y contenido

Una vez que se ha verificado que la navegación proporcionada por la aplicación Web es la adecuada, se ha de verificar que los contenidos que se muestran en cada una de las páginas gene-

radas por ella se adecúan a los requisitos especificados. En los siguientes apartados se hace va a exponer por medio de ejemplos la forma de verificar los tipos de documentos Web que pueden encontrarse con mayor frecuencia.

### 11.3.3.1. Prueba de páginas Web dinámicas

Este tipo de páginas representan un gran porcentaje de las páginas que un usuario puede encontrarse al navegar por Internet. Se trata de páginas construidas por el servidor de forma dinámica, es decir, utilizando información que varía en cada acceso. Esta información se divide principalmente en dos grupos: información del usuario (típicamente contenida en las cookies) e información obtenida de la base de datos de aplicación o bien de algún otro recurso Web con el que trabaje la aplicación. Un claro ejemplo de estas páginas son aquellas generadas por el sistema software del Apéndice B, en el que se generan documentos XML cuyo contenido depende de la información contenida en el archivo de registros o en la base de datos de la aplicación. Para la construcción de páginas dinámicas es necesario que exista código de script<sup>15</sup> (código ejecutable) en el servidor. El objetivo de este apartado es, justamente, mostrar la forma de realizar pruebas funcionales sobre dicho código examinando la estructura y contenido de las páginas que genera. En particular, interesa únicamente verificar el contenido y la estructura de las páginas generadas, y nunca detalles relacionados con la presentación. En este apartado la discusión va a estar centrada en páginas HTML generadas dinámicamente, más adelante se puede ver el procedimiento análogo para páginas XML dinámicas en el apartado “Pruebas de documentos XML generados dinámicamente”.

Es bien sabido que el lenguaje HTML es un lenguaje de marcado, basado en etiquetas y atributos, que se utiliza para crear documentos con información basada en texto (aunque también proporciona mecanismos para referenciar elementos multimedia). El lenguaje HTML proporciona mecanismos para definir la estructura del documento así como la presentación del mismo y la localización de otros elementos en la Web. Se trata por tanto de un lenguaje que aúna información de muy diverso tipo. La primera cuestión es: ¿qué aspectos interesa probar en una página HTML construida dinámicamente y en general en cualquier página HTML? A continuación se listan algunos de ellos:

- Que el título de la página sea el adecuado.
- Que la página contenga ciertos enlaces en ciertas localizaciones. (Esto está relacionado con la, anteriormente comentada, prueba de navegación).
- Que la página contenga cierto formulario con cierto nombre y con una determinada serie de campos.
- Que ciertas tablas hayan sido definidas y que contengan determinada información.

Alternativamente surge la cuestión complementaria: ¿qué aspectos no interesa probar? La respuesta es bien sencilla, todos los aspectos relacionados con la presentación, entre los cuales se encuentran los siguientes:

- Fuentes y tipos de letra escogidos.
- Colores de tablas, celdas y fondos.

---

<sup>15</sup> Existen multitud de tecnologías específicamente creadas para el desarrollo de código que se ejecuta dentro de un servidor Web. Los ejemplos más conocidos son: PHP, ASP y las más populares del mundo Java: JSP y Servlets.

- Información relativa a contenidos sujetos a cambios. Como se verá más adelante, este tipo de información, debido a su naturaleza no permite ser probada de una forma sencilla.

Una vez definidos los aspectos hacia los que se va a orientar la prueba es necesario señalar que, debido a la propia naturaleza dinámica de estas páginas, existen elementos que pueden ser verificados y elementos que no. Por ejemplo, si una página HTML se genera mostrando una tabla con información horaria cuyo contenido depende obviamente de la hora en la que se ha creado, es imposible definir un caso de prueba para verificar que dicha información es correcta. Sin embargo, sí que es posible, y esto es válido como norma general, verificar que el nombre de las filas y de las columnas de dicha tabla es el correcto, ya que dichos nombres son de naturaleza invariable. Debe tenerse presente que, por ejemplo, si se tiene una página HTML que contiene una tabla con la información obtenida al ejecutar una sentencia SQL sobre la base de datos de aplicación, el contenido de dicha tabla sí que puede ser verificado mediante un caso de prueba definido para tal efecto. Esto es así dado que es posible conocer *a priori* el estado de la base de datos y por tanto la información que se espera se obtenga como resultado de ejecutar la sentencia SQL <sup>16</sup>.

Para realizar este tipo de pruebas en principio se puede utilizar cualquiera de las herramientas mencionadas al principio de este capítulo, es decir: JWebUnit, HttpUnit o HtmlUnit. Si embargo, como se va a ver en los ejemplos, cada herramienta presenta ventajas e inconvenientes según el contexto. Mientras que JWebUnit es la más cómoda de utilizar, HtmlUnit y HttpUnit, gracias a su flexibilidad, permiten probar aspectos de la página a mayor nivel de detalle, lo que con JWebUnit resulta imposible. A continuación se va a ver la prueba de una página dinámica generada por el sistema software del Apéndice B. En la Figura 11.3 se puede observar un ejemplo de cómo debería ser esta página según la especificación de requisitos software.

Se trata de la página de información del servidor, que ha de mostrar el formato correcto de las peticiones HTTP que se pueden realizar sobre el mismo así como la fecha y hora de arranque del sistema y el número de peticiones atendidas desde ese momento. Como puede verse esta página contiene información que sólo puede ser generada dinámicamente:

- La dirección de red así como el puerto (`localhost:1100` en la Figura 11.3) dependen de los datos contenidos en el archivo de configuración de la aplicación <sup>17</sup>.
- La fecha y hora de arranque del sistema.
- El número de peticiones atendidas por el servidor hasta ese momento.

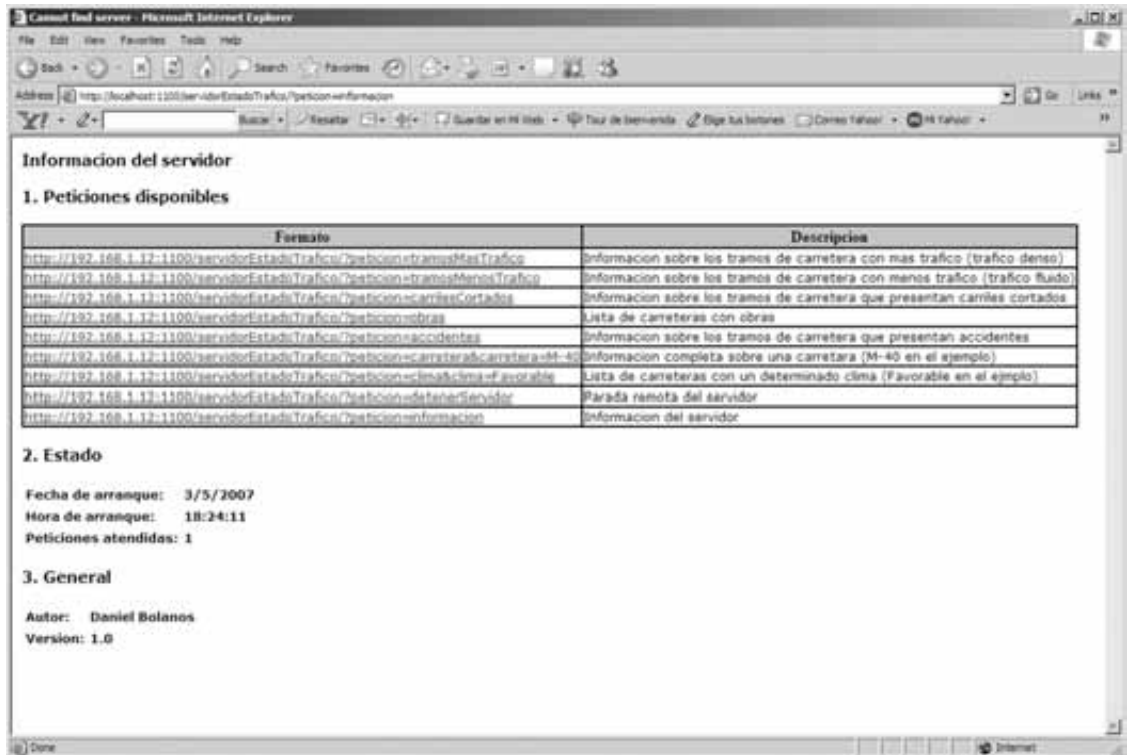
Para la prueba de esta página se ha definido un método cuyo código fuente se lista a continuación. Se trata del método `peticionPaginaInformacion` definido en la clase `PeticionesTest` incluida dentro del paquete de pruebas funcionales `pruebasSistemaSoftware.funcionales`.

---

<sup>16</sup> En el Capítulo 9 se describe con todo detalle el proceso de prueba del código que accede a una base de datos. Si bien en ese capítulo el enfoque es de pruebas unitarias, la herramienta utilizada (DBUnit) es de gran utilidad para definir el estado de una base de datos y por tanto fijar expectativas sobre el resultado obtenido al ejecutar sentencias SQL.

<sup>17</sup> Véase Apéndice B para más detalles.





**Figura 11.3.** Página de información del servidor

```
pruebas sistema software/src/pruebasSistemaSoftware/funcionales/PeticionesTest.java
```

```
/**
 * Prueba de la pagina de informacion
 */
@Test public void petitionPaginaInformacion() {

    // se realiza la peticion
    beginAt("/?peticon=informacion");
    // título de la pagina
    assertEquals("Informacion");

    // tabla "peticiones"
    assertTablePresent("peticiones");
    assertTextInTable("peticiones", "Formato");
    assertTextInTable("peticiones", "Descripcion");

    assertLinkPresent("tramosMasTráfico");
    assertLinkPresent("tramosMenosTráfico");
    assertLinkPresent("carrilesCortados");
    assertLinkPresent("obras");
    assertLinkPresent("accidentes");
    assertLinkPresent("carretera");
```

```

assertLinkPresent("clima");
assertLinkPresent("informacion");
assertLinkPresent("detenerServidor");

assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=tramosMasTrafico");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=tramosMenosTrafico");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=carrilesCortados");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=obras");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=accidentes");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=carretera&carretera=M-40");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=clima&clima=Favorable");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=informacion");
assertLinkPresentWithText("http://192.168.1.12:1100/servidorEstadoTrafico/
    ?peticion=detenerServidor");
assertTablePresent("estado");

// tabla "estado"
assertTablePresent("estado");
// comprobacion por contenido
assertTextInTable("estado", "Fecha de arranque:");
assertTextInTable("estado", "Hora de arranque:");
assertTextInTable("estado", "Peticiones atendidas:");
// comprobacion por contenido y estructura
Table tabla = getTable("estado");
assertEquals("Numero de filas incorrecto",3,tabla.getRowCount());
ArrayList filas = tabla.getRows();
ListIterator listaFilas = filas.listIterator();
ArrayList celdas[] = new ArrayList[filas.size()];
for(int i=0 ; listaFilas.hasNext() ; ++i) {
    Row fila = (Row)listaFilas.next();
    celdas[i] = fila.getCells();
}
assertEquals(2,celdas[0].size());
assertEquals(2,celdas[1].size());
assertEquals(2,celdas[2].size());
assertTrue(((Cell)(celdas[0].get(0))).equals("Fecha de arranque:"));
assertTrue(((Cell)(celdas[1].get(0))).equals("Hora de arranque:"));
assertTrue(((Cell)(celdas[2].get(0))).equals("Peticiones atendidas:"));

//tabla "general"
assertTablePresent("general");
assertTableEquals("general",
    new String[][] {{"Autor:", "Daniel Bolanos"},
                    {"Version:", "1.0"}});

//Prueba de navegacion
clickLink("informacion");
assertTitleEquals("Informacion");

```

```

assertTablePresent("peticiones");
assertTablePresent("estado");
}

```

En primer lugar, se utiliza el método `beginAt` para cargar la página de información. La primera comprobación que se realiza es sobre el título de la página, para ello se utiliza el método `assertTitleEquals`. El título de una página (`<title>`) es normalmente utilizado con identificador único entre páginas, por lo que debe ser generado de forma única. Posteriormente, se realiza la verificación de cada una de las tablas<sup>18</sup>. La primera tabla que debe aparecer es la tabla “peticiones”, lo que se comprueba mediante el método `assertTablePresent`<sup>19</sup> que comunica un fallo a JUnit en caso de que la tabla no exista. Adicionalmente, se comprueba que las palabras “formato” y descripción” (nombres de las columnas de la tabla `peticiones`) están contenidas dentro de dicha tabla. El siguiente paso es comprobar que los enlaces de las peticiones al servidor se encuentran dentro del documento. Idealmente, se debería comprobar no solo que están dentro del documento sino que están dentro de la tabla `peticiones`, desgraciadamente JWebUnit no permite hilar tan fino. Esa comprobación se puede realizar de varias formas, en el ejemplo se ve como se han utilizado complementariamente los métodos `assertLinkPresent` y `assertLinkPresentWithText` que comprueban si un determinado enlace se encuentra en la página HTML dado el identificador del enlace (valor del atributo `id` de la etiqueta `<a>`) o el texto asociado al enlace respectivamente. La siguiente tabla a verificar es la tabla “estado” en la que debe aparecer la fecha y hora de arranque y el número de peticiones tal y como aparece en la Figura 11.3 (se trata de una tabla transparente de tres filas y dos columnas). La verificación de esta tabla presenta una peculiaridad y es que mientras que los valores de la primera columna son fácilmente verificables ya que son de naturaleza estática (van a ser siempre los mismos), los valores de la segunda columna son de naturaleza dinámica y por lo tanto no pueden ser verificados. Se trata de hacer una verificación selectiva por celdas, para lo que se ha hecho uso de las clases `Table`, `Row` y `Cell` que representan una tabla, una fila y una columna respectivamente. Claramente, se trata de una verificación muy costosa en términos de líneas de código escritas y que es fruto de la escasa flexibilidad de JWebUnit. La última tabla a verificar es la tabla “general” que es muy sencilla ya que su contenido es completamente estático. El último paso es una simple verificación de navegación.

Como se ha visto, JWebUnit presenta una serie de problemas a la hora de verificar ciertos elementos de una página HTML con cierto nivel de precisión y detalle. A continuación, se lista el código fuente del método `peticionPaginaInformacion`, pero esta vez haciendo uso de la herramienta `HttpUnit`.

```
pruebas sistema software/src/pruebasSistemaSoftware/funcionales/PeticionesTest.java
```

```

/**
 * Prueba de la pagina de informacion
 */
@Test public void peticionPaginaInformacionHttpUnit() {

    try {

```

<sup>18</sup> Para poder realizar verificaciones sobre tablas es necesario que estas puedan ser identificadas de forma única dentro del documento HTML en el que se han definido. Para ello se ha de utilizar el atributo `id` de la etiqueta `<table>`.

<sup>19</sup> La presencia de este tipo de métodos `assert` altamente especializados en la clase `WebTestCase` es una de las causas de que hacen de JWebUnit una herramienta extremadamente cómoda de utilizar.

```

WebConversation wc = new WebConversation();
WebRequest petition = new GetMethodWebRequest("http://192.168.1.12:1100/
    servidorEstadoTrafico/?peticion=informacion");
WebResponse respuesta = wc.getResponse(petition);

//título
assertTrue(respuesta.isHTML());
assertEquals("Informacion",respuesta.getTitle());

//tabla "peticiones"
WebTable tablaPeticiones = respuesta.getTableWithID("peticiones");
assertEquals("rows",10,tablaPeticiones.getRowCount());
assertEquals("columns",2,tablaPeticiones.getColumnCount());
WebLink links[] = tablaPeticiones.getTableCell(1,0).getLinks();
assertEquals(1,links.length);
assertEquals("http://192.168.1.12:1100/
    servidorEstadoTrafico/?peticion=tramosMasTrafico",links[0].getText());

//tabla "estado"
WebTable tablaEstado = respuesta.getTableWithID("estado");
assertEquals("Numero incorrecto de filas",3,tablaEstado.getRowCount());
assertEquals("Numero incorrecto de columnas",2,tablaEstado.getColumnCount());
String[][] celdas = tablaEstado.asText();
assertEquals("Fecha de arranque:",celdas[0][0]);
assertEquals("Hora de arranque:",celdas[1][0]);
assertEquals("Peticiones atendidas:",celdas[2][0]);

//Prueba de navegacion
WebLink link = respuesta.getLinkWith("informacion");
link.click();
WebResponse respuesta2 = wc.getCurrentPage();
assertTrue(respuesta2.isHTML());
assertEquals("Informacion",respuesta2.getTitle());

} catch (IOException e) {

    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (org.xml.sax.SAXException e) {

    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

}
}

```

En primer lugar, es posible obtener una lista de los enlaces asociados a una determinada celda de una tabla mediante el método `getLinks` de la clase `TableCell` (en realidad se trata de un método de la clase `BlockElement`<sup>20</sup> de la que hereda `TableCell`). Una vez obtenida la lista de enlaces se pueden realizar verificaciones acerca del numero de enlaces (debe ser uno en

---

<sup>20</sup> La clase `BlockElement` es la clase base de las clases `TableCell` y `BlockElement` que se utilizan para representar párrafos y celdas respectivamente en el documento HTML, mirando la interfaz pública de esta clase es fácil hacerse una idea del potencial de `HttpUnit` para la verificación detallada del contenido de un documento HTML.

el ejemplo) y del texto asociado a dichos enlaces. Lo segundo que llama la atención es la facilidad con la que se ha verificado el contenido de la tabla “estado” (compárese el número de líneas utilizado respecto al ejemplo anterior que hace uso de la herramienta JWebUnit). El método `asText` de la clase `WebTable` (que representa una tabla) devuelve un array bidimensional que contiene el texto de cada una de las celdas de la tabla. Una vez obtenido este array, realizar una verificación selectiva por celdas resulta muy sencillo.

Una vez vista la forma de utilizar ambas herramientas para la prueba de contenidos, es responsabilidad del desarrollador elegir aquella que más se adapte a las particularidades de los documentos generados por la aplicación a probar.

### 11.3.3.2. Prueba de páginas HTML estáticas

Este tipo de páginas son construidas típicamente mediante editores Web y desplegadas sobre una carpeta del servidor. Se trata de páginas que contienen información invariante en el tiempo, como por ejemplo páginas de error. La ventaja que presentan estas páginas es que, debido a su naturaleza estática, no dependen del código de la aplicación situado en el servidor Web por lo que pueden ser probadas al margen de este. A menudo, poner en marcha el servidor Web y configurarlo así como desplegar el software de la aplicación (típicamente Servlets y páginas JSP) y todos los recursos, como bases de datos, de los que este hace uso, requiere de un considerable esfuerzo que en este caso puede ser evitado.

En realidad, durante el proceso de pruebas en sí, lo único que diferencia la prueba de una página dinámica de la prueba de una página estática es la forma de obtener dichas páginas. Mientras que la primera se obtendrá abriendo una conexión HTTP con el servidor Web, la segunda se obtiene simplemente leyendo un archivo de disco. De esta forma la prueba se va a realizar utilizando las herramientas descritas en el apartado anterior y modificándolas de forma que no exista tal conexión HTTP. Esta modificación es necesaria ya que ninguna de tales herramientas ha sido diseñada con este objetivo. A continuación, se muestra un método de prueba encargado de la prueba del documento estático:

```
pruebas sistema software/src/pruebasSistemaSoftware/funcionales/PeticionesTest.java
```

```
/**
 * Prueba de una pagina html estatica cargada desde disco
 */
@Test public void pruebaPaginaEstatica() {

    try {

        //lectura de la pagina HTML desde disco
        File archivo = new File("./testData/paginaEstatica.html");
        FileInputStream entrada = new FileInputStream(archivo);
        BufferedInputStream buffer = new BufferedInputStream(entrada);
        DataInputStream data = new DataInputStream(buffer);

        byte[] datos = new byte[10000];
        int iBytes = data.read(datos);
        String strPaginaHTML = new String(datos, 0, iBytes);

        //obtencion de la pagina
```

```

WebClient webClient = new WebClient();
webClient.setJavaScriptEnabled(true);
com.gargoylesoftware.htmlunit.WebResponse webResponse = new
    StringWebResponse(strPaginaHTML);
HtmlPage page = HTMLParser.parse(webResponse,webClient.getCurrentWindow());

//verificacion de condiciones
assertEquals("Esto es el título!",page.getTitleText());

//liberacion de recursos
entrada.close();
buffer.close();
data.close();

} catch (FileNotFoundException e) {

    fail("Error al ejecutar un caso de prueba");
    e.printStackTrace();

} catch (IOException e) {

    fail("Error al ejecutar un caso de prueba");
    e.printStackTrace();

}
}

```

La gran diferencia respecto a lo visto anteriormente reside en la creación del objeto que representa la página HTML, es decir, la obtención de una instancia de la clase `HtmlPage`. Para ello ahora se utiliza el método estático `parse` perteneciente a la clase `HTMLParser`. Este método recibe como parámetros la ventana (que toma el valor por defecto en el ejemplo) y un objeto que implemente la interfaz `WebResponse`<sup>21</sup>. Este último es obtenido al crear una instancia de la clase `StringWebResponse` que construye una respuesta Web a partir del contenido de un `String` que se supone está en formato HTML y que no es otro que la página HTML cargada desde el disco. A continuación, se lista el código HTML de dicha página, en el que destaca el valor de la etiqueta `<title>` que es utilizado para verificar el contenido de la página.

pruebas sistema software/testData/paginaEstatica.html

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>Esto es el título!</title>
  </head>
  <body>
    <font face="Verdana" size=3><b>Soy una pagina de prueba!</b></font>
  </body>
</html>

```

<sup>21</sup> En el código fuente del ejemplo, la clase `WebResponse` ha sido referida como `com.gargoylesoftware.htmlunit.WebResponse` para distinguirla de la clase de igual nombre (`com.meterware.httpunit.WebResponse`) perteneciente a la herramienta `HttpUnit`. Esto es necesario ya que ambas herramientas son utilizadas dentro del archivo del ejemplo.

### 11.3.3.3. Prueba de documentos XML generados dinámicamente

En este apartado se va a ver como, mediante la utilización combinada de las herramientas HtmlUnit, JtestCase y XMLUnit<sup>22</sup>, se pueden realizar pruebas funcionales completamente automatizadas sobre una aplicación Web que genera documentos XML con información dinámica. De nuevo, la aplicación Web a probar es la descrita en el Apéndice B y los documentos XML son aquellos generados en respuesta a peticiones sobre el estado del tráfico. De esta forma el objetivo de la prueba es verificar que todas las peticiones HTTP correctas que pueden realizarse sobre el servidor devuelven documentos XML con la información adecuada según el contenido del archivo de registros que la aplicación utiliza para generar dichos documentos. Téngase en cuenta en lo sucesivo que el contenido del archivo de registros se ha fijado como una constante y no interviene en el caso de prueba del ejemplo.

La gran ventaja de utilizar estas tres herramientas en combinación es que, en caso de que se añadan nuevas peticiones al servidor que necesiten ser probadas, no va a ser necesario modificar el método de pruebas sino únicamente definir un nuevo caso de prueba en el documento XML.

A continuación, se lista el código fuente del método de prueba `peticionesCorrectas` perteneciente a la clase `PeticionesTest` y encargado, como su propio nombre indica, de ejecutar casos de prueba de peticiones correctas<sup>23</sup>.

`pruebas sistema software/src/pruebasSistemaSoftware/funcionales/PeticionesTest.java`

```
/**
 * Casos de prueba de peticiones correctas
 */
@Test public void peticionesCorrectas() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases = m_jtestcase.getTestCasesInstancesInMethod("peticionesCorrectas");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {
```

<sup>22</sup> Esta herramienta se utiliza para la comparación de documentos XML y es analizada en profundidad en el Capítulo 10 de este libro.

<sup>23</sup> Más adelante se verá como realizar la prueba de las peticiones incorrectas. Se ha decidido separar la ejecución de los casos de prueba de peticiones correctas e incorrectas debido a que los documentos obtenidos al ejecutar dichas peticiones están en formato XML y HTML respectivamente por los que la naturaleza de la prueba y la herramienta más propicia a utilizar (HtmlUnit y JWebUnit respectivamente) varía.

```

// obtencion de los datos asociados al caso de prueba
TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);

try {

    HashMap params = testCase.getTestCaseParams();
    String strURL = (String)params.get("url");

    // ejecucion de la peticion
    WebClient webClient = new WebClient(BrowserVersion.FULL_FEATURED_BROWSER);
    URL url = new URL(strURL);
    Page paginaRespuesta = webClient.getPage(url);
    assertEquals(XmlPage.class,paginaRespuesta.getClass());
    XmlPage paginaXml = (XmlPage)paginaRespuesta;
    Document docObtenido = paginaXml.getXmlDocument();

    // carga del documento esperado
    MultiKeyHashtable asserts = testCase.getTestCaseAssertParams();
    String[] strClaves = {"documentoEsperado", "EQUALS"};
    String strDocumentoEsperado = (String)(asserts.get(strClaves));
    DocumentBuilderFactory docFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder docBuilder = docFactory.newDocumentBuilder();
    Document docEsperado = docBuilder.parse(strDocumentoEsperado);

    // verificacion
    Diff diff = new Diff(docEsperado,docObtenido);
    boolean bResultado = diff.identical();
    assertTrue("Error en el formato del documento XML generado: " +
        diff,bResultado);

} catch (JTestCaseException e) {

    // Ha ocurrido un error en el procesado de los datos del caso de prueba
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (MalformedURLException e) {

    // Ha ocurrido un error en el procesado de los datos del caso de prueba
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (IOException e) {

    // Ha ocurrido un error en el procesado de los datos del caso de prueba
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (ParserConfigurationException e) {

    // Ha ocurrido un error en el procesado de los datos del caso de prueba
    e.printStackTrace();
    fail("Error al ejecutar un caso de prueba");

} catch (SAXException e) {

```



```

        // Ha ocurrido un error en el procesado de los datos del caso de prueba
        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    }
}
}

```

Inicialmente se utiliza la herramienta JTestCase<sup>24</sup> para cargar los casos de prueba asociados al método. Como puede observarse en el fragmento de documento XML que se muestra al final de este párrafo, cada caso de prueba contiene lo siguiente:

- Un parámetro de tipo `String` que recibe el nombre `url` y contiene la petición HTTP a realizar sobre el servidor.
- Un `assert` (etiqueta `<assertparam>` en este caso) cuyo parámetro es la ruta en disco del documento XML que se espera obtener del servidor al ejecutar la anterior petición HTTP.

Una vez se ha cargado la url de la petición a ejecutar, se utiliza la herramienta `HtmlUnit` para realizar dicha tarea. Si se ha utilizado esta herramienta en lugar de `JWebUnit` es porque esta última no permite trabajar con documentos XML directamente, solo con páginas HTML. El resultado es un objeto de la clase `XmlPage` que representa el documento XML obtenido y del cual, a través del método `getXmlDocument`, se puede obtener un objeto que implementa la interfaz `Document` (`org.w3c.dom.Document`). Este objeto ya puede ser utilizado mediante `XMLUnit` para ser comparado con el documento XML esperado que es cargado desde el disco. En el siguiente fragmento de código XML se han definido los casos de prueba mediante la sintaxis de `JTestCase`.

pruebas sistema software/testCases/PeticionesTest.xml

```

<method name="peticionesCorrectas">
  <test-case name="peticionTramosMasTrafico">
    <params>
      <param name="url" type="java.lang.String">http://localhost:1100/
        servidorEstadoTrafico/?peticion=tramosMasTrafico
      </param>
    </params>
    <asserts>
      <assertparam name="documentoEsperado" type="java.lang.String"
        action="EQUALS">./testData/peticionTramosMasTrafico.xml
      </assertparam>
    </asserts>
  </test-case>
  <test-case name="petitionTramosMenosTrafico">
    <params>
      <param name="url" type="java.lang.String">http://localhost:1100/
        servidorEstadoTrafico/?peticion=tramosMenosTrafico
      </param>
    </params>
  </test-case>
</method>

```

<sup>24</sup> Véase Capítulo 8 Mejora de la mantenibilidad mediante JTestCase.

```

    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionTramosMenosTrafico.xml
        </assertparam>
    </asserts>
</test-case>
<test-case name="peticionCarrilesCortados">
    <params>
        <param name="url" type="java.lang.String">http://localhost:1100/
            servidorEstadoTrafico/?peticion=carrilesCortados
        </param>
    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionCarrilesCortados.xml
        </assertparam>
    </asserts>
</test-case>
<test-case name="peticionObras">
    <params>
        <param name="url" type="java.lang.String">http://localhost:1100/
            servidorEstadoTrafico/?peticion=obras
        </param>
    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionObras.xml
        </assertparam>
    </asserts>
</test-case>
<test-case name="peticionAccidentes">
    <params>
        <param name="url" type="java.lang.String">http://localhost:1100/
            servidorEstadoTrafico/?peticion=accidentes</param>
    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionAccidentes.xml
        </assertparam>
    </asserts>
</test-case>
<test-case name="peticionCarretera">
    <params>
        <param name="url" type="java.lang.String">http://localhost:1100/
            servidorEstadoTrafico/?peticion=carretera&amp;carretera=M-40
        </param>
    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionCarretera.xml
        </assertparam>
    </asserts>
</test-case>
<test-case name="peticionClima">
    <params>

```

```

        <param name="url" type="java.lang.String">http://localhost:1100/
            servidorEstadoTrafico/?peticion=clima&clima=Favorable
        </param>
    </params>
    <asserts>
        <assertparam name="documentoEsperado" type="java.lang.String"
            action="EQUALS">./testData/peticionClima.xml
        </assertparam>
    </asserts>
</test-case>
</method>

```

De forma análoga a como se ha realizado la prueba de las peticiones correctas, se puede realizar las pruebas de las peticiones incorrectas. Para este último caso, puesto que la respuesta esperada de la aplicación Web es un documento HTML con una estructura muy sencilla (véase Figura 11.4), se ha decidido utilizar JWebUnit. A continuación, se lista el código del método `peticionesIncorrectas` encargado de ejecutar los casos de prueba definidos para este tipo de peticiones.

`pruebas sistema software/src/pruebasSistemaSoftware/funcionales/PeticionesTest.java`

```

/**
 * Casos de prueba de peticiones correctas
 */
@Test public void peticionesIncorrectas() {

    // comprobacion
    if (m_jtestcase == null)
        fail("Error al cargar los casos de prueba");

    // carga de los casos de prueba correspondientes a este metodo
    Vector testCases = null;
    try {
        testCases = m_jtestcase.getTestCasesInstancesInMethod("peticiones
            Incorrectas");
        if (testCases == null)
            fail("No se han definido casos de prueba");
    } catch (JTestCaseException e) {
        e.printStackTrace();
        fail("Error en el formato de los casos de prueba.");
    }

    // ejecucion de los casos de prueba
    for (int i=0; i<testCases.size(); i++) {

        // obtencion de los datos asociados al caso de prueba
        TestCaseInstance testCase = (TestCaseInstance)testCases.elementAt(i);

        try {

            // se carga la pagina
            HashMap params = testCase.getTestCaseParams();

```

```

        String strURL = (String)params.get("url");

        // se realiza la peticion
        beginAt(strURL);

        // verificacion
        assertEquals("Mensaje");
        assertTextPresent("Error: formato de peticion incorrecto.");

    } catch (JTestCaseException e) {

        // Ha ocurrido un error en el procesado de los datos del caso de prueba
        e.printStackTrace();
        fail("Error al ejecutar un caso de prueba");
    }
}
}

```

Únicamente cabe destacar la forma en que se ha verificado el documento HTML obtenido al ejecutar las peticiones incorrectas. Se trata de una verificación muy sencilla basada en el título del documento y en el texto que aparece en el cuerpo del mismo. A continuación, se muestra el fragmento del documento XML que contiene la definición de los casos de prueba para este método.

pruebas sistema software/testCases/PeticionesTest.xml

```

<method name="peticionesIncorrectas">
  <test-case name="sintaxisIncorrecta">
    <params>
      <param name="url"
type="java.lang.String">/?peticimosMasTrafico=23</param>
    </params>
  </test-case>
  <test-case name="parametrosExtra">
    <params>
      <param name="url" type="java.lang.String">/?peticion=carretera&
carretera=M-40&clima=Favorable</param>
    </params>
  </test-case>
  <test-case name="parametrosInsuficientes">
    <params>
      <param name="url" type="java.lang.String">/?peticion=carretera</param>
    </params>
  </test-case>
</method>

```

La Figura 11.4 presenta la página con el mensaje de error esperado en caso de que se realicen peticiones incorrectas sobre el sistema.

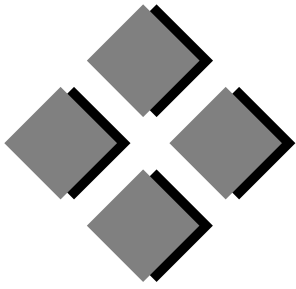


**Figura 11.4.** Documento HTML esperado al ejecutar peticiones HTTP incorrectas.

## 11.4. Bibliografía

- Rainsberger, J. B. (con contribuciones de Scott Stirling): *JUnit Recipes. Practical Methods for Programmer Testing*, Capítulo 13, Manning Publications, Greenwich, 2005.
- Nguyen, H. Q.: *Testing Applications on the Web: Test Planning for Internet-Based*, Wiley, 16 de octubre de 2000.
- Dustin, E.; Rashka, J.; McDiarmid, D. y Nielson, J.: *Quality Web Systems: Performance, Security, and Usability*, Addison-Wesley, agosto de 2001.
- <http://httpunit.sourceforge.net/>.
- <http://htmlunit.sourceforge.net/>.
- <http://jwebunit.sourceforge.net/>.





# Capítulo 12

## Pruebas de validación

### SUMARIO

12.1. Introducción

12.2. JFunc

12.3. JUnitPerf

12.4. JMeter

12.5. Bibliografía

## 12.1. Introducción

Siguiendo la estrategia de pruebas expuesta en el Capítulo 1, una vez realizadas las pruebas unitarias y de integración, se procede a realizar las pruebas de validación. Estas pruebas consisten en comprobar si se satisface la especificación de requisitos. Conviene recordar que dicha especificación incluye las funciones que el software debe realizar y los criterios de calidad que dichas funciones deben satisfacer<sup>1</sup>. Es decir, con las pruebas de validación se comprueba si el software en pruebas realiza las funciones definidas, lo que normalmente se conoce como pruebas funcionales, y si lo hace bajo los criterios de calidad establecidos. En este sentido, en muchas ocasiones se utiliza la expresión *pruebas funcionales* como sinónimo de pruebas de validación, cuando en realidad las pruebas de validación son algo más amplio.

Para automatizar las pruebas funcionales se dispone de herramientas como JFunc, HttpUnit, HtmlUnit o JWebUnit<sup>2</sup>. Estas tres últimas herramientas están especializadas para probar los requisitos funcionales de aplicaciones Web y ya se han descrito en el Capítulo 11, mientras que la

<sup>1</sup> Los criterios de calidad también son conocidos como requisitos no funcionales.

<sup>2</sup> Nótese que, en ocasiones, la herramienta DBUnit que fue presentada en el Capítulo 9 de este libro también se utiliza para realizar pruebas funcionales.

herramienta JFunc se describirá a lo largo de este capítulo. Una de las características más llamativas de JFunc y que la hace muy interesante desde el punto de vista de las pruebas funcionales, es que, al contrario que JUnit<sup>3</sup>, cuando se produce un fallo, la ejecución de las pruebas no se detiene y todos los fallos producidos se informan al final de la ejecución.

Uno de los criterios de calidad es el rendimiento. Dicho de forma muy sencilla, ¿cuántos usuarios pueden usar la aplicación a la vez sin que aumente el tiempo de respuesta? La mayor parte de las pruebas de carga o de estrés<sup>4</sup> comprueban si la aplicación puede procesar un gran número de entradas. En el caso de las aplicaciones Web, donde las entradas son peticiones al sistema, las pruebas de carga consisten en comprobar si se puede atender un gran número de dichas peticiones en un corto periodo de tiempo. Las pruebas de estrés se suelen realizar en un entorno separado del entorno de producción, pero de las mismas características, ya que de no ser así, las mediciones de tiempo de respuesta no servirían de mucho.

Por otro lado, herramientas como JUnitPerf y JMeter, que serán explicadas también a lo largo de este capítulo, permiten automatizar las pruebas de rendimiento:

- JUnitPerf permite incorporar pruebas de carga a las pruebas unitarias, para así determinar si alguna de las unidades probadas supone un cuello de botella o resulta crítica para el rendimiento global del software. De hecho, las pruebas desarrolladas con JUnitPerf se definen sobre métodos de pruebas definidos con JUnit y, posteriormente, se pueden añadir a una misma batería de pruebas junto con el conjunto de pruebas unitarias. De esta forma, las pruebas de carga pueden ejecutarse a la vez que las unitarias cada vez que se modifica el software de producción. Esto ayuda a determinar si los cambios realizados afectan al rendimiento de la unidad software modificada.
- JMeter envía peticiones preprogramadas y recoge datos del tiempo de respuesta del software desarrollado. Habitualmente, al usar JMeter no se comprueba si la respuesta es la esperada, puesto que eso ya se debe haber comprobado con otras pruebas, ya sean unitarias o funcionales y, por tanto, queda fuera del alcance.

En este capítulo se describirán estas herramientas: cómo instalarlas, sus elementos básicos y cómo escribir las pruebas. Los ejemplos que se utilizan se basan, una vez más, en el ejemplo del Servidor del Estado del Tráfico descrito en el Apéndice B.

## 12.2. JFunc

JFunc es un *framework* definido como una extensión de JUnit para realizar pruebas funcionales<sup>5</sup>. Es una herramienta de libre distribución y se puede descargar desde <http://jfunc.sourceforge.net/>. JFunc ofrece las siguientes características respecto a JUnit:

- Permite fallos múltiples. Esto quiere decir que cuando un método de prueba encuentra un fallo al ejecutar un caso de prueba, la ejecución del resto de casos de prueba definidos den-

<sup>3</sup> Véase Capítulo 2.

<sup>4</sup> Así se denominan en muchas ocasiones las pruebas de rendimiento.

<sup>5</sup> La documentación de JFunc indica que las pruebas funcionales también se conocen como pruebas de integración (“Functional testing (also called integration testing)...”). Esto no es exacto puesto que las pruebas funcionales abarcan algo más que las pruebas de integración como se explicó en el Capítulo 1. De acuerdo a las definiciones que se dieron en dicho capítulo, JFunc facilita las pruebas funcionales.



tro del mismo método no se interrumpe sino que continua, al contrario de lo que ocurre con JUnit.

- Facilita la construcción de la *suite* de pruebas mediante:
  - El uso de *proxies* para la construcción de la *suite*.
  - La utilización de un objeto de prueba para varias pruebas, en vez de una instancia por cada prueba.
  - Métodos de prueba que pueden aceptar argumentos.
- Test Runner mejorado:
  - Aserciones detalladas.
  - Paso de argumentos para las pruebas.

Existen desarrolladores de la opinión de que JFunc cubre ciertas carencias presentes en JUnit a la hora de realizar pruebas unitarias sobre métodos para los que se definen varios casos de prueba. Sin embargo, esto no puede tomarse al pie de la letra, dado que, estrictamente, según la forma en que JUnit fue diseñado, no se ha de definir más de un caso de prueba para cada método de prueba. Por otro lado, es posible ahorrar tiempo de ejecución gracias al hecho de que permitir múltiples fallos ofrece la posibilidad de observar varios defectos dentro del mismo método a probar ejecutando una única vez el método de pruebas. Esto último es importante en el caso de las pruebas funcionales que, típicamente, se componen de un gran número de casos de prueba y por tanto conllevan un tiempo de ejecución asociado considerable. Sin embargo, para el caso de las pruebas unitarias el tiempo de ejecución no suele ser tan crítico ya que están enfocadas a una clase y pueden ser ejecutadas repetidamente en un proceso iterativo de corrección de fallos.

En los apartados siguientes se describen estas diferencias de JFunc respecto a JUnit que hacen a esta herramienta más adecuada para la realización de pruebas funcionales.

### 12.2.1. Procedimiento de utilización de JFunc

A la hora de utilizar JFunc se han de seguir una serie de pasos muy sencillos.

1. La clase de prueba ha de heredar de `JFuncTestCase`. Si se está trabajando con versiones de JUnit anteriores a la versión 4.0, se deberá heredar de `JFuncTestCase` en lugar de `TestCase`. En caso contrario simplemente se heredará de `JFuncTestCase`.
2. Se ha de invocar el método `setFatal` perteneciente a la clase `JFuncAssert` con el argumento `false`, de esta forma se permite la detección de múltiples fallos. El lugar más adecuado desde el que invocar este método es desde el constructor de la clase de pruebas. Cuando se utiliza JUnit, una vez que ocurre un fallo al ejecutar un caso de prueba dentro de un método de prueba, el resto de casos de prueba no se siguen ejecutando y la ejecución continúa con el siguiente método de prueba. Esto es así porque según la filosofía de JUnit se puede considerar que todos los fallos son “fatales”. Con JFunc gracias a la clase `JFuncAssert` (que es equivalente a la clase `Assert` en JUnit) se pueden establecer los fallos como “fatales” o “no-fatales”, de esta forma la ejecución de un método de prueba sólo se detendrá tras un fallo si ese fallo es “fatal”.
3. Una vez realizados los pasos anteriores se procederá a definir la clase de pruebas de forma tradicional, es decir, tal y como se hacía con JUnit. La única diferencia es que, esta vez, la clase de pruebas se dedica a hacer pruebas funcionales sobre un sistema en lugar de pruebas unitarias sobre una única clase.

A continuación, se muestra un ejemplo de utilización de JFunc para la realización de pruebas funcionales. Se trata de una clase muy esquemática con el único objetivo de mostrar el uso de la herramienta.

```
pruebas sistema software/src/pruebasSistemaSoftware/
validacion/funcionales/FuncionalesTest.java
```

```
//JUnit
import junit.framework.*;

//JFunc
import junit.extensions.jfunc.*;

public class FuncionalesTest extends JFuncTestCase {

    /**
     * constructor de la clase
     */
    public FuncionalesTest() {

        //Llamamos al constructor de la superclase
        super("testVariosFallos");
        //Se van a permitir multiples fallos
        setFatal(false);
    }

    /**
     * Metodo de prueba que realiza varias pruebas funcionales
     */
    public void testVariosFallos() {

        //caso de prueba 1
        fail("primer fallo");
        //caso de prueba 2
        fail("segundo fallo");
        //caso de prueba 3
        fail("tercer fallo");
    }
}
```

Como puede observarse la clase `FuncionalesTest` hereda de `JFuncTestCase` y desde su constructor se ha invocado el método `setFatal` de la clase `JFuncAssert` con el valor `false` de modo que, cuando se produzca un fallo dentro de cualquier método de prueba, la ejecución de este continuará. Dentro de la clase está definido el método de prueba `testVariosFallos` que se encarga de ejecutar varios casos de prueba sobre el sistema software a probar. Nótese que, a modo ilustrativo, dentro de este método de prueba existen varias sentencias `fail` definidas de modo que, dado que los fallos son “no-fatales”, todas ellas han de ejecutarse. A continuación, se muestra el informe de resultados de la ejecución del método de prueba `testVariosFallos` mediante Ant<sup>6</sup> y JUnitReport<sup>7</sup>.

<sup>6</sup> El Capítulo 3 está dedicado a la ejecución de pruebas de software con ayuda de la herramienta Ant.

<sup>7</sup> El Capítulo 6 de este libro describe el modo de utilización de JUnitReport para generar automáticamente informes con los resultados de las pruebas.

```

primer fallo
junit.framework.AssertionFailedError: primer fallo
at junit.extensions.jfunc.JFuncAssert.fail(JFuncAssert.java:147)
...

segundo fallo
junit.framework.AssertionFailedError: segundo fallo
at junit.extensions.jfunc.JFuncAssert.fail(JFuncAssert.java:147)
...

tercer fallo
junit.framework.AssertionFailedError: tercer fallo
at junit.extensions.jfunc.JFuncAssert.fail(JFuncAssert.java:147)
...

```

El resultado de la ejecución muestra que, en efecto, la ejecución del método de prueba no se ha detenido tras la primera sentencia `fail` sino que ha continuado hasta el final pasando por todos ellos. Como efecto paralelo, cabe observar que a diferencia con JUnit ahora ya no se verifica la propiedad de que el número de métodos de prueba ejecutados sea mayor o igual al número de fallos más el número de errores.

## 12.2.2. Ejecución de las clases de prueba mediante JFunc

JFunc presenta un runner especializado llamado `JFuncRunner` que puede utilizarse en lugar de los runners en modo texto o modo gráfico presentados en el Capítulo 2. Sin embargo, la forma más sencilla de utilizar JFunc es mediante el uso de Ant. Para la ejecución de la clase de pruebas del ejemplo anterior simplemente se ha seguido el procedimiento de ejecución de una clase de pruebas tal y como viene descrito en el Capítulo 3. En particular, para el ejemplo anterior se ha escrito el siguiente fragmento de código Ant.

```
pruebas sistema software/src/pruebasSistemaSoftware/build.xml
```

```

<!-- Ejecucion de las pruebas de validacion -->
<target name="validation" depends="compile" description="Ejecucion de
las pruebas de validacion" >
  <junit printsummary="yes" fork="true" haltonerror="no"
    haltonfailure="no" showoutput="yes">
    <formatter type="xml"/>
    <classpath refid="project.junit381.class.path"/>
    <batchtest todir="${reports}/junitreport">
      <fileset dir="${build}">
        <include
          name="pruebasSistemaSoftware/validacion/funcionales/*Test
            .class" />
      </fileset>
    </batchtest>
  </junit>
<!-- Generacion de informes de resultados en formato html -->

```

```

<junitreport todir="${reports}/junitreport">
  <fileset dir="${reports}/junitreport">
    <include name="TEST-*.xml"/>
  </fileset>
<report format="frames" todir="${reports}/junitreport"/>
</junitreport>
...
</target>

```

### 12.2.3. Mensajes detallados

Otra de las aportaciones de JFunc respecto a JUnit es la posibilidad de incluir mensajes con información detallada sobre las comprobaciones que se hacen dentro de los métodos de prueba. A la hora de presentar la información resultante de la ejecución de las pruebas puede interesar mostrar información no solo asociada a los métodos `assert` que han fallado, sino también los que se han verificado. Esto resulta especialmente útil en las pruebas funcionales, ya que estas pueden ser realizadas por personal que no esté familiarizado con el código, por ejemplo, miembros del equipo de calidad. Conviene, por tanto, saber qué es lo que ha sido probado, lo que ha fallado y lo que se ha ejecutado de forma esperada. JFunc cubre esta necesidad mediante un conjunto de métodos de aserción que básicamente extienden la funcionalidad de aquellos presentes en la clase `Assert` de JUnit. Estos métodos están definidos en la clase `VerboseAssert` del paquete `junit.extensions.jfunc`. Un ejemplo de uso de estas aserciones es el siguiente:

```

/**
 * Metodo de prueba que muestra informacion detallada sobre
 * aserciones
 */
public void testAsercionesDetalladas() {

    //Informacion de valores obtenidos/esperados
    vassert("mensaje informativo del fallo", 2, 1);
    assertEquals("mensaje informativo del fallo",2,1);

    //Informacion de verificacion y de no verificacion
    vassert("Se ha verificado la condicion", "mensaje informativo del
        fallo", 1 == 1);
    vassert("Se ha verificado la condicion", "mensaje informativo del
        fallo99", 1 == 2);
}

```

En las dos primeras sentencias del método anterior se observa cómo el método `vassert` se puede comportar de forma similar al método `assertTrue` de la clase `Assert` de JUnit. Sin embargo, en las dos sentencias siguientes se ve la característica de la que se ha hablado antes. Es posible asociar mensajes de texto que serán mostrados durante la ejecución cuando se verifican ciertas condiciones. La ejecución del método anterior producirá los siguientes mensajes de texto:

```

mensaje informativo del fallo: EXPECTED(2) ACTUAL(1)
mensaje informativo del fallo expected:<2> but was:<1>
se ha verificado la condicion
mensaje informativo del fallo

```

Finalmente, conviene señalar que es posible conseguir este mismo efecto sin la necesidad de utilizar JFunc. Para ello, basta con coger cada método `assert` y duplicarlo negando la condición a verificar y cambiando el mensaje. No obstante resulta mucho más sencillo y cómodo utilizar JFunc.

## 12.3. JUnitPerf

Una forma de realizar pruebas de rendimiento sobre una aplicación es comprobar el rendimiento de los componentes para los que se han desarrollado pruebas unitarias con JUnit. Si, además, estas pruebas de rendimiento son también realizadas con JUnit, se pueden añadir al conjunto de pruebas que se ejecutan continuamente sobre la aplicación. Esto se puede lograr con JUnitPerf, que permite medir el rendimiento y escalabilidad de la funcionalidad contenida en las pruebas JUnit. Se pueden crear dos tipos de pruebas usando JUnitPerf: pruebas de tiempo de respuesta y pruebas de carga. Ambos tipos de pruebas están basados en el patrón de diseño *Decorador*, y utilizan el mecanismo de *suites* de JUnit. Es decir, JUnitPerf es un conjunto de decoradores que se usan para medir el rendimiento y la escalabilidad. Dichos decoradores son:

- **TimedTest**  
Un `TimedTest` es un decorador que ejecuta una prueba y mide el tiempo transcurrido durante la prueba. Por omisión un `TimedTest` esperará a la finalización de la prueba y fallará si se excede un tiempo máximo ya definido. También se puede construir de modo que cuando el tiempo máximo es alcanzado se indique un fallo inmediatamente.
- **LoadTest**  
Un `LoadTest` es un decorador que ejecuta una prueba con un número simulado de iteraciones y de usuarios concurrentes.

Decorando pruebas JUnit existentes es rápido y fácil crear un conjunto de pruebas de rendimiento en una *suite* de pruebas de rendimiento. La *suite* de pruebas de rendimiento puede ejecutarse independientemente de las pruebas JUnit. De hecho, se suele evitar agrupar las pruebas con JUnitPerf de las pruebas con JUnit para poder ejecutar las suites de prueba de forma independiente.

Las pruebas con JUnitPerf están pensadas para situaciones donde hay requisitos de rendimiento y escalabilidad definidos de forma cuantitativa y que se desean mantener cuando, por ejemplo, se reestructura o modifica un algoritmo (rendimiento); o cuando se reorganizan los recursos de un sistema (escalabilidad).

Antes de usar JUnitPerf conviene analizar qué partes del código pueden dar problemas respecto a la escalabilidad y al rendimiento. Entonces se escriben las pruebas con JUnitPerf para probar si dichos requisitos se mantienen tras los cambios de diseño y codificación. Si no es así, se modificará el software y de nuevo se pasarán las pruebas de JUnitPerf. Estas acciones se realizarán hasta conseguir los requisitos ya definidos. Es decir, JUnitPerf sirve además para facilitar la ejecución de las pruebas de regresión.

### 12.3.1. Instalación y configuración de JUnitPerf

Para trabajar con JUnitPerf se necesita Java 2 y JUnit 3.5 o superior. Una vez que estas herramientas están instaladas, se procede de la siguiente manera:

1. Descargar JUnitPerf 1.9 desde <http://clarkware.com/software/junitperf-1.9.1.zip>. Esta distribución contiene un fichero .jar, el código fuente, algunos ejemplos y el javadoc.
2. Descomprimir el fichero `junitperf-1.9.1.zip` en un directorio creado para tal efecto. Este directorio se referenciará después como `%JUNITPERF_HOME%`.
3. Añadir el archivo `%JUNITPERF_HOME%\lib\junitperf-1.9.1.jar` a la variable de entorno `CLASSPATH`.

Para comprobar si JUnitPerf se ha instalado correctamente ha de ejecutarse un conjunto de pruebas que viene a modo de ejemplo incluido en la distribución de JUnitPerf. Desde el directorio de JUnitPerf se ejecutan dichas pruebas con la siguiente orden:

```
c:\> ant test
```

## 12.3.2. Creando pruebas con JUnitPerf

En los apartados siguientes se describe, mediante ejemplos, cómo definir pruebas de tiempo de ejecución y pruebas de carga con JUnitPerf. Al final de cada apartado se analiza la eficiencia de las distintas formas de definir dichas pruebas.

### 12.3.2.1. Pruebas de tiempo de respuesta: `TimedTest`

Para crear una prueba de tiempo de respuesta se utiliza la clase `TimedTest` de JUnitPerf. Esta clase es básicamente un decorador sobre uno o varios casos de prueba contenidos en un objeto que implemente la interfaz `Test`. El objeto `TimedTest` se encarga de medir el tiempo de ejecución de dichos casos de prueba y de comunicar un fallo en caso de que exceda cierto límite de tiempo. El constructor de `TimedTest` toma como parámetros un objeto que implemente la interfaz `Test` y un umbral de tiempo en milisegundos.

En el siguiente ejemplo se va a realizar una prueba de tiempo sobre el método de prueba `testPeticiónPaginaInformación`<sup>8</sup> perteneciente a la clase `PeticiónTest`. Este método se encarga de realizar una petición HTTP al sistema software descrito en el Apéndice B de este libro y comprobar que la página de información obtenida es la correcta. La prueba va a consistir en invocar el método de prueba y asegurar que su ejecución no se prolonga mas allá de un determinado tiempo especificado en milisegundos.

```
pruebas sistema software/src/pruebasSistemaSoftware/
validación/rendimiento/TiempoTest.java
```

```
//JUnitPerf
import com.clarkware.junitperf.*;

import pruebasSistemaSoftware.junit381.*;
```

<sup>8</sup> Nótese que este método es equivalente al método `peticiónPaginaInformación` perteneciente a la clase `PeticionesTest` contenida en el paquete `pruebasSistemaSoftware.funcionales`. La diferencia es que la clase `PeticiónTest` está escrita con la versión 3.8.1 de JUnit y, por tanto, permite construir objetos `TestSuite` con un único método en lugar de con todos los métodos de prueba de la clase, como ocurre con las versiones 4.x de JUnit.

```

public class TiempoTest {

    public static Test suite() {

        TestSuite suite = new TestSuite();
        suite.addTest(tiempoBloqueante());
        suite.addTest(tiempoNoBloqueante());
        return suite;

    }

    /**
     * Realiza una prueba temporizada utilizando el metodo
     * testPeticionPaginaImnformacion, la
     * prueba es bloqueante por lo que la ejecucion de la prueba
     * espera a que el metodo de
     * prueba finalice
     */
    public static Test tiempoBloqueante() {

        long tiempoMaximo = 1000;
        Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
        Test tiempo = new TimedTest(pruebaBasica, tiempoMaximo);

        return tiempo;
    }

    /**
     * Realiza una prueba temporizada utilizando el metodo
     * testPeticionPaginaImnformacion, la
     * prueba es no bloqueante por lo que al consumirse el tiempo
     * maximo, la ejecucion de la
     * prueba finaliza
     */
    public static Test tiempoNoBloqueante() {

        long tiempoMaximo = 1000;
        Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
        Test tiempo = new TimedTest(pruebaBasica, tiempoMaximo, false);

        return tiempo;
    }
}

```

Se ha definido una clase llamada `TiempoTest` que contiene dos métodos para realizar las pruebas de tiempo de ejecución. Estos métodos son `tiempoBloqueante` y `tiempoNoBloqueante` que se encargan de decorar el método de prueba `testPeticionPaginaInformacion` de modo que falle en caso de que su ejecución se prolongue mas allá de un segundo (1000 milisegundos). La diferencia entre estos dos métodos es que mientras que el primero decora el método de prueba de forma que se ejecute completamente, el segundo le hace detenerse y comunicar fallo en cuanto su ejecución se prolonga más allá del límite de tiempo establecido (en este caso un segundo). Para establecer esta diferencia basta con instanciar el objeto



`TimedTest` con el último parámetro a `false` u omitiéndolo según se quiera un decorador no bloqueante o bloqueante respectivamente. En general las pruebas de tiempo de ejecución se pueden dividir en dos categorías:

- Pruebas de tiempo de ejecución que esperan (bloqueantes): por omisión, un `TimedTest` decora un método de prueba de forma que cuando se ejecute lo haga hasta el final, es decir, la ejecución se bloquea hasta que el método de prueba termine. Una vez ha terminado, en caso de que su tiempo de ejecución exceda el límite prefijado, el método fallará. Esta forma de definir la prueba de tiempo de ejecución permite acumular todos los resultados de las pruebas y posibilita revisar los tiempos de ejecución de los métodos de prueba al final. Si la prueba decorada lanza hilos, ya sea directa o indirectamente, entonces la prueba decorada debe esperar la finalización de todos esos hilos para que, de esta forma, el control retorne a la prueba de tiempo. De otro modo, la prueba esperará indefinidamente. Como regla general, las pruebas unitarias deberían esperar la finalización de los hilos usando, por ejemplo, `Thread.join()` para mayor precisión en los resultados de las pruebas.
- Pruebas de tiempo de ejecución que no esperan (no bloqueantes): si la prueba de tiempo de ejecución se define de forma que falle en cuanto se cumpla el tiempo máximo, la prueba no esperará a que la prueba decorada finalice al superarse ese tiempo. Estas pruebas son más eficientes en tiempo que las anteriores. Sin embargo, a diferencia de aquellas, en caso de fallo no permiten observar el tiempo de ejecución de los métodos de prueba contenidos en el `TestSuite` y, por tanto, no dan una idea de por cuanto tiempo han fallado.

Obsérvese la similitud entre este tipo de pruebas de tiempo de ejecución y los métodos de prueba con *timeout* tal como están definidos en las versiones 4.x de JUnit. Al echar la vista atrás (véase el Capítulo 2), este tipo de métodos se definían mediante la anotación `@Test(timeout=X)` de forma que su ejecución se detenía en caso de que la prueba durase más de *x* milisegundos. En realidad esto no es más que una forma de prueba de tiempo de ejecución, completamente equivalente a la vista en JUnitPerf con la única salvedad de que el *timeout* se define de forma local a cada método de prueba y nunca sobre un *TestSuite*.

Finalmente, es conveniente tener en cuenta que el tiempo medido por un `TimedTest` que “decora” un método de prueba `testMetodoPrueba()` incluye el tiempo de ejecución de los métodos `setUp()`, `tearDown()` así como la ejecución del propio método de prueba. Así mismo si el objeto `TimedTest` “decora” un objeto de tipo `TestSuite` compuesto por varios métodos de prueba, el tiempo medido incluye la ejecución de los métodos `setUp()`, `testMetodoPrueba()` y `tearDown()` para todos los métodos de prueba contenidos en el `TestSuite`.

### 12.3.2.2. Pruebas de carga: `LoadTest`

La clase `LoadTest` permite definir pruebas de carga simulando un determinado número de usuarios concurrentes y un número determinado de iteraciones. En su forma más simple se construye sobre un método de prueba y un determinado número de usuarios. Por omisión, cada usuario realiza una sola ejecución del método de prueba.

En el siguiente ejemplo se realiza una prueba de carga sobre el método de prueba `testPetitionPaginaInformacion` utilizado en los ejemplos anteriores. La prueba consiste en el



acceso concurrente y simultáneo de 10 usuarios diferentes que ejecutan el método `testPeticiónPaginaInformación` una sola vez. Esta prueba resulta interesante porque emula un escenario real de uso de la aplicación descrita en el Apéndice B en el que 10 usuarios diferentes están accediendo al sistema Web desde su navegador para obtener la página de información.

```
pruebas_sistema_software/src/pruebasSistemaSoftware/validacion/rendimiento/CargaTest.java
```

```
/**
 * Este metodo decora el metodo de prueba
 * testPeticiónPaginaInformación con una carga de 10
 * usuarios y un retardo de 0 segundos entre usuarios (acceso
 * simultaneo)
 */
public static Test carga10Usuarios0Retardo1Petición() {

    int usuarios = 10;

    Test pruebaBasica = new PeticiónTest("testPeticiónPagina
        Información");
    Test carga = new LoadTest(pruebaBasica, usuarios);

    return carga;
}
```

Como se puede observar, para realizar el proceso de decoración simplemente se instancia un objeto de la clase `LoadTest` pasándole como parámetros un objeto que implementa la interfaz `Test` (que contiene en este caso un único método de prueba) y el número de usuarios concurrentes. Este tipo de pruebas permite verificar que el sistema desarrollado es robusto frente al volumen de carga que se desee.

Es posible invocar al constructor de `LoadTest` con un parámetro adicional con información sobre el tiempo entre accesos de diferentes usuarios, es decir, el intervalo de tiempo que transcurre entre la incorporación de dos usuarios al sistema. Existen dos posibilidades:

- Mediante un objeto de la clase `ConstantTimer` que especifica un intervalo constante. Cuando este intervalo es cero indica que todos los usuarios empiezan simultáneamente.
- Mediante un objeto de la clase `RandomTimer` que especifica un intervalo de tiempo aleatorio que sigue una distribución aleatoria.

A continuación, se analizan diversas posibilidades combinando el ritmo de incorporación de los usuarios al sistema, y el número de veces que ejecutan el método que se está probando. En todos los casos se considerará que el número de usuarios concurrentes es diez.

En el caso de que cada usuario ejecute el método una vez y el tiempo transcurrido entre la incorporación de dos usuarios sea de un segundo, la creación del objeto `LoadTest` se realizará de la siguiente manera:

```
pruebas_sistema_software/src/pruebasSistemaSoftware/validacion/rendimiento/CargaTest.java
```

```

/**
 * Este metodo decora el metodo de prueba
 * testPeticionPaginaInformacion con una carga de 10
 * usuarios y un retardo de 1 segundo entre usuarios
 */
public static Test carga10Usuarios1Retardo1Peticion() {

    int usuarios = 10;
    Timer timer = new ConstantTimer(1000);

    Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
    Test carga = new LoadTest(pruebaBasica, usuarios, timer);

    return carga;
}

```

Para simular que cada usuario realiza un número especificado de ejecuciones del método de prueba, la prueba `LoadTest` se construye con un objeto de la clase `RepeatedTest`. Si, por ejemplo, cada usuario ejecuta el método veinte veces y el tiempo transcurrido entre la incorporación de dos usuarios es de un segundo, la prueba `LoadTest` se crea de la siguiente forma.

```
pruebas sistema software/src/pruebasSistemaSoftware/validacion/rendimiento/CargaTest.java
```

```

/**
 * Este metodo decora el metodo de prueba testPeticionPagina
 * Informacion con una carga de 10
 * usuarios, 20 iteraciones por usuario y un retardo de 0.5 segundos
 * entre usuarios
 */
public static Test carga10Usuarios05Retardo20Peticiones() {

    int usuarios = 10;
    int iteraciones = 20;
    Timer timer = new ConstantTimer(500);

    Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
    Test pruebaRepetida = new RepeatedTest(pruebaBasica, iteraciones);
    Test carga = new LoadTest(pruebaRepetida, usuarios, timer);

    return carga;
}

```

Una forma alternativa y más breve de indicar el número de iteraciones es utilizando directamente el constructor de la clase `LoadTest`.

```
pruebas sistema software/src/pruebasSistemaSoftware/validacion/rendimiento/CargaTest.java
```

```

/**
 * Este metodo decora el metodo de prueba
 * testPeticionPaginaInformacion con una carga de 10
 * usuarios, 20 iteraciones por usuario y un retardo de 0 segundos
 * entre usuarios (utiliza
 * el constructor de LoadTest para indicar el numero de iteraciones)
 */
public static Test carga10Usuarios0Retardo20Peticiones() {
    int usuarios = 10;
    int iteraciones = 20;
    Timer timer = new ConstantTimer(1000);
    Test pruebaBasica = new
    PeticionTest("testPeticionPaginaInformacion");
    Test carga = new LoadTest(pruebaBasica, usuarios, iteraciones, timer);
    return carga;
}

```

En ocasiones, la prueba que se pretende decorar con el objeto `LoadTest` necesita encontrarse en un estado particular que se define en el método `setUp()`. En estos casos se debe utilizar una instancia de la clase `TestFactory` para asegurar que cada hilo de usuario utiliza una instancia local de la prueba. Si, por ejemplo, se desea crear una prueba con diez usuarios que ejecutan una instancia local de `PeticionTest`, se realizará lo siguiente:

```

int usuarios = 10;
Test factory = new TestFactory(PeticionTest.class);
Test loadTest = new LoadTest(factory, usuarios);

```

Si lo que se desea es crear una prueba con diez usuarios pero únicamente para un determinado método de una clase de prueba, se realizará lo siguiente:

```

int usuarios = 10;
Test factory = new TestMethodFactory(PeticionTest.class,
    "testPeticionPaginaInformacion");
Test loadTest = new LoadTest(factory, usuarios);

```

El rendimiento de las pruebas se puede degradar de manera significativa si hay demasiados usuarios concurrentes en una prueba de carga. El umbral de carga es específico de JVM.

### 12.3.2.3. Pruebas combinadas de tiempo y carga

Los ejemplos anteriores utilizan un único decorador para una prueba JUnit. Sin embargo, es posible aplicarlos en varios niveles. Es decir, que un decorador se aplique sobre otro decorador en lugar de directamente sobre una prueba. Cuando los decoradores `TimedTest` y `LoadTest`, se aplican uno sobre otro se obtienen escenarios de pruebas más reales.

En el siguiente ejemplo se prueba el tiempo de ejecución de una prueba de carga para el método `testPeticionPaginaInformacion`. El decorador `LoadTest` añade diez usuarios simultáneos que ejecutan el método una vez. `LoadTest` se “decora” con un `TimedTest` para probar el *throughput*<sup>9</sup> del método `testPeticionPaginaInformacion`. La prueba fallará si el tiempo total de la prueba de carga excede un segundo.

<sup>9</sup> Número de llamadas al método ejecutadas (y terminadas) por unidad de tiempo.

```
pruebas sistema software/src/pruebasSistemaSoftware/
validacion/rendimiento/CombinadasTest.java
```

```
/**
 * Este metodo decora el metodo de prueba
 * testPeticionPaginaInformacion con una carga de 10
 * usuarios de modo que el tiempo de ejecucion de todos los usuarios
 * debe ser inferior a 1
 * segundo
 */
public static Test cargaTemporizada () {

    int usuarios = 10;
    long tiempoMaximo = 1000;

    Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
    Test carga = new LoadTest(pruebaBasica, usuarios);
    Test tiempo = new TimedTest(carga, tiempoMaximo);

    return tiempo;
}
```

Si lo que se quiere probar es el tiempo de respuesta de un método para cada usuario concurrente, se realiza el proceso en orden inverso, es decir, se decora el objeto `TimedTest` con un decorador `LoadTest`. `TimedTest` mide el tiempo de ejecución del método `testPeticionPaginaInformacion`. `LoadTest` decora la prueba `TimedTest` simulando una carga de diez usuarios. La prueba fallará si el tiempo de respuesta para cualquier usuario es superior a 50 milisegundos.

```
pruebas sistema software/src/pruebasSistemaSoftware/
validacion/rendimiento/CombinadasTest.java
```

```
/**
 * Este metodo decora el metodo de prueba testPeticionPagina
 * Informacion con una carga de 10
 * usuarios de modo que se ejecutan 10 pruebas temporizadas con un
 * maximo de 50 milisegundos
 * por prueba
 */
public static Test tiempoEnCarga () {

    int usuarios = 10;
    long tiempoMaximo = 50;

    Test pruebaBasica = new PeticionTest("testPeticionPagina
        Informacion");
    Test carga = new TimedTest(pruebaBasica, tiempoMaximo);
    Test tiempo = new LoadTest(carga, usuarios);

    return carga;
}
```

Si, por ejemplo, para la prueba anterior se reduce el tiempo máximo por usuario y petición de 50 milisegundos a 10 milisegundos, se produciría un fallo<sup>10</sup> y aparecería el siguiente mensaje de error:

```
Maximum elapsed time exceeded! Expected 10ms, but was 16ms.
junit.framework.AssertionFailedError: Maximum elapsed time exceeded!
Expected 10ms, but was 16ms.
at com.clarkware.junitperf.TimedTest.runUntilTestCompletion
(TimedTest.java:161)
at com.clarkware.junitperf.TimedTest.run(TimedTest.java:138)
```

El mensaje indica que el tiempo esperado eran 10 milisegundos mientras que para uno de los usuarios el tiempo ha sido de 16 milisegundos. De esta forma el desarrollador sabe que debe mejorar el tiempo de ejecución de la prueba en al menos 60 milisegundos. Nótese que estos valores son orientativos ya que para garantizar un tiempo de ejecución inferior a un umbral dado es necesario jugar con márgenes que permitan compensar los diferentes estados de carga en los que se pueda encontrar el sistema en un momento dado.

#### 12.3.2.3.1. PRUEBAS DE CARGA NO ATÓMICAS

Por omisión, un `LoadTest` no fuerza la atomicidad<sup>11</sup> de la prueba cuando decora pruebas que lanzan hilos, ya sea directa o indirectamente. Este tipo de prueba de carga no atómica supone que la prueba que se decora es transaccionalmente completa cuando se devuelve el control, aunque no se espere a la finalización de los hilos.

Como regla general, las pruebas unitarias siempre deben esperar la finalización de los hilos lanzados, por ejemplo usando `Thread.join()`, para asegurar la precisión de los resultados de la prueba. Sin embargo, esto no es posible en algunas situaciones. Por ejemplo, un servidor puede lanzar un hilo para responder a una petición. Si el nuevo hilo pertenece al mismo grupo de hilos que el hilo ejecutándose para la prueba decorada, entonces una prueba de carga no atómica esperará a la finalización de los hilos lanzados directamente por la prueba de carga y el nuevo hilo se ignora.

Para concluir, una prueba de carga no atómica solo espera la finalización de los hilos lanzados directamente por la prueba de carga para simular más de un usuario concurrente.

#### 12.3.2.3.2. PRUEBAS DE CARGA ATÓMICAS

Es necesario que la prueba decorada no se trate como completa mientras no hayan terminado de ejecutarse todos los hilos. Se debe utilizar el método `setEnforceTestAtomicity(true)`, que asegure la atomicidad de la prueba. Esto hace que la prueba espere a la finalización de todos los hilos que pertenecen al mismo grupo así como los lanzados directamente por la prueba de carga para simular más de un usuario concurrente. Las pruebas de carga atómicas tratan la salida prematura de cualquier hilo como un fallo. Si un hilo termina bruscamente, todos los hilos que

---

<sup>10</sup> Obviamente, el número de milisegundos a partir del cual aparece el fallo depende de la capacidad de la máquina en la que esté funcionando la aplicación Web. Nótese asimismo que para las pruebas de rendimiento es necesario arrancar el sistema y pararlo como paso previo y posterior al proceso de prueba. Esto se realiza dentro de la tarea `performance` del documento ant localizado en la ruta `pruebas sistema software/build.xml`.

<sup>11</sup> Como se define en el procesamiento de transacciones.

pertenecen al mismo grupo se interrumpen inmediatamente. Si la prueba decorada lanza hilos que pertenecen al mismo grupo que el hilo que ejecuta la prueba decorada, la prueba de carga esperará indefinidamente a que el hilo lanzado finalice.

#### 12.3.2.4. Ejecución de las pruebas de rendimiento

A la hora de ejecutar las pruebas de rendimiento creadas con JUnitPerf, basta definir un método `suite` que devuelva un objeto de la clase `TestSuite` con el conjunto de métodos de prueba decorados que se desea ejecutar. Posteriormente es posible ejecutar la clase que contiene a ese método `suite` mediante Ant como si fuera una clase de pruebas típica. En el ejemplo siguiente se muestra el método `suite` definido en la clase `CargaTest` encargada de realizar las pruebas de carga. Es importante recordar que Ant siempre busca primero un método `suite` del que tomar los casos de prueba a ejecutar y que, en caso de no encontrarlo, utiliza el proceso de *reflection* para descubrir dinámicamente los métodos de prueba (aquellos con el prefijo `test` en versiones previas a JUnit 4.x)

```
pruebas sistema software/src/pruebasSistemaSoftware/validacion/rendimiento/CargaTest.java
```

```
public class CargaTest extends TestCase {

    public static Test suite() {

        TestSuite suite = new TestSuite();
        suite.addTest(carga10Usuarios0Retardo1Peticion());
        suite.addTest(carga10Usuarios1Retardo1Peticion());
        suite.addTest(carga10Usuarios05Retardo20Peticiones());
        suite.addTest(carga10Usuarios0Retardo20Peticiones());

        return suite;
    }

    ...
}
```

Se puede ver cómo el método `suite` simplemente agrupa todos los métodos de decoración de la clase en un objeto `TestSuite` que es el que será utilizado por Ant para la ejecución de las pruebas de carga.

## 12.4. JMeter

JMeter es una herramienta Java, incluida dentro del proyecto Jakarta, que permite realizar pruebas de rendimiento y pruebas funcionales. Inicialmente fue diseñada para realizar pruebas de estrés sobre aplicaciones Web, sin embargo, actualmente permite llevar a cabo pruebas de carga y rendimiento sobre servidores FTP o HTTP, consultas sobre bases de datos, pruebas sobre aplicaciones desarrolladas en Perl, Servlets y prácticamente cualquier otro medio. Además, posee la capacidad de realizar tanto solicitudes sencillas como secuencias de peticiones que permiten diagnosticar el comportamiento de una aplicación en condiciones de producción. En

este sentido, puede simular las funcionalidades de un navegador, o de cualquier otro cliente, siendo capaz de manipular los resultados de determinada petición y reutilizarlos para ser empleados en una nueva secuencia. JMeter también permite la ejecución de pruebas de rendimiento distribuidas entre distintas máquinas.

El componente principal de JMeter se denomina plan de pruebas (en inglés *Test Plan*<sup>12</sup>). En él se definen todos los aspectos relacionados con una prueba de carga. Por ejemplo, los parámetros empleados para una petición, el tipo de informes que se desea generar con los resultados obtenidos, etc. Un plan de pruebas se basa en el concepto de elemento y se estructura en forma de árbol. Cualquier parte o rama del árbol se puede guardar separadamente, para ser reutilizada en otras pruebas. Básicamente, un plan de pruebas consta de una lista ordenada de peticiones *Samplers*, que representa los pasos a ejecutar en el plan. Estas peticiones se organizan mediante *Controllers* que deciden cuándo ejecutar las peticiones. Algunos tipos de *Controllers* modifican el orden de ejecución de los elementos que controlan.

JMeter es extensible, y ofrece la posibilidad de incorporar nuevos *Controllers*. Además permite realizar pruebas distribuidas en distintas máquinas.

En este apartado se describe cómo configurar JMeter, los elementos disponibles para crear un plan de pruebas y cómo crear pruebas de estrés utilizando esta herramienta. No se trata de un manual exhaustivo de la herramienta, sino de una descripción ilustrada con ejemplos prácticos de los elementos disponibles más comúnmente utilizados y una serie de consejos sobre cómo puede ayudar JMeter en las pruebas de rendimiento. Una descripción más extensa de la herramienta se puede encontrar en <http://jakarta.apache.org/jmeter/usermanual>.

### 12.4.1. Instalación y configuración de JMeter

Para poder utilizar JMeter es necesario disponer de JDK 1.4 o superior y añadir el path del JDK a la variable de entorno JAVA\_HOME y el directorio bin de Java al path del sistema. La instalación de JMeter requiere los siguientes pasos:

1. Descargar el archivo<sup>13</sup> `jakarta-jmeter-2.3RC4.zip` desde la página [http://jakarta.apache.org/site/downloads/downloads\\_jmeter.cgi](http://jakarta.apache.org/site/downloads/downloads_jmeter.cgi). Este archivo contiene la versión 2.3RC4 de JMeter.
2. Verificar la integridad de los ficheros descargados utilizando PGP o MD5. Para Windows, MD5 se puede descargar desde <http://www.fourmilab.ch/md5/>. Descargar el archivo `md5.zip` y descomprimir. Ejecutar desde la línea de comandos:

```
md5 jakarta-jmeter-2.3RC4.zip
```

El resultado debe ser igual al contenido de la página <http://www.apache.org/dist/jakarta/jmeter/binaries/jakarta-jmeter-2.3RC4.zip.md5>.

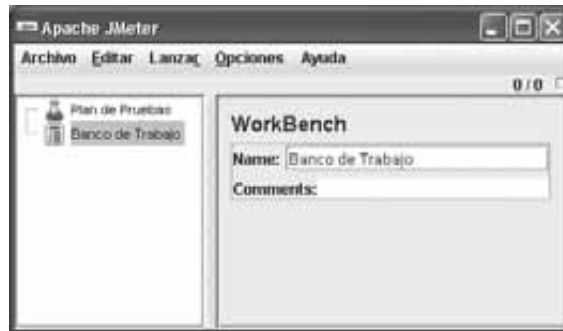
3. Descomprimir el archivo zip de JMeter en un directorio creado para ello. Por ejemplo, `C:\DESCARGAS\jakarta-jmeter-2.3RC4`. No conviene que el nombre del

<sup>12</sup> No confundir con el *Test Plan* o Plan de Pruebas global de un sistema software tal y como se vio en el Capítulo 1 de este libro.

<sup>13</sup> La versión de JMeter utilizada a lo largo de este capítulo es la 2.3RC4.

directorio de instalación de JMeter tenga espacios en blanco, ya que puede causar problemas en pruebas remotas.

4. Incluir el directorio `C:\DESCARGAS\jakarta-jmeter-2.3RC4\bin` en la variable de entorno `Path`<sup>14</sup>.
5. Ejecutar el archivo `jmeter.bat`. Con esta instrucción se arranca la herramienta y se abre la pantalla principal, que puede verse en la Figura 12.1. Esta ventana se divide en dos partes. La parte de la izquierda muestra el árbol del plan de pruebas y la parte de-  
recha contiene la definición del elemento seleccionado en el árbol.



**Figura 12.1.** Pantalla principal de JMeter.

Para configurar JMeter se deben modificar las propiedades del fichero `jmeter.properties` situado en el directorio `bin` de la instalación de JMeter. Nótese que a partir de la versión 2.2 se pueden definir propiedades adicionales en el fichero definido en la propiedad de JMeter `user.properties`. El nombre por omisión de ese fichero es `user.properties`. El fichero se carga automáticamente si está en el directorio actual. Algo similar ocurre con la propiedad de JMeter `system.properties`. Los comentarios de los ficheros `jmeter.properties`, `user.properties` y `system.properties` contienen más información sobre las propiedades que se pueden definir o modificar.

Algunas de las propiedades que se pueden definir son:

- `xml.parser`. Parser que debe usar JMeter. El valor por omisión de esta propiedad es `org.apache.xerces.parsers.SAXParser`.
- `not_in_menu`. En esta variable se especifica una lista de los componentes que no se quiere que aparezcan en los menús de JMeter. Se puede especificar los nombres de las clases o las etiquetas (la que aparece en la interfaz de JMeter).
- `user.properties`. Contiene el nombre del fichero que contiene propiedades adicionales de JMeter. Estas propiedades se añaden después del fichero inicial de propiedades.
- `system.properties`. Contiene el nombre del fichero que contiene las propiedades del sistema.

---

<sup>14</sup> Véase Apéndice A Variables de Entorno.



## 12.4.2. Elementos de un plan de pruebas

En JMeter un plan de pruebas se define mediante una lista de elementos. Los elementos disponibles son los siguientes:

- Elementos jerárquicos:
  - *Listeners* (elementos en escucha)
  - *Configuration Elements* (elementos de configuración)
  - *Post-processors* (post-procesadores)
  - *Pre-processors* (pre-procesadores)
  - *Assertions* (afirmaciones)
  - *Timers* (cronómetros)
- Elementos ordenados:
  - *Controllers* (controladores)
  - *Samplers* (agentes de pruebas)

En este apartado se exponen brevemente estos elementos. Después, mediante ejemplos, se describen las reglas de alcance y el orden de ejecución de los elementos del árbol de pruebas. Una descripción completa de todos los elementos disponibles en JMeter y de los parámetros que los definen se puede consultar en la dirección [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html). Además de la clasificación anterior, existen, entre otros, los conceptos de *ThreadGroup* imprescindible para definir un plan de pruebas, y de *Workbench*, muy útil para guardar elementos temporales.

### 12.4.2.1. ThreadGroup

Un *ThreadGroup* es el elemento inicial de cualquier plan de pruebas. Todos los elementos del plan de pruebas deben estar bajo un *ThreadGroup*. Un *ThreadGroup* controla el número de hilos que usará JMeter para ejecutar la prueba. Se puede considerar como el grupo de usuarios que se desea simular en la prueba de carga. Los parámetros que definen un *ThreadGroup* son los siguientes:

- **Number of threads.** Indica el número de peticiones concurrentes que se desean simular. Cada hilo ejecuta el plan de pruebas independientemente de otros hilos de la prueba.
- **Ramp-up period.** Es el periodo de tiempo en segundos que se desea tener en cada grupo de usuarios (*ThreadGroup*). Si por ejemplo, hay 30 hilos y el *ramp-up* es de 120 segundos, los hilos serán lanzados cada 4 segundos. El periodo *ramp-up* debe ser lo suficientemente largo para evitar un trabajo de carga demasiado costoso al principio de la prueba y lo suficientemente corto para que el último hilo empiece antes de que finalice el primero, a menos que sea esto lo deseado.
- **Forever.** Indica el número de iteraciones que se ejecuta el *ThreadGroup*. Puede ser que se decida que el plan de pruebas se ejecute indefinidamente. Por omisión se ejecuta una sola vez.
- **Scheduler.** Se define mediante un **Start Time**, un **End Time**, una **Duration** en segundos y un **Startup delay**, también en segundos. Cuando se arranca la prueba, ésta espera hasta que llegue el **Start Time**. Al final de cada ciclo, JMeter comprueba si se ha alcanzado el **End Time**. Si es así, la ejecución se detiene, si no se continúa hasta

que se termine o se alcance el *End Time*. Los campos de *Duration* y *Startup delay* indican el tiempo de ejecución disponible para ejecutar el *ThreadGroup* y el retardo entre la ejecución de los hilos respectivamente.

### 12.4.2.2. Controllers

Los *Controllers* son los elementos que envían las peticiones y las organizan dentro del plan de pruebas. En JMeter existen dos tipos de *Controllers*:

- *Samplers*. Solicitan el envío de peticiones a un servidor. Por ejemplo, para una petición http, se añade al plan de pruebas un *HTTP Request Sampler*. La petición se puede configurar añadiendo, además, uno o varios *Configuration Elements*.
- *Logical Controllers*. Sirven para ayudar a JMeter a decidir cuándo se envían las peticiones. Por ejemplo, se puede usar un *Interleave Logic Controller* para alternar dos *HTTP Request Samplers*. O un controlador *If Controller* que permite decidir si realizar o no una petición http en función de una condición. Cada controlador puede tener uno o más *Default Elements*.

#### 12.4.2.2.1. SAMPLERS

Un *Sampler* solicita el envío de peticiones a un servidor. Los *Samplers* de JMeter son los siguientes:

- *FTP Request*
- *HTTP Request*
- *JDBC Request*
- *Java object request*
- *LDAP Request*
- *SOAP/XML-RPC Request*
- *WebService (SOAP) Request*

Cada *Sampler* dispone de un conjunto de propiedades para su configuración (para más detalles consultar [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html)). También se pueden añadir *Configuration Elements*<sup>15</sup> para definir el *Sampler*.

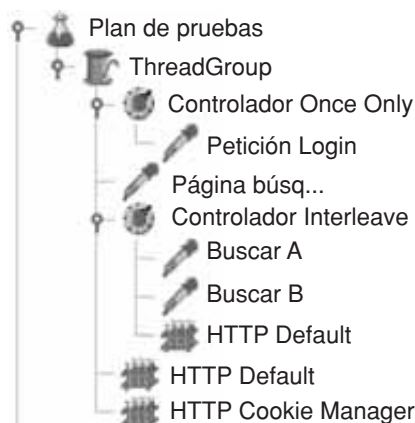
Las peticiones serán enviadas en el orden en que aparecen en el árbol de pruebas. Si se solicitan varias peticiones del mismo tipo al mismo servidor, se recomienda el uso de un *Defaults Configuration Element* que define los parámetros comunes de todas las peticiones. En un *Controller* puede haber varios elementos *Default*.

Para poder ver los resultados de las peticiones hay que añadir un *Listener* al *ThreadGroup*. Si interesa validar la respuesta a la petición se añade una *Assertion* al *Controller*. Por ejemplo, en pruebas de estrés de una aplicación Web, se pueden añadir aserciones que comprueben, por ejemplo, ciertas etiquetas HTML o la presencia de ciertas cadenas de caracteres de errores. Estas aserciones se construyen con expresiones regulares.

<sup>15</sup> Véase el Apartado 12.4.2.6.

#### 12.4.2.2.2. LOGIC CONTROLLERS

Con un *Logic Controller* se indica a JMeter cuándo enviar las peticiones. Dentro de un *Logic Controller* se pueden definir: *Samplers* (peticiones), *Configuration Elements* y otros *Logic Controllers*. Los *Logic Controller* pueden cambiar el orden de las peticiones, las peticiones mismas, repetir las peticiones, etc. Suponer el ejemplo siguiente:



**Figura 12.2.** Orden de ejecución usando Logic Controllers.

En esta prueba se usa un *Controlador Once Only*. Este *Controller* hace que la *Petición login* se realice la primera vez, pero no en las siguientes iteraciones. Después del *login*, el siguiente *Sampler* carga una página de búsqueda, *Página búsqueda*. Esta petición no se ve afectada por ningún *Logic Controller*.

Después de cargar la página de búsqueda se procede a hacer dos búsquedas, A y B. Después de cada búsqueda se quiere recargar la página de búsqueda. Esto se logra usando el *Controlador Interleave*, que pasa cada uno de sus hijos de forma alternativa. Al final de la rama *Controlador Interleave* se ha definido un *HTTP Default*. Se trata de un elemento que permite definir una sola vez propiedades comunes a todas las peticiones de una misma rama. Es decir, cuando se ejecuta la petición A, datos como por ejemplo, el servidor, el puerto, la ruta, etc se buscan en el elemento *HTTP Default*. Lo mismo ocurre con la petición B. El uso de *HTTP Request Defaults* hace más rápido y cómodo definir el plan de pruebas.

El siguiente elemento del árbol es otra *HTTP Request Defaults* añadida al *ThreadGroup*. En este caso afecta a todas las peticiones del *ThreadGroup* puesto que cuelga directamente de él. Es muy útil dejar en blanco la información del servidor de los elementos *HTTP Sampler* y definirlo en un elemento que defina datos por omisión. De este modo, se puede cambiar de servidor con solo cambiar un campo del plan de pruebas.

Por último, se tiene un elemento *HTTP Cookie Manager*. Se debería añadir un elemento de este tipo a todas las pruebas web. Si no es así, JMeter ignora las cookies. Añadiéndolo al nivel del *ThreadGroup* se asegura que todas las peticiones compartirán las mismas cookies.

La lista completa de elementos de configuración se puede consultar en la página [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html).

### 12.4.2.3. Listeners

Un *Listener* da acceso a la información generada durante la ejecución de la prueba. Existen varios *Listeners* que muestran esa información de diferentes formas. Algunos ejemplos son *Graph Results* que muestra en un gráfico los tiempos de respuesta o *View Results Tree* que muestra las peticiones y las respuestas, estas últimas en formato HTML o XML. Un *Listener* puede direccionar los resultados a un fichero para su uso posterior. Cada *Listener* tiene un campo para indicar el nombre del fichero que almacenará los datos. También se pueden elegir los campos que se quieren guardar y el formato. Los *Listeners* se pueden añadir en cualquier parte del plan de pruebas y solo recogerán datos de los elementos de su mismo nivel o inferior.

### 12.4.2.4. Timers

Por omisión, JMeter lanza los hilos sin pausas entre peticiones. Añadiendo un *Timer* al *ThreadGroup* se establece el intervalo de tiempo entre las peticiones que lanza un hilo.

### 12.4.2.5. Assertions

Las *Assertions* permiten comprobar si la aplicación en pruebas da los resultados esperados. Por ejemplo, se puede comprobar si una respuesta a una consulta contiene un texto o código determinado. Las *Assertions* se pueden añadir a cualquier *Sampler*. Para ver el resultado de las aserciones se debe utilizar un *Assertion Listener* al *ThreadGroup*. Las aserciones que fallan se muestran también en *Tree View Listener* y *Table Listener*.

Un ejemplo de *Assertion* es *Response Assertion*. Con esta aserción se puede comprobar la respuesta de la petición. Puede comprobarse el texto, la URL, el código o el mensaje de respuesta, e indicar si coincide con una serie de patrones o no. Por ejemplo se puede comprobar si el código de respuesta es 200, que corresponde a una página servida correctamente.

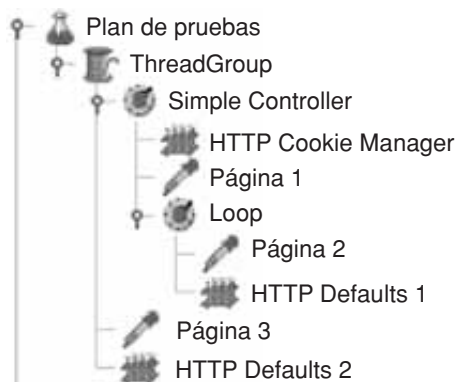
La lista completa de *Assertions* disponibles en JMeter se puede consultar en la página [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html).

### 12.4.2.6. Configuration Elements

Un *Configuration Element* puede añadir o modificar peticiones. Un *Configuration Element* es accesible desde la rama del árbol donde se pone el elemento. Por ejemplo, si se pone un *HTTP Cookie Manager* en un *Simple Logic Controller*, el *Cookie Manager* solo será accesible para los *Controllers HTTP Request* que se ponen dentro del *Simple Logic Controller*. En el ejemplo de la Figura 12.3 el elemento *HTTP Cookie Manager* es accesible para las peticiones *Página 1* y *Página 2*, pero no para *Página 3*.

El elemento *HTTP Cookie Manager* sirve para simular las cookies. Las aplicaciones Web suelen utilizar *Cookies* para guardar información sobre la navegación del usuario. Si se añade este elemento al *ThreadGroup* se pueden controlar las cookies, pudiendo borrar la cookie en cada iteración de la prueba o establecer valores para las cookies.

Un *Configuration Element* de una rama del árbol tiene mayor precedencia que el mismo elemento en una rama padre. En el ejemplo anterior, se definen dos elementos *HTTP Request Defaults: HTTP Defaults 1* y *HTTP Defaults 2*. Puesto que *HTTP Defaults 1* está dentro de un *Loop*, solo *Página 2* puede acceder a *HTTP Defaults 2*. Las otras peticiones



**Figura 12.3.** Accesibilidad de los elementos de configuración.

HTTP usarán *HTTP Defaults 2*, puesto que está situado en el *ThreadGroup*, que es el padre de las demás ramas.

El elemento *HTTP Request Default* sirve para definir las propiedades comunes a varias peticiones representadas por elementos *HTTP Request*. De esta forma, cuando se definen las distintas peticiones, no será necesario rellenar todos los campos de información, ya que heredarán las propiedades definidas aquí.

Para obtener una lista completa de los elementos de configuración disponibles en JMeter se puede consultar la página [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html).

#### 12.4.2.7. Pre-Processor Elements

Un *Pre-Processor* realiza alguna acción antes de ejecutar el *Sampler* al que se ha adjuntado. A menudo se usan para establecer el entorno de la petición antes de ejecutarla o actualizar las variables que no se extraen de un texto de respuesta.

Una lista completa de los *Pre-Processors* disponibles en JMeter se puede encontrar en la dirección [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html).

#### 12.4.2.8. Post-Processor Elements

Un *Post-Processor* realiza alguna acción después de ejecutar el *Sampler* al que se ha adjuntado. Se suele usar para procesar datos de respuesta o extraer valores de la misma.

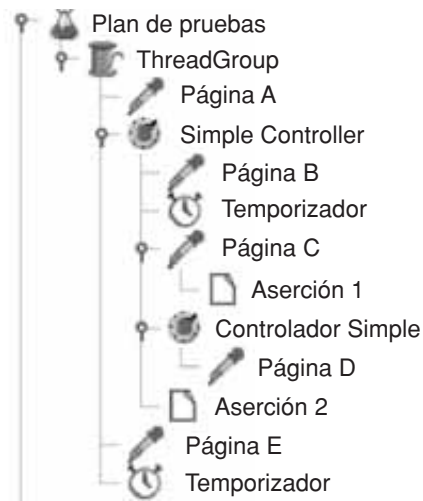
Una lista completa de los *Post-Processors* disponibles en JMeter se puede encontrar en la dirección [http://jakarta.apache.org/jmeter/usermanual/component\\_reference.html](http://jakarta.apache.org/jmeter/usermanual/component_reference.html).

#### 12.4.2.9. Reglas de alcance

Ya se ha visto en el Apartado 12.4.2 que algunos elementos de JMeter son jerárquicos y otros ordenados. Cuando se crea un plan de pruebas se crea una lista ordenada de peticiones (usando *Samplers*) que representa los pasos que se van a ejecutar. Estas peticiones se organizan en

*Controllers* que pueden afectar al orden de ejecución de sus subelementos. Además, las peticiones se pueden ver afectadas por elementos jerárquicos que, dependiendo dónde se definan, afectan a unas peticiones o a otras. En este apartado se verá cuál es el alcance de cada elemento de JMeter según la rama del árbol de pruebas donde se defina.

Si, por ejemplo, se añaden aserciones, estas afectan a los elementos de la rama en que se define la aserción. Si el padre de una *Assertion* es una petición, la aserción se aplica a la petición. Si el padre es un *Controller* afecta a todas las peticiones que son descendientes de ese *Controller*. En el ejemplo de la Figura 12.4, Aserción 1 afecta a la petición Página C, mientras que la Aserción 2 afecta a las peticiones Página B, Página C y Página D.



**Figura 12.4.** Alcance de elementos jerárquicos en un plan de pruebas.

Los temporizadores afectan a las peticiones descendientes del elemento en que están definidos. En el ejemplo de la Figura 12.4, Timer 1 afecta a las peticiones Página B, Página C y Página D mientras que Timer 2 afecta a todas las peticiones.

Los elementos de configuración *Header Manager*, *Cookie Manager* y *Authorization manager* se tratan de forma diferente que los elementos de configuración por omisión (por ejemplo *HTTP Request Defaults*). Los datos de un elemento de configuración de valores por omisión se mezclan en un conjunto de valores a los que el *Sampler* puede acceder. Sin embargo, los valores de varios *Managers* no se mezclan. Si hay más de un *Manager* afectando a un *Sampler*, solo se usa un *Manager*, pero no se puede especificar cuál se usará.

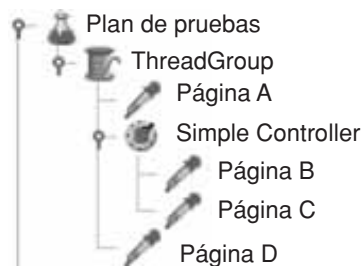
#### 12.4.2.10. Orden de ejecución

El orden de ejecución de los elementos de JMeter es el que se muestra a continuación:

1. Pre-Processors.
2. Timers.
3. Sampler.

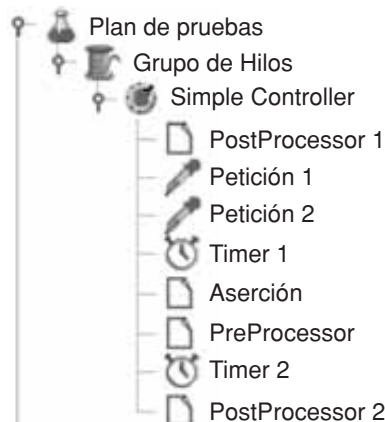
4. Post-Processors (a menos que SampleResult sea null).
5. Assertions (a menos que SampleResult is null).
6. Listeners (a menos que SampleResult is null).

Para elementos del mismo tipo, el orden de ejecución es el de definición en el plan de pruebas. En el ejemplo de la Figura 12.5 las peticiones se ejecutarán en el orden en que se han añadido: Página A, Página B, Página C, Página D. Pero este orden de ejecución se puede modificar con el uso de *Controllers* distintos al definido en este ejemplo. En la Figura 12.3, se vio el caso de un *Loop Controller*.



**Figura 12.5.** Ejemplo de plan de pruebas.

Hay que recordar que *Timers*, *Assertions*, *Pre-* y *Post-Processors* solo se procesan si se aplican a un *Sampler*. *Logic Controllers* y *Samplers* se procesan en el orden en que aparecen en el árbol del plan de pruebas. Otros elementos se procesan según las reglas de alcance y el tipo de elemento. En el ejemplo de la Figura 12.6, las dos peticiones están en la misma rama y afectadas por los mismos elementos jerárquicos.



**Figura 12.6.** Orden de ejecución de un plan de pruebas.

Por tanto, el orden de ejecución sería:

```
Pre-Processor 1
Timer 1
Timer 2
Sampler 1

Post-Processor 1
Post-Processor 2
Assertion
Pre-Processor 1
Timer 1
Timer 2
Sampler 2
Post-Processor 1
Post-Processor 2
```

#### 12.4.2.11. *WorkBench*

Un *WorkBench* proporciona un lugar para almacenar temporalmente elementos del plan de pruebas. Cuando se guarda un plan de pruebas, los elementos del *Workbench* no se almacenan. Si se desea, se puede guardar de forma independiente.

Algunos elementos como *HTTP Proxy Server* y *HTTP Mirror Server* solo están disponibles como parte de un *Workbench*.

### 12.4.3. Creando pruebas con JMeter

Para crear un plan de pruebas se crea una lista ordenada de peticiones utilizando *Samplers* que representa los pasos a ejecutar en el plan. Normalmente, las peticiones se organizan dentro de *Controllers*. Algunos tipos de *Controllers* modifican el orden de ejecución de los elementos que controlan.

Para ilustrar cómo se define un plan de pruebas con JMeter, se describirán algunos ejemplos.

#### 12.4.3.1. Una prueba simple

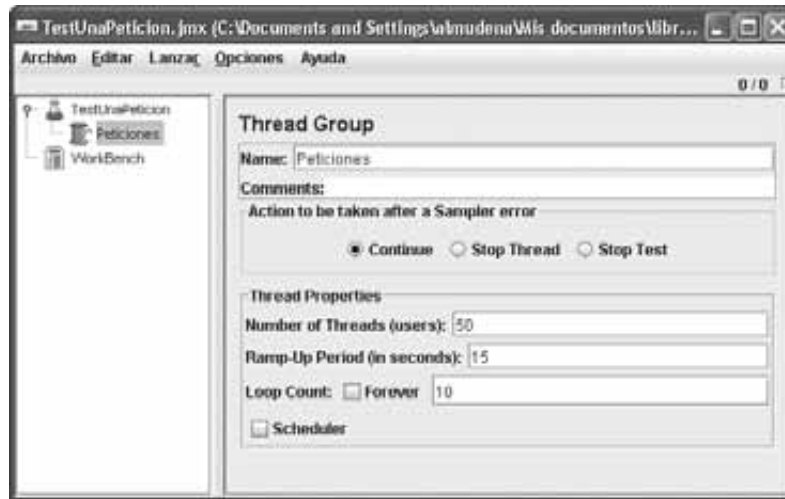
Este ejemplo será una prueba de una petición al servidor de tráfico descrito en el Apéndice B de este libro. Se simularán cincuenta usuarios solicitando información sobre el tramo con más tráfico. Los pasos a seguir son los siguientes:

1. Para crear un plan de pruebas se selecciona en la parte izquierda de la ventana el icono **Test Plan**. En la parte derecha aparecen las opciones para definir ese plan de pruebas, entre ellas el nombre. En el ejemplo *TestCargaPeticiones*, el objeto **Test Plan** dispone de una opción llamada **Functional Test Mode**. Si se selecciona, JMeter grabará los datos devueltos desde el servidor para cada petición. Si hay seleccionado un fichero en los *Listeners* de la prueba, esos datos se escribirán en ese fichero. Esto es útil cuando se hace una pequeña ejecución para asegurarse de la correcta configuración de JMeter. Pero incide negativamente en el rendimiento de JMeter. Por esta ra-



zón se debe deshabilitar esta opción cuando se hacen pruebas de estrés. En el Listener utilizado se puede decidir qué campos salvar mediante la opción *Configuration*.

2. Todo plan de pruebas debe tener un elemento *ThreadGroup* que representa el grupo de usuarios que se simularán en la prueba. Para añadir uno a *TestCargaPeticones* se selecciona el icono *TestPlan* y en el menú contextual se selecciona la opción *Add/Thread Group*.

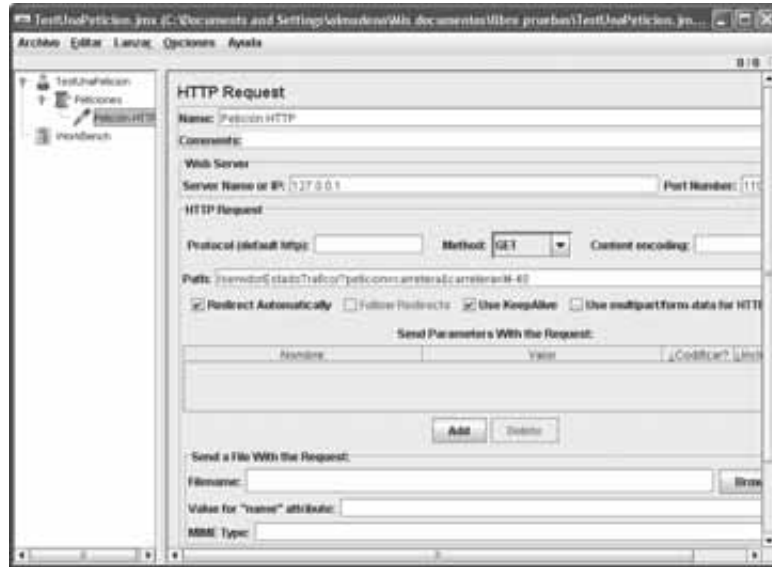


**Figura 12.7.** Definición de un *ThreadGroup*.

3. Seleccionando el icono *ThreadGroup*, la parte derecha de la ventana muestra las opciones para definirlo. En el Apartado 12.4.2.1 se describen dichas opciones, que, para el ejemplo en creación, toman los valores de la Figura 12.7.
4. Se añaden al *ThreadGroup* los elementos que representan las peticiones que harán los usuarios del *ThreadGroup*. Para ello se selecciona el icono *ThreadGroup* en la parte izquierda de la ventana y en el menú contextual se selecciona la opción *Add/Sampler* y el tipo de petición que se va a simular, por ejemplo, *HTTP Request*. Igual que antes, en la parte derecha de la ventana aparece la definición del elemento seleccionado. La Figura 12.8 muestra los valores que definen la petición añadida.

El significado de estos parámetros es el siguiente:

- **Server Name or IP.** Dirección I.P o nombre de red del servidor donde se realiza la prueba de carga. Se utilizará 127.0.0.1 para indicar un servidor local.
  - **port number.** Puerto TCP de operación del servidor, será empleado el 80 o el puerto que corresponda según esté configurado el servidor.
  - **path.** Ruta de acceso de la aplicación de servidor para realizar la prueba. Se define */index.html*, que es la página principal habitual de todo sitio Web.
5. Para que JMeter muestre los resultados de la prueba se añade un *Listener* seleccionando el icono *HTTP Request* en el menú contextual y ejecutando las opciones *Add/Listener/View Results in Table* y *Add/Listener/Graph Results*. JMeter generará una tabla y un gráfico para mostrar los resultados de la simulación.



**Figura 12.8.** Definición de una petición HTTP.

- Finalmente, se guarda el plan de pruebas y se ejecuta con la opción **Run/Star**. Si se selecciona el icono **Graph Results** mientras se ejecuta la prueba, se ve cómo se construye el gráfico de resultados.

La información sobre errores producidos al ejecutar un plan de pruebas, se almacena en el fichero `jmeter.log` o en el resultado de la prueba. Algunos errores, como por ejemplo **HTTP 404 - file not found**, no se incluyen en `jmeter.log`, sino que se almacenan como atributos del *Sampler* resultado, a los que se puede acceder mediante *Listeners*.

#### 12.4.3.2. Uso de parámetros en las peticiones

La prueba que se acaba de definir realiza peticiones idénticas en todas las iteraciones. Esta prueba se puede mejorar aproximándola a una situación más real. Con JMeter se pueden definir pruebas para que cada petición tenga parámetros únicos por usuario. Un ejemplo típico de esto es la prueba que simula varios usuarios autenticándose a la vez en una aplicación. En este caso cada usuario emplea un *usuario* y *contraseña* distintos.

A continuación se ilustra paso a paso cómo definir esta prueba con JMeter.

- La definición de los usuarios con sus respectivos parámetros de autenticación se hace en el fichero `users.xml`. Este fichero se debe colocar en el directorio `bin` de JMeter. Por ejemplo, el contenido de este fichero puede ser el siguiente:

```
<?xml version="1.0"?>
<!DOCTYPE allthreads SYSTEM "users.dtd">

<allthreads>

<thread>
```

```

    <parameter>
      <paramname>usuario</paramname>
      <paramvalue>ialarcon</paramvalue>
    </parameter>
    <parameter>
      <paramname>contraseña</paramname>
      <paramvalue>azul</paramvalue>
    </parameter>
  </thread>

  <thread>
    <parameter>
      <paramname>usuario</paramname>
      <paramvalue>asierra</paramvalue>
    </parameter>
    <parameter>
      <paramname>contraseña</paramname>
      <paramvalue>amarillo</paramvalue>
    </parameter>
  </thread>

  ...
</allthreads>

```

El fichero se inicia con la etiqueta `<allthread>` que anida estructuras `<thread>`. Cada thread representa una petición. A su vez, dentro de cada elemento `<thread>` se definen los elementos `<parameter>` que incluyen las etiquetas `<paramname>` y `<paramvalues>` para especificar el nombre y valor de los parámetros utilizados. En el fichero del ejemplo se incluyen dos peticiones.

- Después de haber creado el fichero, se procede a crear el plan de pruebas, definiendo el *ThreadGroup* con un `Number of threads` igual al número de hilos definidos en el fichero `users.xml`.
- Una vez definido el grupo de usuarios, se añade un *Sampler* de tipo *HTTP Request al ThreadGroup*. Ahora es necesario asociar el grupo de usuarios definidos con las peticiones. Para ellos, se selecciona el elemento `HTTP Request` y en el menú contextual se selecciona la opción `Add/Pre Processors/HTTP User Parameter Modifier`. Se observará que, por omisión, los datos se leen del archivo `users.xml`. Después se termina de definir la petición `HTTP Request`:
  - `Server Name or IP`: 127.0.0.1 para indicar un servidor local.
  - `Port Number`: 80.
  - `HTTP Request/Method`: POST, ya que el proceso a simular es el de autenticación.
  - `Path`: /login.jsp.
  - `Send Parameters With the Request`: con el botón Add se agregan, en la tabla situada encima del botón los parámetros que serán enviados por solicitud como fueron definidos en `users.xml`. Solo hay que definir sus nombres, los valores son tomados automáticamente del archivo XML.
- Para ver los resultados de la simulación se añade un *Listener*: seleccionar el icono de `HTTP Request` y seleccionar la opción `Add/Listener/Graph Results` en el

menú contextual. Si también se desean los resultados en forma de tabla se añadirá también *View Results in Table*.

Con esto quedaría definido el plan de pruebas y se podría ejecutar.

### 12.4.3.3. Una prueba con varias peticiones

Probar que todos los usuarios se registren a la vez en una aplicación no es una prueba muy realista. Lo habitual es que cada uno de ellos haga un tipo de petición diferente.

Siguiendo con el servidor de tráfico, se puede definir un plan de pruebas que incluya 50 usuarios, de forma que cada uno haga varias peticiones diferentes. Por ejemplo, se pueden mezclar peticiones del estado de una carretera, de los tramos con más tráfico, del clima, etc. En la Figura 12.9 se muestra el árbol de peticiones para este plan de pruebas. En la parte de la derecha se muestra el *ThreadGroup* definido.

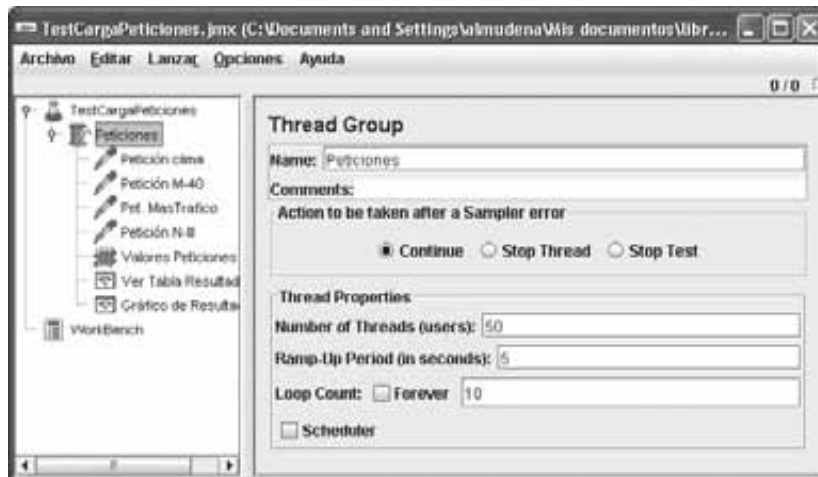


Figura 12.9. Definición del *ThreadGroup* para el plan de pruebas *TestCargaPeticiones*.

Puesto que todas las peticiones se hacen sobre el mismo servidor, se usará un elemento del tipo *HTTP Request Default*, que en el ejemplo se denomina *Valores Peticiones HTTP*, para facilitar la definición del plan. La definición de este elemento se muestra en la Figura 12.10.

Los *Listeners* *Ver Tabla Resultados* y *Gráfico de Resultados*, permite analizar el resultado de la prueba. Por ejemplo, en la Figura 12.11 se muestran esos resultados en forma de tabla. Esta tabla informa del momento de arranque de cada hilo, la petición que se ejecutó en dicho hilo, el tiempo que se ha tardado en satisfacer dicho hilo y si la petición se ejecutó correctamente o no. En el ejemplo, todas las peticiones se ejecutaron bien. En caso contrario aparecería un icono amarillo con una admiración cerrada. En la columna *Tiempo de Muestra* indica, en milisegundos, el tiempo de respuesta para cada *thread*. Para ver como se comporta el sistema cuando aumenta el número de usuarios, sería suficiente cambiar el número de hilos en el *ThreadGroup* o el número de veces que se repite cada hilo.



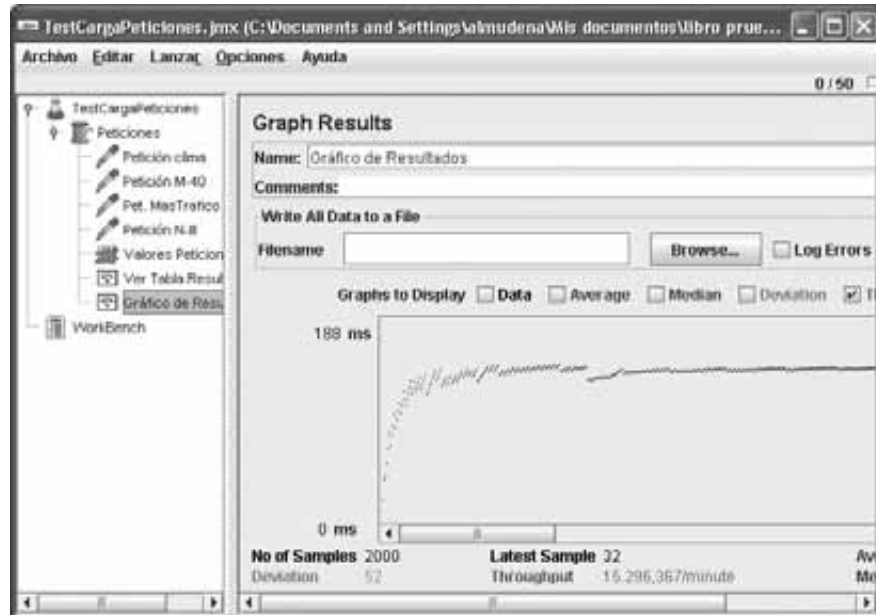
Figura 12.10. El elemento Valores Peticones HTTP.

Sample #	Start Time	Thread Name	Label	Sam	Status
1	07:54:25.640	Peticones 1-3	Petición clima	688	
2	07:54:25.750	Peticones 1-4	Petición clima	609	
3	07:54:25.437	Peticones 1-1	Petición clima	938	
4	07:54:25.943	Peticones 1-5	Petición clima	532	
5	07:54:25.546	Peticones 1-2	Petición clima	844	
6	07:54:25.937	Peticones 1-6	Petición clima	453	
7	07:54:26.046	Peticones 1-7	Petición clima	344	
8	07:54:26.140	Peticones 1-8	Petición clima	266	
9	07:54:26.250	Peticones 1-9	Petición clima	156	
10	07:54:26.343	Peticones 1-10	Petición clima	78	
11	07:54:26.328	Peticones 1-3	Petición M-40	109	
12	07:54:26.359	Peticones 1-4	Petición M-40	79	

Figura 12.11. Resultados de la ejecución del plan de pruebas TestCargaPeticones.

Otro modo de ver el resultado de la ejecución del plan de pruebas es en forma de gráfico. Por ejemplo, la Figura 12.12 muestra el rendimiento asociado a la ejecución del plan de pruebas. Al principio, cuando no hay demasiadas peticiones la curva de rendimiento sube. Llegado un cierto número de ellas, aunque la curva desciende, el rendimiento sigue siendo bueno. De hecho se atienden 15.296 peticiones por minuto (este dato se muestra en verde en la parte inferior de la

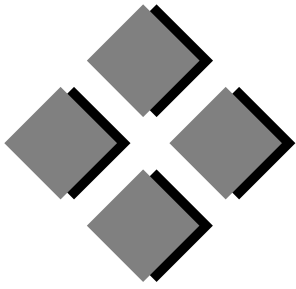
ventana). Por supuesto, el rendimiento de una aplicación siempre depende de la máquina en la que se esté ejecutando el sistema. A este respecto se ha de tener en cuenta el entorno hardware en el que se supone que el sistema va a estar operativo tal y como haya sido descrito en el Documento de Especificación de Requisitos.



**Figura 12.12.** Gráfico de resultados mostrando el rendimiento de la aplicación.

## 12.5. Bibliografía

- Pfleeger S. L. y Atlee, J. M.: *Software Engineering*, 3.<sup>a</sup> edición, Prentice Hall, 2006.
- <http://jfunc.sourceforge.net/>.
- <http://clarkware.com/software/JUnitPerf.html>.
- <http://jakarta.apache.org/jmeter/>.
- Clark, M.: «Continuous Performance Testing With JUnitPerf», *JavaProNews*, 2003. <http://www.javapronews.com/javapronews-47-20030721ContinuousPerformanceTesting-withJUnitPerf.html>.



## Apéndice A

# Variables de entorno

### SUMARIO

A.1 Linux

A.2 Windows

Las variables de entorno son variables que contienen información general que el sistema operativo u otros procesos utilizan para la localización de recursos necesarios en un determinado contexto de ejecución. Por ejemplo la variable PATH, y esto vale para sistemas basados en Linux y Windows, contiene una lista con los directorios que el sistema operativo utiliza para buscar los archivos ejecutables asociados a comandos introducidos desde una ventana de comandos.

Para aquellos usuarios no familiarizados, a continuación se introduce el uso de variables de entorno tanto en sistemas basados en Linux como en Windows.

## A.1 Linux

En Linux existen una serie de comandos de consola muy sencillos que permiten trabajar con variables de entorno. A continuación, se explica paso a paso cómo utilizar estos comandos para realizar las tareas más frecuentes. El primer paso siempre es abrir una ventana de terminal.

### a) Visualizar las variables de entorno definidas en el sistema

```
> NOMBRE_DE_VARIABLE
```

por ejemplo:

```
> env
```

**b) Ver el valor de una variable de entorno en particular**

```
> echo $NOMBRE_DE_VARIABLE
```

por ejemplo:

```
> echo $PATH
```

**c) Crear una variable de entorno**

```
> export NOMBRE_VARIABLE=valor_variable
```

por ejemplo:

```
> export JAVA_HOME=C:\Archivos de programa\Java\jdk1.5.0_08
```

Este comando se ha de utilizar con mucho cuidado ya que crear una variable de entorno con el nombre de una variable que ya existe produciría que aquella fuese reemplazada por esta, con la consiguiente pérdida de información.

**d) Modificar una variable de entorno**

En este caso el procedimiento es idéntico al de creación, solo que el antiguo valor de la variable será sustituido por el nuevo. Se puede entender como un caso particular de creación de una variable en el que la variable ya existe.

**e) Concatenar un valor a una variable de entorno**

Se utiliza igualmente el comando export, pero añadiendo el valor actual de la variable como primer elemento del nuevo valor.

```
> export NOMBRE_VARIABLE=${NOMBRE_VARIABLE}VALOR_A_AÑADIR
```

por ejemplo:

```
> export PATH=${PATH}:C:\Archivos de programa\Java\
jdk1.5.0_08\bin
```

## A.2 Windows

A la hora de trabajar con variables de entorno en Windows se han de seguir los siguientes pasos:

1. Hacer clic con el botón derecho del ratón sobre el icono de *MiPC* situado en el escritorio. Se abrirá un menú desplegable del que se seleccionará la opción *Propiedades* haciendo clic con el botón izquierdo. En ese momento se abrirá una ventana de diálogo con el título *Propiedades del Sistema*.
2. Dentro de la ventana de diálogo se ha de seleccionar la pestaña avanzado para, posteriormente, hacer clic sobre *Variables de entorno de forma* para que un nuevo cuadro de diálogo se abra con el mismo nombre.
3. Desde este cuadro de diálogo ya es posible crear, cambiar o eliminar una variable de entorno.



A continuación se describe cada caso por separado:

**a) Creación de una variable de entorno**

El primer paso a la hora de crear una variable de entorno en Windows es decidir qué tipo de variable se necesita. Existen dos posibilidades, variables de sistema y variables de usuario. A continuación se describen las diferencias entre ellas:

- Variables de sistema: están siempre definidas sin importar cuál es el usuario que esté logueado en el sistema. Son por tanto comunes a todos los usuarios. Por motivos de seguridad este tipo de variables solo pueden ser creadas, modificadas o borradas por usuarios con privilegios de administrador; aunque pueden ser leídas por cualquier usuario.
- Variables de usuario: estas variables se definen para cada usuario por lo que solo son visibles por el usuario que las ha creado. A la hora de evitar conflictos entre variables, es importante tener en cuenta que las variables de usuario siempre tienen precedencia sobre las variables de sistema. Es decir, si existe una variable de sistema y una de usuario con el mismo nombre, el usuario que definió esta última, no verá la de sistema.

En términos generales parece lógico definir una variable como variable de sistema en aquellos casos en los que la variable responde a una necesidad común de todos los usuarios. En caso contrario, se definirá una variable de usuario. De esta forma, cada usuario podrá asignarle valor convenientemente sin interferir en el trabajo de los demás.

Los pasos a realizar son los siguientes:

1. Dentro del cuadro de diálogo *Variables de entorno* hacer clic sobre el botón Nuevo; aparecerá un cuadro de diálogo en el que se debe introducir el nombre de la variable así como su valor.
2. Una vez introducida la información necesaria simplemente pulsar Aceptar.

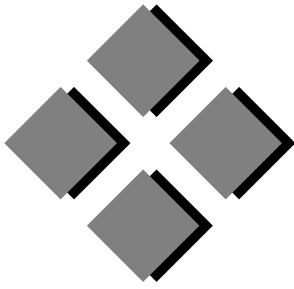
**b) Modificar una variable de entorno**

Simplemente seleccionar la variable de la lista y hacer clic sobre el botón modificar, aparecerá un cuadro de diálogo desde el cual cambiar su valor.

**c) Eliminar una variable de entorno**

Se ha de seleccionar la variable a eliminar y hacer clic sobre el botón Eliminar.





## Apéndice B

# Sistema a probar

---

### SUMARIO

<b>B.1</b> Descripción general del sistema	<b>B.4</b> Características del sistema y su relevancia en la pruebas de software
<b>B.2</b> Arquitectura del sistema	<b>B.5</b> Arquitectura interna del sistema
<b>B.3</b> Configuración del sistema	<b>B.6</b> Documento de especificación de requisitos software

Como complemento a la información contenida en las páginas de este libro se ha considerado de gran utilidad el desarrollo de un sistema software. El objetivo de este sistema es presentar al lector una aplicación en la que se muestre de una forma práctica los diferentes procedimientos y herramientas de prueba que se describen a lo largo de este libro. Este sistema software, llamado Servidor del Estado del Trafico (en adelante será también referido como SET) pretende ser una versión manejable y rápidamente asimilable de lo que sería un sistema software real. Presenta la funcionalidad, obviamente simplificada por razones prácticas, que se puede observar en cualquier sistema software real equivalente. De esta forma, el conjunto de pruebas a realizar sobre este sistema, en el contexto de un plan de pruebas convencional, se puede entender como un puzzle, del que para cada una de sus piezas, entendidas como técnicas o procedimientos de prueba complementarios, existe un capítulo correspondiente en el libro, que describe como construir la pieza y como integrarla con las demás.

Cada uno de los capítulos del libro presenta al lector procedimientos, técnicas, herramientas y recomendaciones sobre cómo realizar y automatizar pruebas sobre un sistema software genérico. No obstante, los fragmentos de código fuente proporcionados a modo de ejemplo a lo

largo de este libro están basados, prácticamente en su totalidad, en la prueba de un sistema software único y completo, que es el sistema SET. El objetivo de este apéndice es describir este sistema y familiarizar al lector con el mismo.

Aunque los ejemplos contenidos en este libro siempre tratan de ser autodescriptivos y autocontenidos, no cabe duda de que integrar dichos ejemplos en un todo, entendido como el sistema software completo al que pertenecen, es de gran ayuda para comprender el enfoque global de la prueba y los objetivos últimos de la misma. Por este motivo, inicialmente, se recomienda al lector echar un vistazo a la descripción del sistema SET para, de esta forma, obtener una idea general sobre su arquitectura y funcionalidad. Progresivamente, y a medida que el lector profundice en los contenidos de este libro, podrá adentrarse en los detalles de las distintas partes de las que se compone el sistema y realizar una lectura más detallada de aquellas que le interesen.

## B.1 Descripción general del sistema

La aplicación utilizada en el libro como ejemplo para ilustrar el desarrollo de las pruebas funciona en modo servidor (aunque como se verá, también desempeña el rol de cliente) y está escrita en el lenguaje Java. Dicha aplicación proporciona información en tiempo real en formato XML acerca del estado del tráfico, climatología, obras, etc. de carreteras y sus tramos.

El funcionamiento del servidor es muy sencillo, básicamente se encarga de recoger la información del estado del tráfico, que genera otro sistema llamado Sistema de Observación y Registro de Tráfico (SORT), y de publicarla mediante HTTP a través de un puerto determinado. De esta forma, la información puede ser accedida por potenciales usuarios de las carreteras mediante un navegador Web convencional como Mozilla FireFox o Internet Explorer. El SORT es un sistema externo e independiente que se encarga de medir en tiempo real el tráfico en las carreteras, así como las condiciones meteorológicas de las mismas, etc. Toda esa información la almacena en un fichero en formato texto<sup>1</sup> que va cambiando en tiempo real. Este fichero actúa de interfaz entre el sistema objetivo de las pruebas (SET) y el SORT, por tanto la especificación de su formato es todo lo que interesa conocer acerca de este último. Este fichero se compone de una serie de registros, cada uno de los cuales contiene información asociada a una carretera en particular y sus tramos. A continuación se muestra un ejemplo de registro, que es básicamente una serie de campos separados mediante el carácter “|”:

```
N-401|17:37:22|8/2/2007|Nieve|Si|0|9|3|0|Trafico lento|Sin
accidentes|10|18|3|2|Retenciones|Camion volcado en mitad de la
calzada|19|27|3|1|Trafico fluido|Turismo averiado en el
arcen|28|36|3|1|Retenciones|Turismo averiado en el arcen
```

En este registro aparece información sobre una carretera y sus cuatro tramos. Los primeros 5 campos del registro contienen la información asociada a la carretera, mientras que los restantes 24 corresponden a sus tramos, es decir, existen 6 campos de información por tramo de carretera. En lo que respecta a la información de la carretera, el primer campo es el nombre (“N-401”), el segundo campo es la hora de captura del registro de información<sup>2</sup>

<sup>1</sup> Un ejemplo completo de este tipo de ficheros se encuentra en la ruta ./sistema software/data/registros.txt

<sup>2</sup> Puesto que se trata de un sistema que funciona en tiempo real existen múltiples observaciones y capturas de datos para cada carretera y sus tramos.

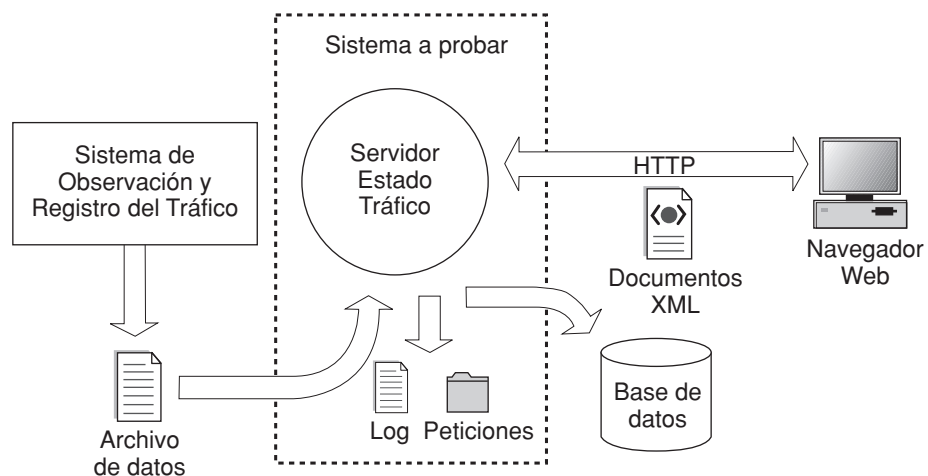
("17:37:22"), el tercer campo es la fecha de captura del registro de información ("8/2/2007"), el cuarto campo contiene las condiciones meteorológicas de la carretera ("Nieve") y el quinto y último campo contiene información acerca de la presencia de obras en la carretera ("Si"). Por lo que respecta a la información asociada a cada uno de los tramos de la carretera, los dos primeros campos contienen el punto kilométrico inicial y final del tramo respectivamente ("10" y "18" en el segundo tramo del ejemplo), el tercer y cuarto campo indican el número total de carriles presentes en el tramo y el número de ellos que están cortados al tráfico respectivamente ("3" y "2" en el segundo tramo del ejemplo), el quinto campo indica la situación del tráfico en el tramo ("Retenciones" en el segundo tramo del ejemplo), finalmente, el sexto campo contiene información acerca de accidentes o situaciones extraordinarias que se han observado en el tramo ("Camión volcado en mitad de la calzada" en el segundo tramo del ejemplo).

El sistema a probar, por tanto, es una aplicación que consume datos en un formato de representación de bajo nivel, el formato producido por el sistema de captura de datos y convierte, bajo demanda, dicha información a un formato de representación de alto nivel como es XML. El formato XML es ideal para ser consumido y transformado por aplicaciones Web, que mezclan la información contenida en ellos con elementos visuales para presentar la información de manera agradable al usuario.

Adicionalmente, el sistema SET hace uso de una base de datos para almacenar la información que obtiene periódicamente del archivo de registros y, de esta forma, mantener un histórico sobre la evolución del estado de las carreteras y tramos que pueda ser consultado posteriormente. Los datos de conexión y localización de la base de datos se encuentran en el archivo de configuración del sistema.

## B.2 Arquitectura del sistema

La arquitectura básica del sistema así como la forma en que interactúa con los sistemas externos se puede observar en la Figura 1.



**Figura 1.** Arquitectura del sistema SET y esquema de interacción con los sistemas externos.

En la Figura 1, el sistema objeto de las pruebas (sistema SET), se encuentra dentro del recuadro en línea discontinua. Como puede observarse, recibe como entrada el archivo de datos con la información de las carreteras y sus tramos producido por el sistema SORT, y genera documentos en formato XML en base a esta información. Estos documentos se construyen bajo demanda, es decir, en respuesta a peticiones HTTP que el sistema recibe de un cliente Web. Típicamente los clientes Web serán navegadores. Sin embargo, dado que el formato de intercambio es XML, dichos documentos pueden ser solicitados por otra aplicación y ser utilizados con otro objetivo que el de ser presentados directamente al usuario en la pantalla del ordenador. Nótese que el sistema SET, en la figura, actúa en modo cliente respecto al sistema SORT y en modo servidor respecto a las aplicaciones Web que consumen información en formato XML.

Paralelamente, existe un archivo de log en el que el SET almacena todos los eventos relevantes que ocurren durante su ejecución. Existe una carpeta, que en la figura aparece con el nombre de Peticiones, que el sistema utiliza para almacenar aquellos documentos XML que va generando en respuesta a peticiones HTTP recibidas. Al igual que el archivo de log, los documentos con la información servida, serán típicamente monitorizados por el responsable del mantenimiento del sistema para comprobar que todo funciona correctamente y conocer la causa del problema en caso de que una situación anormal se produjera.

Por último, existe una base de datos relacional externa asociada al sistema a probar que permite almacenar y cargar información sobre el estado del tráfico, es decir, información de carreteras y tramos. Esta base de datos es muy sencilla y consta únicamente de dos tablas, como se verá más adelante. La ventaja de almacenar la información de registros y tramos en una base de datos es que posibilita hacer consultas selectivas sobre información mediante el lenguaje SQL y de esta forma agilizar el proceso de construcción de los documentos XML bajo demanda.

## B.3 Configuración del sistema

Diferentes aspectos de configuración del sistema se pueden seleccionar previamente a la puesta en marcha del mismo mediante un archivo de configuración (no presente en la figura). Dichos aspectos se listan a continuación:

- Puerto de escucha del sistema SET. Es el puerto al que se han de conectar los clientes Web mediante el protocolo HTTP para realizar peticiones de información en formato XML. Típicamente, al igual que en cualquier servidor Web, este puerto será el puerto 80<sup>3</sup>.
- Ruta y nombre del archivo de información que actúa de interfaz entre el sistema SORT y el sistema SET.
- Ruta y nombre del archivo de log en el que el sistema almacena los eventos que se producen durante su ejecución.
- Directorio donde el sistema almacena los documentos XML generados como respuesta a peticiones HTTP recibidas.

---

<sup>3</sup> A la hora de poner en marcha el sistema software se ha de tener en cuenta que en caso de que el Puerto 80 esté ya en uso por otra aplicación (por ejemplo, el servidor Web Apache en Linux o el servidor WEB IIS en el caso de Windows) se deberá seleccionar un puerto diferente.

- Nombre de usuario y password para el acceso a la base de datos utilizada por el sistema para crear el histórico con la información de carreteras y tramos.

## B.4 Características del sistema y su relevancia en las pruebas de software

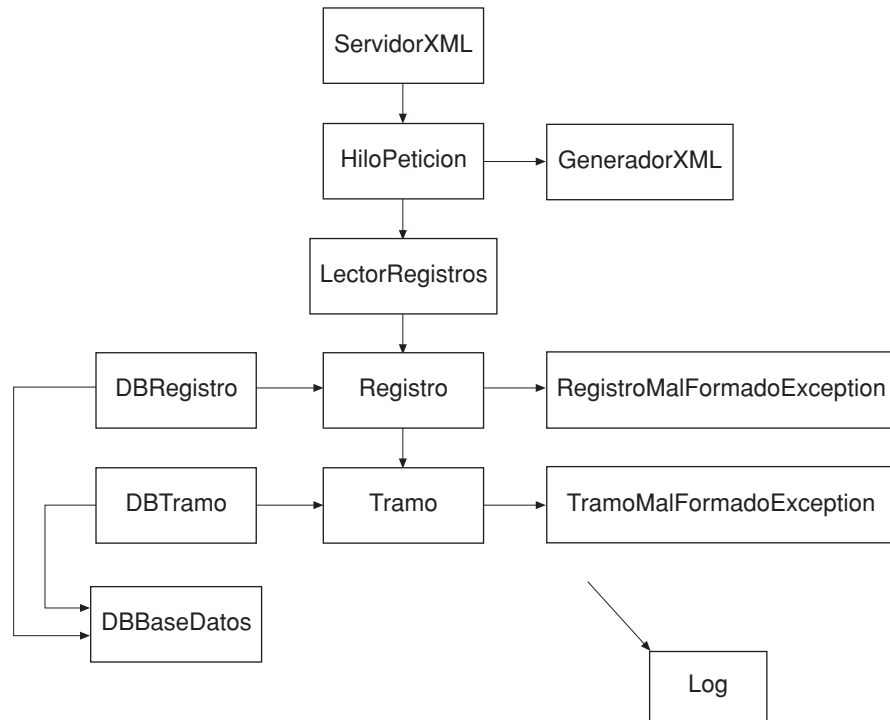
A continuación, se presenta una lista con las principales características del sistema SET. Dichas características han sido introducidas con el único objetivo de construir un sistema lo más completo y real posible sobre el que poder aplicar el amplio abanico de técnicas y herramientas de prueba que se describen a lo largo de este libro.

- Funcionamiento en modo concurrente: el servidor permite atender más de una petición simultáneamente. Para ello, un hilo de ejecución es lanzado para atender cada petición recibida. Esta característica permite realizar pruebas de carga y rendimiento con varios usuarios concurrentes. En el Capítulo 12 de este libro se presentan las herramientas JPerf y JMeter acompañadas de varios ejemplos que permiten explorar este tipo de pruebas.
- Interfaz HTTP: gracias a esta característica el sistema permite la interoperabilidad con cualquier cliente Web. El servidor Web está construido sobre la clase `com.sun.net.httpserver.HttpServer`, que se encarga de ocultar los detalles de manejo del protocolo HTTP inherentes a cualquier servidor Web. El Capítulo 11 de este libro está dedicado por entero a las pruebas de aplicaciones Web. En él se presentan herramientas como HTTPUnit, JWebUnit o HTMLUnit que permiten realizar pruebas en profundidad sobre este tipo de aplicaciones.
- Entrada/Salida desde/a fichero.
- Utilización de métodos privados y manejo de excepciones. Como se ha visto en el Capítulo 2 este es un aspecto muy importante en la prueba de código Java en particular y en la prueba de código escrito en lenguajes orientados a objetos en general.
- Configuración externa.
- Generación de documentos en formato XML y HTML. Estos dos formatos de documento se encuentran, sin duda, entre los más utilizados hoy en día para el almacenamiento y representación de la información, de ahí la importancia de conocer cómo probar eficazmente las aplicaciones que trabajan con ellos. El Capítulo 10 de este libro introduce al lector en la prueba de este tipo de documentos mediante la herramienta XMLUnit.
- Acceso a bases de datos. El acceso a bases de datos es una característica cada vez más común en cualquier aplicación software. Desafortunadamente, el hecho de que la base de datos sea una entidad externa a la aplicación en sí, dificulta el proceso de prueba sustancialmente. En el Capítulo 9, se abordó en profundidad este escenario de pruebas.
- Relaciones de asociación entre clases. Este tipo de relaciones entre clases dificultan las pruebas unitarias de clases que hacen uso de clases colaboradoras. En el Capítulo 2 de este libro “Pruebas unitarias en aislamiento mediante Mock Objects” se trató en profundidad esta técnica de pruebas.

## B.5 Arquitectura interna del sistema

### a) Clases del sistema

Internamente el sistema está formado por doce clases para un total de 2500 líneas de código aproximadamente. En la Figura 2 se muestra el diagrama de clases del sistema en el que las flechas muestran las relaciones de asociación entre clases.



**Figura 2.** Diagrama de clases del sistema SET.

A continuación, se va a describir brevemente cada una de las clases de las que se compone el sistema.

**ServidorXML.** Esta es la clase principal del sistema y en la que se encuentra definida la función global `main`. Esta clase se encarga de cargar la configuración del sistema y de arrancar un servidor Web (instanciando un objeto de la clase `com.sun.net.httpserver.HttpServer`) que se encargará de recibir las peticiones HTTP por el puerto indicado. Durante la inicialización de este objeto es necesario instanciar un objeto de la clase `HiloPetición` que es el que procesa las peticiones que le llegan al servidor Web.

**HiloPetición.** Esta clase implementa la interfaz `com.sun.net.httpserver.HttpHandler` y se encarga de procesar las peticiones que le llegan al servidor. Dependiendo del tipo de petición a procesar, esta clase se encarga de generar páginas HTML con información sobre el estado/características del servidor o bien documentos XML con información sobre el estado del tráfico. En este último caso, el procedimiento consiste en dos pasos. Inicialmente una



instancia de la clase `LectorRegistros` es utilizada para cargar la información sobre el estado del tráfico desde el archivo en formato texto generado por el SORT (Sistema de Observación y Registro del Tráfico). Posteriormente, esta información es utilizada para la generación de los documentos XML mediante la clase `GeneradorXML`.

**LectorRegistros.** Esta clase se encarga de leer desde el archivo de registros generado por el SORT, los registros con información del estado del tráfico de carreteras y tramos. Para el almacenamiento de esta información utiliza objetos de las clases `Registro` y `Tramo`.

**GeneradorXML.** Esta clase construye documentos XML con la información asociada a las consultas que el sistema puede recibir en forma de peticiones HTTP. Básicamente, transforma la información contenida en un conjunto de objetos `Registro` y `Tramo` (se trata de información instantánea del estado de carreteras y tramos que fue tomada del archivo de registros) en un documento XML según la petición que se esté procesando. Por ejemplo, si la petición es acerca del estado del tráfico en una carretera en particular, esta clase se encarga de iterar el conjunto de registros en búsqueda de aquel correspondiente a la carretera indicada y construir un documento XML con la información contenida en él. En situaciones en las que no sea posible construir un documento con la información solicitada, debido a que no hay información disponible, se generará un documento XML de error indicando el problema.

**Registro.** Esta clase sirve de contenedor de información sobre un registro leído del archivo de registros (mediante la clase `LectorRegistros`), es decir, información sobre el estado de una carretera y sus correspondientes tramos. Internamente, almacena objetos `Tramo` que actúan de contenedores de la información de cada tramo de la carretera. Adicionalmente, esta clase permite comprobar si la información de un registro está bien formada<sup>4</sup> (por ejemplo que la carretera tenga definida información para al menos un tramo). En caso de que se encuentre alguna anomalía en esta información, se utiliza la clase colaboradora `RegistroMalFormadoException` para comunicar este problema.

**Tramo.** Esta clase sirve de contenedor de información sobre un tramo de carretera que forma parte del conjunto de tramos leídos de un registro. Adicionalmente, esta clase permite comprobar si la información del tramo está bien formada (por ejemplo que no haya más carriles cortados que el número total de carriles en el tramo). En caso de que se encuentre alguna anomalía en esta información, se utiliza la clase colaboradora `TramoMalFormadoException` para comunicar este problema.

**RegistroMalFormadoException.** Se trata de una excepción que hereda de la clase `Exception` y contiene información acerca de la naturaleza de la inconsistencia que presenta la información contenida en un determinado objeto `Registro`.

**TramoMalFormadoException.** Se trata de una excepción que hereda de la clase `Exception` y contiene información acerca de la naturaleza de la inconsistencia que presenta la información contenida en un determinado objeto `Tramo`.

**DBBaseDatos.** Esta clase encapsula los detalles de conexión con la base de datos a partir de la información de autenticación y localización de la misma que reside en el archivo de configuración.

---

<sup>4</sup> Se trata de una forma de asegurar que la información que fue generada y almacenada en el archivo de registros por el sistema SORT y posteriormente leída por la clase `LectorRegistros`.

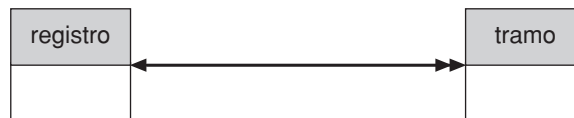
**DBRegistro.** Esta clase sirve de interfaz entre la base de datos y el sistema. Permite cargar información sobre registros desde la base de datos. Asimismo permite almacenar objetos registro en la base de datos.

**DBTramo.** Esta clase sirve de interfaz entre la base de datos y el sistema. Permite cargar información sobre tramos desde la base de datos. Asimismo permite almacenar objetos tramo en la base de datos.

**Log.** Esta clase, como su propio nombre indica, se encarga de serializar los eventos que se producen durante la ejecución del sistema a un archivo en formato texto que resulta de gran utilidad para la realización de las tareas de mantenimiento del sistema.

### b) Estructura interna de la base de datos

A pesar de que la base de datos no forma parte del sistema en sí sino que está alojada en un servidor de bases de datos externo, es interesante conocer su estructura para entender de qué forma interactúa el sistema con ella. La base de datos consta de dos tablas, *registro* y *tramo* que contienen información relativa a registros con información sobre el estado del tráfico y sus correspondientes tramos. En la Figura 3 se puede observar el diagrama entidad-relación de la base de datos.



**Figura 3.** Diagrama entidad relación.

Como puede observarse existe una relación de cardinalidad uno a *n* entre las tablas *registro* y *tramo*. Además cada elemento de la tabla *registro* ha de tener al menos un elemento de la tabla *tramo* asociado. Finalmente, cada elemento de la tabla *tramo* ha de estar asociado a uno y solo un elemento de la tabla *registro*.

Por último cabe mencionar que el sistema gestor de bases de datos utilizado para los ejemplos de este libro es MySQL, que se encuentra disponible para libre descarga desde la página <http://www.mysql.com/>.

## B.6 Documento de especificación de requisitos software

Uno de los pilares en los que se sustenta todo proceso de pruebas es el Documento de Especificación de Requisitos Software<sup>5</sup>. Es justo este documento el que informa al desarrollador del comportamiento y características esperadas del sistema y, por tanto, de las condiciones que han de verificarse en el proceso de la prueba. Por este motivo se ha incluido una versión sim-

<sup>5</sup> Para obtener información detallada sobre el papel que dicho documento juega en la elaboración del Plan de Pruebas véase Capítulo 1 Fundamentos de las pruebas de software.

plificada<sup>6</sup> del mismo (básicamente una lista de requisitos ordenados por tipo) en este apéndice. Dicho documento debe entenderse como el conjunto de características que el cliente espera que el sistema presente en el momento en el que le sea entregado. Conviene recordar que el objetivo final de las pruebas es verificar que el sistema desarrollado va a satisfacer las expectativas del cliente.

No se recomienda al lector hacer una lectura literal del documento sino consultarlo únicamente en aquellas situaciones en las que, al seguir los ejemplos de este libro, le aparezcan dudas acerca del objetivo de una prueba en particular.

#### a) **Configuración del servidor**

Debe existir un fichero desde el cual se puedan configurar los siguientes parámetros del servidor:

- Puerto de escucha del servidor.
- Nombre del archivo desde el cual se cargará la información relativa al estado de las carreteras.
- Nombre del archivo de log.
- Carpeta donde se almacenarán los XML enviados como respuestas a peticiones.

#### b) **Tipos y formato de las peticiones a las que debe responder el servidor así como formato de los documentos de respuesta**

El servidor deberá responder a peticiones HTTP (en modo GET) con un documento XML<sup>7</sup> conteniendo la información correspondiente:

- Información acerca del estado de una carretera.
- Información acerca de los tramos de carretera con más tráfico.
- Información acerca de los tramos de carretera con menos tráfico.
- Información acerca de los tramos de carretera con carriles cortados al tráfico.
- Información acerca de los tramos de carretera con accidentes.
- Información acerca de las carreteras con obras.
- Información acerca de las carreteras con una climatología dada.

En caso de que el SET reciba una petición HTTP con un formato incorrecto, dicha petición será respondida con una página HTML de error que contendrá el siguiente mensaje de texto: *"Error: formato de petición incorrecto"*.

Adicionalmente debe existir la posibilidad de detener el sistema de forma remota mediante una petición HTTP (en modo GET) de detención. Esta petición debe mostrar un formulario de

---

<sup>6</sup> Una versión completa y detallada del Documento de Especificación de Requisitos Software está disponible para consulta desde el sitio Web dedicado a este libro:

<sup>7</sup> Información detallada acerca de la sintaxis exacta requerida para las peticiones HTTP así como el formato de los documentos XML que se han de generar asociadamente, se puede encontrar en la versión completa del Documento de Especificación de Requisitos Software.

autenticación con nombre de usuario y password de forma que solo usuarios autorizados sean capaces de detener el sistema. Una vez el sistema haya sido detenido con éxito, deberá mostrar una página HTTP con un mensaje informativo.

Por último, el servidor debe ser capaz de proporcionar información acerca de su estado en cualquier momento de su ejecución mediante un documento HTML. La información que se debe mostrar será, al menos, la siguiente:

- Peticiones disponibles en el sistema con su correspondiente formato a modo de ejemplo.
- Información acerca del estado del servidor, incluyendo la hora y fecha de arranque del mismo y el número de peticiones atendidas desde ese momento.

#### **c) Log**

El servidor debe almacenar en un archivo de Log, una línea de texto por cada incidencia que ocurra durante su funcionamiento, las incidencias a incluir serán, al menos, las siguientes:

- Peticiones que se realizan al servidor.
- Peticiones con un formato incorrecto que se realizan al servidor.
- Cualquier tipo de fallo en el servidor

Adicionalmente, cada línea que se almacene en el archivo de Log deberá ir precedida de la hora y la fecha en la que se produjo la incidencia y, cuando sea oportuno, de la localización en el código fuente dentro de la cual se generó dicha línea, es decir, clase y método.

#### **d) Parada del servidor**

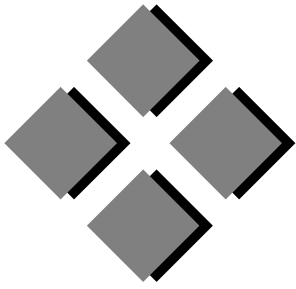
El servidor debe poder detenerse mediante la combinación de teclas ctrl+c, liberando todos los recursos utilizados.

#### **e) Registro de información enviada**

Deben almacenarse en una carpeta todas aquellas respuestas a peticiones que se hayan producido, las respuestas han de ir numeradas, y serán sobrescritas en cada arranque del servidor. Las respuestas serán archivos XML con la información enviada o bien páginas HTML con las respuestas de error que se hayan producido.

#### **f) Verificación de la información contenida en el archivo de registros**

Como mecanismo de seguridad adicional, el sistema debe ser capaz de verificar el correcto formato de la información sobre el estado del tráfico leída desde el archivo de registros producido por el Sistema de Observación y Registro del Tráfico. Este tipo de verificaciones incluyen el formato de la hora y fecha así como la coherencia en la relación entre carreteras y sus tramos.



# Apéndice C

## Estándares de nomenclatura y normas de estilo en Java

---

### SUMARIO

C.1 Conceptos de clase y objeto

C.2 Normas de estilo

Este apéndice tiene como objetivo, por un lado, familiarizar al lector con los términos utilizados a lo largo de este libro para referirse a los distintos elementos que componen el lenguaje Java y, por otro lado, presentar brevemente las normas de estilo de codificación en el lenguaje Java que son habitualmente utilizadas.

## C.1 Conceptos de clase y objeto

Para la correcta comprensión de este libro es de vital importancia tener claros los conceptos de clase y objeto. Estos conceptos son, quizás, los primeros que se enseñan en cualquier curso básico de Java. Sin embargo, a menudo son fuente de malentendidos. La razón es que, en ocasiones, por error, se utilizan indistintamente a pesar de tener significados muy diferentes. La forma más clara de ver esta diferencia para aquellos no familiarizados con la orientación a objetos es estableciendo una analogía entre clase y tipo de dato. Mientras que una clase se puede entender como un tipo de dato, un objeto es una variable declarada con ese tipo de dato. A menudo los objetos son referidos como instancias ya que son instancias de una clase.

Una clase puede verse como una estructura de datos en el lenguaje C (declaradas con la palabra reservada `struct`) cuyos miembros pueden ser variables o funciones que realizan operaciones sobre estas variables<sup>1</sup>. Sin embargo, una clase es mucho más que eso ya que, por ejem-

---

<sup>1</sup> Nótese que se trata de una simplificación considerable únicamente válida a modo ilustrativo.

plo, permite declarar variables llamadas “variables de clase” que existen independientemente de si la clase es instanciada o no. Una variable de clase es como una variable estática en el lenguaje C (declaradas con la palabra reservada `static`). Estas variables existen de forma única dentro de una instancia de la máquina virtual de Java de modo que no se duplican al instanciar objetos de la clase. En contraposición, las variables de objeto pertenecen al ámbito del objeto por lo que se crean e inician con cada instanciación de la clase.

Se puede establecer un paralelismo similar con los llamados “métodos de clase” y los “métodos de objeto”. Mientras que los métodos de clase son métodos que únicamente pueden acceder a variables de clase, los métodos de objeto pueden acceder a variables de clase o de objeto indistintamente.

## C.2 Normas de estilo

El objetivo de las normas de estilo es hacer los programas más fácilmente legibles y, en última instancia, más mantenibles. Estas normas permiten conocer la naturaleza de un identificador (constante, clase, método, etc.) sin tener presente cómo ha sido declarado (en ocasiones fuera del alcance de la vista del desarrollador). De esta forma, el desarrollador con solo ver el identificador puede hacerse una idea de su función y del papel que desempeña dentro del código.

### a) Paquetes

Los paquetes son conjuntos de clases que tienen relación entre sí. Normalmente, los nombres de paquete están compuestos de una serie de palabras escritas con letras minúsculas separadas por puntos. La serie de palabras normalmente denota una jerarquía, siendo la primera palabra el nombre del elemento de nivel superior en la jerarquía. Nótese que, a menudo, esta jerarquía refleja la forma en la que los componentes del paquete están almacenados en disco.

Como ejemplo de lo anterior se puede mencionar el bien conocido paquete: `org.junit.runner.manipulation`, cuyo nombre está compuesto por cuatro elementos de una jerarquía.

### b) Clases

El nombre de las clases ha de estar compuesto siempre por una secuencia de palabras concatenadas que describan adecuadamente el propósito de dicha clase. Estas palabras no deben ser abreviaturas sino siempre palabras completas. La forma de separar unas palabras de otras es mediante el uso de mayúsculas para la primera letra de cada una de las palabras. De esta forma, si se quiere hacer una clase que se encargue de la gestión de colas de pedidos, el nombre de dicha clase será: `GestorColasPedidos`.

### c) Interfaces

La nomenclatura de las interfaces es idéntica a la de las clases, es decir, una secuencia de nombres escritos con la primera letra en mayúscula.

### d) Métodos

Mientras que el nombre de las clases suele estar compuesto de palabras con la función de nombre (debido a la naturaleza estática del concepto de clase) el nombre de los métodos nor-

malmente se ha de componer de al menos un verbo (debido a la naturaleza dinámica de los mismos). La primera palabra del nombre del método ha de estar escrita siempre con minúsculas mientras que las palabras subsiguientes deben comenzar con mayúscula. Por ejemplo, un método de la clase `GestorColasPedidos` que se encargue de vaciar las colas recibirá el nombre de `vaciarColas`.

### e) Variables

El nombre de las variables debe ser lo mas descriptivo posible ya que a menudo juegan un papel fundamental para facilitar la lectura del código. Sin embargo, a ser posible, se recomienda utilizar nombres cortos para acortar el tiempo de lectura/escritura. Al igual que los métodos, su nombre ha de estar compuesto de una serie de palabras: la primera escrita con minúsculas y las siguientes con la primera letra en mayúscula. No se recomienda el uso de nombres compuestos de una única letra salvo para variables de tipo contador (`i`, `j`, `k`, etc.) o de “usar y tirar”.

Una consideración a tener en cuenta a la hora de asignar nombre a una variable es el tipo de dato al que pertenece. De una forma sencilla es posible incorporar el nombre del tipo de dato de la variable dentro de su nombre, esto se realiza utilizando un prefijo. El objetivo es que con sólo mirar la variable, el desarrollador se haga una idea de qué operaciones están disponibles para esa variable sin necesidad de buscar su declaración (que no tiene por qué estar accesible en un lugar cercano dentro del código fuente). En la siguiente tabla se muestran los prefijos a utilizar para los tipos de datos/clases más utilizados en Java.

Tipo de dato	Nombre de la variable
<code>int</code>	<code>iVariable</code>
<code>float</code>	<code>tfVariable</code>
<code>char</code>	<code>cVariable</code>
<code>byte</code>	<code>variable</code>
<code>String</code>	<code>strVariable/sVariable</code>

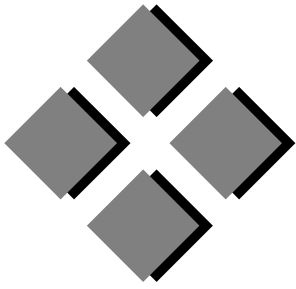
Adicionalmente, se suele utilizar el prefijo `m_` o simplemente `m` para declarar variables de clase o de objeto es decir, “variables miembro”. Por ejemplo, si la variable `iContador` se definiera como miembro de la clase `GestorColaPedidos`, el nombre que se utilizaría es `m_iContador`.

### f) Constantes

Al igual que en el lenguaje de programación C, las constantes han de ser definidas como una secuencia de palabras (normalmente nombres) escritas en mayúscula y separadas por medio del carácter “\_”. Por ejemplo, una constante para que indique el valor máximo para el tamaño de una cola recibirá el nombre `TAMANO_MAXIMO_COLA`.







## Apéndice D

# Novedades en Java 5.0

---

### SUMARIO

**D.1** Introducción

**D.3** *Import* estático

**D.2** Anotaciones en Java

## D.1 Introducción

Este apéndice no pretende ser un compendio de todas y cada una de las novedades introducidas en el lenguaje Java en su versión 5.0, sino presentar y servir de primera toma de contacto con dos de ellas cuyo conocimiento resulta clave para la utilización de, por ejemplo, las versiones 4.x de JUnit. Se refieren a las anotaciones y al *import* estático y, probablemente, constituyen las principales novedades de Java 5.0. Se presentarán en los apartados siguientes.

## D.2 Anotaciones en Java

Han sido introducidas por Sun en la versión 1.5 de Java SDK. Se trata, básicamente, de un método para añadir información adicional o metadatos en el código fuente. Lo interesante es que esta información puede ser posteriormente accedida e interpretada por herramientas externas o incluso desde el propio compilador, con el objetivo de reducir la escritura de código fuente redundante por el programador. Este mecanismo permite disminuir el tamaño del código fuente, lo que lleva consigo un aumento de la legibilidad, reducción de la probabilidad de introducir errores y disminución del tiempo de desarrollo. Básicamente, permite trasladar el “trabajo sucio”, que habitualmente ha de realizar el desarrollador al compilador, herramientas generadoras de código fuente, etc. Por ejemplo, en el caso de JUnit 4.x las anotaciones sirven, entre

otras cosas, para indicar a la herramienta qué métodos de aquellos pertenecientes a una clase de pruebas son métodos de prueba que han de ejecutarse como parte de las pruebas de la clase (para este propósito se utiliza la anotación `@test` al comienzo de la definición de un método de prueba).

Mirando atrás en el tiempo, la inclusión de anotaciones en el código fuente no es nada novedoso, lo que sí es nuevo es la definición de un estándar de anotación inherente al lenguaje Java. Dos ejemplos claros de herramientas ya existentes con una funcionalidad similar son JavaDoc y XDoclet. JavaDoc utiliza principalmente etiquetas incluidas en los comentarios para generar automáticamente documentación sobre el código fuente. Posteriormente, surgió XDoclet, que utiliza etiquetas especiales similares a las usadas en JavaDoc para definir propiedades dentro del código fuente. XDoclet proporciona mecanismos que permiten procesar el código fuente para obtener estas propiedades y utilizarlas en la generación de elementos software como son: código fuente, archivos XML, descriptores, etc.<sup>1</sup>.

Sin embargo, estos mecanismos de definición de metadatos en los comentarios del código fuente presentan una serie de inconvenientes:

- La información contenida en los comentarios no está disponible en tiempo de ejecución ya que se pierde en el proceso de compilación.
- El procesado de esta información incrementa el tiempo de compilación.
- La sintaxis empleada en la definición de los metadatos no se puede verificar fácilmente en tiempo de compilación por lo que si el desarrollador comete, por ejemplo, un error al escribir una etiqueta, el compilador probablemente no se dé cuenta y no sea capaz de notificar el problema.

El sistema de anotaciones incluido en Java 5.0 soluciona todos estos problemas a la vez que introduce un estándar de anotación único y común para todos los desarrolladores Java.

A la hora de trabajar con anotaciones existen tres pasos: definición de las anotaciones, utilización de las anotaciones y procesado de las mismas para realizar la tarea en sí. En la mayoría de los casos, los desarrolladores Java no necesitan definir sus propias anotaciones sino que simplemente hacen uso de anotaciones definidas de antemano en las herramientas que ellos utilizan. Este es el caso cuando se trabaja con JUnit 4.x. Sin embargo, definir anotaciones no es difícil y en ocasiones puede resultar de gran utilidad. En cualquier caso, el proceso de definición de anotaciones y su procesado queda fuera de los objetivos de este libro.

## D.3 *Import* estático

Quizás la aparición del *import* estático en Java ha sido una de las fuentes de mayor controversia desde la aparición de este popular lenguaje de programación. El *import* estático es en el fondo algo muy sencillo, consiste en permitir a una clase hacer uso de métodos o variables estáticas de otra clase sin el correspondiente cualificador (nombre de la clase a la que pertenecen) y sin he-

---

<sup>1</sup> Conviene destacar que las anotaciones no son un mecanismo mediante el cual almacenar información de configuración dentro del código fuente sino una forma de expresar información o propiedades de elementos del código fuente. Esto se debe tener siempre presente ya que las anotaciones son, con más frecuencia de la deseada, utilizadas para almacenar información de configuración, la cual, por razones obvias nunca ha de estar presente en el código fuente.

redar de dicha clase. Por ejemplo, normalmente si se quiere hacer uso de la constante  $\pi$  (3.1416) desde un método que calcula el área de polígonos en una clase llamada, por ejemplo, `PropiedadesPoligonos`, se hace referencia a esta constante mediante el nombre `Math.PI`. En este caso, el cualificador `Math` es el nombre de la clase en el que la constante<sup>2</sup> está definida (`java.lang.Math`). Una forma de facilitar el uso de esta constante es mediante el uso del *import* estático. En este caso bastaría con añadir la sentencia

```
import java.lang.Math.*;
```

al principio de la definición de la clase `PropiedadesPoligonos` y automáticamente todas las variables estáticas (variables de clase) y métodos estáticos (métodos de clase) estarán accesibles para ser utilizados dentro de la clase sin necesidad de cualificador.

Como puede observarse, el *import* estático es respecto a las variables y propiedades estáticas de una clase lo que el *import* tradicional (*import* de paquetes/clases) es a las clases pertenecientes a un paquete.

Si bien el *import* estático permite hacer uso de miembros estáticos de otras clases de una forma breve, prescindiendo del cualificador, es justo esto lo que lo ha hecho objetivo de fuertes críticas. Y es que un desarrollador java que esté leyendo código que no haya sido escrito por él o bien que haya sido escrito por él hace cierto tiempo, puede no conocer o recordar a qué clase pertenece determinado miembro que es referenciado sin cualificador, y esto es fuente de confusiones, desorientación y, en definitiva, pérdida de legibilidad y de mantenibilidad.

Por ejemplo, si un desarrollador no familiarizado con la clase `Math`, se encuentra con el uso de la constante `PI` sin el cualificador `Math`, pensará que esa constante pertenece a la clase `PropiedadesPoligonos` y al buscarla dentro de la clase y ver que no se encuentra allí, puede que no sepa bien dónde seguir buscando. Por este motivo, existen muchos detractores del *import* estático y este sólo debe utilizarse en situaciones muy excepcionales y, en cualquier caso, nunca más de uno o dos *imports* estáticos dentro de la definición de una clase.

---

<sup>2</sup> Nótese que las constantes en Java no son más que variables estáticas declaradas como públicas y finales, es decir, `public final static nombreConstante`.

