

# Apuntes de pruebas

Macario Polo Usaola

Beatriz Pérez Lamancha

Pedro Reales Mateo

**Escuela Superior de Informática**

**Universidad de Castilla-La Mancha**

## ÍNDICE

Capítulo I. Introducción.....	4
1 Imposibilidad de las pruebas exhaustivas.....	4
2 Error, defecto o falta, fallo .....	5
3 Caso de prueba .....	5
4 Objetivo de las pruebas .....	6
Capítulo II. Niveles de prueba.....	8
1 Pruebas de caja negra .....	8
2 Pruebas estructurales o de caja blanca .....	10
3 Pruebas unitarias.....	11
4 Pruebas de integración.....	12
5 Pruebas de sistema .....	12
Capítulo III. Criterios de cobertura para artefactos software .....	14
1 Criterios de cobertura .....	14
2 Utilidad de los criterios de cobertura .....	14
3 Un posible modelo de trabajo .....	15
4 Criterios de cobertura para código fuente .....	16
4.1 Cobertura de sentencias.....	16
4.2 Cobertura de decisiones, de ramas o de todos los arcos.....	17
4.3 Cobertura de condiciones.....	18
4.4 Cobertura de condiciones/decisiones ( <i>Decision/Condition coverage</i> , o <i>DCC</i> ).....	19
4.5 Cobertura múltiple de condiciones ( <i>Multiple Condition Coverage</i> , <i>MCC</i> ) .....	20
4.6 Cobertura modificada de condiciones/decisiones ( <i>Modified Condition/Decision Coverage</i> , <i>MC/DC</i> ). .....	21
5 Criterios de cobertura para máquinas de estado .....	24
5.1 Cobertura de estados.....	24
5.2 Cobertura de transiciones .....	25
5.3 Cobertura de pares de transiciones.....	25
5.4 Cobertura de secuencia completa.....	26
Capítulo IV. Valores de prueba .....	27
1 Clases o particiones de equivalencia .....	27
2 Valores límite ( <i>Boundary values</i> ) .....	29
3 Conjetura de errores ( <i>error-guessing</i> ) .....	30

4	Aplicación de las técnicas al conjunto de datos de salida .....	31
5	Criterios de cobertura para valores de prueba .....	31
5.1	Cada uso ( <i>each use</i> ).....	32
5.2	Todos los pares ( <i>pairwise</i> ) .....	32
5.3	Todas las tuplas de $n$ elementos ( <i>n-wise</i> ) .....	33
Capítulo V. Estrategias de combinación para la obtención de casos de prueba		34
1	Estructura de un caso de prueba .....	34
1.1	Ejemplo.....	34
2	El oráculo.....	39
2.1	Obtención de casos de prueba con oráculos a partir de máquinas de estado .....	40
3	Estrategias de combinación .....	44
3.1	Todas las combinaciones ( <i>All combinations</i> ).....	45
3.2	Cada elección ( <i>Each choice</i> ) .....	49
3.3	AETG ( <i>Automatic Efficient Test Generator</i> ).....	50
3.4	PROW ( <i>Pairwise with Restrictions, Order and Weight</i> ).....	53
3.5	Antirandom.....	55
3.6	Algoritmo del peine ( <i>Comb</i> ).....	57
3.7	Algoritmos aleatorios.....	58
4	CTWeb, una aplicación web para testing combinatorio .....	58

## Capítulo I. Introducción

*El problema fundamental respecto a la prueba de software es que no se puede probar completamente un sistema, por lo que, en el momento de realizar las pruebas, se deben tomar decisiones respecto a qué partes del sistema probar y el grado de profundidad de los casos de prueba. La actitud que debe tomar la persona frente al sistema bajo prueba no es la de demostrar que el programa funciona correctamente, sino la de encontrarle fallos.*

*En este capítulo se definen los principales términos utilizados en las pruebas de software y se presenta el objetivo de las pruebas.*

### 1 Imposibilidad de las pruebas exhaustivas

Para probar completamente un sistema se deben ejercitar todos los caminos posibles del programa a probar. Myers mostró en 1979 un programa que contenía un bucle y unas pocas instrucciones condicionales, y que tenía 100 trillones de caminos. La prueba exhaustiva de este programa podría requerir un billón de años.

La prueba exhaustiva requiere probar el comportamiento de un programa para todas las combinaciones válidas e inválidas de todos sus posibles puntos de entrada, y teniendo en cuenta todos los estados posibles del programa.. En la práctica esto resulta imposible, ya que el tiempo requerido es prohibitivo.

En un mundo ideal, desearíamos probar un programa con todo el rango posible de datos de entrada. Incluso un programa aparentemente simple puede tener centenares o millares de combinaciones posibles de entrada y de salida. Crear los casos de prueba para todas estas posibilidades no es económicamente factible, por lo que este problema, fundamental, tiene serias implicaciones en la economía de las pruebas, en las suposiciones que el tester tendrá que hacer sobre el programa y en la manera en la cual se diseñan los casos de prueba. El objetivo debe ser maximizar la “producción de las pruebas”: esto es, maximizar el número de los errores encontrados por un número finito y, deseablemente, pequeño, de casos de

prueba<sup>1</sup>. La forma en que se construyen y seleccionan los casos de prueba es una de las principales decisiones que debe tomarse.

## 2 Error, defecto o falta, fallo

Se utilizan distintos términos para distinguir entre la causa de un mal funcionamiento en el sistema, que es percibida por los desarrolladores y la observación de ese mal funcionamiento, que es percibida por quien usa el sistema.

Un *error* es una equivocación realizada por una persona (en este contexto, un programador); un *defecto o falta* es un error al realizar alguna actividad concerniente al software; y un *fallo* es un desvío respecto al comportamiento requerido del sistema<sup>2</sup>.

Con estas definiciones, el programador introduce errores, que son percibidos como defectos por el tester, mientras que el fallo lo detecta el usuario por una diferencia de comportamiento. No todos los defectos corresponden a un fallo: si un trozo de código defectuoso nunca se ejecuta, el defecto nunca provocará el fallo del sistema.

La prueba puede revelar fallos, pero son los defectos los que pueden y deben ser detectados por el tester y eliminados por el desarrollador<sup>3</sup>.

## 3 Caso de prueba

Un *caso de prueba (test case)* es un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y poscondiciones de ejecución, desarrollados con un objetivo particular o condición de prueba, tal como ejercitar un camino de un programa particular o para verificar que se cumple un requisito específico<sup>4</sup>.

Un *conjunto de prueba (test suite)* es un conjunto de uno o más casos de prueba, con un propósito y base de datos común que usualmente se ejecutan en conjunto.

---

<sup>1</sup> Myers G. "The art of software testing, 2nd edition", ISBN 0-471-46912-2, John Wiley & Sons Inc., 2004.

<sup>2</sup> Pfleeger S. "Software Engineering, 2nd Edition", ISBN: 0130290491, Prentice Hill, 2001.

<sup>3</sup> Guide to the Software Engineering Body of Knowledge SWEBOK, 2004 version. IEEE Computer Society. <http://www.swebok.org>, 2004.

<sup>4</sup> IEEE Standard Glossary of Software Engineering Terminology Institute of Electrical and Electronics Engineers, ISBN: 155937067X, 1990.

Los *datos de prueba* (*test data*) son los datos que existen (por ejemplo, en una base de datos) antes de que una prueba sea ejecutada, y que afecta o es afectado por el componente o sistema bajo prueba<sup>5</sup>.

Uno de los aspectos a considerar al realizar pruebas es decidir cuándo el programa falla para un conjunto de datos de entrada, o sea, conocer cuál es la salida esperada del programa. Este problema es conocido como el *problema del oráculo*. Un *oráculo* es cualquier agente (humano o mecánico) que decide si un programa se comportó correctamente en un caso de prueba determinado, y que por consiguiente produce un veredicto de "paso" o de "fallo". Existen diversas clases de oráculos, y la automatización del oráculo puede llegar a ser muy compleja y costosa<sup>6</sup>. El oráculo más común es el oráculo de entrada/salida, que especifica la salida esperada para una entrada específica<sup>7</sup>.

#### **4    Objetivo de las pruebas**

La prueba debe ser vista como el proceso destructivo de encontrar errores (cuya presencia se asume) en un programa. Si el propósito del tester es verificar que el programa funciona correctamente, entonces el tester está fallando al conseguir su propósito cuando encuentra defectos en el programa. En cambio, si el tester piensa que su tarea es encontrar fallos, los buscará con mayor ahínco que si piensa que su tarea es verificar que el programa no los tiene.

Si el objetivo de la prueba es encontrar errores, un caso de prueba exitoso es uno que hace que el programa falle (es decir, que no se comporte de la forma esperada). Por supuesto que eventualmente se puede usar la prueba para establecer algún grado de confianza de que el programa hace lo que se supone que debe hacer y no hace nada que no se suponga que no deba hacer<sup>8</sup>.

---

<sup>5</sup> International Software Testing Qualifications Board, Certified Tester Foundation Level Syllabus, Versión 2005.

<sup>6</sup> Guide to the Software Engineering Body of Knowledge SWEBOK, 2004 version. IEEE Computer Society. <http://www.swebok.org>, 2004.

<sup>7</sup> Beizer B. "Software testing techniques (2nd ed.)", ISBN:0-442-20672-0, Van Nostrand Reinhold Co, 1990.

<sup>8</sup> Myers G. "The art of software testing, 2nd edition", ISBN 0-471-46912-2, John Wiley & Sons Inc., 2004.

El tester debe adoptar una actitud destructiva hacia el programa, debe querer que falle, debe esperar que falle y debe concentrarse en encontrar casos de prueba que muestren sus fallos<sup>9</sup>.

---

<sup>9</sup> Kaner C., Falk J., Nguyen H. "Testing Computer Software, 2nd Edition", ISBN: 0471358460, Wiley, 1999 .

## Capítulo II. Niveles de prueba

*Tradicionalmente, se distinguen dos tipos básicos de pruebas: pruebas de caja blanca y pruebas de caja negra que, además, suelen subdividirse en niveles aun más específicos. Así, las pruebas de caja blanca se aplican, normalmente, a pruebas unitarias y a ciertas pruebas de integración, mientras que las de caja negra hacen referencia, en general, tanto a pruebas unitarias, como funcionales, de integración, de sistema e, incluso, de aceptación.*

*En este capítulo se presenta una breve introducción a las pruebas de caja blanca y negra, y se describen algunas características de los niveles de prueba.*

### 1 Pruebas de caja negra

En este tipo de pruebas, el elemento que se va a probar se entiende como una caja negra de la que sólo se conocen sus entradas y sus salidas. Así, al elemento bajo prueba se lo somete a una serie de datos de entrada, se observan las salidas que produce y se determina si éstas son conformes a las entradas introducidas.

Un conocido problema que se utiliza con frecuencia en el contexto de las pruebas de software es el de la determinación del tipo de un triángulo, que fue originalmente propuesto por Bertrand Myers<sup>10</sup>: adaptado a la orientación a objetos, se trata de probar una clase que dispone de una operación que calcula el tipo de un triángulo según las longitudes de los lados. Esta operación devuelve un entero que representa si el triángulo es equilátero, isósceles, escaleno o si no es un triángulo (porque tenga lados de longitud cero o negativa, o porque la suma de dos lados sea menor o igual a la suma del tercero). En la Figura 1 se muestra la estructura de la clase Triángulo, que consta de un constructor, tres operaciones *set* que asignan una longitud a los lados del triángulo y un método *getTipo*, que devuelve un entero que representa el tipo del triángulo.

Bajo un enfoque de caja negra, el punto de vista que interesa al ingeniero de software se ilustra en las cuatro imágenes de la Figura 1, en las que se pasan diferentes ternas de valores a los métodos que asignan la longitud a los lados del

---

<sup>10</sup> Myers GJ. (1979). *The Art of Software Testing*. John Wiley & Sons.



triángulo (*setI*, *setJ*, *setK*) y luego se comprueba únicamente si el resultado devuelto por *getTipo* es el correcto.

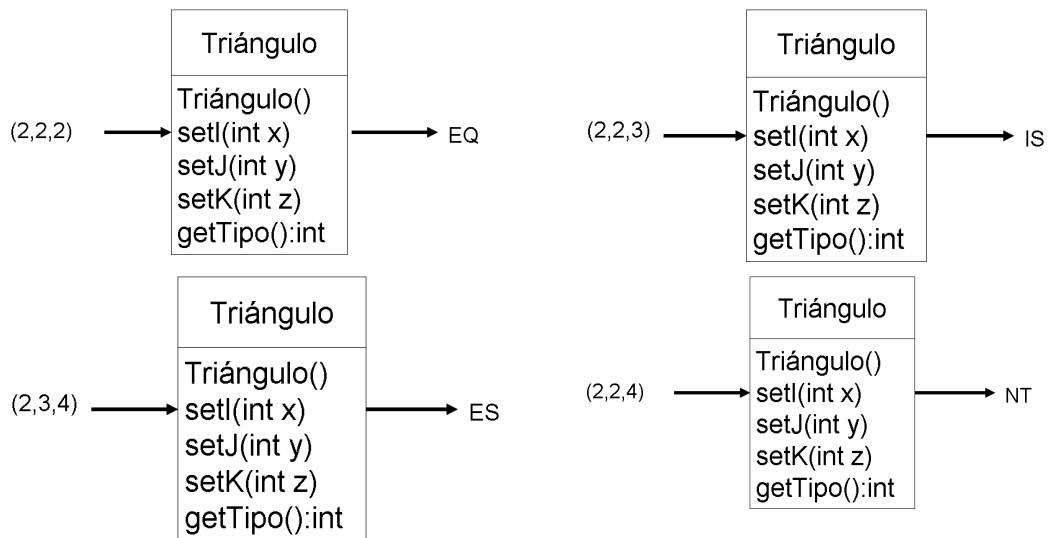


Figura 1. El problema del triángulo, bajo un enfoque de caja negra

Como puede comprobarse, si los casos de prueba de caja negra se superan, el ingeniero de pruebas estará seguro de que la clase bajo prueba (la *Class Under Test* en inglés, que habitualmente se abrevia con sus siglas *CUT*) se comporta correctamente para los datos de prueba utilizados en esos casos de prueba: en el caso del triángulo isósceles, por ejemplo, en la figura se comprueba que un triángulo con longitudes 2, 2 y 3 es isósceles; ahora bien, ¿la implementación de la clase determinará también que el triángulo es isósceles para lados de longitudes 2, 3 y 2? ¿Y para 3, 2 y 2? Las pruebas deben ser tan completas como sea posible, si bien la *prueba completamente exhaustiva* (comprobación de absolutamente todo el comportamiento del sistema o la clase para cualesquiera posibles valores de entrada) es normalmente imposible.

Las pruebas de caja negra pueden aplicarse no sólo a una clase o pequeño programa, sino que son aplicables también a un subsistema o, incluso, al sistema completo mediante *pruebas funcionales* o *testing exploratorio*:

- En las *pruebas funcionales*, se ejecutan las diferentes funcionalidades del sistema deteniéndonos únicamente en la comprobación de que los resultados proporcionados por cada una de las funcionalidades del sistema sean correctos. Estas pruebas requieren conocer los requisitos del sistema: los casos de prueba se diseñan a partir de dichos requisitos.

En las segundas, no es necesario contar con los casos de prueba antes de ejecutar las pruebas. El diseño de las pruebas se realiza a medida que se ejecutan los propios casos de prueba, utilizando la información obtenida mientras se va probando para diseñar nuevas y mejores pruebas. El testing exploratorio se define como el aprendizaje, el diseño de casos de prueba y la ejecución de las pruebas de forma simultánea<sup>11</sup>. Debe tomarse nota de lo que se hizo y lo que sucedió.

Con el fin de alcanzar una mayor seguridad respecto del comportamiento correcto del sistema se introduce la idea de las pruebas estructurales o de caja blanca.

## 2 Pruebas estructurales o de caja blanca

Las pruebas de caja blanca realizan, de alguna manera, un seguimiento del zonas del sistema que van ejecutando los casos de prueba. Si el artefacto que se está probando es un programa, se determinan de manera concreta las instrucciones, bloques, etc. que han sido ejecutados por los casos de prueba. En este caso, este tipo de pruebas permiten conocer *cuánto código se ha recorrido*.

Así, en el mismo problema del triángulo que comentábamos antes, el ingeniero de pruebas se fijará ahora en el código que implementa su funcionalidad y observará, para cada terna de entradas, el recorrido seguido por los casos de prueba en la implementación de la clase (Figura 2).

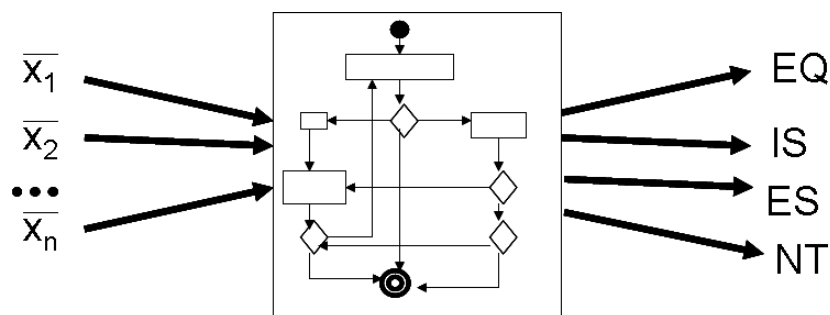


Figura 2. El problema del triángulo, desde un punto de vista de caja blanca

Dependiendo por ejemplo de las sentencias visitadas por los casos de prueba o (si se está considerando el diagrama de flujo del programa), dependiendo de las ramas, caminos o nodos visitados por los casos de prueba, el ingeniero estará más o menos seguro de la buena, muy buena o mediana calidad del software

<sup>11</sup> Bach J. "What is Exploratory Testing? And How it differs from Scripted Testing" StickyMinds, Enero 2001.

objeto de estudio. Existen formas muy variadas de medir esa “cantidad” de código recorrido mediante lo que se llaman “criterios de cobertura”.

Para Cornet<sup>12</sup>, el análisis de cobertura del código es el proceso de:

- Encontrar fragmentos del programa que no son ejecutados por los casos de prueba.
- Crear casos de prueba adicionales que incrementen la cobertura.
- Determinar un valor cuantitativo de la cobertura (que es, de manera indirecta, una medida de la calidad del programa).

Adicionalmente, el análisis de cobertura también permite la identificación de casos de prueba redundantes, que son aquellos que recorren zonas del sistema ya recorridas por otros casos, por lo que no incrementan la cobertura.

### 3 Pruebas unitarias

Las pruebas unitarias centran su aplicación en lo que se denomina la “unidad de prueba” que, dependiendo del contexto, puede ser una clase, un método o un subsistema. El estándar ANSI/IEEE 1008/1987<sup>13</sup>, define la unidad de prueba de la siguiente forma:

Un conjunto de uno o más módulos de un programa, junto a los datos de control asociados (por ejemplo, tablas), procedimientos de uso y procedimientos de operación que satisfagan las siguientes condiciones:

- (1) Todos los módulos pertenecen a un único programa
- (2) Al menos uno de los módulos nuevos o cambiados del conjunto no ha pasado las pruebas unitarias (puesto que una unidad de prueba puede contener uno o más módulos previamente probados)
- (3) El conjunto de módulos junto con sus datos y procedimientos asociados son el único objetivo del proceso de pruebas

En general, en orientación a objetos se asume que la unidad de prueba es la clase, por lo que se comprueba que el *estado en el que queda la instancia de la clase que se está probando* es correcto para los datos que se le pasan como entrada. En el caso de las pruebas unitarias de caja negra para un sistema orientado a objetos, se entiende la clase como, en efecto, una caja cuyo interior no interesa: lo único que

---

<sup>12</sup> Cornett S. (2002). *Code Coverage Analysis*. Disponible en: <http://www.bullseye.com/webCoverage.html>

<sup>13</sup> IEEE (1987). *IEEE Standard for Software Unit Testing*.

importa desde este punto de vista es el conjunto de entradas suministradas y las salidas obtenidas.

#### **4 Pruebas de integración**

Las pruebas de integración se emplean para comprobar que las unidades de prueba, que han superado sus pruebas de unidad, funcionan correctamente cuando se integran, de manera que lo que se tiende a ir probando es la arquitectura software. Durante la integración, las técnicas que más se utilizan son las de caja negra, aunque se pueden llevar a cabo algunas pruebas de caja blanca para asegurar que se cubren los principales flujos de comunicación entre las unidades<sup>14</sup>.

En el contexto de la orientación a objetos, las pruebas de integración pretenden asegurar que los mensajes que fluyen desde los objetos de una clase o componente se envían y reciben en el orden adecuado en el objeto receptor, así como que producen en éste los cambios de estado que se esperaban<sup>15</sup>.

#### **5 Pruebas de sistema**

Las pruebas de sistema tienen por objetivo comprobar que el sistema, que ha superado las pruebas de integración, se comporta correctamente con su entorno (otras máquinas, otro hardware, redes, fuentes reales de información, etc.). Las pruebas funcionales forman parte de las pruebas del sistema, al igual que las siguientes:<sup>14</sup>:

- 1) Pruebas de recuperación. Consisten en forzar el fallo del software y comprobar que la recuperación se lleva a cabo de manera correcta, devolviendo al sistema a un estado coherente.
- 2) Pruebas de seguridad. Intentan verificar que los mecanismos de protección incorporados al sistema lo protegerán, de hecho, de penetraciones inadecuadas.
- 3) Pruebas de resistencia. Estas pruebas están diseñadas para que el sistema requiera recursos en cantidad, frecuencia o volumen

---

<sup>14</sup> Pressman RS. (1993). *Ingeniería del Software, un enfoque práctico (3ª Edición)*. McGraw-Hill.

<sup>15</sup> Gallagher L, Offutt J and Cincotta A. (2006). *Integration testing of object-oriented components using finite state machines*. Software Testing, Verification and Reliability, 16.

anormales. La idea es intentar que el sistema se venga abajo por la excesiva tensión a la que se lo somete.

- 4) Pruebas de rendimiento. En sistemas de tiempo real o sistemas empujados, es inaceptable que el software proporcione las funciones requeridas fuera de las condiciones de rendimiento exigidas.

## Capítulo III. Criterios de cobertura para artefactos software

*A la hora de probar un artefacto software, al ingeniero de pruebas le interesa conocer las porciones del sistema que sus casos de prueba están recorriendo. De este modo, el tester puede añadir más casos de prueba para buscar errores en esas zonas inexploradas. Para esto se utilizan los denominados criterios de cobertura, que permiten conocer cuantitativamente la “cantidad de producto” que se está probando. En este capítulo se repasan algunos de ellos.*

### 1 Criterios de cobertura

Cuando el *tester* ejecuta un conjunto de casos de prueba sobre un sistema, aquellos avanzan a través de éste recorriendo las regiones que correspondan a los servicios del sistema a los que llama cada caso. La cobertura alcanzada puede medirse de acuerdo con muy diversos criterios, unos más estrictos y otros más flexibles: para probar el comportamiento de una clase, por ejemplo, el *tester* puede conformarse con invocar una sola vez a cada método público (sería, claramente, un criterio de cobertura muy pobre) o con pasar una vez por cada una de sus sentencias (el cual es otro criterio algo más estricto que el anterior).

De manera general, se utilizará un criterio u otro dependiendo de alguna característica del sistema o de los elementos que lo componen, como su criticidad o su frecuencia de uso. Por otro lado, y aunque lo habitual es utilizar criterios para medir la cobertura de código fuente, también se han definido criterios para otro tipo de artefactos, como máquinas de estado, diagramas de clase o diagramas de interacción.

### 2 Utilidad de los criterios de cobertura

Conocer la cobertura que un conjunto de casos de prueba (*test suite*) alcanza sobre el sistema que se está probando (el *system under test*, en inglés, o *SUT*) permite al *tester* determinar las porciones del sistema que no están siendo recorridas por los casos de prueba. Para probar de manera más exhaustiva el sistema, el *tester* debería añadir más casos de prueba al *test suite* con objeto de que se compruebe el funcionamiento del sistema en esas zonas cuyo comportamiento no se ha explorado.

Además, alcanzar una cobertura alta en un sistema con un *test suite* que no encuentra errores permite disponer de una garantía más o menos alta acerca de la calidad de dicho sistema, que dependerá además del criterio de cobertura utilizado.

### **3 Un posible modelo de trabajo**

A la hora de probar un sistema o algunas de sus porciones (funcionalidades, componentes, módulos, subsistema, etcétera), el *tester* debe determinar un criterio y un valor umbral para la cobertura que utilizará como criterio de parada: así, por ejemplo, puede establecerse que se dejará de probar cuando se haya recorrido el 90% de las sentencias del programa, o cuando se hayan ejecutado todos los escenarios de un cierto requisito funcional, o cuando se haya pasado por todas las transiciones de una máquina de estados.

Existen varios modelos de proceso para llevar a cabo las pruebas (dependen, por ejemplo, de si el equipo de pruebas está integrado en el equipo de desarrollo o es un equipo independiente, o de si la organización que desarrolla el sistema externaliza el proceso a una tercera empresa, o de si se sigue alguna metodología como TMAP o TPI); la Figura 3 muestra una propuesta de modelo de proceso para la realización de pruebas:

1. Una vez seleccionado el sistema o la porción de él que se va a probar, se determina a priori el valor umbral que se utilizará como criterio de parada:
2. Una vez determinado dicho valor, se construye el conjunto de casos de prueba (ciertamente, los casos podrían haber sido contruidos antes: si se está probando el código que implementa un requisito funcional, quizá los casos ya estaban diseñados desde la etapa inicial de especificación de requisitos) y se ejecutan sobre el sistema con objeto de encontrar errores.
3. Si se encuentran errores, el sistema se debe corregir por parte del equipo de desarrollo (en este caso, si el equipo de test está separado del equipo de desarrollo o pertenece a una tercera organización a la que se ha contratado, puede que se redacte un solo informe con todos los fallos encontrados).
4. Cuando los casos ya no encuentran errores, se mide la cobertura alcanzada:

- a. Si se ha alcanzado el umbral que se preestableció, el proceso termina, y que los casos de prueba han recorrido suficientemente el sistema sin hallar error alguno.
- b. En otro caso, deben añadirse más casos de prueba al *test suite* para que se incremente la cobertura y se alcance el valor umbral, de manera que los casos recorran “tanto sistema” como se deseaba.

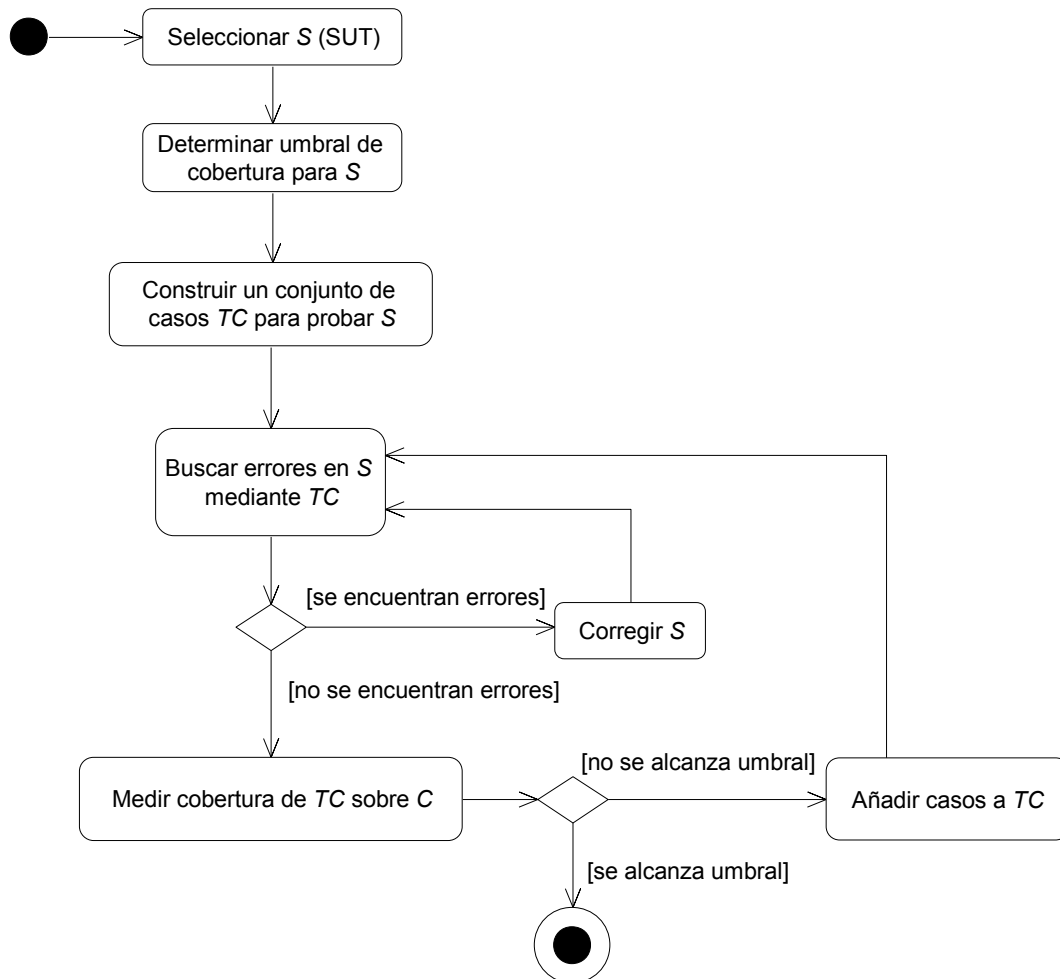


Figura 3. Un posible proceso de pruebas

## 4 Criterios de cobertura para código fuente

### 4.1 Cobertura de sentencias

Este criterio se verifica cuando los casos de prueba recorren todas las sentencias del programa. El lado izquierdo de la siguiente figura muestra un trozo de código que implementa el método *getType* de la clase *TriTyp*, que representa el problema de la determinación del tipo de un Triángulo (*Triangle-Type*) que ya comentamos



en la página 8. En color verde aparecen las sentencias del programa que han sido recorridas por los casos de prueba de la derecha; las sentencias que aparecen en rojo no han sido alcanzadas. Así, si establecemos como valor umbral el recorrer el 100% de las sentencias del programa, el *tester*, de acuerdo con el proceso propuesto en la Figura 3, debería añadir nuevos casos de prueba al *test suite*.

<pre> /**  *  * @return 1 if <u>scalene</u>; 2 if <u>isosceles</u>;  3 if <u>equilateral</u>; 4 if not a triangle  */ public void getType() {     if (i == j) {         trityp = trityp + 1;     }     if (i == k) {         trityp = trityp + 2;     }     if (j == k) {         trityp = trityp + 3;     }      if (i &lt;= 0    j &lt;= 0    k &lt;= 0) {         trityp = 4;         return;     }      if (trityp == 0) {         if (i + j &lt;= k    j + k &lt;= i                i + k &lt;= j) {             trityp = 4;             return;         } else {             trityp = 1;             return;         }     }      if (trityp &gt; 3) {         trityp = 3;     } else if (trityp == 1 &amp;&amp; i + j &gt; k) {         trityp = 2;     } else if (trityp == 2 &amp;&amp; i + k &gt; j) {         trityp = 2;     } else if (trityp == 3 &amp;&amp; j + k &gt; i) {         trityp = 2;     } else {         trityp = 4;     } } </pre>	<pre> import junit.framework.TestCase;  public class TriTypTest extends TestCase {      public void testEQUILATERO() {         TriTyp t=new TriTyp();         t.setI(5);         t.setJ(5);         t.setK(5);         t.getType();         assertTrue(t.trityp==TriTyp.EQUILATERAL);     }      public void testISOSCELES() {         TriTyp t=new TriTyp();         .setI(5);         t.setJ(5);         t.setK(8);         t.getType();         assertTrue(t.trityp==TriTyp.ISOSCELES);     }      public void testESCALENO() {         TriTyp t=new TriTyp();         t.setI(4);         t.setJ(5);         t.setK(8);         t.getType();         assertTrue(t.trityp==TriTyp.SCALENE);     }      public void testNOTTRIANGULO() {         TriTyp t=new TriTyp();         t.setI(4);         t.setJ(5);         t.setK(9);         t.getType();         assertTrue(t.trityp==TriTyp.NOT_A_TRIANGLE);     } } </pre>
--	--

Figura 4. Código recorrido (izquierda) por los casos de prueba de la derecha

#### 4.2 Cobertura de decisiones, de ramas o de todos los arcos

Una *decisión* es una secuencia de condiciones. Una *condición* es una expresión algebraica que consta de dos expresiones relacionadas con un operador relacional (<, >, =, >=, <=, <>). El criterio de *cobertura de decisiones*

(también llamado de todas las ramas, *all branches*, o de todos los arcos, *all edges*) se satisface cuando cada decisión se evalúa a *true* y a *false* al menos una vez.

En el ejemplo del problema *TriTyp* (Figura 4), la decisión  $(i+j \leq k \ || \ j+k \leq i \ || \ i+k \leq j)$  se considerará recorrida de acuerdo con este criterio cuando se entre a los dos bloques *true* y *false*: para recorrer la rama *true*, basta con hacer cierta cualquiera de las tres condiciones de la decisión (pues son tres condiciones separadas con *or*); para que sea *false*, deberán ser también *false* las tres condiciones.

Así, para esa decisión, con los casos de prueba *testNOTRIANGULO* y *testEQUILATERO* (mostrados en el lado derecho de la Figura 4) se verifica este criterio, ya que el primero consigue hacer cierta la decisión, y el segundo la hace falsa.

#### 4.3 Cobertura de condiciones

Este criterio requiere que cada condición de cada decisión se evalúe a *true* y a *false* al menos una vez.

Volviendo al ejemplo de la decisión anterior  $(i+j \leq k \ || \ j+k \leq i \ || \ i+k \leq j)$ , los casos de prueba habrán cumplido el criterio de condiciones cuando  $i+j \leq k$  haya sido verdadero y falso,  $j+k \leq i$  haya sido también verdadero y falso, e  $i+k \leq j$  haya sido también verdadero y falso: respecto de los valores *false*, el mismo caso *testEQUILATERO* es sencillo, pues las tres condiciones toman valor *false*. Respecto de los valores *true*, un caso en el que le pasáramos los valores  $(0, 0, 0)$  la cubriría en apariencia, pero realmente la decisión sería inalcanzable porque el valor del campo *trityp* sería 4 por la decisión anterior  $(i \leq 0 \ || \ j \leq 0 \ || \ k \leq 0)$ , señalada con tres asteriscos (\*\*\*) en la Figura 5. Para alcanzar la decisión que nos interesa, debemos llegar a ella con *trityp*=0, para lo que ni *i*, ni *j*, ni *k* pueden ser iguales (dos de ellas) ni ser menores o iguales a cero. Los valores  $(2, 3, 5)$  hacen que *trityp* sea cero y que se entre en la rama *true* de la decisión gracias a que la primera condición  $(i+j \leq k)$  es verdadera; para cubrir las otras dos condiciones, los casos podrían ser, respectivamente,  $(5, 3, 2)$  y  $(2, 3, 5)$ . Por tanto, esa decisión sería cubierta con el criterio de cobertura de condiciones con cuatro casos de prueba.

```

public void getType() {
    if (i == j) {
        trityp = trityp + 1;
    }
    if (i == k) {
        trityp = trityp + 2;
    }
    if (j == k) {
        trityp = trityp + 3;
    }

    if (i <= 0 || j <= 0 || k <= 0) {          ***
        trityp = 4;
        return;
    }
    if (trityp == 0) {
        if (i + j <= k || j + k <= i || i + k <= j) {
            trityp = 4;
            return;
        } else {
            trityp = 1;
            return;
        }
    }
}
...

```

Figura 5. Fragmento del código de `getType()`

#### 4.4 Cobertura de condiciones/decisiones (*Decision/Condition coverage*, o *DCC*)

Este criterio requiere que cada condición de cada decisión se evalúe a *true* y a *false* al menos una vez, y que cada decisión se evalúe también a *true* y a *false* al menos una vez.

Aunque en apariencia pueda parecer que el cubrimiento de uno u otro criterio pudiera implicar el cubrimiento del contrario, los siguientes contraejemplos demuestran que no es así:

- En el caso de la Tabla 1, se verifica el criterio de *decisiones* (la decisión completa toma los valores *true* y *false*), pero no el de *condiciones*, pues la condición *C* no llega a tomar el valor *true*.

A and (B or C)			
Condiciones			Decisión
A	B	C	A and (B or C)
true	true	false	true
false	false	false	false

Tabla 1. Se verifica el criterio de *decisiones*, pero no el de *condiciones*

- La decisión de la Tabla 2, que consta de dos condiciones separadas por un *or exclusivo* (la decisión toma valor *true* cuando exclusivamente uno de los dos valores es *true*), se verifica el criterio de *condiciones* (ambas toman los valores *true* y *false*), pero no el de *decisiones* (la decisión siempre vale *false*).

A xor B		
Condiciones		Decisión
A	B	A xor B
true	true	false
false	false	false

Tabla 2. Se verifica el criterio de *condiciones*, pero no el de *decisiones*

Para que se verifique condiciones/decisiones en el segundo ejemplo, el *tester* debería añadir a los casos de la Tabla 2 o bien *(true, false)* o bien *(false, true)*, ya que se haría cierta la decisión completa. Para este mismo ejemplo, otro posible *test suite* estaría formado por los casos mostrados en la Tabla 3, ya que cada condición se evalúa a *true* y a *false* al menos una vez, y también la decisión toma los dos valores booleanos.

A xor B		
Condiciones		Decisión
A	B	A xor B
true	true	false
true	false	true
false	true	true

Tabla 3. Se verifican condiciones, decisiones y condiciones/decisiones

Para el caso de la decisión  $(i \leq 0 \parallel j \leq 0 \parallel k \leq 0)$  del problema del triángulo (Figura 5), un posible *test suite* que verifica condiciones/decisiones podría estar formado por los dos siguientes casos:  $(0, 0, 0)$  (que hace ciertas las tres condiciones y también la decisión) y  $(7, 8, 9)$  (que hace falsas las tres condiciones y también la decisión): sin embargo, este *test suite* resulta a todas luces muy pobre, por lo que el *tester* debería aplicar cierta “picardía” a la hora de diseñar los casos de prueba.

#### 4.5 Cobertura múltiple de condiciones (*Multiple Condition Coverage, MCC*)

Este criterio requiere que todas las condiciones tomen valor *true* y *false*, de manera que se recorra la tabla de verdad completa de la decisión.

Para la misma decisión de la Tabla 1 se necesitarían los  $2^3=8$  casos que se muestran en la Tabla 4. Este criterio de cobertura es sin duda el más completo, pero requiere un número de casos muy grande para cubrir cada decisión.

A and (B or C)			
Condiciones			Decisión
A	B	C	A and (B or C)
true	true	true	true
true	true	false	true
true	false	true	true
true	false	false	false
false	true	true	false
false	true	false	false
false	false	true	false
false	false	false	false

Tabla 4. Cobertura de condiciones

#### 4.6 Cobertura modificada de condiciones/decisiones (*Modified Condition/Decision Coverage, MC/DC*).

Este criterio se verifica cuando cada posible valor de una condición determina la salida de la decisión al menos una vez.

Supongamos la decisión  $D=A \text{ or } B \text{ or } C$ . En primer lugar, debemos conseguir que  $A$  sirva para determinar la salida de  $D$  como *true* y como *false*, independientemente del valor de  $B$  y de  $C$ . Para que  $D$  valga *true* dependiendo de  $A$ ,  $A$  debe valer *true* y las otras dos condiciones deben ser *false*; para que  $D$  valga *false* gracias a  $A$ ,  $A$  debe valer *false*, así como las otras dos. Lo mismo podemos decir de  $B$  y  $C$ . Así, cuando relacionamos dos condiciones con el operador *or*, el valor *false* hace que no se afecte a la condición que se está considerando: el valor *false* es el denominado *valor neutral* del operador *or*.

Para el caso de la decisión de la Tabla 4 ( $A \text{ and } (B \text{ or } C)$ ) sucede lo mismo:

1. Debemos ir a la rama *true* gracias a la contribución de  $A$  (para lo cual  $A$  debe valer *true*) sin fijarnos en las contribuciones ni de  $B$  ni de  $C$ . Para que esto ocurra, o bien  $B$  o bien  $C$  deben valer *true*. Tomamos, por ejemplo, el caso (*true, true, false*).

2. Debemos ir a la rama *false* gracias a la contribución de *A* y no a la de *B* ni a la de *C*. Para ello, hacemos *A=false, B=true, C=true*. El caso es, por tanto, *(false, true, true)*.
3. Debemos recorrer la rama *true* gracias a la contribución de *B*, sin fijarnos ni en *A* ni en *C*. Para ello, hacemos *C=false*, no teniendo más remedio que poner *A=true* (porque *A* está separado de *(B or C)* con un *and*). El caso de prueba, ahora, es también *(true, true, false)*, que es el que ya construimos en el punto 1.
4. Del mismo modo, debemos recorrer la rama *false* gracias a la contribución de *B*, sin fijarnos ni en *A* ni en *C*: hacemos *B=false, A=false, B=true*, quedando el caso *(false, false, true)*.
5. Para la condición *C*, tenemos que recorrer la rama *true* de la decisión gracias a que *C* tome valor *true*: hacemos *C=true, B=false, A=true*, quedando por tanto *(true, false, true)*.
6. Para ir a la rama *false* de la decisión gracias a *C*, hacemos *C=false, B=true, A=false*, con lo que el caso es *(false, true, false)*.

Los casos se resumen en la Tabla 5: obsérvese que la decisión toma los valores *true* y *false* gracias a los valores *true* y *false* que van tomando las condiciones que, en la tabla, hemos denominado “dominantes”.

A and (B or C)				
Condiciones			Decisión	
A	B	C	A and (B or C)	Condición “dominante”
true	true	false	true	A, B
false	true	true	false	A
false	false	true	false	B
true	false	true	true	C
false	true	false	false	C

Tabla 5. Casos de prueba para conseguir cobertura modificada de condición/decisión

Para el ejemplo de la decisión *A xor B* el mecanismo es parecido:

1. Hay que tomar *true* gracias a la contribución de *A* y no a la de *B*: tomamos el caso *(true, false)*. Para ir a la rama *false*, tomamos el caso *(false, false)* o *(true, true)*, que sirve como contribución de las dos condiciones.

- Hay que tomar las ramas *true* y *false* gracias a *B*, para lo que construimos el caso *(false, true)*.

A xor B		
Condiciones		Decisión
A	B	A xor B
true	false	true
false	false	false
false	true	true

Tabla 6. Casos de prueba para conseguir *cobertura modificada de condición/decisión* en *A xor B*

La decisión *A xor B* es equivalente a *(A and not B) or (not A and B)*. Por tanto, la aplicación de este criterio de cobertura a la versión larga de la decisión debería dar los mismos resultados. Veámoslo:

- Para irnos a la rama *true* gracias a *A*, *A* debe valer *true* y *B* debe valer *false*, con lo que tenemos el caso *(true, false)*.
- Para irnos a la rama *false* gracias a *A*, *A* debe valer *false* y *B* *false*, con lo que el caso es *(false, false)*, o bien *A true* y *B true*, siendo el caso *(true, true)*, pudiéndonos quedar con cualquiera de los dos. Por obtener el mismo resultado que en la Tabla 6, nos quedamos con el primero.
- Procedemos del mismo modo con *B*, quedándonos con *(false, true)* para alcanzar la rama *true*, y con *(false, false)* o *(true, true)* para la rama *false*.

(A and not B) or (not A and B)					
Condiciones				Decisión	
A	B	A and not B	not A and B	(A and not B) or (not A and B)	Condición dominante
true	false	true	false	true	A
false	false	false	false	false	A, B
false	true	false	true	true	B

Tabla 7. Casos de prueba para conseguir *cobertura modificada de condición/decisión* en la versión extendida de *A xor B*

El criterio *MC/DC* produce menos casos de prueba que *MCC* y, sin embargo, es muy difícil encontrar un fallo con *MCC* que pase inadvertido para *MC/DC*. Por ello, este criterio de cobertura se exige en la prueba de sistemas críticos.

## 5 Criterios de cobertura para máquinas de estado

Una máquina de estados es un grafo dirigido cuyos nodos representan estados y cuyos arcos representan transiciones entre dichos estados. En desarrollo de software, las máquinas de estado tienen dos usos principales:

1. Describir el comportamiento de instancias de clases. De manera general (hay muchos tipos de estados en la especificación de UML 2.0), los nodos representan los diferentes estados en los que puede encontrarse la instancia, siendo el estado de la instancia una función de los valores de sus campos. También de manera general (porque hay muchos tipos de transiciones), las transiciones representan operaciones de la clase que provocan el paso de un estado a otro y, quizá, la ejecución de otras operaciones “colaterales”.
2. Describir los pasos en la ejecución de casos de uso. En este caso, cada estado representa un paso, y la transición entre un estado y otro significa que el paso previo ha terminado de ser ejecutado.

La estructura de la máquina de estados puede utilizarse para diseñar casos de prueba que la recorran atendiendo a algún criterio de cobertura. A continuación se describen los criterios propuestos por Offutt et al.<sup>16</sup>

### 5.1 Cobertura de estados

Este criterio se satisface cuando los casos de prueba recorren todos los estados de la máquina. Así, para la máquina de la Figura 6, un *test suite* que satisface este criterio estaría formado por la secuencia de transiciones ( $m1, m2, m3, m4$ ).

---

<sup>16</sup> Offutt AJ, Liu S, Abdurazik A and Amman P. (2003). *Generating test data from state-based specifications*. Software Testing, Verification and Reliability (13), 25-53.



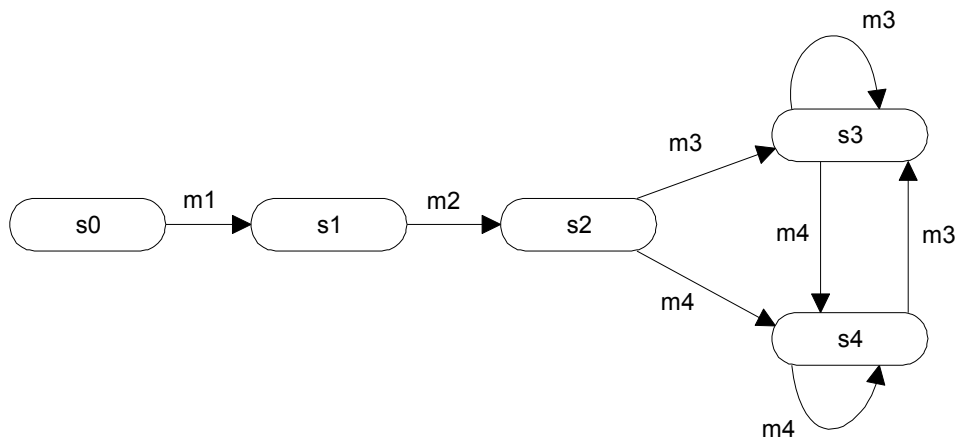


Figura 6. Una sencilla máquina de estados

## 5.2 Cobertura de transiciones

Este criterio se satisface cuando el conjunto de casos de prueba contiene casos que provocan el disparo de todas las transiciones contenidas en la máquina de estados.

Para la máquina de la Figura 6, un test suite que satisface este criterio necesitaría al menos dos casos de prueba, como por ejemplo:  $(m1, m2, m3, m3, m4, m4, m3)$  y  $(m1, m2, m4)$ .

## 5.3 Cobertura de pares de transiciones

Este criterio se satisface cuando el conjunto de casos de prueba contiene casos de prueba que recorren, para un estado dado, todos los pares de transiciones de entrada y salida de ese estado.

Así, para la máquina de estados de la Figura 7, este criterio será satisfecho si el conjunto de casos de prueba contiene casos que recorran  $(m1, m3)$ ,  $(m1, m4)$ ,  $(m1, m5)$ ,  $(m2, m3)$ ,  $(m2, m4)$  y  $(m2, m5)$ .

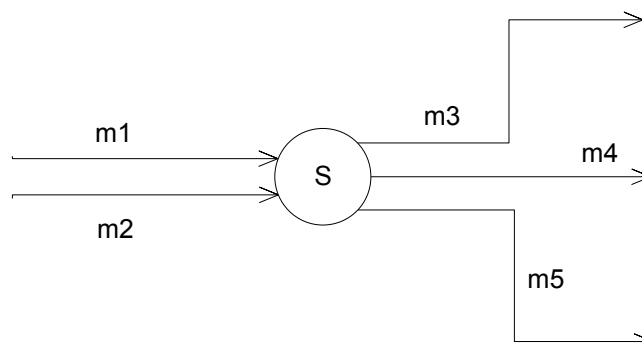


Figura 7. Un fragmento de una máquina de estados

Para el caso de la máquina de la Figura 6:

1. Como  $s0$  no tiene transiciones de entrada, el criterio no es aplicable.
2. Respecto de  $s1$ , hay que recorrer el par  $(m1, m2)$ .
3. Respecto de  $s2$ , hay que recorrer los pares  $(m2, m3)$  y  $(m2, m4)$ .
4. Respecto de  $s3$ , consideramos tres transiciones  $m3$  (que pueden corresponder a la misma operación de la clase, pero que se disparan desde estados distintos):
  - a. Respecto de la transición  $m3$  que procede de  $s2$ , construiremos los pares  $(m3, m3)$ ,  $(m3, m4)$ .
  - b. Respecto de la transición  $m3$  que procede de  $s4$ , construimos también los pares  $(m3, m3)$ ,  $(m3, m4)$ , pero que representan instancias diferentes de la misma operación.
  - c. Análogamente, para la transición  $m3$  que procede de  $s3$  y que también incide en  $s3$ , consideramos los pares  $(m3, m3)$  (la misma instancia de  $m3$  en este caso: es decir, la misma transición que entra es la que sale) y  $(m3, m4)$ .
5. La situación para  $s4$  es muy parecida, debiendo recorrer los pares  $(m4, m4)$ ,  $(m4, m3)$  (respecto de la transición procedente de  $s2$ ),  $(m4, m4)$ ,  $(m4, m3)$  (respecto de la procedente de  $s3$ ) y  $(m4, m4)$ ,  $(m4, m3)$  (respecto de la que sale de  $s4$ ).

#### 5.4 Cobertura de secuencia completa

Este criterio se satisface cuando el conjunto de casos prueba contiene casos que recorren los caminos (formados por transiciones) más significativos, que son aquellos que ha seleccionado el ingeniero de pruebas, que ha propuesto el usuario o cliente, etcétera.

## Capítulo IV. Valores de prueba

*Con objeto de encontrar pronto los errores en el sistema, el tester debe escribir buenos casos de prueba que traten de ejercitar el sistema en todas sus formas posibles. Para ello, los servicios invocados sobre el sistema deben recibir buenos valores como parámetros, que procederán de la propia experiencia del ingeniero de pruebas, de entrevistas con los usuarios o clientes, o de la aplicación de determinadas técnicas. En este capítulo se describen algunas de ellas.*

### 1 Clases o particiones de equivalencia

Con la técnica de particiones de equivalencia, se divide el dominio de entrada de cada parámetro en conjuntos disjuntos (cada uno es una *partición* o *clase de equivalencia*), asumiéndose que el sistema bajo prueba se comportará igual para cualquier valor de la clase.

Supongamos una clase que representa una cuenta corriente de un banco. La Figura 8 muestra el código de su operación *transferir*, que realiza una transferencia de dinero desde la instancia de *Cuenta* sobre la que se ejecuta la operación, y un ingreso sobre la *cuentaDestino* que se pasa como primer parámetro. Por la realización de la transferencia el banco cobra una comisión del 1% sobre el importe transferido, con un mínimo de 2 euros. Esto produce dos movimientos sobre la cuenta origen y uno sobre la cuenta destino:

- Sobre la cuenta origen se producen dos movimientos de retirada: uno por el *importe* que se pasa como parámetro, y otro por la comisión.
- Sobre la cuenta destino se produce un ingreso por el *importe* transferido.

Para que la transferencia se lleve a cabo deben darse varias condiciones: (1) el *importe* que se transfiere debe ser positivo (en caso contrario, la operación *retirar*, a la que se llama desde *transferir*, lanza una *ImporteNegativoException*); (2) la cuenta debe disponer de saldo suficiente (mayor o igual que el *importe* más la *comisión*; en caso contrario, se lanza una *SaldoInsuficienteException*); y (3) la cuenta destino debe existir (si no, se lanza la *CuentaInexistenteException*). Si se dan las dos condiciones, se realizan dos movimientos de retirada sobre la cuenta

origen y uno de ingreso sobre la de destino. Si se produce algún error de acceso a la base de datos, se lanza una *SQLException*.

```
...
public class Cuenta {
    protected String numero;
    protected java.util.Date fechaApertura;
    protected Cliente titular;
    protected double saldo;
    ...

    public void transferir(Cuenta cuentaDestino, double importe, String concepto)
        throws SQLException, ImporteNegativoException,
            SaldoInsuficienteException, CuentaInexistenteException {
        double comision=2.0;
        if (importe>300)
            comision=importe*0.01;
        if (!existe(cuentaDestino))
            throw new CuentaInexistenteException();
        this.retirar(importe, "Transferencia emitida a " +
            cuentaDestino.getNumero());
        this.retirar(comision, "Comisión por transferencia");
        cuentaDestino.ingresar(importe, concepto);
    }
    ...
}
```

Figura 8. Código de la operación *transferir* de la clase *Cuenta*

De acuerdo con la técnica de las clases de equivalencia, debemos obtener particiones para los tres parámetros:

1. Para *cuentaDestino*, los conjuntos disjuntos pueden estar formados por *todas las cuentas que existen en el banco y todas las cuentas que no existen*.
2. Para el *importe* tenemos diferentes escenarios: por un lado, podemos suponer una cuenta que tiene saldo suficiente para realizar una transferencia con la comisión que corresponda; por otro, podemos suponer una cuenta que no tenga saldo suficiente. En ambos casos, puede ocurrir que se retiren más de 300 euros o menos, consideración que habría que tomar en cuenta porque la comisión que se percibe es distinta en ambos casos.
3. Del punto anterior se observa que los posibles valores que tome el *importe* afectan al comportamiento de la instancia de la cuenta emisora que, si bien no se pasa como parámetro, sí influye decisivamente en el escenario de ejecución. Por ello, añadiremos el objeto *this* (que representará la cuenta ordenante) a la colección de elementos para los que identificaremos clases de equivalencia.

4. Para el *concepto* identificaremos valores de prueba más adelante mediante la técnica de conjetura de errores.

Parámetro	Clases de equivalencia	Valores seleccionados
<i>this</i> (cuenta ordenante)	cuenta con saldo en $[0, 300]$ cuenta con saldo en $(300, +\infty)$	cuenta con saldo 200 cuenta con saldo 500
<i>cuentaDestino</i>	una cuenta que existe una cuenta que no existe	cuenta "123456" cuenta "000000"
<i>importe</i>	$(-\infty, 0]$ $(0, 300)$ $[300, +\infty)$	-100 +100 +400
<i>concepto</i>	"Pago de alquiler"	"Pago de alquiler"

Figura 9. Clases de equivalencia identificadas para la operación *transferir* y valores seleccionados de cada una

## 2 Valores límite (*Boundary values*)

La técnica de valores límite complementa la anterior: en lugar de elegir cualquier valor de la clase de equivalencia, se seleccionan los valores situados en los límites de las clases de equivalencia. Existen dos variantes de esta técnica:

1. En la *variante ligera*, por cada clase de equivalencia se toman como valores de prueba los propios valores límite de la clase y los valores adyacentes de las clases de equivalencia adyacentes
2. En la *variante pesada*, se toma además el valor inmediatamente inferior al límite, perteneciente a la clase de equivalencia que se está considerando.

Para el caso de la operación *retirar*, consideraríamos los parámetros que puedan ser enumerables (ni *cuentaDestino* ni *concepto* lo son; incluso la cuenta ordenante tampoco lo es, aunque sí su saldo, que es el elemento compositivo de su estado que nos interesa). En la siguiente tabla se han añadido, a los resultados de la tabla anterior, los valores procedentes de la variante ligera de los valores límite, señalándose con dos asteriscos (\*\*).

Parámetro	Clases de equivalencia	Valores seleccionados
<i>this</i> (cuenta ordenante)	cuenta con saldo en $[0, 300]$ cuenta con saldo en $(300, +\infty)$	cuenta con saldo 200 cuenta con saldo 500 cuenta con saldo 300 ** cuenta con saldo 301 **
cuentaDestino	una cuenta que existe una cuenta que no existe	cuenta "123456" cuenta "000000"
importe	$(-\infty, 0]$ $(0, 300)$ $[300, +\infty)$	-100 +100 +400 0 ** 1 ** 300 ** 301 **
concepto	"Pago de alquiler"	"Pago de alquiler"

Tabla 8. Valores de prueba seleccionados, a los que se han añadido los procedentes de la variante ligera de valores límite

### 3 Conjetura de errores (*error-guessing*)

Mediante la técnica de la *conjetura de errores*, el ingeniero de pruebas propone valores para los que, de alguna manera, intuye que el sistema puede mostrar un comportamiento erróneo.

Para el *concepto*, por ejemplo, que se trata de una cadena de caracteres, pueden utilizarse la cadena vacía y una cadena con más de 255 caracteres. Para la *cuentaDestino* puede utilizarse la propia cuenta ordenante o un objeto *null*. En la Tabla 9 se han marcado con dos asteriscos estos nuevos valores.

Parámetro	Clases de equivalencia	Valores seleccionados
<i>this</i> (cuenta ordenante)	cuenta con saldo en $[0, 300]$ cuenta con saldo en $(300, +\infty)$	cuenta con saldo 200 cuenta con saldo 500 cuenta con saldo 300 cuenta con saldo 301
cuentaDestino	una cuenta que existe una cuenta que no existe	cuenta "123456" cuenta "000000" la propia cuenta ordenante **
importe	$(-\infty, 0]$ $(0, 300)$ $[300, +\infty)$	-100 +100 +400 0 1 300 301
concepto	"Pago de alquiler"	"Pago de alquiler" "" ** Cadena de más de 255 caracteres **

Tabla 9. Valores de prueba seleccionados, a los que se han añadido los procedentes de la conjetura de errores

Para valores de otros tipos, como los numéricos, suelen utilizarse el valor 0 (ya que, por ejemplo, puede que se utilice como denominador en alguna división), algún número negativo, etcétera.

#### 4 Aplicación de las técnicas al conjunto de datos de salida

Tanto la técnica de valores límite como la de clases de equivalencia pueden ser aplicadas al conjunto de salida, de forma que lo que se particiona no es el dominio de entrada, sino el de salida, con lo que los casos de prueba que se construyan (que serán las entradas del programa bajo prueba) deberán ser capaces de generar las salidas en cada una de las clases de equivalencia de salida.

En este punto podemos tener en cuenta que los eventos asíncronos (excepciones) también pertenecen al posible conjunto de salidas de una operación, de manera que debemos intentar que nuestros casos de prueba lancen, con cada operación, todas sus posibles excepciones. La operación *transferir* que veíamos en la Figura 8 puede lanzar cuatro excepciones: *SQLException*, *ImporteNegativoException*, *SaldoInsuficienteException* y *CuentaInexistenteException*. Para este ejemplo, y teniendo en cuenta que estas excepciones forman también parte del posible conjunto de salidas de la operación, deberíamos construir, al menos, cuatro casos de prueba para que se arrojen las cuatro excepciones:

- Para lanzar la *SQLException*, en un caso de prueba tendríamos que, por ejemplo, detener el servidor de base de datos.
- Para lanzar la *ImporteNegativoException*, tendríamos que tratar de hacer una transferencia con un importe negativo.
- Para lanzar la *SaldoInsuficienteException*, el importe de la transferencia más la comisión que corresponda debe superar el saldo disponible en la cuenta.
- Para la *CuentaInexistenteException*, deberíamos tratar de hacer una transferencia a una cuenta que no exista.

#### 5 Criterios de cobertura para valores de prueba

Una vez que el *tester* ha propuesto los valores de prueba conforme a alguna de las técnicas repasadas en las secciones anteriores, los debe combinar con alguna *estrategia de combinación* (se verán en el capítulo siguiente) con objeto de obtener

los casos de prueba. Los casos de prueba deben utilizar, “de alguna manera”, los valores interesantes que el *tester* ha propuesto: puede, por ejemplo, utilizar cada valor en un caso, cada valor dos veces, utilizar diferentes combinaciones de valores, etcétera. Para medir el grado en que se utilizan los valores de prueba, se utilizan criterios de cobertura para valores.

Con objeto de ilustrar los criterios que se exponen a continuación, utilizaremos, a modo de ejemplo, los parámetros  $A$ ,  $B$  y  $C$  cuyos valores son los siguientes

$$A=\{1, 2, 3, 4\}, B=\{5, 6, 7\}, C=\{8, 9\}$$

### 5.1 Cada uso (*each use*)

Cuando se construyen casos de prueba conforme a este criterio, cada valor interesante debe utilizarse, al menos, en un caso de prueba. Para el caso de los parámetros  $A$ ,  $B$  y  $C$ , dos *test suites* que verifican este criterio son los siguientes:

$$\text{test suite 1} = \{ (1, 5, 8), (2, 6, 9), (3, 7, 8), (4, 5, 9) \}$$

$$\text{test suite 2} = \{ (2, 5, 9), (3, 6, 8), (1, 7, 8), (4, 7, 8) \}$$

### 5.2 Todos los pares (*pairwise*)

Con este criterio, los casos de prueba deben visitar, al menos una vez, todos los pares de valores de dos parámetros cualesquiera. Este criterio se utiliza con mucha frecuencia, ya que muchos errores aparecen por la interacción de los valores inesperados de dos parámetros.

A primera vista, un algoritmo para alcanzar este criterio de cobertura de valores requiere la construcción de las tablas de pares de los parámetros; a continuación, se van construyendo casos de prueba de manera que, en cada uno, se visite el mayor número posible de pares no visitados. Si hay  $n$  parámetros, habrá  $n \cdot (n-1)/2$  tablas de pares. Para los parámetros  $A$ ,  $B$  y  $C$ , un *test suite* que satisface el criterio *pairwise* es el siguiente:

$$(2,6,9), (2,5,8), (2,7,9), (1,6,8), (1,5,9), (1,7,8), (4,6,9), (4,5,8), (4,7,9), (3,6,8), \\ (3,5,9), (3,7,8)$$



En la Tabla 10 aparecen las tablas de pares de los tres parámetros, así como el número de visitas que recibe cada par: el caso (2, 6, 9), por ejemplo, visita los pares (2, 6) de la tabla (A, B), (2, 9) de la tabla (A, C) y (6, 9) de la tabla (B, C).

(A, B)		(A, C)		(B, C)	
Pares	Visitas	Pares	Visitas	Pares	Visitas
(1, 5)	1	(1, 8)	2	(5, 8)	2
(1, 6)	1	(1, 9)	1	(5, 9)	2
(1, 7)	1	(2, 8)	1	(6, 8)	2
(2, 5)	1	(2, 9)	2	(6, 9)	2
(2, 6)	1	(3, 8)	2	(7, 8)	2
(2, 7)	1	(3, 9)	1	(7, 9)	2
(3, 5)	1	(4, 8)	1		
(3, 6)	1	(4, 9)	2		
(3, 7)	1				
(4, 5)	1				
(4, 6)	1				
(4, 7)	1				

Tabla 10. Tablas de pares y número de visitas para los parámetros A, B y C

### 5.3 Todas las tuplas de $n$ elementos ( $n$ -wise)

Este criterio es una generalización del anterior (*pairwise*), cuyo objetivo es construir casos de prueba que visiten todas las tuplas de  $n$  elementos de  $n$  parámetros cualesquiera. Procediendo de modo parecido al caso anterior, se construirán no tablas de pares, sino tablas de  $n$  elementos (cada elemento es una tupla), de manera que se visiten todas las tuplas.

Para el ejemplo de A, B y C, el mayor valor que puede tomar  $n$  es 3, ya que hay tres parámetros. Para un sistema con cuatro conjuntos A, B, C y D y  $n=3$ , existirán las tablas (A, B, C), (A, B, D) y (B, C, D). El criterio  $n$ -wise subsume al criterio  $(n-1)$ -wise: es decir, si se visitan todas las tuplas de  $n$  elementos, también se están visitando todas las tuplas de  $n-1$  elementos.

## Capítulo V. Estrategias de combinación para la obtención de casos de prueba

*Una vez que se han propuesto los valores más adecuados para probar el sistema, el tester debe combinarlos con objeto de obtener buenos casos de prueba que lo ayuden a encontrar errores en el sistema. En general, los casos de prueba ejecutan servicios del sistema bajo prueba. Tras cada ejecución, el caso debe ser capaz de determinar, mediante lo que se llama un oráculo, si el sistema se ha comportado de forma correcta (no ha encontrado error) o de forma incorrecta (sí lo ha encontrado).*

### 1 Estructura de un caso de prueba

Una de las características deseables de los casos de prueba es que sean reproducibles, en el sentido de que, independientemente del momento en que se ejecuten, se obtengan siempre los mismos resultados. Por ello, un buen caso de prueba establece, en su comienzo, la situación inicial que se requiere para su ejecución (por ejemplo, eliminar todos los registros de la base de datos de prueba); a continuación, lo habitual es ejecutar servicios sobre el sistema; finalmente, al caso de prueba se lo dota de un *oráculo*, que ayuda a determinar si el caso ha encontrado o no error en el sistema.

#### 1.1 Ejemplo

Supongamos que el sistema bancario que venimos utilizando manipula una base de datos relacional cuyo esquema entidad-interrelación es el que aparece en la Figura 10: la base de datos almacena información de *Clientes*, cada uno de los cuales puede tener varias *Cuentas*; cada *Cuenta* tiene una serie de *Movimientos* (ingresos y retiradas de fondos) y puede tener varias tarjetas asociadas, que pueden ser de *Débito* o de *Crédito*. Ambos tipos de tarjetas tienen también una serie de *Movimientos de tarjeta* asociados. Las tarjetas de *Débito* no aportan nada respecto de la definición de *Tarjeta*, mientras que las de *Crédito* tienen una columna adicional (*crédito*) que denota el crédito concedido a esa tarjeta. Hay, además, una tabla aislada (*ÚltimosNúmeros*) en la que se almacena un contador para asignar el número de cuenta a las cuentas nuevas que se van creando.

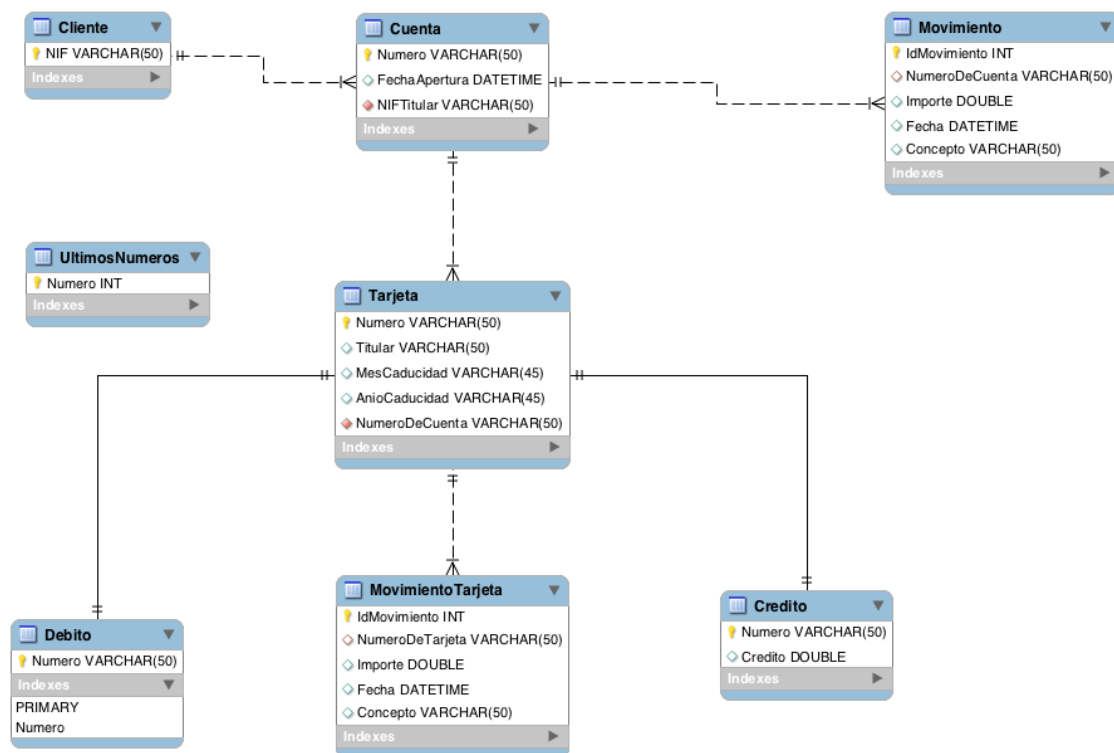


Figura 10. Esquema entidad-interrelación de la base de datos bancaria

La Figura 11 muestra el código de un caso de prueba en formato *JUnit* (un framework que permite automatizar la ejecución de casos de prueba para aplicaciones Java), que permite probar parte de la funcionalidad de la operación *retirar* de *Cuenta*: en su primera sentencia (*double saldoPepePre=this.cuentaPepe.getSaldo()*) se almacena en la variable *saldoPepePre* el saldo de la cuenta identificada por el objeto *cuentaPepe*. A continuación, se retiran 500 euros de ella. En la última instrucción del bloque *try* se comprueba que el nuevo saldo de la cuenta sea el anterior menos los 500 euros que se han retirado. Si el sistema bajo prueba se comporta adecuadamente con este caso de prueba (es decir, el saldo se decrementa efectivamente en 500 euros), el framework *JUnit* mostrará, para este caso, una barra verde; en otro caso mostrará una barra roja.

```

public void testRetirarDeCuenta() {
    try {
        double saldoPepePre=this.cuentaPepe.getSaldo();
        this.cuentaPepe.retirar(500, "Retirada de 500");
        assertTrue(this.cuentaPepe.getSaldo()==saldoPepePre-500);
    } catch (Exception e) {
        fail(e.toString());
    }
}

```

Figura 11. Un caso de prueba para probar la operación *retirar* de la clase *Cuenta*

El caso que se muestra en la figura anterior adolece, sin embargo, de un elemento importante: como hemos dicho, los casos deben ser reproducibles en cualquier momento y, también en cualquier momento, deben devolver siempre el mismo resultado. Para ello es preciso especificar la situación inicial que, como apuntábamos más arriba, puede consistir en situar la base de datos en un estado inicial conocido. JUnit y otros frameworks de pruebas incluyen la posibilidad de ejecutar operaciones de inicialización antes de cada caso de prueba, así como operaciones de finalización después de cada uno (liberación de conexiones o recursos, por ejemplo). En JUnit, la situación inicial se especifica en un método llamado *setUp*; las operaciones finales se incluyen en un método llamado *tearDown*.

En la Figura 12 se muestra el código correspondiente al método *setUp* para el sistema bancario: en primer lugar, se eliminan todos los registros de las tablas de la base de datos (excepto la tabla *ÚltimosNúmeros*). A continuación, se crean dos clientes (objetos *pepe* y *ana*) y se insertan en la base de datos (en la tabla *Cliente*). Luego se crea una cuenta para cada uno ellos, se les ingresan 1000 y 2000 euros respectivamente a *pepe* y a *ana* y se les crean unas tarjetas de débito y crédito.

```

protected void setUp() {
    System.out.println("setUp");
    try {
        Connection bd=Broker.getConnection();
        String SQL1="Delete from Cliente where nif='ana' or nif='pepe'";
        String SQL2="Delete from Cuenta where nifTitular='ana' or nifTitular='pepe'";
        String SQL3="Delete from Tarjeta";
        String SQL4="Delete from Credito";
        String SQL5="Delete from Debito";
        String SQL6="Delete from Movimiento";
        String SQL7="Delete from MovimientoTarjeta";
        PreparedStatement p1=bd.prepareStatement(SQL1);
        PreparedStatement p2=bd.prepareStatement(SQL2);
        PreparedStatement p3=bd.prepareStatement(SQL3);
        PreparedStatement p4=bd.prepareStatement(SQL4);
        PreparedStatement p5=bd.prepareStatement(SQL5);
        PreparedStatement p6=bd.prepareStatement(SQL6);
        PreparedStatement p7=bd.prepareStatement(SQL7);
        p1.executeUpdate(); p2.executeUpdate(); p3.executeUpdate();
        p4.executeUpdate(); p5.executeUpdate(); p6.executeUpdate();
        p7.executeUpdate();
    } catch (Exception e) {
        fail(e.toString());
    }
    pepe=new Cliente();
    pepe.setNIF("pepe");
    ana=new Cliente();
    ana.setNIF("ana");
    try {
        pepe.insert();
        ana.insert();
    } catch (Exception e) {
        fail(e.toString());
    }

    try {
        cuentaPepe=new Cuenta(pepe);
        cuentaAna=new Cuenta(ana);
        cuentaPepe.insert();
        cuentaAna.insert();
    } catch (Exception e) {
        fail(e.toString());
    }

    try {
        cuentaPepe.ingresar(1000, "Ingreso a Pepe");
        cuentaAna.ingresar(2000, "Ingreso a Ana");
    } catch (Exception e) {
        fail(e.toString());
    }

    tdPepe=new TarjetaDeDebito("tdPepe", "10", "2010", cuentaPepe);
    tdAna=new TarjetaDeDebito("tdAna", "10", "2010", cuentaAna);
    tcPepe=new TarjetaDeCredito("tcPepe", "10", "2010", cuentaPepe, 500);
    tcAna=new TarjetaDeCredito("tcAna", "10", "2010", cuentaAna, 800);
    try {
        tdPepe.insert();
        tdAna.insert();
        tcPepe.insert();
        tcAna.insert();
    } catch (Exception e) {
        fail(e.toString());
    }
}

```

Figura 12. Código del método *setUp* para los casos de prueba del banco

La estructura de la clase *CuentaTest*, que prueba algunas de las funcionalidades ofrecidas por la clase *Cuenta* aparece en la Figura 13. Los métodos cuyo nombre empieza con el prefijo *test* son casos de prueba; antes de ejecutar cada uno, JUnit ejecuta el método *setUp* (Figura 12), que vacía la base de datos. Tras ejecutar cada caso, ejecuta el método *tearDown*. Obsérvese la presencia, en la clase, de los campos *pepe*, *ana*, *cuentaPepe*, *cuentaAna*, *tdPepe*, *tdAna*, *tcPepe* y *tcAna* que representan clientes, cuentas, tarjetas de débito y tarjetas de crédito. En este tipo de frameworks, a estos objetos que se declaran como miembros de la clase de prueba y que se reutilizan en varios casos de prueba se los conoce con el nombre de *fixtures*. En el caso de la Figura 13, el orden de ejecución de los métodos será el siguiente: *setUp*, *testRetirarDeCuenta*, *tearDown*, *setUp*, *testRetirarConDebito*, *tearDown*, *setUp*, *testRetirarConCredito*, *tearDown*, *setUp*, *testTransferir*, *tearDown*, *setUp*, *testForzarExcepcionEnInsercionDeCuenta*, *tearDown*.

```
package bancoupm.dominio.tests;

import java.sql.Connection;

public class CuentaTest extends TestCase {
    Cliente pepe, ana;
    Cuenta cuentaPepe, cuentaAna;
    TarjetaDeDebito tdPepe, tdAna;
    TarjetaDeCredito tcPepe, tcAna;

    protected void setUp() {}

    protected void tearDown() {}

    public void testRetirarDeCuenta() {}

    public void testRetirarConDebito() {}

    public void testRetirarConCredito() {}

    public void testTransferir() {}

    public void testForzarExcepcionEnInsercionDeCuenta() {}
}
```

Figura 13. Campos y operaciones en la clase *CuentaTest*, que prueba algunas funcionalidades de la clase *Cuenta* incluida en el sistema bancario

## 2 El oráculo

Como se ha comentado, el oráculo es el mecanismo de que se dota a los casos de prueba para determinar si el propio caso ha encontrado o no un error en el sistema bajo prueba. El oráculo debe ser lo suficientemente “fino” como para comprobar adecuadamente el comportamiento obtenido por el sistema. En el ejemplo de la Figura 11, el oráculo viene dado por la instrucción

`assertTrue(this.cuentaPepe.getSaldo()==saldoPepePre-500)`, que comprueba, para una retirada de 500 euros de la cuenta de *Pepe*, que el saldo, en efecto disminuye en esa cantidad. Un oráculo menos riguroso podría comprobar, sencillamente, que el saldo de la cuenta después de la retirada es menor que el saldo antes:

`assertTrue(this.cuentaPepe.getSaldo()<saldoPepePre)`. Como se observa, este oráculo tiene una granularidad mucho más “gruesa” que el anterior, y tiene una probabilidad más alta de pasar por alto errores en el sistema.

La escritura de buenos oráculos es una de las tareas más tediosas en la escritura de casos de prueba. De hecho, gran parte de los casos de prueba que, en apariencia, encuentran errores, incluyen un error en la definición del oráculo.

Por lo general, el oráculo compara el resultado obtenido tras la ejecución del caso de prueba con el resultado que se esperaba. Estos resultados pueden expresarse en función del estado de los objetos que intervienen en el escenario de ejecución que se está probando. Además, puesto que el estado de un objeto es una función de los valores de los campos de dicho objeto, los resultados (esperado u obtenido) pueden expresarse como una función de los valores de los campos de los objetos involucrados. En el ejemplo del oráculo escrito para la operación *retirar*, se toma el estado del objeto *cuentaPepe* que, en este caso, es el único objeto que interviene en el escenario de ejecución. El estado de *cuentaPepe* viene dado por una llamada a su método *getSaldo* el cual, como se ve en la Figura 14, devuelve la suma de los importes de los movimientos asociados a la cuenta: en la determinación del valor del estado de la instancia, entonces, intervienen los estados de otros objetos, que son los movimientos, pero que forman parte de la definición de la *Cuenta*.

```

public double getSaldo() throws SQLException {
    String SQL="Select sum(Importe) from Movimiento where NumeroDeCuenta=?";
    Connection bd=null;
    SQLException ex=null;
    PreparedStatement p=null;
    double resul=0;
    try {
        bd=Broker.getConnection();
        p=bd.prepareStatement(SQL);
        p.setString(1, this.numero);
        ResultSet r=p.executeQuery();
        if (r.next()) {
            resul=r.getDouble(1);
        }
    }
    catch (SQLException e) {
        ex=e;
    }
    finally {
        p.close();
        if (ex!=null)
            throw ex;
    }
    return resul;
}

```

Figura 14. Código del método *getSaldo* de la clase *Cuenta*, que calcula el saldo en función de los importes de los movimientos asociados

## 2.1 Obtención de casos de prueba con oráculos a partir de máquinas de estado

Cuando el comportamiento de las instancias de clases se describen mediante máquinas de estados, la propia máquina puede entenderse como un artefacto más o menos similar a un autómata finito, de los que pueden derivarse expresiones regulares que describen un lenguaje regular. En el caso de la máquina de estados, el lenguaje que acepta puede derivarse a partir de las operaciones que etiquetan las transiciones entre estados. De este modo, recorriendo la máquina de diferentes formas (siempre teniendo presente algún criterio de cobertura para máquinas de estado, como los que vimos en la Sección Capítulo III.5 del Capítulo III, página 24), obtenemos diversos escenarios de ejecución de la instancia.

Además, puesto que cada transición conecta dos estados, la definición del estado origen puede utilizarse como precondition del caso de prueba, y la definición del estado destino, como postcondición.

Fijándonos en el ejemplo de la Figura 15, que representa el posible comportamiento de una supuesta cuenta corriente, observamos que hay tres estados (además del estado inicial), y que se va de uno a otro en función de las transiciones que se vayan disparando, algunas de las cuales cuentan además con condiciones “de guarda”.



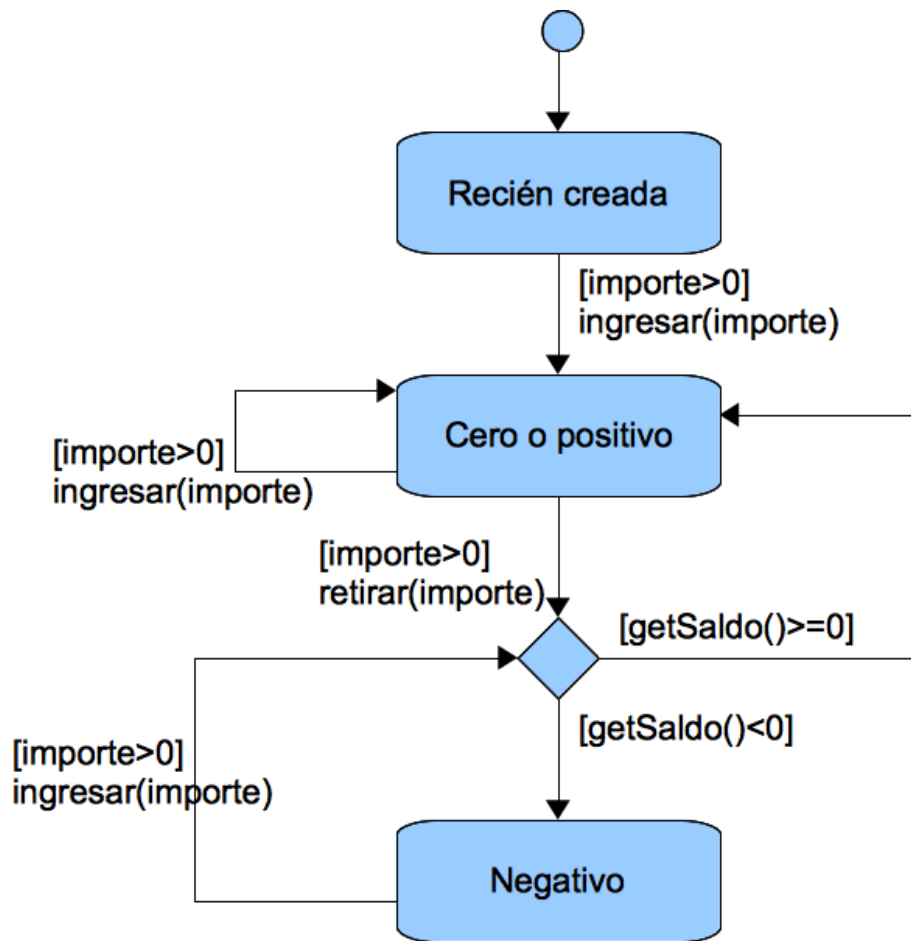


Figura 15. Una posible máquina de estados para una clase *Cuenta*

Asumiendo que cada *cuenta* tiene una colección de *movimientos* asociados y una operación *getSaldo*, los tres estados mostrados en la figura se pueden describir como en la Tabla 11: la cuenta está en el estado *Recién creada* cuando no ha habido movimientos sobre ella; en *Cero o positivo* cuando el saldo es mayor o igual a cero, y en *Negativo* cuando el saldo es menor que cero.

Estado	Descripción
Recién creada	<code>movimientos.size()==0</code>
Cero o positivo	<code>getSaldo()&gt;=0</code>
Negativo	<code>getSaldo()&lt;0</code>

Tabla 11. Descripción de los estados de una clase *Cuenta*

Suponiendo que la transición desde el estado inicial hasta el final se corresponda con una llamada al constructor de la clase, un caso de prueba con el que se logra cobertura de estados (página 24) es el que se muestra en la Figura 16: obsérvese que usamos las transiciones para construir la secuencia de operaciones del caso de

prueba, mientras que la descripción de los estados la utilizamos para escribir los oráculos, que se resaltan en letra **negrilla**.

```
Cuenta c=new Cuenta();  
assertTrue(c.movimientos.size()==0);  
c.ingresar(10);  
assertTrue(c.getSaldo())>=0;  
c.retirar(-10);  
assertTrue(c.getSaldo())<0;
```

Figura 16. Un caso de prueba para la cuenta, procedente de la Figura 15 y de la Tabla 11

Por otro lado, las máquinas de estado se utilizan también para describir casos de uso: los estados representan pasos en la ejecución del casos de uso, y las transiciones representan que el paso o actividad anterior se ha terminado de ejecutar y se pasa a ejecutar la siguiente actividad que corresponda. Puesto que los casos de uso se corresponden de forma muy aproximada con los requisitos funcionales del sistema, pueden utilizarse las máquinas de estado que describen casos de uso para preparar, desde la captura de requisitos, casos de prueba funcionales, que quizá se ejecuten en un momento mucho más tardío (cuando, por ejemplo, el sistema esté completamente terminado).

En la Figura 17 se muestra la máquina de estados correspondiente a un supuesto caso de uso *Identificar*, que recibe un *login* y una contraseña del usuario y verifica si la combinación de ambos es correcta. En caso negativo puede realizar dos intentos más antes de que se le bloquee la cuenta; en caso afirmativo, se le cierra su sesión anterior si es la tenía abierta y en todo se le crea una nueva, que se añade a una lista de sesiones. Para este requisito, podríamos crear casos de prueba que logren cobertura de pares de transiciones entrada/salida (sección 5.3, página 25). Para ello, construimos para cada estado una lista con las transiciones que entran y salen (Tabla 12). Nótese que las transiciones que salen y entran a nodos *choice* (nodos de elección) no se consideran salidas, y que pasamos directamente a considerar las transiciones que salen de los *choice*.

Estado	Entradas	Salidas	Pares
Comprobar <i>login</i> y contraseña	1, 12	3,4	(1, 3), (1, 4), (12, 3), (12, 4)
Comprobar si ya está conectado	4	(6, 7)	(4, 6), (4, 7)
Cerrar sesión	7	8	(7,8)
Crear sesión	6, 8	9	(6, 9), (8, 9)
Añadir sesión a una lista de sesiones	9	10	(9, 10)
Incrementar contador	3	12, 13	(3, 12), (3, 13)
Bloquear usuario	13	14	(13, 14)

Tabla 12. Transiciones de entrada y salida de los estados de la Figura 17

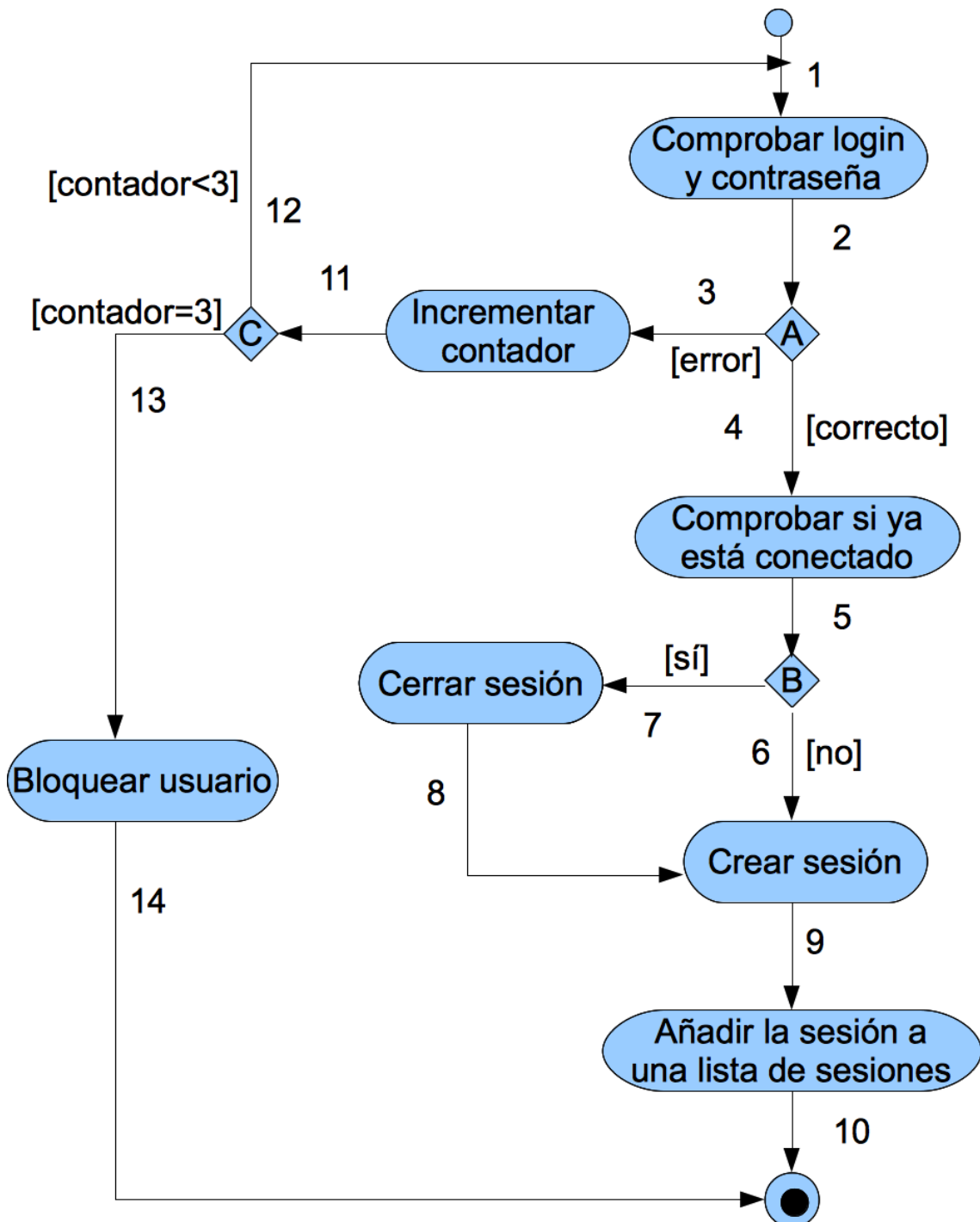


Figura 17. Una máquina de estados que representa el caso de uso *Identificar*

A continuación, construimos los casos de prueba para ir visitando todos los pares de transiciones. En la Tabla 13 se muestran la secuencia de transiciones que componen los casos de prueba y se tachan los pares que van siendo visitados por cada caso.

Caso de prueba	Pares visitados
1-3-12-3-13-14	<del>(1,3)</del> , (1, 4), <del>(12, 3)</del> , (12, 4), (4, 6), (4, 7), (7,8), (6, 9), (8, 9), (9, 10), <del>(3,12)</del> , <del>(3,13)</del> , <del>(13,14)</del>
1-4-6-9-10	<del>(1,3)</del> , <del>(1,4)</del> , <del>(12, 3)</del> , (12, 4), <del>(4,6)</del> , (4, 7), (7,8), <del>(6,9)</del> , (8, 9), <del>(9,10)</del> , <del>(3,12)</del> , <del>(3,13)</del> , <del>(13,14)</del>
1-3-12-4-7-8-9-10	<del>(1,3)</del> , <del>(1,4)</del> , <del>(12, 3)</del> , <del>(12,4)</del> , <del>(4,6)</del> , <del>(4,7)</del> , <del>(7,8)</del> , <del>(6,9)</del> , <del>(8,9)</del> , <del>(9,10)</del> , <del>(3,12)</del> , <del>(3,13)</del> , <del>(13,14)</del>

Tabla 13. Casos de prueba y pares visitados

Los casos de prueba contruidos cubren entonces el criterio de cobertura que nos habíamos prefijado. Por el propio diseño del sistema, el primer caso es infactible, ya que sólo se recorre la transición 13 cuando el *contador* valga 3. Habrá entonces que modificar ese primer caso y dejarlo, por ejemplo, de esta manera: 1-3-12-3-13-12-3-12-3-13-14.

### 3 Estrategias de combinación

Una estrategia de combinación es un algoritmo que toma como entradas un conjunto de datos de prueba y produce, mediante algún medio de combinación de dichos valores, un conjunto de casos de prueba. Por lo general, las estrategias de combinación combinan los valores de prueba con objeto de alcanzar algún criterio de cobertura para valores (*each use, pairwise, n-wise*, página 31).

Puesto que los casos de prueba deben ser mantenidos y ejecutados, lo cual tiene un coste asociado, por lo general es deseable obtener *test suites* que no sean demasiado grandes (es decir, que no tengan demasiados casos de prueba) y que, a la vez, sean capaces de encontrar tantos errores como sea posible.

A continuación se revisan algunas estrategias de combinación. A modo de ejemplo, supondremos que disponemos de los parámetros  $A$ ,  $B$  y  $C$ , cuyos valores son los siguientes:

$$A=\{1, 2, 3, 4\}, B=\{5, 6, 7\}, C=\{8, 9\}$$

### 3.1 Todas las combinaciones (*All combinations*)

Mediante este algoritmo, se generan todos los posibles casos de prueba, mediante el cálculo del producto cartesiano de los parámetros. Para el ejemplo, habría  $4 \times 3 \times 2 = 24$  combinaciones, que corresponden a los 24 casos de prueba que se muestran en la Tabla 14

i	A[i]	B[i]	C[i]
0	1	5	8
1	1	5	9
2	1	6	8
3	1	6	9
4	1	7	8
5	1	7	9
6	2	5	8
7	2	5	9
8	2	6	8
9	2	6	9
10	2	7	8
11	2	7	9
12	3	5	8
13	3	5	9
14	3	6	8
15	3	6	9
16	3	7	8
17	3	7	9
18	4	5	8
19	4	5	9
20	4	6	8
21	4	6	9
22	4	7	8
23	4	7	9

Tabla 14. Casos de prueba procedentes de aplicar *All combinations* a los parámetros de ejemplo

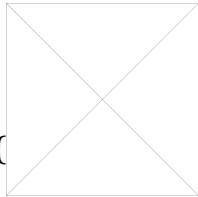
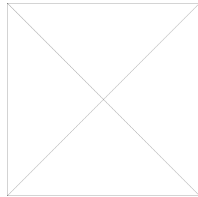
Resulta relativamente sencillo escribir un algoritmo recursivo para calcular el producto cartesiano de  $n$  conjuntos; sin embargo, es posible también construir un algoritmo iterativo si añadimos a la Tabla 14 una columna adicional que nos muestra las posiciones que los elementos de cada conjunto ocupan en el caso de prueba (véase Tabla 15): por ejemplo, el caso 0 está formado por los elementos 1, 5 y 8, que corresponden a los elementos situados en las posiciones 0, 0 y 0 de los conjuntos  $A$ ,  $B$  y  $C$ ; el caso 1 corresponde a los elementos situados en 0, 0, 1; el caso 7, a los elementos situados en las posiciones 1, 0 y 1 de, respectivamente,  $A$ ,  $B$  y  $C$ . Observemos, además, que los elementos del tercer conjunto ( $C$ ) aparecen una vez sí, una vez no (es decir: 8, 9, 8, 9...); que los del segundo ( $B$ ) aparecen tantas veces seguidas como número de elementos haya en el tercero (como hay dos elementos

en  $C$ : 5, 5, 6, 6, 7, 7, 5, 5, 6, 6, 7, 7...) y que los del primero ( $A$ ) aparecen tantas veces seguidas como número de elementos haya en el segundo multiplicado por el número de elementos en el tercero ( $3 \times 2 = 6$  veces: hay 6 unos seguidos, seis doses seguido, seis treses y seis cuatros).

i	A[i]	B[i]	C[i]	Posiciones
0	1	5	8	{0, 0, 0}
1	1	5	9	{0, 0, 1}
2	1	6	8	{0, 1, 0}
3	1	6	9	{0, 1, 1}
4	1	7	8	{0, 2, 0}
5	1	7	9	{0, 2, 1}
6	2	5	8	{1, 0, 0}
7	2	5	9	{1, 0, 1}
8	2	6	8	{1, 1, 0}
9	2	6	9	{1, 1, 1}
10	2	7	8	{1, 2, 0}
11	2	7	9	{1, 2, 1}
12	3	5	8	{2, 0, 0}
13	3	5	9	{2, 0, 1}
14	3	6	8	{2, 1, 0}
15	3	6	9	{2, 1, 1}
16	3	7	8	{2, 2, 0}
17	3	7	9	{2, 2, 1}
18	4	5	8	{3, 0, 0}
19	4	5	9	{3, 0, 1}
20	4	6	8	{3, 1, 0}
21	4	6	9	{3, 1, 1}
22	4	7	8	{3, 2, 0}
23	4	7	9	{3, 2, 1}

Tabla 15. La Tabla 14, ampliada con una columna que representa la posición de los elementos (valores) en cada conjunto (parámetros)

De manera general, si hay  $n$  conjuntos  $\{S_1, S_2, \dots, S_n\}$ , cada elemento de  $S_1$  aparece  $|S_2| \cdot |S_3| \cdot \dots \cdot |S_n|$  veces seguidas (donde  $|S_i|$  es el cardinal del conjunto  $S_i$ ); los elementos de  $S_2$ ,  $|S_3| \cdot |S_4| \cdot \dots \cdot |S_n|$ , veces seguidas; etcétera. Los elementos de  $S_n$  aparecen una vez cada uno, alternando. Ya que el número de combinaciones es

conocido a priori (  ), dado un índice arbitrario  $i$ ,  , es posible conocer la combinación o caso de prueba correspondiente a ese  $i$ .

El primer bucle de la Figura 18 evalúa el número de combinaciones (es decir, el cardinal del producto cartesiano); luego, en el segundo bucle, durante

$númeroDeCombinaciones$  veces, se añade al conjunto de resultados la combinación  $i$ -ésima.

```
function productoCartesiano({S}) : Set(Combinación) {
    númeroDeCombinaciones: int = 1;
    for (i=0; i<|S|; i++) {
        númeroDeCombinaciones = númeroDeCombinaciones
        * |Si|
    }

    resultado : Set(Combinación)=∅
    for (i=0; i<númeroDeCombinaciones; i++) {
        resultado= resultado ∪ getCombinación(S, i)
    }
    return resultado
}
```

Figura 18. Algoritmo iterativo para obtener el producto cartesiano de  $n$  conjuntos (versión 1)

Para completar el algoritmo de la Figura 18, se debe implementar la función *getCombination*, que devuelve la combinación correspondiente al caso de prueba  $i$ -ésimo, recibiendo también como parámetro la lista de conjuntos  $\{S\}$ . Cada combinación está compuesta de  $|S|$  elementos (el primero del primer conjunto, el segundo del segundo conjunto...); para el último conjunto ( $S_n$ ), el elemento que se debe incluir en la combinación  $i$ -ésima (y que aparecerá en la última posición de la combinación) es el elemento de  $S_n$  dado por la expresión  $i\%|S_n|$  (en donde  $\%$  representa el resto de la división entera). Dada una posición arbitraria  $i$  de un caso de prueba (en el ejemplo, dado un número arbitrario entre 0 y 23), la posición de cada conjunto  $S_j$  que se debe incluir en la combinación depende del número de valores en  $S_{j+1}, S_{j+2}, \dots, S_n$ . El valor seleccionado del conjunto  $j$ -ésimo para la combinación  $i$ -ésima es el elemento de  $S_j$  situado en la posición dada por la expresión que aparece en la Figura 19.

**Figura 19. Cálculo del valor del conjunto  $j$ -ésimo para la combinación  $i$ -ésima**

$divisores_j$  es el producto de los cardinales de los conjuntos que van desde  $S_j$  hasta  $S_n$ , y es 1 para  $S_n$ . Para los conjuntos  $A$ ,  $B$  y  $C$  de nuestro ejemplo (página 44), los  $divisores$  son:

$$divisores_A = |B| \cdot |C| = 3 \cdot 2 = 6 \quad divisores_B = |C| = 2 \quad divisores_C = 1$$

Por tanto, la función mostrada en la Figura 18 se puede reescribir y dejarla como en la Figura 20: ahora, dentro del primer bucle hay otro bucle anidado que va calculando los  $divisores$  de cada conjunto. En el segundo bucle externo, *getCombinación* toma ahora tres parámetros:

- La lista de conjuntos ( $S$ )
- La posición de la combinación que se desea obtener ( $i$ )
- Los valores de  $divisores$ .



```

function productoCartesiano({S}) : Set(Combinación) {
    númeroDeCombinaciones : int = 1;
    divisores : int[] = new int[|S|]

    for (i=0; i<|S|; i++) {
        númeroDeCombinaciones = númeroDeCombinaciones * |Si|
        divisores[i]=1
        for (j=0; j<|S|; j++) {
            divisores[i]=divisores[i]*|Sj|
        }
    }

    resultado : Set(Combinación)=∅
    for (i=0; i<númeroDeCombinaciones; i++) {
        resultado= resultado ∪ getCombinación(S, i, divisores)
    }
    return resultado
}

```

Figura 20. Algoritmo iterativo para obtener el producto cartesiano de  $n$  conjuntos (versión 1)

Finalmente, la función *getCombinación* queda como en la Figura 21:

```

function getCombinación(S, posición, divisores) : Combinación {
    resultado : Combinación = new int[|S|]
    for (int i=0; i<|S|; i++) {
        resultado[i]=(posición/divisores[i])%|Si|
    }
    return resultado
}

```

Figura 21. Algoritmo para obtener una combinación arbitraria

Para un conjunto dado de valores de prueba, *All combinations* consigue siempre la máxima cobertura posible en el sistema bajo prueba, además de conseguir cobertura total *n-wise* (donde  $n$  es el número de parámetros) y, desde luego, *each use* y *pairwise*. Sin embargo, es también la que produce más casos de prueba y, probablemente, muchos de ellos sean redundantes.

### 3.2 Cada elección (*Each choice*)

El objetivo de esta estrategia de combinación es la creación de casos de prueba, de manera que cada valor interesante se utilice, al menos, en un caso de prueba, logrando por tanto cobertura *each use*. El algoritmo, entonces, es sencillo: mientras haya valores no visitados, se construye una combinación vacía. Si un conjunto tiene algún valor no utilizado, se coloca en la posición que corresponda de la combinación; en otro caso, se toma otro valor (puede ser aleatoriamente, el que menos veces se haya utilizado, el que el *tester* considere más oportuno, etcétera).

Una aplicación del algoritmo sobre el ejemplo de los conjuntos  $A$ ,  $B$  y  $C$  va produciendo los resultados de la Tabla 16: se muestra, para cada iteración, los casos de prueba que se van seleccionando y, tachados, los valores que visita cada caso en cada iteración. Nótese que el algoritmo para cuando se han visitado todos los valores al menos una vez.

Iteración	Casos de prueba	Valores visitados
0	<i>Ninguno</i>	$A=\{1, 2, 3, 4\}$ $B=\{5, 6, 7\}$ $C=\{8, 9\}$
1	(1, 5, 8)	$A=\{\underline{1}, 2, 3, 4\}$ $B=\{\underline{5}, 6, 7\}$ $C=\{\underline{8}, 9\}$
2	(1, 5, 8) (2, 6, 9)	$A=\{\underline{1}, \underline{2}, 3, 4\}$ $B=\{\underline{5}, \underline{6}, 7\}$ $C=\{\underline{8}, \underline{9}\}$
3	(1, 5, 8) (2, 6, 9) (3, 7, 8)	$A=\{\underline{1}, \underline{2}, \underline{3}, 4\}$ $B=\{\underline{5}, \underline{6}, \underline{7}\}$ $C=\{\underline{8}, \underline{9}\}$
4	(1, 5, 8) (2, 6, 9) (3, 7, 8) (4, 6, 9)	$A=\{\underline{1}, \underline{2}, \underline{3}, \underline{4}\}$ $B=\{\underline{5}, \underline{6}, \underline{7}\}$ $C=\{\underline{8}, \underline{9}\}$

Tabla 16. Aplicación de *Each choice* al ejemplo de los conjuntos  $A$ ,  $B$  y  $C$

### 3.3 AETG (*Automatic Efficient Test Generator*)

Este algoritmo, descrito por Cohen et al.<sup>17</sup> y reproducido en la Figura 22, genera, en tiempo polinomial, un conjunto de casos de prueba de tamaño aceptable que consigue cobertura de valores *pairwise*.

<sup>17</sup> Cohen DM, Dalal SR, Kajla A y Patton GC. (1994). *The automatic efficient test generator (AETG) system*. Proceedings of Fifth International Symposium on Software Reliability Engineering (ISSRE'94). IEEE Computer Society Press: Los Alamitos, California, pp. 303–309.

Sea un sistema con  $k$  parámetros. Supóngase que el parámetro  $i$ -ésimo tiene  $l_i$  valores.

Supongamos que ya se han seleccionado  $r$  casos de prueba. El caso  $(r+1)$ -ésimo se construye a partir de  $M$  casos de prueba candidatos y eligiendo aquel que cubra más pares nuevos. Cada caso candidato se selecciona por medio del siguiente algoritmo voraz:

1. Elegir el parámetro  $f$  y un valor  $l$  de  $f$  tal que el valor del parámetro aparezca en el mayor número de pares no cubiertos.
2. Sea  $f_l = f$ . Elegir un orden aleatorio para los restantes parámetros, de manera que los  $k$  parámetros quedan ordenados:  $f_1, \dots, f_k$ .
3. Supongamos que ya están elegidos los valores para los parámetros  $f_1, \dots, f_j$ . Sea  $v_i$  el valor seleccionado para  $f_i$  ( $1 \leq i \leq j$ ). Para elegir el valor  $v_{j+1}$  del parámetro  $f_{j+1}$ , se cuenta el número de pares  $\{f_{j+1} = v \text{ y } f_i = v_i, \text{ para } 1 \leq i \leq j\}$ . El valor  $v_{j+1}$  que se selecciona es aquel que aparece en mayor número de pares. Obsérvese que, en este paso, sólo se considera una vez la inclusión de cada valor del parámetro, y que, al elegir el valor para el parámetro  $f_{j+1}$ , los valores se comparan con los  $j$  valores ya elegidos para los parámetros  $f_1, \dots, f_j$ .

**Figura 22. Pseudocódigo original del algoritmo AETG**

Como se observa, el segundo paso del algoritmo introduce un factor de aleatoriedad al ordenar al azar los parámetros cuyos valores están pendientes de ser elegidos. En la Figura 23 se ofrece una versión determinista de AETG que elimina el factor azaroso y que aplicaremos al ejemplo de los parámetros  $A, B$  y  $C$ .

1. Construir las tablas de pares (*tablasDePares*) para  $S$ , el conjunto de parámetros
2. Sea  $c$  la primera combinación que se elige, y sea  $c$  la primera combinación (la 0)
3. Añadir  $c$  al *resultado*
4. Actualizar *tablasDePares* con los pares visitados por  $c$
5. Mientras haya pares no visitados en *tablasDePares*
  1. Inicializar  $c$ , colocando el valor que visita más pares no visitados en *pairTables*
  2. Completar  $c$  con valores compatibles de los restantes parámetros, de manera que se visiten el mayor número de pares
  3. Añadir  $c$  al *resultado*
  4. Actualizar *tablasDePares* con los pares visitados por  $c$

**Figura 23. Versión determinista de AETG**

Al aplicar la versión determinista de AETG a  $A, B$  y  $C$ , elegimos en primer lugar la combinación 0:  $\text{resultado} = \{(1, 5, 8)\}$  y marcamos los pares visitados:  $(1, 5)$ ,  $(1, 8)$  y  $(5, 8)$ . En la Tabla 18 se muestran las tablas de pares y la iteración en la que se visita cada par (el 0 corresponde a los pasos 1 a 4 del algoritmo, que se ejecutan antes del bucle). Como quedan pares no visitados, se entra en el bucle y se selecciona el valor que visita más pares. Con objeto de elegir correctamente el valor que visita más pares (paso 5.1 del algoritmo de la Figura 23), en la Tabla 17 se muestra el número de pares visitados por cada valor en cada iteración del bucle mientras:

1.  $\text{resultado} = \{(1, 5, 8)\}$

2. La primera vez que se entra al bucle, el elemento que visita más pares nuevos (línea 5.1 del algoritmo) es el 9 del conjunto  $C$ . Por tanto, se construye una combinación  $c=(-, -, 9)$ . A continuación (línea 5.2), se completa  $c$  con valores compatibles de los restantes parámetros. Puesto que estamos aplicando la versión determinista del algoritmo, continuamos completando la combinación por el primer parámetro ( $A$ ): puesto que en la combinación  $c$  hay un elemento del conjunto  $C$ , nos fijamos en la tabla  $(A, C)$  y seleccionamos el valor que más pares visite de los elementos 1, 2, 3, y 4, ya que los pares  $(1, 9)$ ,  $(2, 9)$ ,  $(3, 9)$  y  $(4, 9)$  no han sido visitados todavía. Se selecciona el primer valor (1) y la combinación queda como  $(1, -, 9)$ . Para completar con el valor de  $B$ , debemos elegir un valor compatible con el 1 de  $A$  y el 9 de  $C$ , pudiendo añadir el 6, con lo que  $c=(1, 6, 9)$  y  $resultado=\{(1, 5, 8), (1, 6, 9)\}$ .
3. En la siguiente iteración, el valor que visita más pares no visitados es el 7 del conjunto  $B$ , con lo que  $c=(-, 7, -)$ . Para elegir el elemento de  $A$  que se colocará en  $c$ , consideramos que, en  $(A, B)$ , el 7 visita los pares no visitados  $(1, 7)$ ,  $(2, 7)$ ,  $(3, 7)$  y  $(4, 7)$ ; al haber empate, el algoritmo elige el 1, con lo que  $c=\{1, 7, -\}$ . De  $C$ , se visitan los pares no visitados  $(7, 8)$  y  $(7, 9)$ , por lo que se elige el 7,  $resultado=\{(1, 5, 8), (1, 6, 9), (1, 7, 8)\}$  y se actualizan las tablas de pares y el número de pares visitados por cada valor de cada parámetro.
4. Se continúa de la misma manera, seleccionando los valores de la manera descrita, actualizando las dos tablas:
  - a.  $c=(2, -, -), (2, 5, -), (2, 5, 9)$
  - b.  $c=(3, -, -), (3, 5, -), (3, 5, 8)$
  - c.  $c=(4, -, -), (4, 5, -), (4, 5, 8)$
  - d.  $c=(-, 6, -), (2, 6, -), (2, 6, 8)$
  - e.  $c=(-, 7, -), (2, 7, -), (2, 7, 9)$
  - f.  $c=(3, -, -), (3, 6, -), (3, 6, 9)$
  - g.  $c=(4, -, -), (4, 6, -), (4, 6, 9)$
  - h.  $c=(-, 7, -), (3, 7, -), (3, 7, 8)$
  - i.  $c=(4, -, -), (4, 7, -), (4, 7, 8)$

5. Finalmente,  $resultado = \{(1, 5, 8), (1, 6, 9), (1, 7, 8), (2, 5, 9), (3, 5, 8), (4, 5, 8), (2, 6, 8), (2, 7, 9), (3, 6, 9), (4, 6, 9), (3, 7, 8), (4, 7, 8)\}$

Iteraciones	Conjuntos (parámetros) y sus elementos (valores)								
	A				B			C	
	1	2	3	4	5	6	7	8	9
0	3	5	5	5	4	6	6	5	7
1	1	5	5	5	4	4	6	5	5
2	0	5	5	5	4	4	4	3	5
3	0	3	5	5	2	4	4	3	3
4	0	3	3	5	1	4	4	3	3
5	0	3	3	3	0	4	4	2	3
6	0	1	3	3	0	2	4	0	3
7	0	0	3	3	0	2	2	0	2
8	0	0	1	3	0	1	2	0	1
9	0	0	1	1	0	0	2	0	0
10	0	0	0	1	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0

Tabla 17. Pares visitados por cada valor en las distintas iteraciones de la versión determinista de AETG

(A, B)		(A, C)		(B, C)	
Pares	Visitas	Pares	Visitas	Pares	Visitas
(1, 5)	0	(1, 8)	0, 2	(5, 8)	0, 4, 5
(1, 6)	1	(1, 9)	1	(5, 9)	3
(1, 7)	2	(2, 8)	6	(6, 8)	6
(2, 5)	3	(2, 9)	3, 7	(6, 9)	1, 8, 9
(2, 6)	6	(3, 8)	4, 10	(7, 8)	2, 10, 11
(2, 7)	7	(3, 9)	8	(7, 9)	7
(3, 5)	4	(4, 8)	5, 11		
(3, 6)	8	(4, 9)	9		
(3, 7)	10				
(4, 5)	5				
(4, 6)	9				
(4, 7)	11				

Tabla 18. Tablas de pares e iteraciones en que son visitados los pares de los parámetros A, B y C

### 3.4 PROW (Pairwise with Restrictions, Order and Weight)

La mayoría de los algoritmos de testing combinatorio no tienen en cuenta las restricciones entre pares. Resulta muy común que determinadas combinaciones de pares no estén permitidas: en el ejemplo de la sección anterior podría ocurrir que, entre los conjuntos B y C, el par (6,9) no estuviera permitido. En este caso, no tiene sentido incluir este par en ningún caso de prueba.

Los casos de prueba siguiendo el algoritmo AETG visto en la sección anterior son:

$\{(1, 5, 8), (1, 6, 9), (1, 7, 8), (2, 5, 9), (3, 5, 8), (4, 5, 8), (2, 6, 8), (2, 7, 9), (3, 6, 9), (4, 6, 9), (3, 7, 8), (4, 7, 8)\}$

Pero si el par  $(6,9)$  no está permitido, esto quiere decir que el caso de prueba  $(1,6,9)$  que aparece en el resultado no es un caso de prueba válido. Si borráramos este caso de prueba, lo que sucedería es que los pares  $(1,6)$  entre  $A$  y  $B$  y el par  $(1,9)$  entre  $A$  y  $C$  quedarían sin ser visitados, por que no se lograría la cobertura pairwise deseada.

El algoritmo PROW ha sido ideado por los autores de este libro. Las principales características de este algoritmo son:

- i. Restricciones entre pares: permite el borrado entre pares de valores de los parámetros que no sean válidos.
- ii. Priorización de los casos de prueba: permite asignar un peso a cada par, de forma que los casos de prueba pueden ser ordenados según el peso.

Para explicar el algoritmo usaremos como ejemplo los parámetros de la Figura 24. La aplicación bajo incluye distintas combinaciones de sistema operativo, navegador y procesador de texto. Por las supuestas restricciones del sistema, los siguientes pares entre valores no son válidos y deben ser eliminados antes de ejecutar el algoritmo:  $(Linux, IExplorer)$ ,  $(Linux, Microsoft Word)$ ,  $(Mac, IExplorer)$ ,  $(Mac, MicrosoftWord)$  y  $(Windows, Opera)$ .

Sistema Operativo	Navegador	Procesador Texto
Linux	Firefox	Microsoft Word
Windows	IE Explorer	Open Office
Mac OS	Opera	Feng Office
	Chrome	Google Docs

Figura 24. Parámetros y valores para PROW

12 pairs in (0, 1)			12 pairs in (0, 2)			16 pairs in (1, 2)		
Elements	Remove	Sel. factor	Elements	Remove	Sel. factor	Elements	Remove	Sel. factor
(Linux, Firefox)	<input type="checkbox"/>	1	(Linux, Microsoft Word)	<input checked="" type="checkbox"/>	0.0	(Firefox, Microsoft Word)	<input type="checkbox"/>	0.0
(Linux, Chrome)	<input type="checkbox"/>	0.3	(Linux, Google Docs)	<input type="checkbox"/>	0.2	(Firefox, Google Docs)	<input type="checkbox"/>	0.0
(Linux, IE Explorer)	<input checked="" type="checkbox"/>	0.0	(Linux, Open Office)	<input type="checkbox"/>	1	(Firefox, Open Office)	<input type="checkbox"/>	0.0
(Linux, Opera)	<input type="checkbox"/>	0.8	(Linux, Feng Office)	<input type="checkbox"/>	0.8	(Firefox, Feng Office)	<input type="checkbox"/>	0.0
(Windows, Firefox)	<input type="checkbox"/>	0.8	(Windows, Microsoft Word)	<input type="checkbox"/>	1	(Chrome, Microsoft Word)	<input type="checkbox"/>	0.0
(Windows, Chrome)	<input type="checkbox"/>	0.3	(Windows, Google Docs)	<input type="checkbox"/>	0.4	(Chrome, Google Docs)	<input type="checkbox"/>	0.0
(Windows, IE Explorer)	<input type="checkbox"/>	1	(Windows, Open Office)	<input type="checkbox"/>	0.5	(Chrome, Open Office)	<input type="checkbox"/>	0.0
(Windows, Opera)	<input checked="" type="checkbox"/>	0.0	(Windows, Feng Office)	<input type="checkbox"/>	0.2	(Chrome, Feng Office)	<input type="checkbox"/>	0.0
(Mac OS, Firefox)	<input type="checkbox"/>	1	(Mac OS, Microsoft Word)	<input checked="" type="checkbox"/>	0.0	(IE Explorer, Microsoft Word)	<input type="checkbox"/>	1
(Mac OS, Chrome)	<input type="checkbox"/>	0.8	(Mac OS, Google Docs)	<input type="checkbox"/>	0.6	(IE Explorer, Google Docs)	<input type="checkbox"/>	0.0
(Mac OS, IE Explorer)	<input checked="" type="checkbox"/>	0.0	(Mac OS, Open Office)	<input type="checkbox"/>	1	(IE Explorer, Open Office)	<input type="checkbox"/>	0.0
(Mac OS, Opera)	<input type="checkbox"/>	0.3	(Mac OS, Feng Office)	<input type="checkbox"/>	0.2	(IE Explorer, Feng Office)	<input type="checkbox"/>	0.0
						(Opera, Microsoft Word)	<input type="checkbox"/>	0.0
						(Opera, Google Docs)	<input type="checkbox"/>	0.0
						(Opera, Open Office)	<input type="checkbox"/>	0.0
						(Opera, Feng Office)	<input type="checkbox"/>	0.0

Figura 25. Tabla de pares con pares borrados y pesos

### 3.5 Antirandom

Antirandom es una estrategia de combinación propuesta por Malayia<sup>18</sup>: partiendo de un caso de prueba  $tc_1$  en el *test suite*, el siguiente en ser elegido (sea  $tc_2$ ) es aquél que resulte “más diferente” de  $tc_1$ . El tercer caso (sea  $tc_3$ ) es el que sea simultáneamente “más diferente” de  $tc_1$  y  $tc_2$ . El siguiente, el “más diferente” de los tres ya seleccionados.

El concepto de diferencia se calcula en función de la *distancia Hamming* entre la codificación binaria de los casos de prueba. Para cada parámetro se utilizan tantos bits como sean necesarios para codificar sus valores: para  $A$ , que tiene 4 elementos, se utilizarán 2 bits; para  $B$ , que tiene 3 elementos, se requieren también

<sup>18</sup> Malayia YK (1995). *Antirandom testing: Getting the most out of black-box testing*. Proceedings of the International Symposium on Software Reliability Engineering (ISSRE'95). Toulouse, Francia. IEEE Computer Society Press: Los Alamitos, California, pp. 86–95.

3 bits (uno de los elementos recibirá dos codificaciones); para  $C$ , que tiene 2 elementos, basta un bit (Tabla 19).

A		B		C	
V	Bits	Valor	Bits	Valor	Bits
1	00	5	00	8	0
2	01	6	01	9	1
3	10	7	10		
4	11	7	11		

Tabla 19. Codificación binaria de los elementos de  $A$ ,  $B$  y  $C$

El primer caso de prueba puede ser el correspondiente a los valores (1, 5, 8), cuya representación binaria es 00000. La distancia Hamming entre dos palabras binarias es el número de bits diferentes entre ambas palabras, por lo que el siguiente caso de prueba en ser elegido será el que corresponda a la palabra 11111, que es el (4, 7, 9).

Para el siguiente caso, se suman las distancias Hamming de cada posible caso a los dos ya elegidos. Como se ve en la Tabla 20, en la primera iteración del algoritmo la suma de distancias Hamming es siempre 5. Para elegir un caso cuando hay empates, como este caso, se calcula la distancia cartesiana, que es la suma de las raíces cuadradas de las distancias Hamming, y que se muestra en la última columna de la tabla: ahora sí, se toma por ejemplo el primer caso con distancia 3,15, que es la palabra 01001 y que corresponde al caso (2, 5, 9).

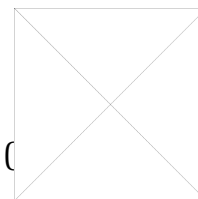


Caso	Distancias Hamming			Distancia cartesiana
	Al caso 00000	Al caso 11111	Total	
00001	1	4	5	3
01000	1	4	5	3
01001	2	3	5	3,15
01010	2	3	5	3,15
01011	3	2	5	3,15
01100	2	3	5	3,15
01101	3	2	5	3,15
01110	3	2	5	3,15
01111	4	1	5	3
10000	1	4	5	3
10001	2	3	5	3,15
10010	2	3	5	3,15
10011	3	2	5	3,15
10110	3	2	5	3,15
10111	4	1	5	3
11000	2	3	5	3,15
11001	3	2	5	3,15
11010	3	2	5	3,15
11011	4	1	5	3
11100	3	2	5	3,15
11101	4	1	5	3
11110	4	1	5	3,15

Tabla 20. Distancias Hamming en la primera iteración del algoritmo *Antirandom*

### 3.6 Algoritmo del peine (*Comb*)

Este algoritmo está ideado por los autores de este libro y está inspirado en *Antirandom*. Una de las desventajas de *Antirandom* es que, para seleccionar el siguiente caso de prueba, deben calcularse las distancias Hamming o cartesiana a cada uno de los casos del *test suite*, lo que supone un coste computacional muy alto.



Puesto que el máximo número de combinaciones es conocido ( $2^n$ ),

donde  $n$  es el número de parámetros y  $S_i$  representa el parámetro  $i$ -ésimo), *Comb* toma, como primer caso de prueba, el que se encuentra en la primera posición (el 0); a continuación, añade al *test suite* el más alejado (en el ejemplo de los conjuntos  $A$ ,  $B$  y  $C$ , el 23); después, el que se encuentre a la misma distancia de ambos (el 12); luego, el que se encuentre equidistante entre el 0 y el 12, y otro que se encuentre equidistante entre el 12 y el 23 (el 18). El algoritmo evoluciona de esta forma hasta que se haya generado el número de casos preestablecido. La equidistancia que se va manteniendo entre las combinaciones elegidas puede recordar a las púas de un peine, motivo por el cual se lo ha bautizado de esta manera.

### 3.7 Algoritmos aleatorios

Los algoritmos aleatorios generan casos de prueba al azar, siguiendo a veces alguna distribución de probabilidad. Un posible algoritmo de este tipo puede guardar en una lista  $n$  números aleatorios entre 0 y el número máximo de combinaciones menos 1 y, luego, recorrer la lista mediante la función *getCombinación* (Figura 21, página 49) para obtener los casos de prueba que correspondan.

## 4 CTWeb, una aplicación web para testing combinatorio

CTWeb es una aplicación web (disponible en <http://alarcosj.esi.uclm.es/CTWeb/>) en la que se implementan diversas estrategias de testing combinatorio, entre ellas todas las mencionadas en la sección anterior.

La siguiente figura muestra la página principal de la aplicación, que se ha cargado con los datos que se obtienen al pulsar el botón *Play example*. Este ejemplo representa los diferentes elementos necesarios para construir juegos de mesa: hay seis parámetros (de la *A* a la *F*) y cada uno tiene una serie de valores: *A* (que representa el juego) toma los valores *Ludo*, *Trivial*, *Checkers* y *Chess* (respectivamente: Parchís, Trivial Pursuit, Damas y Ajedrez); *B* toma *Dice* cuando el juego requiere dados, y *null* en otro caso; *C* denota al jugador oponente, que puede ser una persona o el ordenador; *D* indica el número de jugadores (2 o 3); *E* representa el método de pago (tarjeta Visa, Master Card o American Express) y *F* denota si el juego lleva un sistema de preguntas o no, lo cual es aplicable para el caso del Trivial Pursuit.

El funcionamiento de la aplicación es sencillo: se introducen los datos en la matriz de cajas de texto (se añaden parámetros pulsando el botón *Add set*, y se añaden valores pulsando *Add row*), se selecciona en la izquierda la estrategia de combinación que se desea aplicar, se pulsa el botón *Execute* y la aplicación devuelve, en el lado inferior, los resultados.

## Combinatorial testing page

Cite this site as: Polo M. and Pérez B. (2010). A framework and a web implementation for combinatorial testing. White paper of the Alarcos Research Group. Available at (current date): <http://alarcosj.esi.uclm.es/CTWeb>

First of all, we add so many columns as sets we need (with the *Add set* button), and so many rows (*Add row*) as elements

[Play example](#)

Please, press the *Play example* button to continue.

[Upload feature model file](#) [Upload variables file](#) (see [example of structure](#))

Algorithms		Data																																								
<input checked="" type="radio"/> All combinations (exponential cost) <input type="radio"/> Each choice (very low cost) <input type="radio"/> Antirandom (exponential cost) <input type="radio"/> Comb (lineal cost) <input type="radio"/> Genetic <input type="radio"/> Costly pairwise (exponential cost) <input type="radio"/> AETG (polynomial cost) <input type="radio"/> Customizable AETG (beta, polynomial cost) <input type="radio"/> PROW (polynomial cost) <input type="radio"/> Customizable pairwise (exponential cost) <input type="radio"/> Bacteriologic <input type="radio"/> Random (lineal cost)		<div> <input type="button" value="Add set"/> <input type="button" value="Add row"/> </div> <table border="1"> <thead> <tr> <th></th> <th>A</th> <th>B</th> <th>C</th> <th>D</th> <th>E</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Ludo</td> <td>Dice</td> <td>Person</td> <td>2</td> <td>Visa</td> <td>Quiz</td> </tr> <tr> <td>1</td> <td>Trivial</td> <td>null</td> <td>Computer</td> <td>3</td> <td>Master card</td> <td>-</td> </tr> <tr> <td>2</td> <td>Checkers</td> <td></td> <td></td> <td></td> <td>American express</td> <td></td> </tr> <tr> <td>3</td> <td>Chess</td> <td></td> <td></td> <td></td> <td></td> <td></td> </tr> </tbody> </table> <div> <div>Expression to generate test cases:</div> <div> <input type="text"/> </div> </div> <div> <div>Example</div> <pre> public void testTCNUMBER {     ClassUnderTest o=new ClassUnderTest(#A, #B);     o.method1(#C);     double result=o.method2(#C, #B, #A); } </pre> </div>							A	B	C	D	E	F	0	Ludo	Dice	Person	2	Visa	Quiz	1	Trivial	null	Computer	3	Master card	-	2	Checkers				American express		3	Chess					
	A	B	C	D	E	F																																				
0	Ludo	Dice	Person	2	Visa	Quiz																																				
1	Trivial	null	Computer	3	Master card	-																																				
2	Checkers				American express																																					
3	Chess																																									
<div> <input type="button" value="Execute"/> <input type="checkbox"/> Verbose         </div>																																										
<div>Additional stuff</div> <table border="1"> <tbody> <tr> <td>White paper "<a href="#">A framework and a web implementation for combinatorial testing</a>"</td> <td><a href="#">Source code of the algorithms, test cases, etc.</a></td> <td>Algorithms in Excel, developed by José Antonio Cruz. (Coming soon).</td> </tr> </tbody> </table>								White paper " <a href="#">A framework and a web implementation for combinatorial testing</a> "	<a href="#">Source code of the algorithms, test cases, etc.</a>	Algorithms in Excel, developed by José Antonio Cruz. (Coming soon).																																
White paper " <a href="#">A framework and a web implementation for combinatorial testing</a> "	<a href="#">Source code of the algorithms, test cases, etc.</a>	Algorithms in Excel, developed by José Antonio Cruz. (Coming soon).																																								

Developed by Macario Polo Usaola and Beatriz Pérez Lamancha  
 Alarcos Group  
 Department of Information Systems and Technologies  
 University of Castilla-La Mancha  
 Spain

Figura 26. Página principal del sistema CTWeb, cargada con datos de ejemplo

El *tester* puede utilizar el área de texto situada bajo la etiqueta *Expression to generate test cases* para escribir una expresión que le permita generar el código de datos o casos de prueba. Supongamos que los casos de prueba consistirán en la creación de una instancia de clase *Game* (Juego) y, a continuación, en la asignación de los diferentes posibles elementos (dados, oponente, etcétera). En esa zona de texto podemos escribir código u otro tipo de texto que haga referencia al valor de cada conjunto de cada combinación utilizando el símbolo almohadilla seguido de la letra correspondiente a cada conjunto (Figura 27).

Expression to generate test cases:

```

Game g=new #A();
g.setDice(#B);
Opponent opp=new #C();
g.setOpponent(opp);
g.setPlayers(#D);
g.setPaymentMethod(#E);
g.setQuiz(#F);

```

Figura 27. Código de ejemplo para generar casos de prueba con código

Así, a partir del ejemplo mostrado, y para el ejemplo de los juegos de mesa, la herramienta devuelve los resultados con el algoritmo AETG que aparecen en la Figura 28: en la primera columna aparece el número de combinación; en la segunda, los elementos que la componen; en la tercera y última, el código que

procede de sustituir los valores de los conjuntos  $A$  a  $F$  en el texto mostrado en la Figura 27. Más abajo se indica el tiempo que ha tardado el servidor en realizar el cómputo y el porcentaje de pares que se han visitado.

Algorithm "aetg"

#	Results for 192 combinations
1	{Trivial,null,Computer,3,Master card,-} Game g=new Trivial(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(Master card); g.setQuiz(-);
2	{Trivial,null,Person,3,American express,Quiz} Game g=new Trivial(); g.setDice(null); Opponent opp=new Person(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(American express); g.setQuiz(Quiz);
3	{Trivial,Dice,Person,2,Visa,Quiz} Game g=new Trivial(); g.setDice(Dice); Opponent opp=new Person(); g.setOpponent(opp); g.setPlayers(2); g.setPaymentMethod(Visa); g.setQuiz(Quiz);
4	{Ludo,null,Computer,2,Visa,-} Game g=new Ludo(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(2); g.setPaymentMethod(Visa); g.setQuiz(-);
5	{Ludo,null,Person,3,American express,-} Game g=new Ludo(); g.setDice(null); Opponent opp=new Person(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(American express); g.setQuiz(-);
6	{Ludo,Dice,Computer,3,Master card,Quiz} Game g=new Ludo(); g.setDice(Dice); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(Master card); g.setQuiz(Quiz);
7	{Chess,null,Computer,3,Visa,Quiz} Game g=new Chess(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(Visa); g.setQuiz(Quiz);
8	{Chess,null,Computer,3,American express,-} Game g=new Chess(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(American express); g.setQuiz(-);
9	{Chess,Dice,Person,2,Master card,-} Game g=new Chess(); g.setDice(Dice); Opponent opp=new Person(); g.setOpponent(opp); g.setPlayers(2); g.setPaymentMethod(Master card); g.setQuiz(-);
10	{Checkers,null,Computer,3,Master card,-} Game g=new Checkers(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(Master card); g.setQuiz(-);
11	{Checkers,null,Computer,3,Visa,-} Game g=new Checkers(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(Visa); g.setQuiz(-);
12	{Checkers,null,Computer,3,American express,-} Game g=new Checkers(); g.setDice(null); Opponent opp=new Computer(); g.setOpponent(opp); g.setPlayers(3); g.setPaymentMethod(American express); g.setQuiz(-);
13	{Checkers,Dice,Person,2,American express,Quiz} Game g=new Checkers(); g.setDice(Dice); Opponent opp=new Person(); g.setOpponent(opp); g.setPlayers(2); g.setPaymentMethod(American express); g.setQuiz(Quiz);

Computed in 15 milliseconds  
Pairs visited: 100.0%

Figura 28. Resultados de aplicar AETG a los datos de los juegos de mesa con el código de la Figura 27