# Pruebas de Sistemas de Información


**Universidad de Castilla-La Mancha**

**Departamento de Tecnologías y Sistemas de Información**

**Programa Oficial de Postgrado en Tecnologías Informáticas Avanzadas**

**Macario Polo Usaola**
Departamento de Tecnologías y Sistemas de Información
Paseo de la Universidad, 4
13071-Ciudad Real
macario.polo@uclm.es

# Índice

# Capítulo 1. LA IMPORTANCIA DE LAS PRUEBAS EN EL CICLO DE VIDA

La fase de pruebas es una de las más costosas del ciclo de vida software. En sentido estricto, deben realizarse pruebas de todos los artefactos generados durante la construcción de un producto software, lo que incluye las especificaciones de requisitos, casos de uso, diagramas de diversos tipos y, por supuesto, el código fuente y el resto de elementos que forman parte de la aplicación (como por ejemplo, la base de datos). Obviamente, se aplican diferentes técnicas de prueba a cada tipo de producto software.

En este primer capítulo se enmarca el proceso de pruebas dentro del ciclo de vida software, se describe brevemente una serie de buenas prácticas para el proceso de pruebas, y se presentan algunos retos en cuanto a la automatización del proceso.

## 1. El proceso de pruebas en el ciclo de vida

El estándar ISO/IEC 12207 [1] identifica tres grupos de procesos en el ciclo de vida software:

- Procesos principales, grupo en el que incluye los procesos de Adquisición, Suministro, Desarrollo, Operación y Mantenimiento.
- Procesos de la organización, en donde se encuentran los procesos de Gestión, Mejora, Infraestructura y Formación.
- Procesos de soporte o auxiliares, en donde están los procesos de Documentación, Gestión de la Configuración, Auditoría, Resolución de Problemas, Revisión Conjunta, Aseguramiento de la Calidad, Verificación, Validación,

No define, como vemos, un proceso de Pruebas como tal, sino que aconseja, durante la ejecución de los procesos principales o de la organización, utilizar los procesos de soporte. Entre éstos se encuentran los procesos de Validación y de Verificación:

- El proceso de Validación tiene como objetivo determinar si los requisitos y el sistema final cumplen los objetivos para los que se construyó el producto, respondiendo así a la pregunta *¿el producto es correcto?*
- El proceso de Verificación intenta determinar si los productos software de una actividad se ajustan a los requisitos o a las condiciones impuestas en actividades anteriores. De este modo, la pregunta a la que responde este proceso es *¿se está construyendo el producto correctamente?*

Del proceso de Verificación se observa la importancia de verificar cada uno de los productos que se van construyendo, pues se asume que si lo que se va construyendo es todo ello correcto, también lo será el producto final. Igualmente, se observa que el proceso de Validación resalta la importancia de comprobar el cumplimiento de los objetivos de los requisitos y del sistema final, de suerte que podría construirse un Plan de pruebas de aceptación desde el momento mismo de tener los requisitos, que sería comprobado al finalizar el proyecto. Si tras la fase de requisitos viniese una segunda de diseño a alto nivel del sistema, también podría prepararse un Plan de pruebas de integración, que sería comprobado tras tener (o según se van teniendo) codificados los diferentes módulos del sistema. Esta correspondencia entre fases del desarrollo y niveles de prueba produce el llamado "modelo en V", del que se muestra un ejemplo en la Figura 1. En la figura se muestra cómo, efectivamente, mientras que en el desarrollo se va de lo más general a lo más particular, en las pruebas se va de lo más particular a lo más general: si lo primero que se hace es recolectar los requisitos con el usuario, las últimas pruebas que se hacen son las de aceptación, con el mismo usuario; si lo último que se construye es el código, lo primero que se hace son las pruebas unitarias de dicho código.

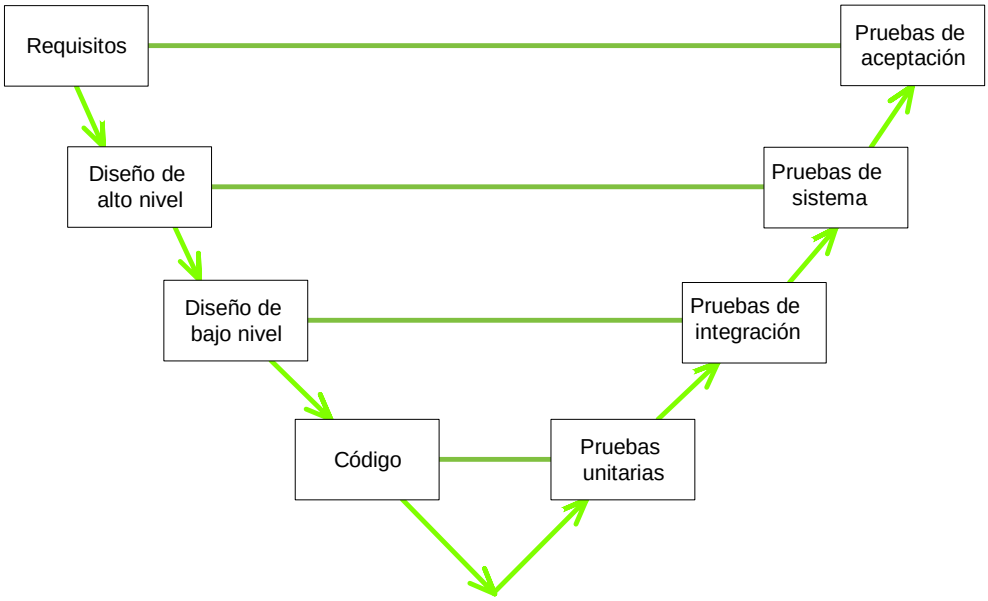



**Figura 1. Modelo en V**

## 2. Las pruebas en algunas modelos

El *Capability Maturity Model* (CMM) hace una consideración especial de las pruebas. Las áreas de este modelo que más relacionadas se encuentran con la fase de pruebas son *Software Product Engineering*, *Training Program*, *Technology Change Management* y *Process Change Management*. CMM establece cuatro niveles para la realización de pruebas: de unidad, de integración, de sistema y de aceptación, además de las de regresión, que se ejecutan para verificar la corrección de los cambios en el sistema. Deben redactarse un plan de pruebas y definirse estándares de pruebas junto al resto del proceso. El grupo de pruebas debe estar bien distinguido respecto del de desarrollo y debe realizar las pruebas independientemente de éste.

## 3. El MTPF (*Minimal Test Practice Framework*)

Karlström et al. describen un proceso para pequeñas organizaciones que deseen adoptar buenas prácticas en pruebas [2]. El MTPF (*Minimal Test Practice Framework*) se estructura en cinco categorías y consta de tres fases (que, aproximadamente, representan tres niveles de madurez). Su objetivo es pasar de una estrategias de pruebas “ad hoc” a un proceso de pruebas claro y estructurado.

A cada categoría le corresponden un conjunto de prácticas de trabajo relacionadas con las pruebas, que deben realizarse según la fase en la que se encuentre la organización. Las fases se describen en función de la cantidad de personal de desarrollo que trabaja en la organización. Para equipos de desarrollo superiores a 30 personas, los propios autores recomiendan el uso de otros frameworks.

La Tabla 1 describe brevemente las actividades de cada fase y categoría.

| | | | | | |
|---|---|---|---|---|---|
| Fase 3<br>(~30 personas) | **Mantenimiento del sistema**<br>Mantener el sistema para que se adapte a la evolución de la compañía | **Definición de equipos**<br>Creación de un equipo de pruebas independiente del de desarrollo.<br>Los miembros de este equipo pueden especializarse (usabilidad, seguridad, etc.). | **Inspecciones**<br>Sustitución progresiva los walkthroughs por inspecciones que, al requerir preparación previa, resultan más eficientes.<br>Se definirán los roles de cada inspector. | **Gestión de riesgos**<br>Se creará y mantendrá una base de datos de problemas y experiencias que ayude a predecir las áreas de mayor riesgo de los proyectos antes de que acontezcan. | **Coordinación del aseguramiento de la calidad**<br>Establecimiento de rutinas de aseguramiento de la calidad, de forma que se asegure que el software no será entregado antes de que alcance el nivel mínimo predeterminado. |
| Fase 2<br>(~20 personas) | **Creación del sistema**<br>Introducción de sistema de recogida y almacenamiento de problemas de acuerdo a estándares de Fase 1.<br>Definición de procedimientos para recolectar, documentar, almacenar y reutilizar conocimiento de cada proyecto. | **Definición de roles**<br>Asignar los roles de responsable de pruebas y de ingenieros de pruebas (*testers)*.<br>Las responsabilidades de los *testers* son: gestión de walkthroughs, gestión de, desarrollo de casos de prueba; gestión del sistema de recogida de problemas; gestión de experiencia. | **Walkthroughs**<br>Realización de walkthroughs antes de que el software esté preparado para su ejecución, idealmente en la fase de diseño.<br>El equipo de walkthroughs se compone de desarrolladores y diseñadores, al que puede asistir un *tester*. | **Casos de prueba**<br>Se construyen casos de prueba para comprobar que se prueban las situaciones y acciones más comunes. A la creación de casos de prueba se le asignarán varios *testers,* ya que es una de las actividades que requiere más tiempo.<br>Se crearán varios casos para cada escenario. | |
| Fase 1<br>(~10 personas) | **Definición de estándares de registro**<br>Definición de terminología, lenguaje, procedimientos, etc.:<br>• Campos que, juntos, describan el problema<br>• Que sigan el flujo del *tester*<br>• Fácil de entender<br>• Que identifique cada problema unívocamente<br>• Que permita agrupar problemas | **Definir responsabilidades**<br>Las responsabilidades son: desarrollo de planes de pruebas para cada proyecto; gestión del entorno de pruebas; gestión del sistema de recogida de problemas; actualización de checklists; evaluación de prácticas; control de necesidades para la siguiente fase | **Uso de checklists**<br>Creación de checklists, o revisión de las que pudieran estar siendo utilizadas. | **Gestión básica**<br>El entorno de pruebas debe estar disponible siempre que se requiera. Las actividades básicas son: organizar el entorno de pruebas para cada proyecto, mantenerlo actualizado y documentar su forma de uso. | **Plan de pruebas**<br>El plan de pruebas permite recoger en un solo documento todo lo relacionado con las pruebas de cada proyecto (p.ej., el IEEE Std. for Soft. Test Documentation).<br>Es importante indicar los diferentes hitos en el plan de pruebas. |
| Fases / Categorías | Registro sistemático de defectos | Definición de roles y organización | Verificación y validación | Gestión de las pruebas | Planificación |

**Tabla 1. Resumen del MTPF**

Cada fase se introduce en la organización cuidadosamente, siguiendo cinco pasos: preparar la fase, introducirla, revisarla, realizarla y evaluarla, como se ilustra en la

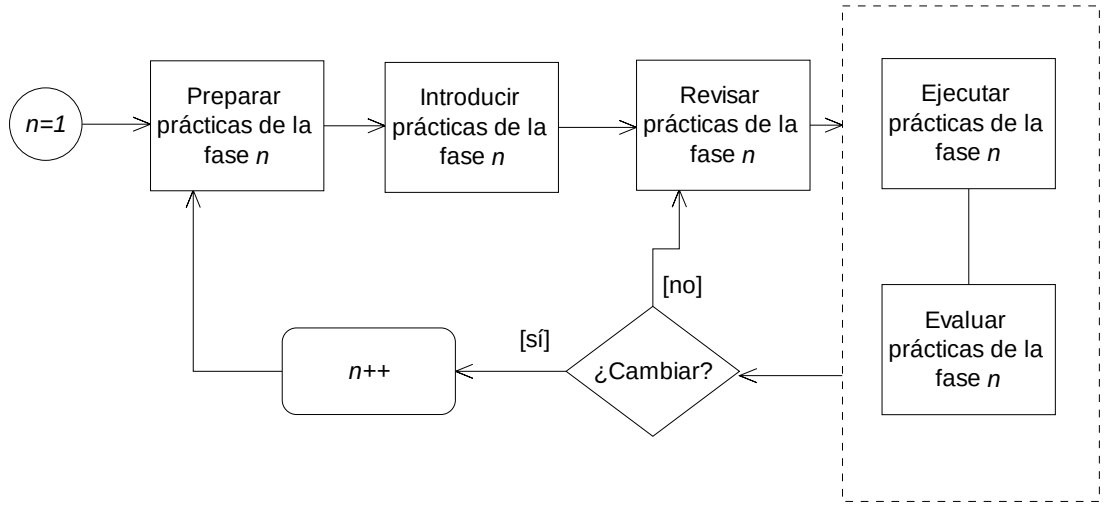



**Figura 2. Método seguido para introducir cada fase**

## 4. El plan de pruebas

El plan de pruebas es un documento que se utiliza para indicar los recursos, los elementos que se van a probar, las actividades, el personal y los riesgos asociados. El estándar IEEE 829 es el estándar para documentación de las pruebas, e indica la siguiente estructura para el plan de pruebas:

1) Identificador del documento.
2) Introducción, resumen de los elementos y las características que se van a probar.
3) Elementos que se van a probar (programas, módulos…)
4) Características que se van a probar.
5) Características que no se van a probar.
6) Enfoque general de la prueba.
7) Criterios de paso o fallo.
8) Criterios de suspensión y de reanudación.
9) Documentación asociada.
10) Actividades de preparación y ejecución de pruebas.
11) Entorno necesario.
12) Responsables.

13) Necesidades de personal y de formación.
14) Esquema de tiempos.
15) Riesgos asumidos y planes de contingencia.
16) Aprobaciones, con las firmas de los responsables.

## 5. Automatización de las pruebas

Diversos estudios recientes han destacado la falta de automatización de las tareas relacionadas con las pruebas de software en la mayoría de las compañías [3, 4]. De acuerdo con Meudec [5], hay tres líneas de automatización en este contexto:

1) **Tareas administrativas**, como el registro de especificaciones o la generación de informes.
2) **Tareas mecánicas**, como la ejecución y la monitorización, o las posibilidades de captura y reejecución de casos.
3) **Tareas de generación de casos de prueba**. Respecto estas tareas, [6] indican tres enfoques:
    a. Generar casos de prueba a partir de código fuente para alcanzar un nivel determinado de cobertura.
    b. Dado el conocimiento del ingeniero de pruebas sobre el programa y sobre su comportamiento esperado, generar automáticamente entradas y comprobar las salidas.
    c. Dada una especificación formal, generar automáticamente casos de prueba para una implementación de esa especificación.

En los últimos años se han desarrollado una serie de herramientas ligadas al *Test Driven Development* (Desarrollo Dirigido por las Pruebas) que han tenido un éxito importante en el desarrollo de software. Estas herramientas, que se agrupan habitualmente bajo la denominación X-Unit, automatizan parcialmente las pruebas unitarias de código orientado a objetos, en los que habitualmente se considera que la unidad de pruebas es la clase. En X-Unit, siendo *K* la clase que se va a probar (abreviadamente *CUT,* de las siglas *Class Under Test*), se construye una clase *TestK* que contiene métodos (que realmente constituyen

casos de prueba) que ejercitan las diversas funcionalidades ofrecidas por *K*. Así, cada caso de prueba suele consistir en la construcción de una instancia de la CUT, en la ejecución de una serie de servicios sobre ésta y en la comprobación del resultado (a lo que se le llama oráculo). La Tabla 2 muestra la clase Java *TestAccount,* que prueba la clase de dominio *Account* (que representa una cuenta bancaria) mediante tres casos de prueba, correspondientes los tres métodos *test1, test2* y *test3*.

- *test1* construye una instancia de la CUT, ingresa 1000 euros en ella y, a continuación, comprueba en su oráculo que el saldo de la cuenta es, en efecto, de 1000 euros.
- *test2* crea una instancia de la CUT, ingresa 300 euros y retira 1000. Puesto que la cuenta no tiene saldo suficiente, se espera que la CUT lance una excepción de saldo insuficiente (que se captura en el bloque *catch)*, superándose en este supuesto el caso de prueba. Si la excepción no se lanza, el control del programa no salta al *catch* y se ejecutaría la instrucción *fail,* que indica al entorno de pruebas que el caso no ha sido superado.
- *test3* crea una instancia de la CUT, ingresa 1000 euros y luego comprueba que el saldo es de 1000 euros; después retira 300 y comprueba que el saldo es de 700 euros. Si se produce alguna excepción, se salta al bloque *catch,* que ejecuta el *fail,* indicando al entorno de pruebas que el caso de prueba ha fallado.

Una importante ventaja de los entornos X-Unit es que los casos de prueba se guardan en ficheros separados, lo que permite reejecutarlos según la CUT va siendo modificada (lo que los convierte en casos válidos para pruebas de regresión). Una desventaja es que se requiere un esfuerzo importante para escribir buenos casos de prueba que consigan probar todo el funcionamiento de la CUT. Por otro lado, estos entornos no generan normalmente demasiados resultados cuantitativos sobre los resultados de las pruebas, informando la mayoría de ellos de si los casos han encontrado o no defectos en la CUT.

```
package samples.results;

import junit.framework.*;
import samples.Account;
import samples.InsufficientBalanceException;

public class TestAccount extends TestCase {

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

  public void test2() {
    try {
       Account o=new Account();
       o.deposit(300);
       o.withdraw(1000);
       fail("InsufficientBalanceException expected");
    }
    catch (InsufficientBalanceException e) {
    }
  }

  public void test3() {
    try {
       Account o=new Account();
       o.deposit(1000);
       assertTrue(o.getBalance()==1000);
       o.withdraw(300);
       assertTrue(o.getBalance()==700);
    }
    catch (Exception e) {
       fail("Unexpected exception");
    }
  }

  public static void main (String [] args) {
    junit.swingui.TestRunner.run(TestAccount.class);
  }
}
```

**Tabla 2. Tres casos de prueba de JUnit para una clase *Account* (Cuenta bancaria)**

La Tabla 3 ofrece una valoración aproximada del grado de automatización que los entornos X-Unit consiguen hacer respecto de la automatización de las pruebas, de acuerdo con las tres líneas mencionadas más arriba.

| | | |
|---|---|---|
| Generación de casos | | 5% |
| Tareas mecánicas | Ejecución y monitorización | 100% |
| | Reejecución | 100% |
| Tareas administrativas | Registro de especificaciones | 80% |
| | Generación de informes | 10% |

**Tabla 3. Automatización de las pruebas conseguida por los entornos X-Unit**

Así pues, queda bastante por hacer en cuanto a automatización de las pruebas. En muchas ocasiones, lo que falta es una cultura de pruebas en las propias empresas. Rice enumera y explica los diez principales obstáculos que existen en las empresas respecto de la automatización del proceso de pruebas:

- **Falta de herramientas**, debida fundamentalmente a su elevado precio o a que las existentes no se ajusten al propósito o entorno para el que se necesitan. La primera razón parece deberse a la no mucha importancia que habitualmente se le da a la fase de pruebas, y eso que el costo de corregir un error puede, en muchos casos, superar al de la licencia de uso. Sería conveniente evaluar el coste de corrección de defectos del software entregado y compararlo con el de la licencia de la herramienta de pruebas.
- **Falta de compatibilidad e interoperabilidad** entre herramientas.
- **Falta de proceso de gestión de la configuración**. Igual que las diferentes versiones del código fuente, las pruebas, especialmente las de regresión, deben someterse a un control de versiones. Recuérdese que el proceso de Gestión de la Configuración es uno de los procesos de soporte del estándar ISO/IEC 12207 , que debería utilizarse en la ejecución de los procesos principales, y muy especialmente en los de Desarrollo y Mantenimiento.
- **Falta de un proceso básico de pruebas** y de conocimiento de qué es lo que se debe probar.
- **Falta de uso de las herramientas de prueba** que ya se poseen, bien por su dificultad de uso, por falta de tiempo para aprender a manejarla, por falta de soporte técnico, obsolescencia, etc.
- **Formación inadecuada** en el uso de la herramienta.
- **La herramienta no cubre todos los tipos de prueba que se desean** (corrección, fiabilidad, seguridad, rendimiento, etc.). Obviamente, a la hora de elegir la herramienta, deberían tenerse priorizados los tipos de pruebas, y entonces hacer la elección de la herramienta basados en esto. A veces también es necesario utilizar no una, sino varias

herramientas de prueba, así como tener en cuenta que es imposible automatizar el 100% de las pruebas.

- **Falta de soporte o comprensión por parte de los gestores**, debido otra vez a la escasa importancia que habitualmente se le da a la fase de pruebas.
- **Organización inadecuada** del equipo de pruebas.
- Adquisición de una **herramienta inadecuada**.

# Capítulo 2. 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 la 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 [7]: 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 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 3 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 3, en las que se pasan diferentes ternas de valores a los métodos que asignan la longitud a los lados del

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

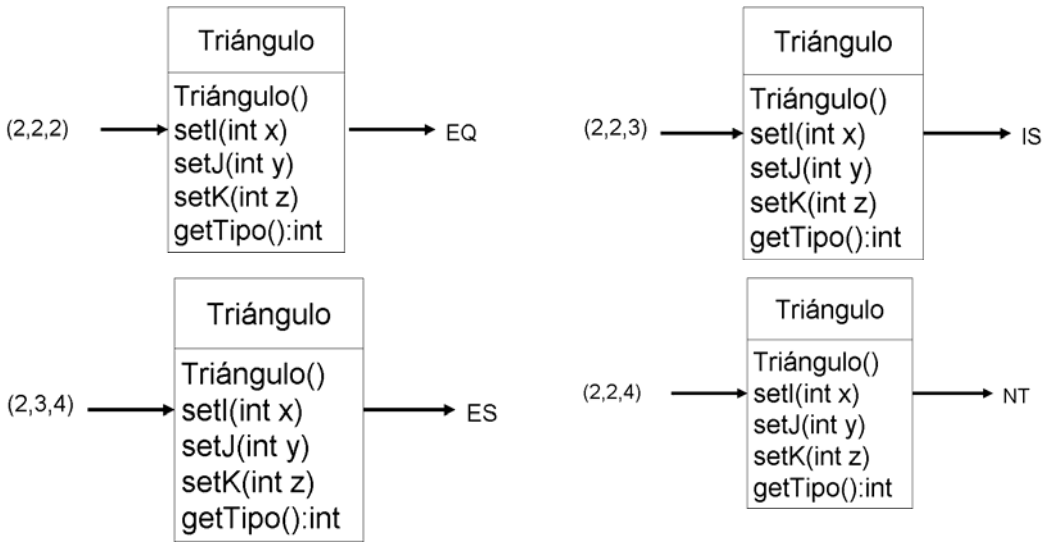



**Figura 3. El problema del triángulo, bajo un enfoque de caja negra**

Como puede comprobarse, si los casos de prueba de caja negra se superan, el ingeniero de pruebas estará seguro de que la 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?

Con el fin de alcanzar una mayor seguridad respecto del comportamiento correcto de la CUT 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 código fuente según se van ejecutando los casos de prueba, de manera que se determinan de manera concreta las instrucciones, bloques, etc. que han sido ejecutados por los casos de prueba. Así pues, mediante este tipo de pruebas se puede saber *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 $x_1$, $x_2$, etc., el recorrido seguido por los casos de prueba en la implementación de la clase (Figura 4).

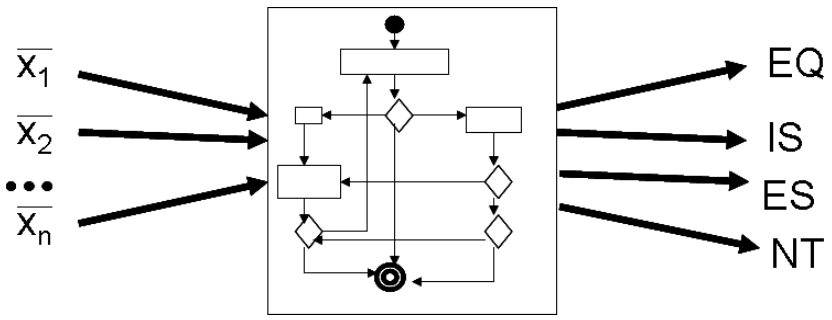



**Figura 4. El problema del triángulo, desde un punto de vista de caja blanca**

Así, dependiendo de las ramas o 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 objeto de estudio. Existen formas muy variadas de medir esa "cantidad" de código recorrido mediante lo que se llaman "criterios de cobertura".

Para Cornett [8], 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 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, define la unidad de prueba de la siguiente forma [9]:

> Un conjunto de uno o más módulos de un programa, junto a con 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*[1] es correcto para los datos que se le pasan como entrada. Así, las pruebas unitarias de caja negra entienden la clase como, en efecto, una caja cuyo interior no interesa: lo único que importa desde este punto de vista es el conjunto de entradas suministradas y las salidas obtenidas.

### 3.1. Un modelo de proceso para pruebas unitarias

Las pruebas de caja negra y de caja blanca no son excluyentes, sino que son complementarias: con las de pruebas de caja negra se buscan errores en la CUT, y con las de caja blanca nos aseguramos de que todo el código (según el criterio de cobertura que se haya seleccionado) ha sido recorrido.

Así, un proceso de pruebas unitarias que combine técnicas de caja negra con técnicas de caja blanca seguirá los siguientes pasos:

1) Sea *C* la *CUT*
2) Escribir un conjunto de casos de prueba *TC* para probar *C*
3) Ejecutar *TC* sobre *C* con una herramienta de caja negra
4) Si *TC* descubre errores en *C*, entonces corregir *C* y volver al paso 3
5) Ejecutar *TC* sobre *C* con una herramienta de caja blanca
6) Si *TC* no alcanza un umbral de cobertura mínimo sobre *C,* entonces es necesario escribir nuevos casos de prueba que se añaden a *TC,* y se vuelve al paso 3
7) Se está en este paso cuando *TC* no ha encontrado errores en *C* y cuando *TC* ha recorrido todo el código de *C.* En esta situación, todo el código ha sido probado sin descubrir errores, con lo que la clase *C* tiene una calidad muy alta.

Este proceso se muestra gráficamente en la Figura 5: obsérvese que, después de ejecutar las pruebas de caja blanca, el ingeniero de pruebas se pregunta si la cobertura obtenida alcanza un determinado "umbral", que deberá haber sido

[1] A la clase que se está probando se la llama en inglés *class under test,* y en muchos artículos y papers se abrevia diciendo *CUT*.

determinado con anterioridad al comienzo del proceso. Dependiendo del criterio de cobertura elegido, el umbral puede ser más o menos exigente: así, matar más del 90% de los mutantes puede ser muy difícil para algunos problemas, por lo que este valor sería un buen umbral al utilizar técnicas de mutación; para cobertura de sentencias, sin embargo, es necesario recorrer el 100% de ellas.

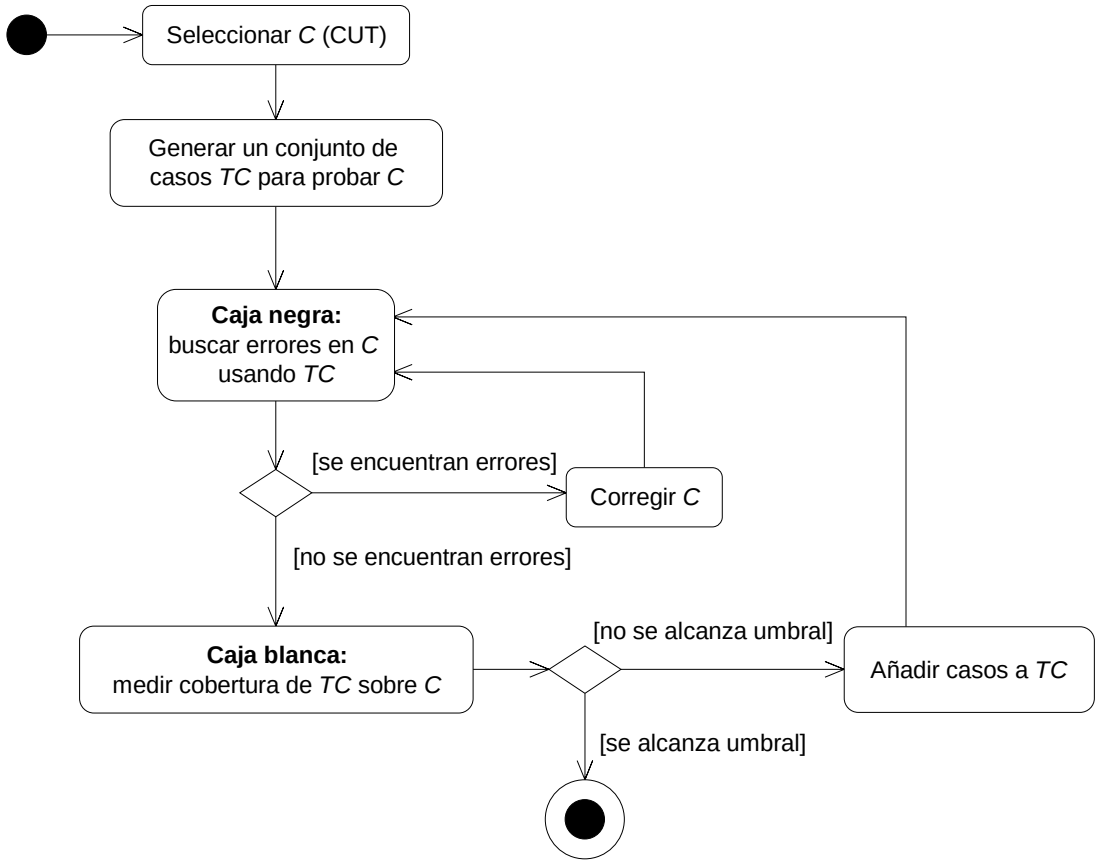



**Figura 5. Estructura del proceso de pruebas unitarias, combinando pruebas funcionales (caja negra) con estructurales (caja blanca)**

## 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 [10].

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 [11].

## 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.).

Bajo este nivel de pruebas encontramos varios subniveles [10]:

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 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 empotrados, es inaceptable que el software proporcione las funciones requeridas fuera de las condiciones de rendimiento exigidas.

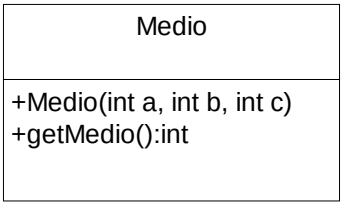
## 6. Ejercicios

1) Escriba una clase para calcular el valor medio de 3 números que se pasan como parámetros. A continuación, someta el programa al proceso de pruebas descrito en la Figura 5 hasta que:

(a) Se hayan recorrido todas las sentencias.

(b) Se hayan recorrido todas las condiciones a *true* y a *false*.

La interfaz de la clase debe ser la siguiente:


Medio
+Medio(int a, int b, int c)
+getMedio():int

# Capítulo 3. PRUEBAS DE CAJA BLANCA

Como se ha comentado en el capítulo anterior, en las pruebas de caja blanca el ingeniero de pruebas se interesa por conocer las regiones del código que han sido recorridas. Así, si se encuentra con que una zona del código no ha sido recorrida por los casos de prueba, se añadirán nuevos casos de prueba que fuercen a que se pase por ahí.

## 1. Medidas de la cobertura

La cobertura es la cantidad de código recorrido por un conjunto de casos de prueba. Existen multitud de formas de medir cuánto código se ha recorrido. A continuación se presentan algunos de ellos.

### 1.1. Cobertura de sentencias

Este criterio se satisface cuando se recorren todas las sentencias del programa al menos una vez.

### 1.2. Cobertura de decisiones, de ramas o de todos los arcos

Una *condición* es un par de expresiones algebraicas relacionadas por un operador relacional (<, >, =, >=, <=, <>).

Una *decisión* es una lista de condiciones conectadas por operadores lógicos (AND, OR).

El criterio de cobertura de decisiones se satisface cuando cada decisión se evalúa a *true* y a *false* al menos una vez.

A este criterio también se le llama de ramas (*branch)* o de arcos (*all-edges).*

El fragmento de código mostrado en la Figura 6 se corresponde con la implementación dada al método *getTipo* en la clase Triángulo. Se alcanzará el criterio de cobertura de decisiones cuando todas las ramas de sus sentencias condicionales hayan sido recorridas.

Así, la decisión de la línea 11ª se considerará recorrida de acuerdo con este criterio cuando se entre al bloque de las líneas 12 y 13 y se entre también al bloque de las líneas 15 y 16. Para que estas dos circunstancias acontezcan, debe ocurrir que la decisión *(i+j<=k || j+k<=i || i+k<=j)* tome tanto valor *true* como *false*. Para que sea *true,* basta con que sea *true* cualquiera de sus condiciones; para que sea *false*, deberán serlo las tres condiciones.

```
 1   public int getTipo() {
 2       if (i==j) { tipo=tipo+1; }
 3       if (i==k) { tipo=tipo+2; }
 4       if (j==k) { tipo=tipo+3; }
 5
 6       if (i<=0 || j<=0 || k<=0) {
 7           tipo=Triangulo.NO_TRIANGULO;
 8           return tipo;
 9       }
10       if (tipo==0) {
11          if (i+j<=k || j+k<=i || i+k<=j) {
12               tipo=Triangulo.NO_TRIANGULO;
13               return tipo;
14             } else {
15                 tipo=Triangulo.ESCALENO;
16                 return tipo;
17             }
18        }
19        if (tipo>3) {
20            tipo=Triangulo.EQUILATERO;
21            return tipo;
22        } else if (tipo==1 && i+j>k) {
23            tipo=Triangulo.ISOSCELES;
24            return tipo;
25        } else if (tipo==2 && i+k>j) {
26            tipo=Triangulo.ISOSCELES;
27            return tipo;
28        } else if (tipo==3 && j+k>i) {
29            tipo=Triangulo.ISOSCELES;
30            return tipo;
31        } else {
32            tipo=Triangulo.NO_TRIANGULO;
33            return tipo;
34        }
35    }
```

**Figura 6. Código del método *getTipo* de la clase Triángulo**

### 1.3. Cobertura de condiciones

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

Volviendo al ejemplo de la línea 11 del fragmento de código de la Figura 6, los casos de prueba habrán cumplido el criterio de condiciones cuando *i+j<=k* haya sido verdadero y falso, *j+k<=i* haya sido también verdadero y falso, e *i+k<=j* haya sido también verdadero y falso, para lo cual se necesitaría un mínimo de seis casos de prueba.

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

Requiere que cada condición de cada decisión se evalúe a *true* y a *false* al menos una vez, y que cada decisión se evalúe a *true* y a *false* al menos una vez.

### 1.5. Cobertura modificada de condiciones/decisiones (MC/DC).

Además de requerir el criterio anterior, se requiere que cada condición afecte independientemente a la decisión.

### 1.6. Cobertura múltiple de condiciones (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.

### 1.7. Cobertura de todos los usos (*all-uses)*

Dada una variable *x,* un conjunto de casos de prueba verifica el criterio de todos los usos para *x* si todos los posibles caminos que hay entre la definición de *x* y todos los usos de esta variable son recorridos.

### 1.8. Cobertura de caminos

Este criterio requiere que se recorran todos los caminos linealmente independientes que se encuentren en el grafo de flujo de la unidad que se está probando. El número de caminos linealmente independientes coincide con la complejidad ciclomática de McCabe, la cual puede calcularse contando el número de instrucciones que crean ramas. En el código del Figura 6, hay 11 instrucciones *if,* con lo que la complejidad ciclomática de ese fragmento de código es 11. Este valor coincide con el número de regiones que pueden encontrarse en el diagrama de flujo correspondiente al código que se está probando: la Figura 7 muestra el diagrama de flujo correspondiente a dicho fragmento de código, en el que pueden apreciarse once áreas (incluyendo la exterior).

Para alcanzar este criterio de cobertura en este ejemplo, deberían utilizarse once casos de prueba adecuados que recorrieran todos los posibles caminos.

### 1.9. Cobertura de funciones

Este criterio se verifica cuando se ha llamado a todas las funciones y procedimientos.

### 1.10. Cobertura de llamadas

Se verifica cuando se han ejecutado todas las llamadas a a funciones y procedimientos. No debe confundirse con la cobertura de funciones: en la cobertura de funciones contamos cuántas funciones de las que hay en nuestro programa han sido llamadas, mientras que la cobertura de llamadas cuenta cuántas de las llamadas a funciones que hay en el programa se han ejecutado.

### 1.11. Cubrimiento de bucles

Se verifica cuando todos los bucles se han ejecutados cero veces (excepto para bucles *do..while*), una vez y más de una vez.

### 1.12. Cubrimiento de carrera

Comprueba el número de tareas o hilos que han ejecutado simultáneamente el mismo bloque de código.

### 1.13. Cobertura de operadores relacionales

Comprueba si se han ejecutado los valores límite en los operadores relacionales (>, <, >=, <=), ya que se asume la hipótesis de que estas situaciones son propensas a errores.

### 1.14. Cobertura de tablas

Comprueba si se ha hecho referencia a todos los elementos de los arrays.

### 1.15. Cobertura basada en mutación

Le dedicaremos a esto una sección aparte.

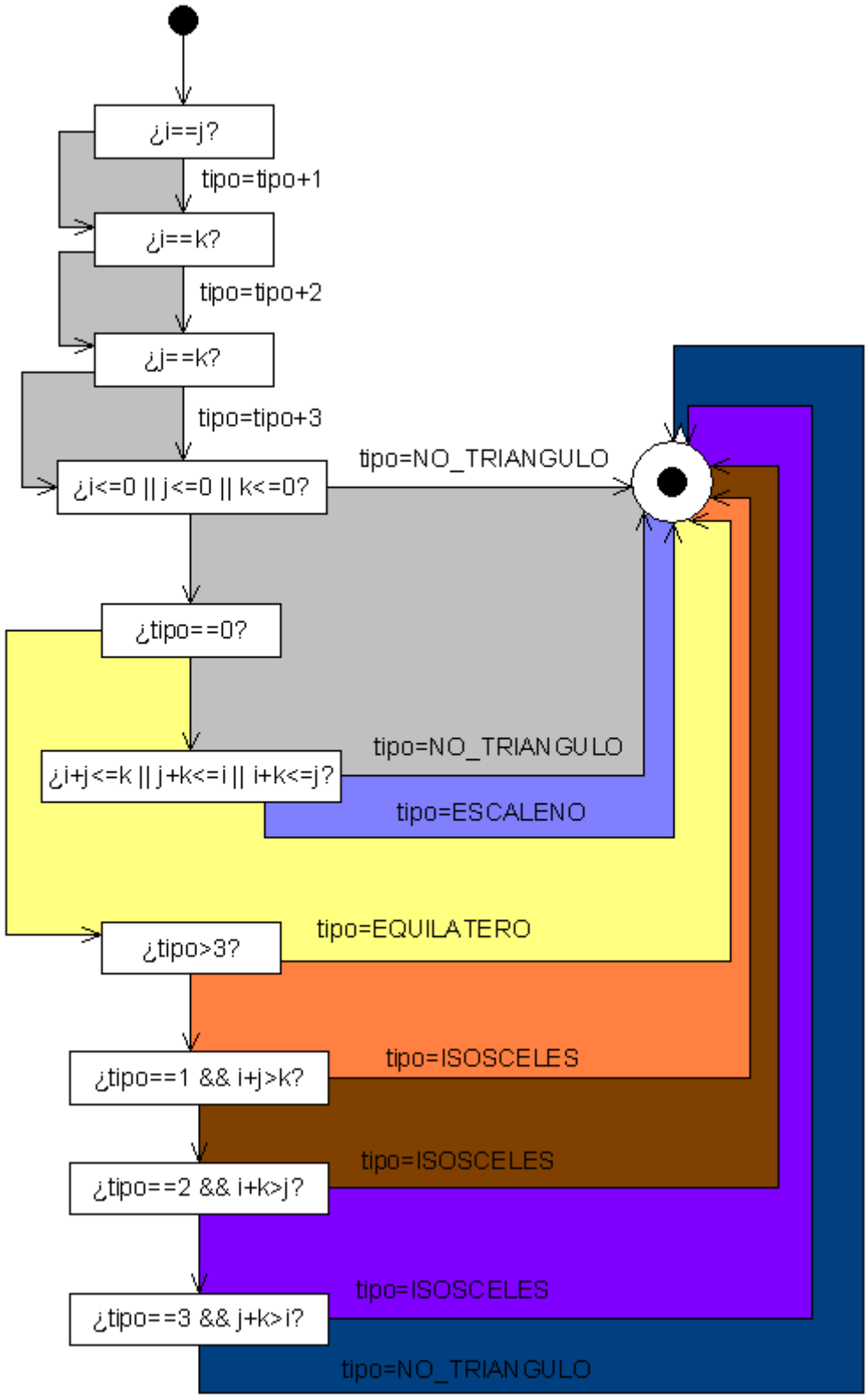



**Figura 7. Diagrama de flujo para el código del método *getType* de la Figura 6**

## 2. La mutación

En el contexto de las pruebas del software, un mutante es una copia del programa que se está probando (programa "original") al que se le ha introducido un único y pequeño cambio sintáctico que no impide que el programa compile (por ejemplo, cambiar un signo + por un * en una expresión aritmética). Así, puede entenderse que el mutante es una versión defectuosa del programa original: es decir, el mutante es el programa original, pero en el que se ha introducido o en el que se ha sembrado un fallo.

El lado izquierdo de la Figura 8 muestra el código del programa que vamos a probar ("Original") y el de algunos mutantes. En el lado derecho aparecen cuatro casos de prueba y los resultados de cada uno sobre las diferentes versiones del programa. Obsérvese que:

- El par (1,1) produce salidas diferentes en el programa original y en los mutantes 1, 2 y 3: esto significa que este caso de prueba es bueno porque ha encontrado los errores introducidos en esas tres versiones.
- El par (0,0) encuentra solamente el error del mutante 3. Respecto de la colección de mutantes mostrada en la figura, es un caso de prueba algo peor que el (1,1), pues encuentra menos errores.
- El mutante 4 ofrece exactamente la misma salida que el programa original para todos los datos de prueba. Además, nos resultará imposible encontrar un caso de prueba que consiga que la salida de este mutante sea distinta.

| Versión | Código |
|---|---|
| Original | int suma(int a, int b)<br>{<br>return a + b;<br>} |
| Mutant 1 | int suma(int a, int b)<br>{<br>return a - b;<br>} |
| Mutant 2 | int suma(int a, int b)<br>{<br>return a * b;<br>} |
| Mutant 3 | int suma(int a, int b)<br>{<br>return a / b;<br>} |
| Mutant 4 | int suma(int a, int b)<br>{<br>return a + b++;<br>} |

**(a) Un programa y unos mutantes**

| | | Datos de prueba (a,b) | | | |
|---|---|---|---|---|---|
| | | (1, 1) | (0, 0) | (-1, 0) | (-1, -1) |
| Versiones | Original | 2 | 0 | -1 | -2 |
| | Mutante 1 | 0 | 0 | -1 | 0 |
| | Mutante 2 | 1 | 0 | 0 | 1 |
| | Mutante 3 | 1 | Error | Error | 1 |
| | Mutante 4 | 2 | 0 | -1 | -2 |

**(b) Resultados con algunos datos de prueba**

**Figura 8. Un programa, varios mutantes y los resultados con algunos casos de prueba**

## 2.1. Algo de terminología

Dado un programa original y un conjunto de mutantes, se dice que un caso de prueba "mata un mutante" cuando la salida del mutante es distinta de la del programa original para ese caso de prueba. Así pues, en la Figura 8, el caso de prueba formado por el par (1,1) mata a los mutantes 1, 2 y 3, mientras que (0,0) mata tan solo al mutante 3.

No hay ningún caso de prueba en nuestro *test suite* que sea capaz de matar al mutante 4, por lo que se dice que este mutante está "vivo". Además, dado el cambio introducido en el código de este mutante, resultará imposible escribir un caso de prueba que lo mate, por lo que se dice que el mutante es "funcionalmente equivalente".

## 2.2. La mutación como criterio de cobertura

Dado un programa *P* formado por *n* líneas de código y dos *test suites* $t_1$ y $t_2$ que sirven para probar *P,* podemos afirmar (con alguna salvedad que no merece la pena entrar a discutir) que $t_1$ es mejor que $t_2$ si $t_1$ recorre más sentencias de *P* que $t_2$. Así pues, la cobertura que un determinado conjunto de casos de prueba

alcanza en un programa es un indicador de la calidad de ese conjunto de casos de prueba.

Del mismo modo, cuantos más mutantes mate un *test suite*, mejor es dicho *test suite*. De hecho, se dice que un *test suite* es adecuado para la mutación (*mutation-adequate*) cuando mata el 100% de los mutantes no equivalentes. Así pues, el grado de adecuación a la mutación se mide calculando el *mutation score*, utilizando la fórmula de la Figura 9.

$$MS(P,T) = \frac{K}{(M - E)}$$ , where:

*P* : programa bajo prueba
*T* : *test suite*
*K* : número de mutantes muertos
*M :* número de mutantes generados
*E* : número de mutantes equivalentes

**Figura 9. Cálculo del *mutation score***

El objetivo de las pruebas utilizando mutación consiste en construir casos de prueba que descubran el error existente en cada mutante. Así pues, los casos de prueba serán buenos cuando, al ser ejecutados sobre el programa original y sobre los mutantes, la salida de éstos difiera de la salida de aquél, ya que esto significará que las instrucciones mutadas (que contienen el error) han sido alcanzadas por los casos de prueba (o, dicho con otras palabras, que los casos de prueba han descubierto los errores introducidos).

## 2.3. Proceso de pruebas basado en mutación

La Figura 10 muestra el proceso de mutación propuesto por Offutt [12]. Las líneas punteadas indican tareas manuales, mientras que las de trazo continuo son tareas automatizables. Inicialmente se genera el conjunto de mutantes, el conjunto de casos de prueba y se define un umbral que representa el porcentaje mínimo de mutantes muertos que debe alcanzarse (el valor mínimo del *mutation score*). Entonces se ejecutan los casos de prueba sobre el programa original y sobre los mutantes, y se calcula el porcentaje de mutantes muertos. Si no se ha alcanzado el umbral previamente definido, se eliminan los casos de prueba que no han matado mutantes y se generan casos de prueba nuevos, especialmente dirigidos a matar los mutantes que han permanecido vivos.

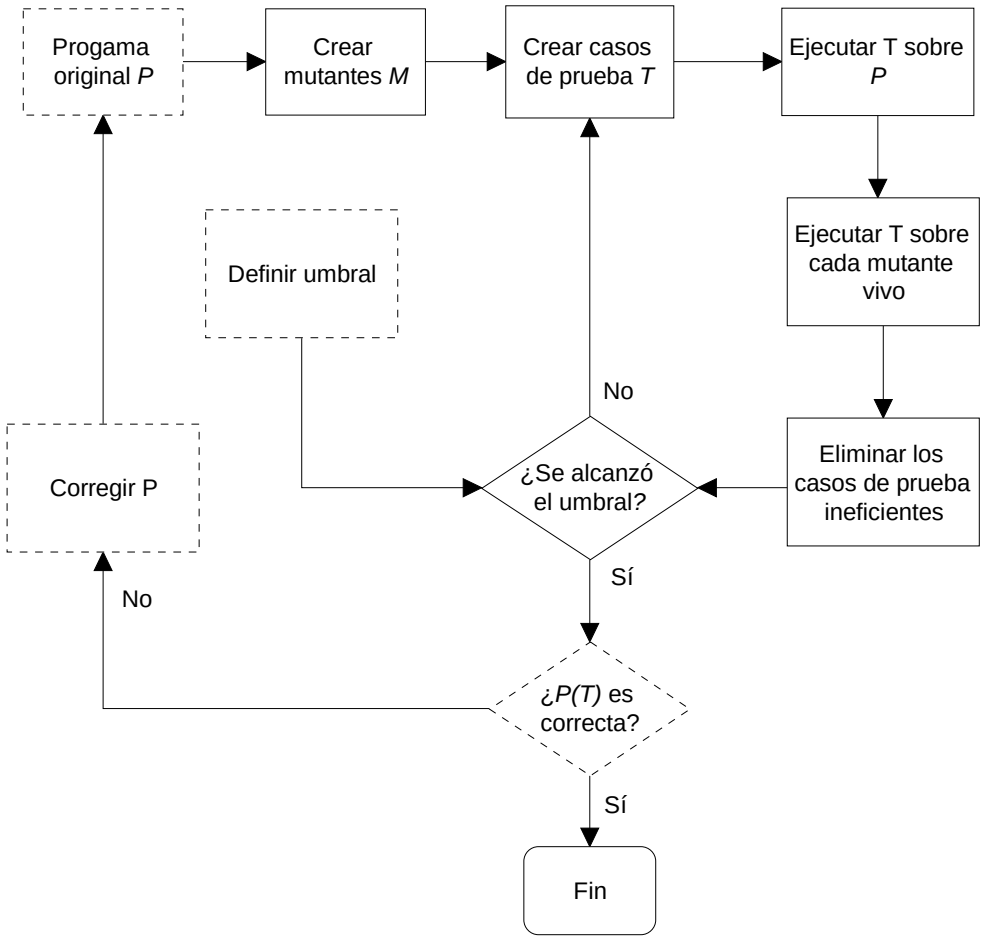



**Figura 10. Proceso de pruebas usando mutación propuesto por Offutt [13]**

El proceso anterior tiene una desventaja importante: si se encuentra que *P(T)* no es correcta, *P* debe ser corregido y, los mutantes, generados de nuevo, con el coste que supone esta tarea. Así pues, como se ha discutido en [14], es preferible utilizar el proceso descrito en la Figura 5.

Para matar a un mutante deben producirse tres condiciones:

- **Alcance**: la sentencia mutada debe ser ejecutada por el caso de prueba.
- **Necesidad**: entre la entrada y la salida del programa debe crearse un estado intermedio erróneo.
- **Suficiencia**: el estado incorrecto debe propagarse hasta la salida del programa.

La condición de *suficiencia* puede ser difícil de conseguir en muchas ocasiones, lo que puede dificultar la obtención de casos de prueba a un coste razona-

ble. La *mutación débil* se queda en la condición de *necesidad* para determinar que un caso de prueba ha matado a un mutante: compara el estado intermedio del mutante y el del programa original inmediatamente después de haber ejecutado la instrucción mutada, marcando el mutante como muerto si los estados son diferentes. Se ha comprobado experimentalmente que esta técnica puede ahorrar hasta el 50% del coste de las pruebas sin degradar excesivamente su calidad.

## 2.4. Operadores de mutación

La generación de mutantes se consigue aplicando *operadores de mutación* al código fuente del programa que queremos probar. La Tabla 4 muestra algunos de los operadores de mutación citados en Offutt et al. [13].

| Operador | Descripción |
|---|---|
| ABS | Sustituir una variable por el valor absoluto de dicha variable |
| ACR | Sustituir una referencia variable a un array por una constante |
| AOR | Sustitución de un operador aritmético |
| CRP | Sustitución del valor de una constante |
| ROR | Sustitución de un operador relacional |
| RSR | Sustitución de la instrucción Return |
| SDL | Eliminación de una sentencia |
| UOI | Inserción de operador unario (p.ej.: en lugar de *x*, poner *–x*) |

**Tabla 4. Algunos operadores de mutación clásicos**

El número de mutantes que puede generarse a partir de un programa sencillo, de pocas líneas de código, es muy grande (piénsese que el operador *AOR*, por ejemplo, puede aplicarse a cada aparición de un operador aritmético en el programa original), resultando también muy costosas tanto la ejecución de los casos de prueba con cada mutante como la comprobación de la salida del programa. Por ello, se han realizado algunos estudios encaminados a disminuir los costes de este tipo de pruebas, evidenciándose que, aplicando pocos operadores, se pueden conseguirse los mismos resultados que si se aplicaran muchos. Los operadores más efectivos son el *ABS, 0* (sustitución de una variable por el valor 0)*, <0* (sustitución de una variable por un valor menor que 0), *>0* (sustitución de una variable por un valor mayor que 0), AOR, ROR y UOI.

Además, también se han propuesto operadores específicos para programas escritos en lenguajes orientados a objeto, algunos de los cuales se muestran en la Tabla 5 .

| Operador | Descripción |
|---|---|
| AMC (Access Modifier Change) | Reemplazo del modificador de acceso (por ejemplo: ponemos *private* en lugar de *public*) |
| AOC (Argument Order Change) | Cambio del orden de los argumentos pasados en la llamada a un método (p.ej.: en lugar de *Persona p=new Persona("Paco", "Pil")* poner *Persona p=new Persona("Pil", "Paco")* |
| CRT (Compatible Reference Type Replacement) | Sustituir una referencia a una instancia de una clase por una referencia a una instancia de una clase compatible (p.ej.: en vez de poner *Persona p=new Empleado()*, poner *Persona p=new Estudiante()*). |
| EHC (Exception Handling Change) | Cambiar una instrucción de manejo de excepciones (try...catch) por un sentencia que propague la excepción (throw), y viceversa |
| EHR (Exception Handgling Removal) | Eliminación de una instrucción de manejo de excepciones |
| HFA (Hiding Field variable Addition) | Añadir en la subclase una variable con el mismo nombre que una variable de su superclase |
| MIR (Method Invocation Replacement) | Reemplazar una llamada a un método por una llamada a otra versión del mismo método |
| OMR (Overriding Method Removal) | Eliminar en la subclase la redefinición de un método definido en una superclase |
| POC (Parameter Order Change) | Cambiar el orden de los parámetros en la declaración de un método (p.ej.: poner *Persona(String apellidos, String nombre)* en vez de *Persona(String nombre, String apellidos)* |
| SMC (Static Modifier Change) | Añadir o eliminar el modificador *static* |

**Tabla 5. Algunos operadores de mutación para orientación a objetos**

## 3. Ejercicios

1) Escriba casos de prueba para lograr cobertura de sentencias, de condiciones y de caminos para el problema del triángulo.

2) Escriba casos de prueba para matar a todos los mutantes del ejercicio 1 del capítulo anterior.

3) Se dice que un criterio de cobertura C1 subsume a otro C2 si, para todo programa P, cualquier caso de prueba T que satisface C1 satisface también C2.

Proponga algunos operadores de mutación para conseguir que la mutación subsuma al criterio de decisiones. ¿Se puede conseguir lo mismo para el criterio de condiciones?

4) Para diagramas de flujo pueden definirse criterios de cobertura como Todos los caminos, Todos los arcos o Todos los nodos. ¿Existen relaciones de subsunción entre estos criterios?

# Capítulo 4. LOS VALORES "INTERESANTES"

En este capítulo se introduce y profundiza en el concepto de "valor interesante", se revisan dos técnicas para utilizarlos y se describen algunos criterios de cobertura para valores.

## 1. Un estudio inicial del problema del triángulo

Si queremos aplicar correctamente el proceso de pruebas descrito en la Figura 5 (página 23) al problema del triángulo (la implementación de su método *getTipo* se daba en la Figura 6, página 28), el ingeniero de pruebas deberá escribir casos de prueba que utilicen, como parámetros de los métodos *setI, setJ* y *setK*, "buenos valores", en el sentido de que con ellos se recorra la mayor cantidad posible de código fuente, en función del criterio de cobertura elegido.

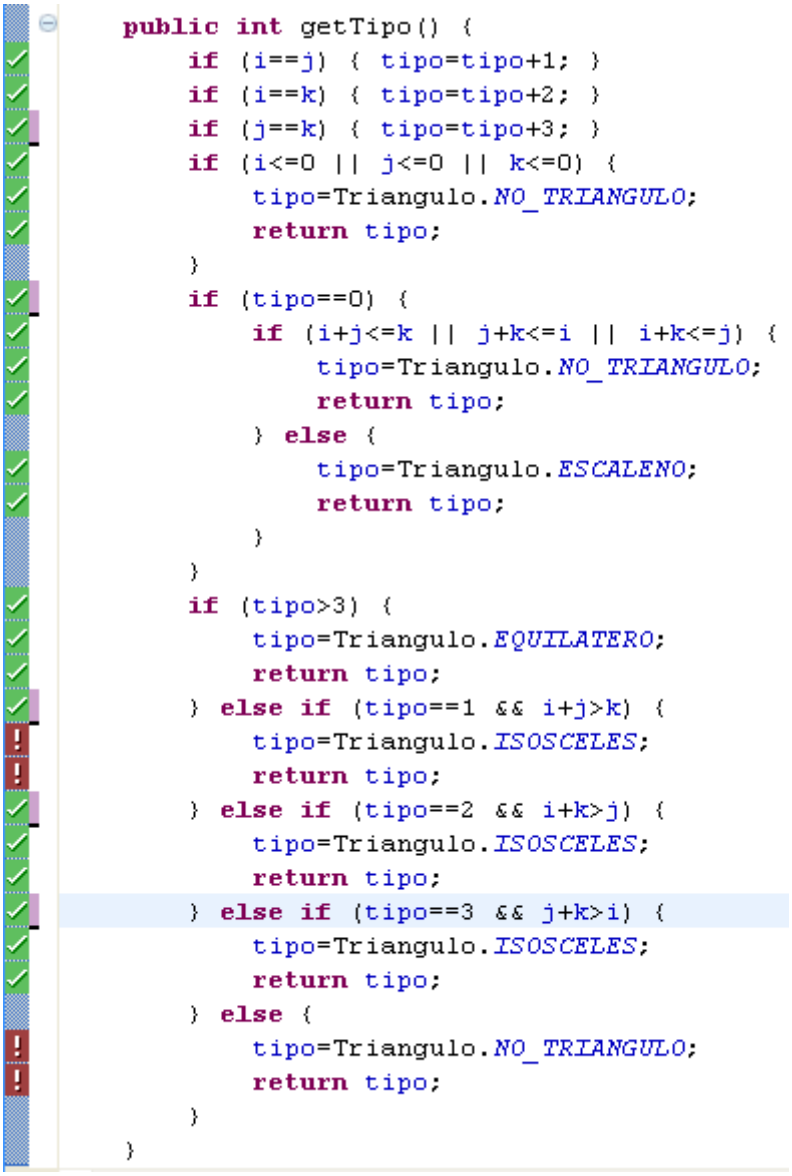
```
public void testEQ1() {
   Triangulo t=new Triangulo();
   t.setI(5);
   t.setJ(5);
   t.setK(5);
   assertTrue(t.getTipo()==t.EQUILATERO);
}

public void testEQ2() {
   Triangulo t=new Triangulo();
   t.setI(3);
   t.setJ(3);
   t.setK(3);
   assertTrue(t.getTipo()==t.EQUILATERO);
}

public void testIS1() {
   Triangulo t=new Triangulo();
   t.setI(3);
   t.setJ(5);
   t.setK(5);
   assertTrue(t.getTipo()==t.ISOSCELES);
}

public void testIS2() {
   Triangulo t=new Triangulo();
   t.setI(5);
   t.setJ(3);
   t.setK(5);
   assertTrue(t.getTipo()==t.ISOSCELES);
}
```

```
public void testES1() {
   Triangulo t=new Triangulo();
   t.setI(4);
   t.setJ(3);
   t.setK(5);
   assertTrue(t.getTipo()==t.ESCALENO);
}

public void testES2() {
   Triangulo t=new Triangulo();
   t.setI(3);
   t.setJ(4);
   t.setK(5);
   assertTrue(t.getTipo()==t.ESCALENO);
}

public void testNT1() {
   Triangulo t=new Triangulo();
   t.setI(1);
   t.setJ(2);
   t.setK(3);
   assertTrue(t.getTipo()==t.NO_TRIANGULO);
}

public void testNT2() {
   Triangulo t=new Triangulo();
   t.setI(0);
   t.setJ(2);
   t.setK(3);
   assertTrue(t.getTipo()==t.NO_TRIANGULO);
}
```

**Figura 11. Ocho casos de prueba en JUnit para el problema del Triángulo**

La Figura 11 muestra ocho casos de prueba en formato JUnit para este problema. Como se observa, se han escrito dos casos para cada una de las posibles

salidas del método *getTipo*. Al medir la cobertura de sentencias alcanzada por estos casos de prueba, se observa que se recorre el 88% de ellas (Figura 12).



```
public int getTipo() {
    if (i==j) { tipo=tipo+1; }
    if (i==k) { tipo=tipo+2; }
    if (j==k) { tipo=tipo+3; }
    if (i<=0 || j<=0 || k<=0) {
        tipo=Triangulo.NO_TRIANGULO;
        return tipo;
    }
    if (tipo==0) {
        if (i+j<=k || j+k<=i || i+k<=j) {
            tipo=Triangulo.NO_TRIANGULO;
            return tipo;
        } else {
            tipo=Triangulo.ESCALENO;
            return tipo;
        }
    }
    if (tipo>3) {
        tipo=Triangulo.EQUILATERO;
        return tipo;
    } else if (tipo==1 && i+j>k) {
        tipo=Triangulo.ISOSCELES;
        return tipo;
    } else if (tipo==2 && i+k>j) {
        tipo=Triangulo.ISOSCELES;
        return tipo;
    } else if (tipo==3 && j+k>i) {
        tipo=Triangulo.ISOSCELES;
        return tipo;
    } else {
        tipo=Triangulo.NO_TRIANGULO;
        return tipo;
    }
}
```

**Figura 12. Sentencias ejecutadas (✓) y no ejecutadas (!) en *getTipo* por los casos de prueba de la Figura 11**

Puesto que no alcanzamos el umbral del 100% con este criterio de cobertura, de acuerdo con la discusión de Cornett que mencionábamos en la página 21, debemos escribir nuevos casos de prueba que ejecuten las líneas no recorridas, que son las que se señalan con la marca ! en la Figura 12. Así, debemos escribir un caso que corresponda a un triángulo isósceles tal que la longitud del tercer lado sea menor que la suma de los otros dos ($i+j>k$), y otro que corresponda al "no triángulo" que se determina en las últimas líneas.

La adición de un nuevo caso con los valores (3, 3, 1) recorre las primeras líneas indicadas, y el caso (5, 5, 10) recorre las últimas. Por tanto, con esta pequeña batería de casos conseguiríamos una cobertura de sentencias del 100%. Con este mismo conjunto de casos de prueba, también conseguimos cobertura del 100% en el criterio "todos los usos" (descrito en la página 29).

Sin embargo, si utilizamos la mutación como criterio de cobertura, estos casos de prueba matan tan sólo el 2% de los mutantes que MuJava genera. MuJava es una herramienta para realizar pruebas de caja blanca de programas Java utilizando mutación [15]. Para el problema del triángulo, MuJava genera 479 mutantes "tradicionales" (correspondientes a aplicar los operadores mostrados en la Tabla 4, página 36) y 82 mutantes "de clase" (correspondientes a los operadores que explotan específicamente las características de los lenguajes de programación orientados a objeto, mostrados en la Tabla 5, página 37).

La Figura 13 muestra, en un pantallazo de MuJava, el código de la clase Triángulo y de unos de sus mutantes tradicionales: obsérvese que la instrucción modificada (se ha sustituido el operador <= por < en la línea 59) supone, en el mutante la simulación de un error introducido involuntariamente por el programador. Puesto que el objetivo de los casos de prueba es encontrar errores en el código, si seguimos el criterio de la mutación deberíamos escribir al menos un caso de prueba que encuentre el error introducido en la zona inferior de la Figura 13.

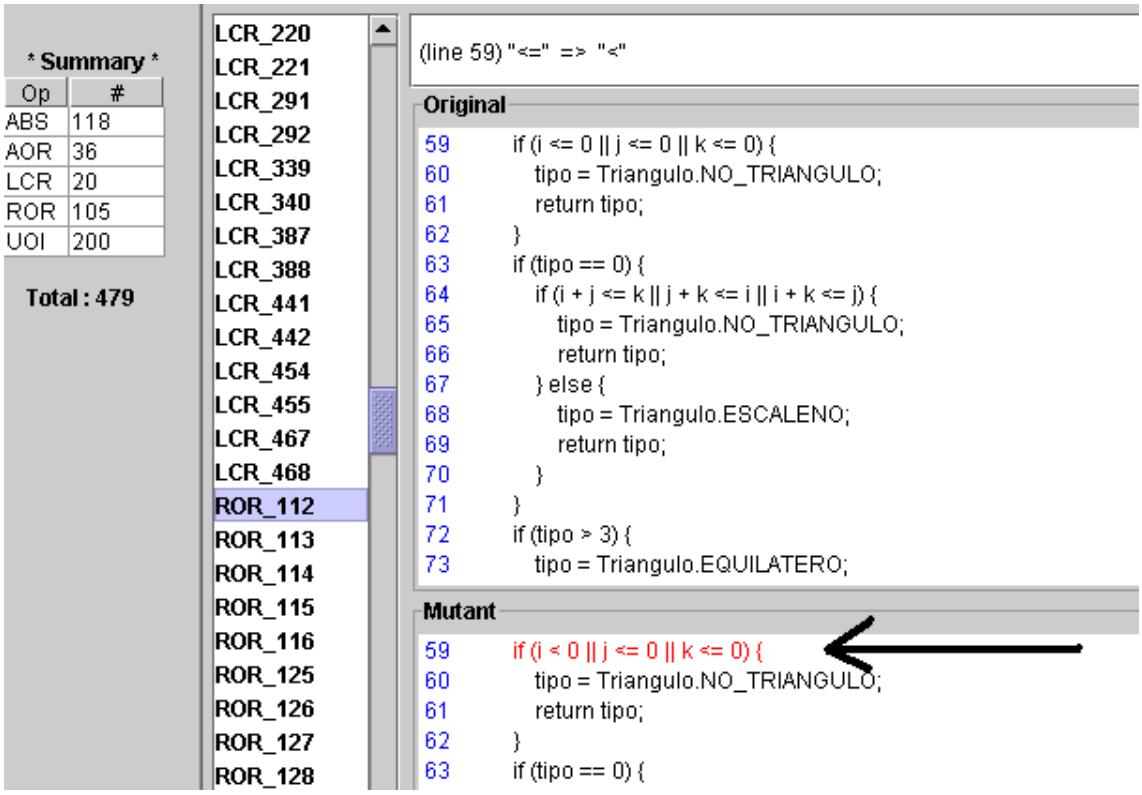



**Figura 13. Código de la clase Triángulo original (arriba) y del mutante ROR_112 generado por MuJava (abajo)**

## 2. Concepto de valor interesante

Desde el punto de vista del testing, un valor interesante es cualquier valor que el ingeniero de pruebas considera que debe ser utilizado para sus casos de prueba [16]. Como ya se ha mencionado, es preciso tener cierta "picardía" para elegir estos valores, ya que deben ir utilizándose aquellos que vayan ejercitando todas las regiones del código.

Así pues, y volviendo a los dos casos de prueba de la Figura 16, la terna de valores (5, 5, 5) no es interesante porque tenemos con anterioridad la (3, 3, 3).

Para proponer valores realmente interesantes existen varias técnicas bien conocidas que, no obstante, vamos a recordar en las siguientes secciones.

### 2.1. Clases de equivalencia

Con esta técnica, se divide el dominio de valores de entrada en un número finito de clases de equivalencia. Se asume que el comportamiento de la CUT ante un elemento cualquiera de una clase de equivalencia dada será el mismo.

## 2.2. Valores límite

Esta técnica complementa a 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.

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.

## 2.3. Fundamento matemático

Hoffman et al. [17] exploran los fundamentos matemáticos de las técnicas de valores límite y clases de equivalencia. Siendo $D=(d_o, ..., d_{n-1})$ un dominio de datos de entrada no vacío y ordenado para un cierto parámetro, se definen las siguientes funciones:

$$límites(D,k) = \begin{cases} \{d_{k-1}, d_{n-k}\}, si\ n\ es\ par\ y\ k < n/2 \\ \{d_{(n/2)-1}, d_{n/2}\}, si\ n\ es\ par\ y\ k \geq n/2 \\ \{d_{k-1}, d_{n-k}\}, si\ n\ es\ impar\ y\ k < (n+1)/2 \\ \{d_{(n-1)/2}\}, si\ n\ es\ impar\ y\ k \geq (n+1)/2 \end{cases}$$

**Ecuación 1**

$$límites(D,k)^* = \bigcup_{i=1}^{k} límites(D,i)$$

**Ecuación 2**

Para un conjunto de dominios con esas características, se definen las siguientes dos funciones:

$$límites(D_0,...,D_{m-1},k) = límites(D_0,k) \times ... \times límites(D_{m-1},k)$$

**Ecuación 3**

$$límites(D_0,...,D_{m-1},k)^* = \bigcup_{i=1}^{k} límites(D_0,...,D_{m-1},k)$$

**Ecuación 4**

Supongamos que tenemos los dominios *X={0, 1, 2, 3, 4}* e *Y={0, 1, 2, 3}*. Los valores de las dos primeras ecuaciones anteriores son:

| *D (dominio)* | *límites(D, 1)* | *límites(D, 2)* | *límites(D, 3)* | *límites(D,1)** | *límites(D,2)** | *límites(D,3)** |
|---|---|---|---|---|---|---|
| *X* | {0, 4} | {1, 3} | {2} | {0,4} | {0, 4, 1, 3} | {0,4, 1, 3, 2} |
| *Y* | {0, 3} | {1,2} | {1,2} | {0,3} | {0, 3, 1, 2} | {0, 3, 1, 2} |

**Tabla 6. Aplicación de la Ecuación 1 y de la Ecuación 2**

Si combinamos los dos dominios, X e Y, podemos aplicar la Ecuación 3:

| *límites(X, Y, 1)* | *límites(X, Y, 2)* | *límites(X, Y, 3)* |
|---|---|---|
| {(0,0), (0, 3), (4, 0), (4, 3)} | {(1, 1), (1, 2), (3, 1), (3, 2)} | {(2, 1), (2, 2)} |

**Tabla 7. Aplicación de la Ecuación 3**

Y, a partir de la Tabla 7, podemos ya aplicar la Ecuación 4:

| *límites(X, Y, 1)** | *límites(X, Y, 2)** | *límites(X, Y, 3)** |
|---|---|---|
| {(0,0), (0, 3), (4, 0), (4, 3)} | {(0,0), (0, 3), (4, 0), (4, 3) (1, 1), (1, 2), (3, 1), (3, 2)} | {(0,0), (0, 3), (4, 0), (4, 3) (1, 1), (1, 2), (3, 1), (3, 2), (2, 1), (2, 2)} |

**Tabla 8. Aplicación de la Ecuación 4**

Los pares de valores que tenemos en la tercera columna de la Tabla 8 representan casos de prueba: cada elemento del par representa un valor de cada uno de los dominios.

En el mismo artículo, los autores definen también el concepto de *perímetro,* que es muy similar al de los *límites* descrito, sólo que, además de los valores límite, también se añaden a los casos de prueba valores pertenecientes al interior de cada dominio.

## 3. Criterios de cobertura para valores

Además de los criterios de cobertura de código, también pueden definirse criterios de cobertura para los valores de los parámetros de los casos de prueba en función de su "interés". Con estos criterios, viene a medirse el grado en que los diferentes valores interesantes seleccionados se utilizan en la batería de casos de prueba.

A continuación se presentan algunos criterios de cobertura para valores.

### 3.1. Cada uso (*each-use*, o *1-wise*)

Es el criterio más simple. Se satisface cuando cada valor interesante de cada parámetro se incluye, al menos, en un caso de prueba.

Supongamos que, para el ejemplo del triángulo, el ingeniero de pruebas ha seleccionado como valores interesantes el conjunto {0, 1, 2, 3, 4, 5} para cada uno de los lados. Un conjunto de casos de prueba que satisface el criterio 1-wise estaría formado por las siguientes ternas:

1. (0, 1, 2)→ con este, en los próximos casos no sería necesario probar con el 0 en la primera posición, ni con el 1 en la segunda, ni con el 2 en la tercera
2. (1, 0, 2)
3. (2, 3, 4)
4. (2, 4, 3)
5. (3, 5, 0)
6. (4, 2, 1)
7. (3, 0, 5)→ este caso vuelve a emplear el 0 en la segunda posición; sin embargo, es necesario utilizarlo para probar con el 5 en la tercera posición
8. (5, 0, 1)

Como se observa, el 0 aparece en la primera, segunda y tercera posiciones (casos 1, 2 y 5); el 1 aparece en la segunda, primera y tercera (casos 1, 2 y 6); el 2 aparece en las posiciones tercera, primera y segunda (casos 1, 3 y 6); el 3 aparece también en todas las posiciones (casos 5, 3 y 4); el 4 aparece en los casos 3, 4 y 6; por último, el 5 aparece en los casos 8, 5 y 7.

Desafortunadamente, la batería de casos formada por las ternas (0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4) y (5, 5, 5) cumple también el criterio 1-wise. Así pues, la calidad del conjunto de casos puede ser muy mala si no ponemos cuidado al escribirlos todos.

### 3.2. *pair-wise* (o *2-wise*)

Este criterio requiere que cada posible par de valores interesantes de cualesquiera dos parámetros sea incluido en algún caso de prueba. Se trata de un criterio ampliamente utilizado. La siguiente figura muestra un algoritmo sencillo que genera casos de prueba con cobertura *pair-wise*.

```
testSuite = ∅
PairTable[] pairTables=buildPairTables(a, b, c)
while ∃ p∈ pairTables / p.numberOfVisits=0
    i = 0
    found = false
    while not found
      if combination[i] visits p
          testSuite= testSuite ∪ {combination[i]}
          visit pairs of combination[i] on
pairTables
          found = true
      end if
      i = i + 1
    end while
end while
```

**Figura 14. Un algoritmo para pair-wise**

Supongamos que, para el problema del Triángulo, utilizamos los valores {0, 1, 2, 3} para asignar la longitud de sus tres lados. Tendríamos las tres siguientes tablas de pares:

| i/j | i/k | j/k |
|---|---|---|
| (0, 0) | (0, 0) | (0, 0) |
| (0, 1) | (0, 1) | (0, 1) |
| (0, 2) | (0, 2) | (0, 2) |
| (0, 3) | (0, 3) | (0, 3) |
| (1, 0) | (1, 0) | (1, 0) |
| (1, 1) | (1, 1) | (1, 1) |
| (1, 2) | (1, 2) | (1, 2) |
| (1, 3) | (1, 3) | (1, 3) |
| (2, 0) | (2, 0) | (2, 0) |
| (2, 1) | (2, 1) | (2, 1) |
| (2, 2) | (2, 2) | (2, 2) |
| (2, 3) | (2, 3) | (2, 3) |
| (3, 0) | (3, 0) | (3, 0) |
| (3, 1) | (3, 1) | (3, 1) |
| (3, 2) | (3, 2) | (3, 2) |
| (3, 3) | (3, 3) | (3, 3) |

**Figura 15. Tablas de pares para el problema del Triángulo**

A partir de las tablas de pares, debemos ir construyendo casos de prueba que vayan visitando cada par el menor número posible de veces. Por ejemplo, podríamos elegir los tres pares (0, 0) de las tres tablas de arriba, con lo que añadiríamos el caso de prueba (0, 0, 0) al conjunto, y marcaríamos esos tres pares en las tablas, con objeto de no volver a visitarlos.

### *3.3. t-wise*

Es una extensión del *2-wise*, en la que cada combinación posible de valores interesantes de los *t* parámetros de la operación de que se trate se incluye en algún caso de prueba. En nuestro ejemplo, ninguna de las operaciones del triángulo toma más de un parámetro, por lo que la cobertura *2-wise* sería la misma que *t-wise*.

### *3.4. N-wise*

Es un caso especial de *t-wise*: siendo *N* el número de parámetros, *N-wise* se satisface cuando todas las combinaciones posibles de todos los parámetros se incluyen en casos de prueba. Es decir, en *t-wise, t≤N*.

## 4. Casos de prueba redundantes

Continuando con el socorrido ejemplo del Triángulo, si realizásemos un seguimiento de los casos de prueba *testEQ1* y *testEQ2* (presentados en la Figura 11, pero que reproducimos en la Figura 16 para comodidad del lector) observaríamos que recorren exactamente las mismas sentencias y que matan exactamente a los mismos mutantes. El caso *testEQ2,* por tanto, no aporta nada a nuestro proceso de pruebas respecto del caso *testEQ1*.

```
public void testEQ1() {
   Triangulo t=new Triangulo();
   t.setI(5);
   t.setJ(5);
   t.setK(5);
   assertTrue(t.getTipo()==t.EQUILATERO);
}
```

```
public void testEQ2() {
   Triangulo t=new Triangulo();
   t.setI(3);
   t.setJ(3);
   t.setK(3);
   assertTrue(t.getTipo()==t.EQUILATERO);
}
```

**Figura 16. Dos de los casos de prueba del Triángulo, que son redundantes**

Así pues, a la hora de escribir casos de prueba para probar cualquier clase, es necesario aplicar cierto sentido común para utilizar valores “buenos”, en el sentido de que cada nuevo caso de prueba que introduzcamos recorra una zona del código que no haya sido recorrida con por los casos de prueba anteriores o, si aplicamos el criterio de la mutación, que mate mutantes que no habían sido matados por los casos de prueba anteriores.

El tema de los casos de prueba redundantes, por otro lado, provoca el problema de su minimización. En el ejemplo mencionado de la Figura 16, es claro que un solo caso de prueba haría exactamente las mismas funciones que los dos

que se muestran. Hay situaciones en las que se dispone de un número muy grande de casos de prueba entre los cuales se encuentran varios subconjuntos de casos que ejercitan exactamente las mismas sentencias o que matan exactamente el mismo conjunto de mutantes. Así pues, en ocasiones es necesario disponer de un conjunto de casos de prueba que consiga la misma cobertura que otro, teniendo éste un mayor número de elementos. Este problema se conoce con el nombre de "Reducción óptima del conjunto de casos" (*Optimal test-suite reduction*), que ha sido ampliamente estudiado en la literatura . El problema resulta ser NP-completo, por lo que su solución óptima no es alcanzable en tiempo polinomial, si bien en las referencias citadas se pueden encontrar diversos algoritmos (normalmente voraces) que logran soluciones buenas en tiempos razonables.

### 4.1. Reducción del conjunto de casos basado en mutación

El siguiente algoritmo recibe como entradas el conjunto completo de casos de prueba, la clase bajo prueba (CUT) y el conjunto completo de mutantes de la CUT.

En la línea 2, ejecuta todos los casos de la prueba contra la CUT y contra los mutantes, guardando los resultados en *testCaseResults*. El algoritmo está ya preparado para seleccionar, en varias iteraciones, los casos de la prueba que matan a más mutantes, lo cual se hace en el bucle de las líneas 5 a 18. La primera vez que el algoritmo entra en el bucle y llega la línea 7, el valor de *n* (que se utiliza como condición de parada las iteraciones) es |*mutants*|: en este caso especial, el algoritmo busca un caso de prueba que mate a todos los mutantes. Si lo encuentra, el algoritmo agrega el caso de la prueba a *requiredTC*, actualiza el valor de *n* y termina; si no, disminuye *n* (línea 16) y entra nuevamente en el bucle.

Supongamos que *n* es inicialmente 100 (es decir, hay 100 mutantes de la clase bajo prueba), y supongamos también que el algoritmo no encuentra casos de prueba que matan a mutantes hasta *n=30*. Con este valor, la función *getTestCasesThatKillN* (llamada en la línea 7) devuelve tantos casos de prueba como casos de la prueba matan a *n* mutantes diferentes: esto es, si hay dos casos de prueba (*tc1* y *tc2*) que matan a los mismos 30 mutantes, *getTestCasesThatKillN* devuelve solamente un caso de la prueba (por ejemplo, *tc1*). Si la intersec-

ción de los mutantes matados por *tc1* y *tc2* no es vacía, el algoritmo devuelve un conjunto formado *tc1* y *tc2*.

Cuando se encuentran casos de prueba que matan *n* mutantes, se agregan a la variable *requiredTC* (línea 9) y se eliminan los mutantes muertos (líneas 10-13). De este modo, el algoritmo no vuelve a considerar los mutantes que han sido muertos en iteraciones anteriores.

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

**Figura 17. Algoritmo para minimizar el conjunto de casos de prueba**

A la función encargada de recoger el conjunto de casos de prueba que matan a *n* mutantes se la llama en la línea 7 de la figura anterior, y se detalla en la Figura 18. Recorre los elementos contenidos en *testCaseResults* y toma aquellos casos cuya lista de *killedMutants* (Figura 17) tenga *n* elementos.

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

**Figura 18. Recuperación de los casos que matan *n* mutantes**

#### 4.1.1 Ejemplo

La Figura 19 muestra los resultados de ejecutar un conjunto de casos de prueba sobre una clase original y sus mutantes. La celda *(i,j)* se marca con un aspa si el caso de prueba (que aparece en la columna *j)* mata al mutante (que aparece en la fila *i-ésima)*. Los algoritmos vistos trabajan con estructuras de datos de este estilo para realizar la minimización del conjunto de casos. En la parte inferior de la figura aparece el conjunto reducido de casos de prueba.

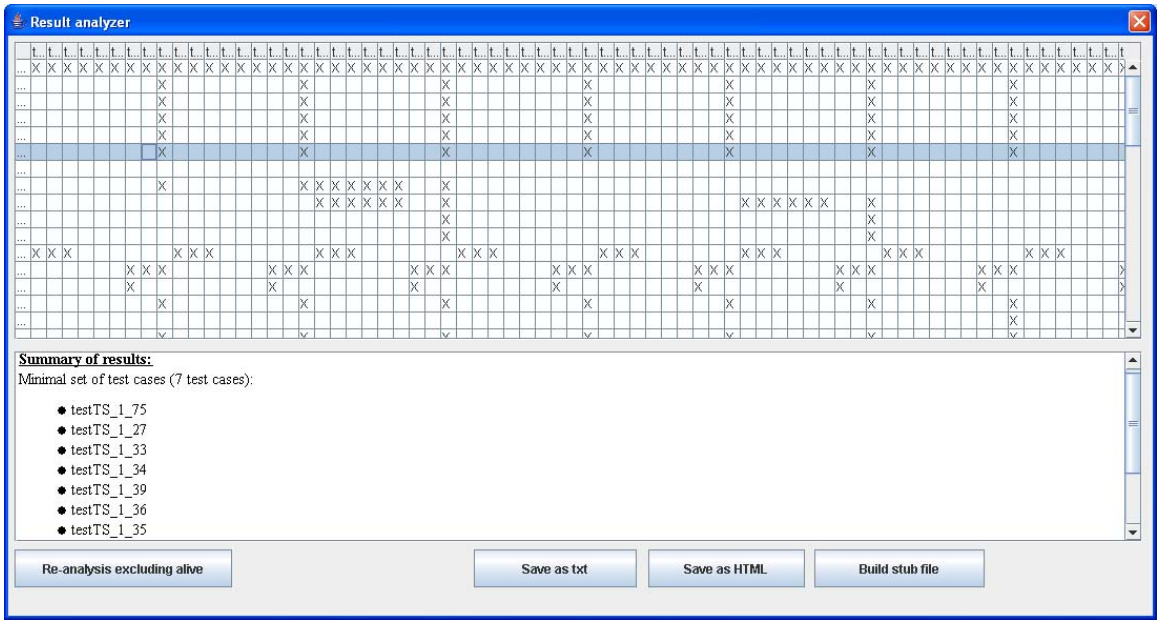



**Figura 19. Tabla de mutantes muertos por un conjunto de casos de prueba**

## 4.2. Algoritmo *HGS*

Harrold, Gupta y Soffa [18] proponen un algoritmo voraz (conocido como *HGS*) para obtener, a partir de un conjunto de casos de prueba dado, otro nuevo conjunto con menor tamaño que conserve los requisitos deseados del conjunto original.

Los pasos principales de este algoritmo son:

1) Inicialmente, todos los requisitos están no marcados.
2) Añadir al resultado los casos de la prueba que verifican un requisito, y marcar los requisitos cubiertos por los casos seleccionados.
3) Ordenar los requisitos no marcados de acuerdo con el número de casos que satisfacen ese requisito. Si varios requisitos son satisfechos por varios casos de prueba, seleccionar el caso de la prueba que marque más requisitos. Marcar los requisitos satisfechos por los casos seleccionados. Eliminar casos redundantes, que son los que no cubren más requisitos no marcados.
4) Repetir el paso 3) hasta que todos los requisitos de prueba están marcados.

## 4.3. Mejoras de Gupta

Con diversos colaboradores, Gupta ha propuesto varias mejoras al algoritmo anterior:

- Con Jeffrey [19], Gupta le agrega "redundancia selectiva". La "redundancia selectiva" permite seleccionar los casos de prueba que, para cualquier requisito dado, proporciona la misma cobertura que otro caso previamente seleccionado. Así, quizá $T'$ alcance el criterio de condiciones, pero quizá no el de todos los mutantes; por tanto, un nuevo caso $t$ puede ser añadido a $T'$ si aumenta la cobertura en cuanto a mutantes: ahora, $T'$ no aumentará el criterio de condiciones, pero sí el de mutantes.
- Con Tallam [20], la selección de los casos de la prueba se basa en técnicas de Análisis de Conceptos. Según los autores, este algoritmo obtiene un conjunto reducido con el mismo tamaño y en un tiempo similares al algoritmo *HGS* original.

### 4.4. Algoritmo de Heimdahl y George

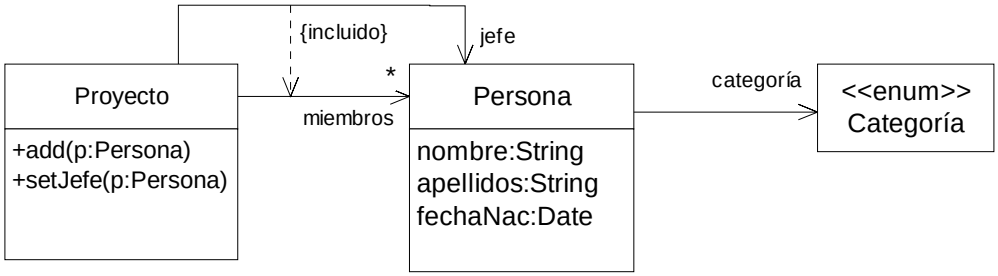
Heimdahl y George [21] también proponen un algoritmo voraz para reducir el conjunto de casos de prueba. Básicamente, toman un caso al azar, lo ejecutan y comprueban la cobertura alcanzada. Si éste es mayor que la cobertura más alta alcanzada, lo agregan al conjunto resultado. El algoritmo se repite cinco veces para obtener cinco conjuntos reducidos distintos, quedándose con el más pequeño. Puesto que el azar es un componente esencial de este algoritmo, la buena calidad de los resultados no está garantizada.

### 4.5. Algoritmo de McMaster y Memon

McMaster y Memon [22] presentan también un algoritmo voraz. El parámetro considerado para incluir los casos de prueba en el conjunto reducido se basa en las "pilas de llamadas únicas" que los casos de prueba producen en el programa bajo prueba.

## 5. Ejercicios

1) Suponga que disponemos de un sistema como el mostrado en la figura. Proponga valores interesantes de tipo *Persona* para las operaciones *add* y *setJefe* del tipo *Proyecto*.




2) Proponga una serie de valores interesantes para el triángulo y para el problema de calcular el máximo que se propuso en un capítulo anterior. (a) Combine dichos valores para obtener casos de prueba cumpliendo los criterios *1-wise choice* y *pair-wise*. (b) Ejecute los casos de prueba y compare la cobertura de código fuente que alcanza cada conjunto de casos utilizando sentencias y mutantes. (c) ¿En alguno de los dos conjuntos hay casos de prueba redundantes? En caso afirmativo, ¿respecto de qué criterio de cobertura? Los casos redundantes respecto de un criterio, ¿lo son también respecto del otro?

# Capítulo 5. ESTRATEGIAS DE COMBINACIÓN PARA LA GENERACIÓN DE CASOS DE PRUEBA

Las estrategias de combinación son métodos que construyen los casos de prueba combinando los valores interesantes de los diferentes objetos que se pasan como parámetros mediante algún algoritmo específico.

## 1. Estrategias de generación de casos de prueba

Grindal et al. presentan un estudio sobre 16 estrategias de combinación, que representan gráficamente de la siguiente forma [16]:

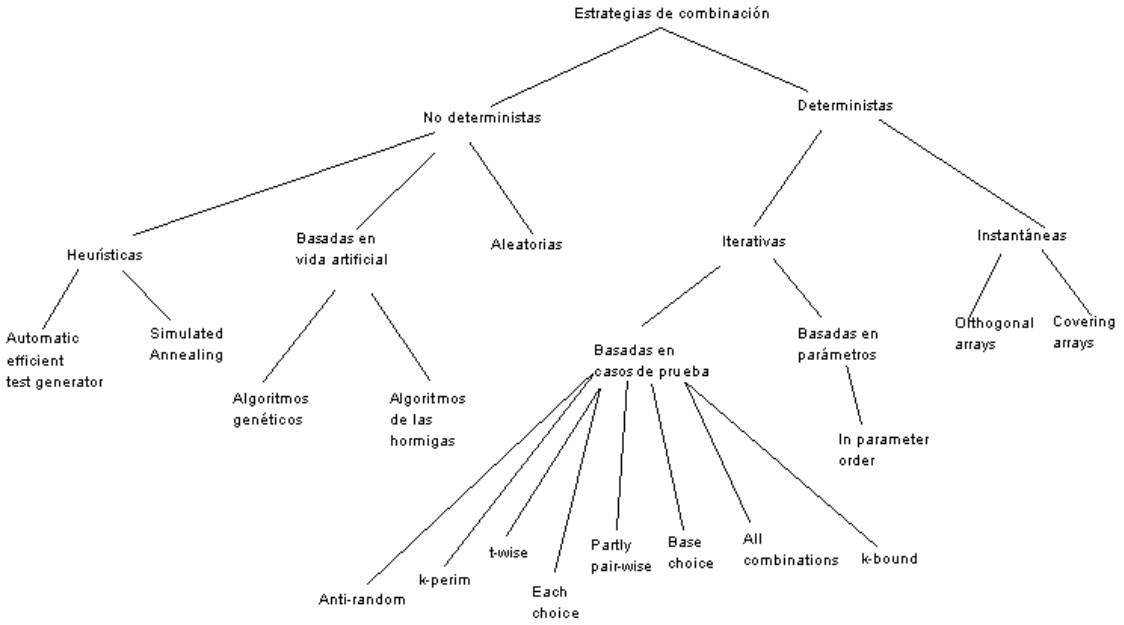



**Figura 20. Clasificación de estrategias de combinación**

Como se ve, las técnicas se clasifican según su algoritmo en deterministas (generan siempre los mismos casos de prueba) y no deterministas (el azar cumple un papel importante, por lo que no se garantiza que en dos ejecuciones del algoritmo se genere el mismo conjunto de casos).

De forma general, el objetivo de todas estas estrategias es conseguir baterías buenas de casos de prueba (donde, como siempre, el concepto de bondad dependerá finalmente del criterio de cobertura elegido).

## 2. Estrategias no deterministas

En este tipo de estrategias, el azar cumple siempre un papel importante en la determinación de qué casos de prueba se generarán. Así, no se asegura que dos ejecuciones del mismo algoritmo produzcan siempre los mismos resultados.

### 2.1. AETG: Automatic efficient test generator

El siguiente algoritmo genera casos de prueba para lograr cobertura *pair-wise*, donde el valor de *k* se fija arbitrariamente. Cuanto mayor es *k,* menor es el tamaño del conjunto de casos, si bien a partir de k=50, el tamaño del conjunto permanece estable.

```
Se parte de un conjunto T de casos de prueba
Sea UC el conjunto de pares de valores de dos parámetros cualesquiera todavía no cubiertos por los casos preseleccionados
while UC≠∅
    UC={pares de valores no cubiertos por T}
    selectedTestCase= λ
    for i=1 to k
        Seleccionar la variable y el valor incluido en más pares de UC
        Ordenar aleatoriamente el resto de variables
        Para cada variable determinada en el paso 5, seleccionar el valor incluido en más pares de UC
            y crear así un caso de prueba, tc
        if tc visita más pares que selectedTestCase then
            selectedTestCase=tc
        endIf
    endFor
    marcar en UC los pares cubiertos por selectedTestCase
    T=T ∪{selectedTestCase}
endWhile
```

**Figura 21. Algoritmo AETG**

### 2.2. Algoritmos Genéticos

De forma general, los algoritmos genéticos constituyen un método de resolución de problemas de minimización. Parten de una solución inicial, que se va aproximando a la óptima según una serie de iteraciones. De la solución inicial van construyéndose soluciones mejores de acuerdo con una función objetivo (*fitness)*.

### 2.2.1 Algoritmos Genéticos (I)

Pargas et al. utilizan el algoritmo mostrado en la Figura 22 para lograr criterios de cobertura altos.

En la primera línea se calcula *CDGPaths*, que representa el grafo de dependencias de control del programa que se va a probar. Un grafo de dependencia de control es un grafo dirigido acíclico cuyos nodos representan sentencias y cuyos arcos representan dependencias de control entre sentencias. Un nodo *Y depende por control* de otro *X* si y sólo si: (1) cualquier nodo intermedio en el camino de *X* a *Y* está postdominado por *Y*; y (2) *X* no está postdominado por Y. Se dice además que un nodo *X* está *postdominado* por otro *Y* si para llegar desde *X* a la salida es preciso pasar siempre por *Y*.

A continuación se inicializa *Scoreboard*, que guarda el registro de los requisitos de prueba *(TestReq*) satisfechos. *Scoreboard* puede ser un vector de bits si se desea lograr cobertura de sentencias, un vector de enteros si deseamos conocer la frecuencia de ejecución de cada sentencia o, en general, una estructura de datos adecuada al criterio de obertura considerado.

El siguiente paso es la generación de la población inicial (conjunto inicial de los valores de entrada). A continuación (bucle de las líneas 4 a 14), comienza el proceso de generación de los casos de prueba, bastante legible sin mayor explicación. Como excepción, hay que resaltar que el cálculo del *fitness* se hace en función del número de nodos de *CDGPaths* alcanzados por el caso de prueba teniendo en cuenta el requisito de prueba actual (variable *r* en el algoritmo).

```
1.  Crear CDGPaths
2.  Crear e inicializar Scoreboard
3.  Generar CurPopulation
4.  while haya requisitos de prueba no marcados and se está en tiempo
5.      seleccionar un requisito de prueba r del conjunto TestReq
6.      while r no esté marcado and no se supere el máximo de intentos
7.          calcular el fitness de de CurPopulation usando CDGPaths
8.          ordenar CurPopulation según su fitness
9.          seleccionar los padres para crear la NewPopulation
10.         generar NewPopulation
11.         ejecutar el programa con cada elemento de NewPopulation
12.         actualizar Scoreboard y marcar aquellos requisitos que se hayan satisfecho
13.     endwhile
14. endwhile
15. final = conjunto de casos de prueba que satisfacen TestReq
16. devolver (TestReq, final)
```

**Figura 22. Algoritmo de generación de casos de prueba de**

#### 2.2.2 Algoritmos Genéticos (II)

Shiba et al. [23] proponen el algoritmo genético mostrado en la Figura 23. *Elite* es el conjunto de los mejores individuos de la población, y se va recalculando en cada iteración; $P_{mating}$ representa la población siguiente, que procede de cruces de *P*.

```
1.  Crear población inicial P con m candidatos
2.  Calcular fitness de P
3.  while no se cumpla la condición de parada
4.      Elite= n mejores individuos de P
5.      Aplicar Selection a los individuos de P para crear P_mating, con m-n individuos
6.      Cruzar P_mating
7.      Mutar P_mating
8.      P=Elite + P_mating
9.      Evaluar P
10.     if fitness(P) se ha estancado mutar masivamente P
11. endwhile
12. devolver el mejor caso
```

**Figura 23. Algoritmo de generación de casos de prueba de**

### 2.2.3 Algoritmo de las hormigas

El algoritmo de las hormigas representa los casos de prueba en un grafo como el siguiente:

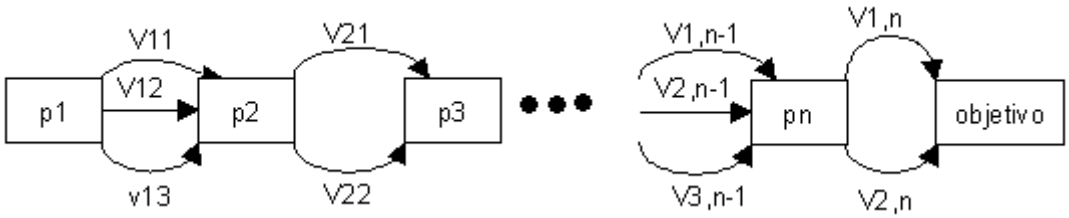



**Figura 24. Representación de los casos de prueba en forma de grafo**

Cada nodo $p_i$ representa un parámetro, y cada arco $V_{j,i}$ representa el j-ésimo valor interesante del parámetro i-ésimo. Cada camino desde el nodo inicial hasta el nodo objetivo se asocia con un caso de prueba. Así, un posible caso de prueba para el problema mostrado en la figura anterior vendría determinado por $\{V_{1,1}, V_{2,1}, ..., V_{1,n-1}, V_{2,n}\}$.

El grafo es recorrido por un conjunto de hormigas. Cuando una hormiga alcanza el nodo objetivo, se deposita en cada arco de los que ha visitado una cantidad de feromonas proporcional a la calidad del solución. Si una hormiga tiene que elegir entre diferentes arcos, se va por aquel que tenga más feromonas.

Como al principio todos los arcos no tienen feromonas, se calcula una heurística para cada arco, que no cambia durante el algoritmo, según la siguiente ecuación:

$$h_{i,j} = \frac{C_{i,\max} - C_{i,j} + 1}{C_{i,\max} - C_{i,\min} + 1}$$

En la ecuación, $C_{i,j}$ representa el número de casos de prueba que contienen el valor $V_{i,j}$. La idea de la heurística es que los parámetros que aparecen menos veces sean añadidos a los nuevos casos de prueba.

El algoritmo para la generación de casos es el siguiente:

```
1.  Calcular heurísticas de cada arco
2.  Inicializar feromonas
3.  while no se alcance la condición de parada
4.      for i=1 to |hormigas|
5.          generar un caso de prueba Sk
6.          evaluar Sk
7.          depositar feromonas
8.      endfor
9.      actualizar feromonas
10.     if la solución se ha estancado then inicializar feromonas
11. endwhile
12. devolver el mejor caso
```

**Figura 25. Algoritmo de las hormigas**

El paso indicado en la línea 5 (*generar un caso de prueba)* lo realiza cada hormiga según la heurística y feromonas de cada arco, de acuerdo con la siguiente función de probabilidad:

$P_{i,j} = \frac{\left[\tau_{i,j}\right]^{\alpha} \cdot \left[\eta_{i,j}\right]^{\beta}}{\sum_{h=1}^{vi} \left[\tau_{i,h}\right]^{\alpha} \cdot \left[\eta_{i,h}\right]^{\beta}}$, en donde $v_i$ es el número de posibles valores del parámetro $p_i$; $\eta_{i,j}$ y $\tau_{i,j}$ son, respectivamente, el valor de la heurística asignada al eje $i,j$ y la cantidad de feromona; $\alpha$ y $\beta$ son factores de peso asignados en función de la importancia que quiera darse a la heurística o a la feromona.

La actualización de las feromonas se realiza cuando cada hormiga ha generado un caso de prueba y se hace con la siguiente ecuación:

$$\tau_{i,j} = \rho \cdot \tau_{i,j} + \Delta\tau_{i,j} + \Delta\tau^{\times}_{i,j}$$

Además de estas ecuaciones, se utilizan otras para representar las tasas de persistencia y evaporación de las feromonas.

## 3. Estrategias deterministas

En estas estrategias, dos ejecuciones del mismo algoritmo producen siempre los mismos resultados con el mismo conjunto de entradas. Se describen a continuación algunas de esta categoría.

### 3.1. Each choice

En esta estrategia, se incluye cada valor de cada parámetro en al menos un caso de prueba: en cada iteración, se escoge un valor que no haya sido utilizado en casos de prueba anteriores. Se logra cobertura 1-wise.

### 3.2. Base choice

Se construye manualmente un caso de prueba base, que contiene los valores más importantes de cada parámetro. A partir de éste, se construyen nuevos casos de prueba variando los valores de un parámetro en cada iteración. Esencialmente, este algoritmo es exactamente igual al anterior, pero el caso de prueba base es elegido por el ingeniero de pruebas componiéndolo con los valores más interesantes. Se logra cobertura 1-wise.

### 3.3. Partly pair-wise

Se seleccionan los dos parámetros más significativos y se construyen casos de prueba con todas los posibles combinaciones de estos dos parámetros. Luego, la batería de casos de prueba se va ampliando añadiendo un valor para cada uno de los parámetros restantes.

### 3.4. All combinations

Se generan todas las combinaciones posibles de valores de prueba mediante el cálculo del producto cartesiano de todos los conjuntos de valores interesantes de cada parámetro.

### 3.5. Anti-random

Esta estrategia, debida a Malaiya [24], se basa en que cada caso de prueba debería seleccionarse de forma que tenga la mayor "distancia" con respecto a los demás. Los parámetros y los valores interesantes se codifican en un vector binario.

Los casos de prueba se van seleccionando en función de su distancia de Hamming o de su distancia cartesiana con respecto a los que se han generado con anterioridad.

Supongamos que, para el problema del triángulo, tenemos los valores {1,2,3}, {4,5} y {6,7} para los tres lados, que se codifican según la Tabla 9. Anti-

random empieza con la selección de un caso arbitrario. Sea el (1,4,6) este primer caso, que corresponde a la palabra 0000; si se utiliza la distancia Hamming, el siguiente caso será aquél que tenga el mayor número diferente de bits, con lo que se selecciona la palabra 1111, correspondiente al caso (3,5,7). El siguiente caso será aquel cuya suma de distancias Hamming a los dos casos seleccionados sea mayor.

| | Valor | Código |
|---|---|---|
| A (*) | 1 | 00 |
| | 2 | 01 |
| | 3 | 10 |
| | 3 | 11 |
| B | 4 | 0 |
| | 5 | 1 |
| C | 6 | 0 |
| | 7 | 1 |

**Tabla 9. Codificación en binario de los valores de prueba. (*) El valor 3 de A recibe dos códigos para utilizar todas las combinaciones de los dos bits**

El trabajo de Malaiya no especifica la condición de parada de Antirandom, y tampoco este algoritmo garantiza la satisfacción de ninguno de los criterios de cobertura tradicionales (pair-wise, etc.).

## 4. Estrategias de generación fuera de la clasificación anterior

De entre los criterios de cobertura de código enumerados en el Capítulo 3, es de destacar la dificultad de conseguir valores aceptables para, por ejemplo, mutación. Piénsese, por ejemplo, en la dificultad de encontrar valores que maten los mutantes de una condición tan simple como *if (a<10)* si esta instrucción se encuentra a una "distancia computacional" considerable de la entrada del programa. La generación de casos de prueba con algoritmos de minimización construye una función de minimización para cada condición, aplicando entonces diferentes heurísticas para conseguir la cobertura, en función de los datos recogidos durante la ejecución de cada caso.

A continuación se hace una revisión rápida de tres métodos basados en algoritmos de minimización: uno basado en Búsqueda Tabú y dos en algoritmos genéticos, si bien existen otras posibilidades, como el uso de *Simmulated Annealing* o enfoques híbridos como el propuesto por [25].

### 4.1. Búsqueda Tabú

Díaz et al. [26] proponen la utilización de un algoritmo basado en *Búsqueda Tabú* para lograr amplia cobertura de decisiones (Figura 26). La Búsqueda Tabú se basa en el algoritmo de los k-vecinos, junto al mantenimiento en memoria de una lista tabú que evita repetir la búsqueda dentro de un área del espacio de soluciones. Al algoritmo se le deben suministrar algunos parámetros, como la función objetivo (que mide el coste de la solución), la estrategia para seleccionar vecinos y la memoria del algoritmo.

```
Generar una solución aleatoria como solución actual
Calcular coste de la solución actual y almacenarlo como mejor coste
Añadir la solución actual como nueva solución
Añadir la nueva solución a la lista tabú
do
        Calcular los vecinos candidatos
        Calcular el coste de los candidatos
        Almacenar el mejor candidato como nueva solución
        Añadir la nueva solución a la lista tabú
        if coste de nueva solución < mejor coste
                Almacenar nueva solución como mejor solución
                Almacenar coste de la nueva solución como mejor coste
        end_if
        Almacenar nueva solución como solución actual
while no se alcance el criterio de parada
```

**Figura 26. Algoritmo de búsqueda tabú**

Puesto que el objetivo de estos autores es alcanzar la mayor cobertura posible, utilizan un grafo que representa el flujo de control del programa, en cuyos nodos se anota si el propio nodo ha sido alcanzado, cuántas veces lo ha sido y cuál es el mejor caso de prueba que lo ha alcanzado. Cuando no hay ramas inalcanzables, el máximo valor posible para la cobertura es el 100%, mientras que será desconocido en caso de que las haya. Por este motivo establecen como cri-

terio de parada o haber alcanzado todas las ramas, o que el algoritmo haya superado un número de iteraciones prefijado. Además, cada solución se caracteriza por su conjunto de valores de entrada. El coste de una solución, fundamental para que el algoritmo funcione eficazmente, se calcula considerando que el mejor caso de prueba es aquel que tiene más posibilidades de que sus vecinos permuten entre ramas o, lo que es lo mismo, aquel que alcanza el nodo con valores límite. Por ejemplo, si la condición es *x!=y*, la función de coste será *|x-y|* (véase la referencia para conocer el resto de detalles de cálculo de la función de coste).

Para calcular los vecinos candidatos, los autores se basan en que, si un caso de prueba cubre al padre de un nodo pero no a su hijo, entonces puede encontrarse un vecino que alcance al hijo utilizando el caso que cubre al padre a partir de la mejor solución. A partir de ésta generan *2n* "vecinos cercanos" y *2n* "vecinos lejanos" (donde *n* es el número de variables de entrada del programa). Los candidatos se comprueban frente a la lista tabú, rechazándose aquellos que ya existen. En la siguiente iteración se repite el proceso, con la diferencia de que el nodo objetivo puede haber cambiado si alguno de los candidatos alcanzó el entonces nodo objetivo.

### 4.2. Generación mediante algoritmos bacteriológicos.

Baudry et al. [27] parten de un conjunto inicial de casos de prueba. Su algoritmo se ejecuta en varias iteraciones, en cada una de las cuales se aplican mutaciones a los casos de prueba. El algoritmo memoriza los casos de prueba que pueden mejorar la calidad del *test suite*, la cual se mide mediante una función de *fitness*.

El algoritmo dispone de cinco funciones principales:

1) *Fitness* del *test suite*, que se mide utilizando algún criterio de cobertura.
2) *Fitness relativo* de un caso de prueba *tc*: se calcula en función del fitness del test suite, *TS:*

   $$relFitness(TS, tc) = fitness(TS \cup \{tc\}) - fitness(TS)$$

3) Memorización, que memoriza o no un caso de prueba en función de *fitness relativo* de dicho caso.

4) Mutación, que genera nuevos casos de prueba a partir de otros. Los casos de prueba origen se toman aleatoriamente del test suite, si bien la probabilidad de tomar un caso u otro depende del fitness relativo de cada caso.
5) Filtrado, que va eliminando los casos que no se utilizan con el fin de preservar la memoria.

## 5. Ejercicios

1) Partiendo del conjunto de casos de prueba (1, 1, 1), (2, 2, 2), (3, 3, 3) y (4,4,4) para el problema del triángulo, construya un conjunto de más calidad aplicando, manualmente, un algoritmo genético. ¿Cómo evoluciona la calidad del conjunto al generar cada nueva población?

# Capítulo 6. PRUEBAS DE CAJA NEGRA

Por lo general, las pruebas de caja negra se consideran pruebas de más alto nivel que las de caja blanca. De hecho, se utilizan técnicas de caja negra tanto para realizar pruebas unitarias, como para otros niveles de prueba, como las funcionales, de integración, de sistema o de aceptación [28].

## 1. Pruebas de componentes

En terminología UML, un componente es "una parte física reemplazable de un sistema que empaqueta su implementación, y que es conforme a un conjunto de interfaces a las que proporciona su realización" [29]. Esto significa, sencillamente, que el componente ofrece una serie de servicios (métodos) a sus usuarios a través de una interfaz; las operaciones incluidas en la interfaz se implementan dentro de la lógica del componente.

Un componente, por tanto, representa una unidad de construcción reutilizable, que puede ensamblarse para formar aplicaciones.

Habitualmente, el componente se percibe como una pieza de la que se conoce la interfaz que ofrece y las salidas que aporta, por lo que pueden considerarse paradigmáticos para considerarlos cajas negras.

### 1.1. Uso de BIT wrappers

Edwards [30] propone la utilización de *BIT wrappers* (*Built-In-Test wrappers,* o adaptadores construidos durante las pruebas) para la prueba de componentes. Esencialmente, un BIT wrapper es una clase que ofrece la misma interfaz que el componente para el que se ha construido. En la fase de pruebas del componente, se ejecutan las operaciones de éste a través del BIT wrapper, que posee funcionalidades como la comprobación de las precondiciones de la operación antes de llamar a la operación real, y la comprobación de las postcondiciones tras su ejecución. Además, el BIT wrapper puede mantenerse para que capture las llamadas que los clientes hacen a las operaciones del componente (Figura 27).

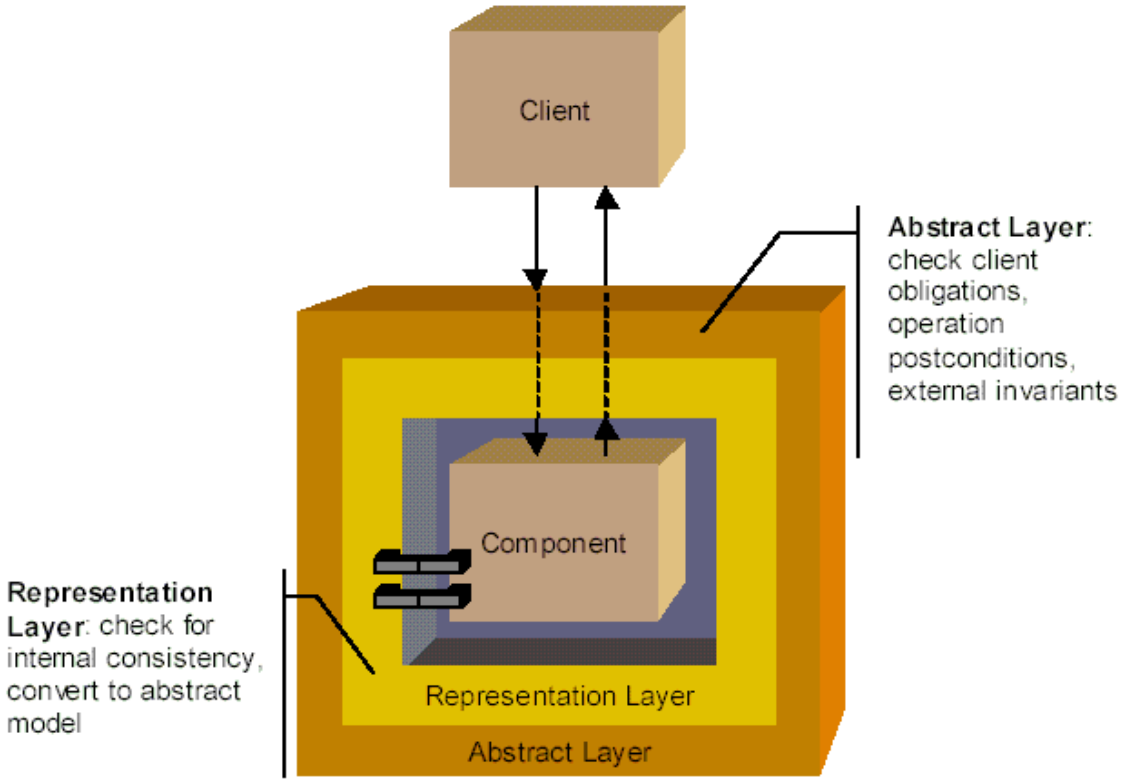



**Figura 27. Ubicación del BIT wrapper alrededor del componente. Figura tomada de [30]**

Algunas características de los BIT wrappers son las siguientes:

- Son transparentes al componente y a los posibles clientes.
- La adición o supresión de BIT wrappers sólo requiere la modificación de las declaraciones en el código de los clientes.
- Se añaden nuevas capacidades de comprobación de restricciones a las que ya realiza el propio componente.
- La violación de las restricciones se detectan en el momento en que ocurren, de manera que se evita su propagación a otros componentes.
- Si se posee una especificación formal del componente, el BIT wrapper se puede generar automáticamente.

Para la fase de pruebas del componente,  propone el proceso que se muestra esquemáticamente en la Figura 28: el conjunto de casos de prueba contenido en el *Test suite* (que se ha podido generar automáticamente) se pasa a un *ejecutor de pruebas* (*Test driver*), que prueba el componente a través del BIT wrapper. El resultado es, por un lado, los resultados obtenidos de la ejecución de cada caso de prueba y, por otro, un informe con los errores encontrados.

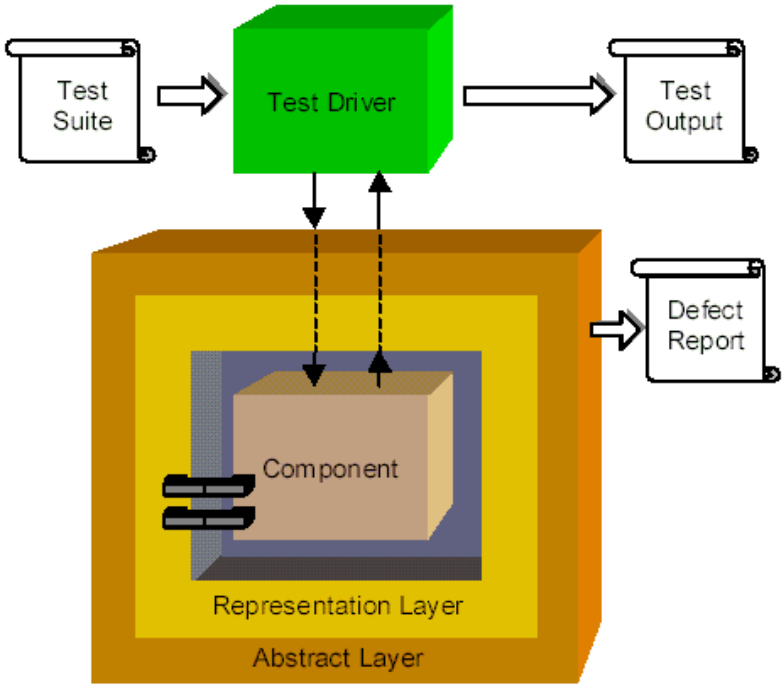



**Figura 28. Proceso de pruebas con BIT wappers**

Tanto el BIT wrapper como el *Test driver* pueden generarse automáticamente: para el primero es preciso disponer de una descripción formal del componente (el autor utiliza el lenguaje *Resolve,* aunque podría emplearse cualquier otro); para la generación del segundo podrían usarse generadores aleatorios, de valores límite, etc.

## 1.2. Mutación de interfaces

Ghosh y Matur [31] proponen aplicar ciertos operadores de mutación a las interfaces de componentes para la realización de pruebas, así como ciertos criterios de cobertura para validar las pruebas realizadas.

Los autores proponen los siguientes operadores de mutación para CORBA-IDL son los siguientes:

- Reemplazar *inout* por *out*.
- Reemplazar *out* por *inout*.
- Intercambiar parámetros de tipos compatibles.
- "Jugar" con un parámetro (operador *twiddle):* por ejemplo, sumarle uno si es entero, añadirle un carácter si es una cadena, etc.
- Poner a cero los valores numéricos.
- Sustituir por *null*.

Los criterios de cobertura que proponen son los siguientes:

- Cobertura de llamadas a métodos de la interfaz del componente.
- Cobertura de excepciones lanzadas por la interfaz del componente.
- Cobertura de llamadas a métodos y de excepciones lanzadas.

La mutación se consigue sustituyendo la interfaz por una nueva versión, como muestra la Figura 29.

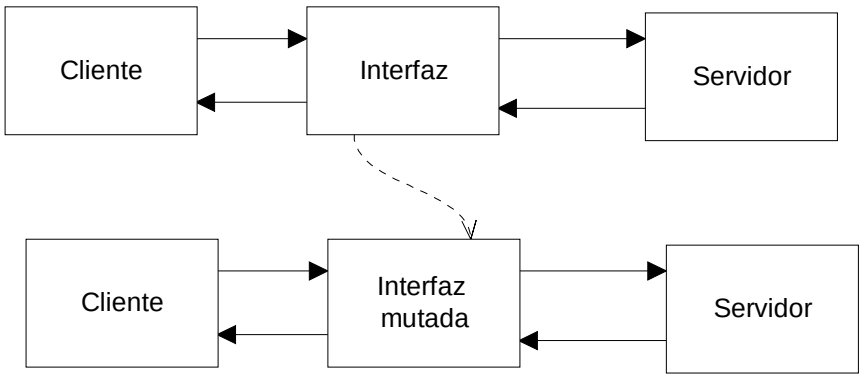



**Figura 29. Mutación de interfaces**

## 2. Pruebas de servicios web

Mediante los servicios web, un cliente puede ejecutar un método en un equipo remoto, transportando la llamada (hacia el servidor) y el resultado (desde el servidor al cliente) mediante protocolo *http* (normalmente). La idea es servir la misma funcionalidad que permiten otros sistemas de invocación remota de métodos, como RMI, pero de un modo más portable.

La portabilidad se consigue gracias a que todo el intercambio de información entre cliente y servidor se realiza en SOAP (*Simple Object Access Protocol*), un protocolo de mensajería basado en XML: así, la llamada a la operación consiste realmente en la transmisión de un mensaje SOAP, el resultado devuelto también, etc. De este modo, el cliente puede estar construido en Java y el servidor en .NET, pero ambos conseguirán comunicarse gracias a la estructura de los mensajes que intercambian.

Los servidores ofrecen una descripción de sus servicios web en WSDL (*Web Services Description Language*), que es una representación en XML de la interfaz del servicio ofrecido. Así, un cliente puede conocer los métodos ofrecidos por el servidor, sus parámetros con sus tipos, etc., simplemente consultando el correspondiente documento WSDL, que se encuentra publicado en alguna URL.

## 2.1. WSDL

Supongamos que un sistema de gestión bancario ofrece, para validar las operaciones realizadas con tarjeta de crédito, el siguiente método remoto:

*public boolean validar(String numeroDeTarjeta, double importe)*

Si este método es accesible como un servicio web, debe estar descrito en WSDL, por ejemplo, como se muestra en la Figura 30.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <definitions name="VisaWS" targetNamespace="http://visa/dominio/Visa.wsdl"
      xmlns="http://schemas.xmlsoap.org/wsdl/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns:tns="http://visa/dominio/Visa.wsdl"
      xmlns:ns1="http://visa.dominio/IVisaWS.xsd">
  - <types>
      <schema targetNamespace="http://visa.dominio/IVisaWS.xsd"
          xmlns="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
          ENC="http://schemas.xmlsoap.org/soap/encoding/" />
    </types>
  - <message name="validar0Request">
      <part name="numeroDeTarjeta" type="xsd:string" />
      <part name="importe" type="xsd:double" />
    </message>
  - <message name="validar0Response">
      <part name="return" type="xsd:boolean" />
    </message>
  - <portType name="VisaPortType">
    - <operation name="validar">
        <input name="validar0Request" message="tns:validar0Request" />
        <output name="validar0Response" message="tns:validar0Response" />
      </operation>
    </portType>
  - <binding name="VisaBinding" type="tns:VisaPortType">
      <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
    - <operation name="validar">
        <soap:operation soapAction="" style="rpc" />
      - <input name="validar0Request">
          <soap:body use="encoded" namespace="VisaWS"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
      - <output name="validar0Response">
          <soap:body use="encoded" namespace="VisaWS"
              encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </output>
      </operation>
    </binding>
  - <service name="VisaWS">
    - <port name="VisaPort" binding="tns:VisaBinding">
        <soap:address location="" />
      </port>
    </service>
</definitions>
```

**Figura 30. Descripción en WSDL**

De la figura anterior merece la pena destacar algunos elementos, que se enumeran y explican en la Figura 31.

| | |
|---|---|
| - <message name="**validar0Request**"><br><part name="**numeroDeTarjeta**"<br>type="**xsd:string**" /><br><part name="**importe**" type="**xsd:double**" /><br></message> | Nombre del método accesible de forma remota, nombres y tipos de los parámetros. El sufijo *Request* denota el formato en que debe enviarse la solicitud al servidor. Cuando el cliente invoca el servicio, envía un mensaje *validar0Request.* |
| - <message name="**validar0Response**"><br><part name="**return**" type="**xsd:boolean**" /><br></message> | Tipo del resultado devuelto por el método. El sufijo *Response* se refiere precisamente a que es el tipo devuelto lo que se está representando. |
| - <portType name="**VisaPortType**"><br>- <operation name="**validar**"><br><input name="**validar0Request**"<br>message="**tns:validar0Request**" /><br><output name="**validar0Response**"<br>message="**tns:validar0Response**" /><br></operation><br></portType> | Operaciones que conforman la interfaz del servicio *validar*, que se corresponden con los dos *messages* anteriores. |

**Figura 31. Significado de algunos elementos del WSDL mostrado en la figura anterior**

Los entornos de desarrollo recientes incluyen los *add-ins* necesarios para generar la especificación WSDL de una clase.

## 2.2. Escritura de un cliente que acceda a un servicio web

El cliente que utiliza el servicio web necesita una clase que actúe como *proxy* entre él mismo y el servicio web ofertado por el servidor. Cuando el *proxy* recibe del cliente una solicitud de llamada al servicio web, el *proxy* la traduce a un mensaje SOAP, que envía al servidor; éste, entonces, lo ejecuta, y devuelve un mensaje SOAP al *proxy*, que traduce el mensaje a objetos Java, .NET, etc. y entrega el resultado al cliente que efectuó la petición.

La siguiente figura muestra la relación entre el *proxy*, el servicio web y la clase que lo implementa: el servicio web (en el centro) ofrece a los clientes acceso a la operación *validar(String, Double);* la operación se encuentra realmente implementada en la clase situada abajo (*Visa*), y podría incluir llamadas a otros métodos de otras clases, acceso a una base de datos, acceso a otros servicios web, etc. El elemento de la izquierda (*VisaWSStub*) es la clase que actúa de *proxy* entre los clientes y el servicio web. Nótese que esta clase incluye, además de otras, la operación *validar(String, Double)*. El *proxy* mostrado se ha obtenido de forma automática con un entorno de desarrollo, por lo que sus miembros pueden variar de unos casos a otros. El atributo *_endpoint* representa la URL en la que se encuentra publicado el servicio web.

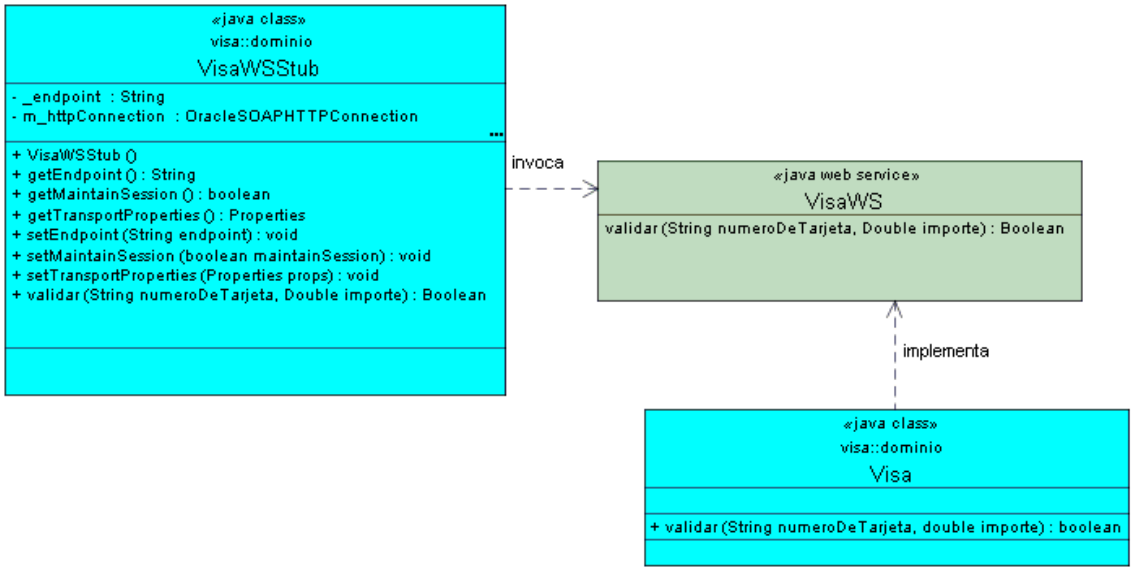



**Figura 32. El proxy, el servicio web y la clase que lo implementa**

La aplicación cliente hace entonces uso del proxy para acceder al servicio web, por ejemplo con un trozo de código como el que sigue:

```
protected void validarDisponibilidadDeCredito(double importe) throws Exception
{
	// Se intancia el proxy
	VisaWSStub stub = new VisaWSStub();
	// Se le dice al proxy dónde puede encontrar el servicio web
	stub.setEndpoint("http://161.67.27.108:8988/soap/servlet/soaprouter");
	// Se llama al método ofertado por el proxy
	if (!stub.validar("", new Double(5.0)).booleanValue())
		throw new Exception("Operación no admitida");
}
```

**Figura 33. Acceso desde el cliente al servicio web a través del *proxy***

## 2.3. Pruebas de servicios web mediante perturbación de datos

Offutt y Xu [32] proponen la utilización de "perturbación de datos" para realizar las pruebas de servicios web. Los mensajes SOAP se modifican y se utilizan como casos de prueba. La técnica de la perturbación de datos utiliza tres métodos para probar los servicios web:

(1) Perturbación de los valores de los datos.

(2) Perturbación de la comunicación RPC.

(3) Perturbación de la comunicación de datos.

En **perturbación de valores**, se sustituyen los valores de los datos en los mensajes SOAP por otros valores basándose en la técnica de valores límite (la Tabla 10 muestra los valores límite de cinco de los diecinueve tipos de datos de SOAP).

| Tipo de dato | Valores límite |
|---|---|
| String | Cadena de máxima longitud, de mínima longitud, toda en mayúsculas, toda en minúsculas |
| Decimal, float, double | Máximo valor, mínimo valor, cero |
| Boolean | true, false |

**Tabla 10. Valores límite para algunos tipos de datos**

En **perturbación de la comunicación RPC**, la prueba se enfoca en el uso que se hace los datos que se envían en el mensaje, distinguiéndose dos usos: usos normales y usos de SQL. Los usos de SQL son aquellos en los que los datos se utilizan para construir consultas SQL. Por ejemplo, si desde un formulario se envían un *login* y una *password* para realizar una consulta y validar a un usuario, se pueden alterar los valores pasados con el fin de encontrar comportamientos inesperados en el servicio web (Figura 34).

```
...
<soapenv:Body>
  <adminLogin soapenv:encodingStyle=...>
    <arg0 xsi:type="xsd:string">turing</arg0>
    <arg1 xsi:type="xsd:string">enigma</arg1>
  </adminLogin>
</soapenv:Body>
...
```

```
...
<soapenv:Body>
  <adminLogin soapenv:encodingStyle=...>
    <arg0 xsi:type="xsd:string">turing'  OR '1'='1</arg0>
    <arg1 xsi:type="xsd:string">enigma'  OR '1'='1</arg1>
  </adminLogin>
</soapenv:Body>
...
```

**Figura 34. Un mensaje SOAP original (izquierda) y modificado**

En los usos normales, los valores se alteran aplicando algunos operadores de mutación. Por ejemplo, si se pasa el valor numérico *n*, puede sustituirse por *1/n, n*n*, etc.

En **perturbación de la comunicación de datos**, se pretende probar las relaciones y la integridad de los datos, ya que muchos de los datos que se envían a los servicios web incluyen, implícitamente, información relativa a la estructura de la base de datos. Por ejemplo, el carrito de una tienda electrónica puede almacenar una lista de los códigos de los productos que compra un cliente sin duplicar ninguno; puede añadirse a la lista de productos un elemento duplicado para observar el comportamiento del sistema.

## 3. Pruebas de integración con máquinas de estado en orientación a objetos

Como ya se ha comentado, 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.

Gallagher et al. describen una metodología para realizar pruebas de integración mediante el modelado del sistema como una máquina de estados [11]. Las máquinas de estado se utilizan con mucha frecuencia para describir el comportamiento de clases individuales. Una máquina de estados para una clase *C* puede describirse mediante una tupla *(V, F, P, S, T)*, donde:

- *V* es el conjunto de los campos de *C*.
- *F* es el conjunto de las operaciones de *C*.
- *P* es el conjunto de parámetros de las operaciones.
- *S* es el conjunto finito de estados. Cada estado se describe como un predicado sobre los valores de los campos de *C*.
- *T* es un conjunto finito de transiciones entre estados. Cada transición *t* se etiqueta una tupla como la siguiente: *t=(estadoOrigen, estadoDestino, operación, guarda, acción)*. En ésta:
    - *estadoOrigen* y *estadoDestino* son elementos de *S*.
    - *operación* es la operación que dispara la transición, si la *guarda* se evalúa como cierta.
    - *guarda* es un predicado sobre los campos de *C* y los parámetros de las operaciones de *F*.
    - *action* es una serie de cómputos que se ejecutan, como consecuencia de que se dispara la transición.

Los autores proponen el concepto de *máquina de estados combinada* (para describir sistemas compuestos por varias clases) mediante la extensión de la definición descrita arriba: a la tupla *(V, F, P, S, T)* se le añade el conjunto de clases *C,* obteniendo *(C, V, F, P, S, T)*. Los elementos *V, F, P, S* y *T* se redefinen, de manera que alberguen la unión de todos los campos, operaciones, parámetros, estados y transiciones de todas las clases que intervienen en el sistema que se pretende someter a pruebas de integración.

El primer paso de la metodología es la identificación del componente cuyas interacciones con el resto del sistema van a ser probadas. A partir de este componente, se identifican las transiciones relevantes de ese componente, que pueden ser de entrada o de salida. Para calcular el conjunto de transiciones relevan-

tes, se parte de las transiciones que salen del componente, y se ejecuta a continuación un proceso iterativo por el que se van añadiendo las transiciones llamadas por las transiciones que ya están en el conjunto.

Después se construye un grafo de flujo a partir de la máquina de estados combinada, en el cual se consideran:

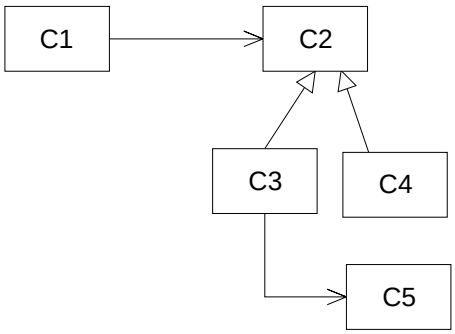
- El conjunto de transiciones relevantes.
- Los estados origen o destino de las transiciones que se han identificado como relevantes.
- El conjunto de guardas de transiciones relevantes.

Entonces, se está casi en condiciones de construir los casos de prueba: el grafo de flujo incluye transiciones y estados, con los que pueden definirse criterios de cobertura, a partir de los cuales se pueden generar los casos de prueba.

## 4. Ejercicios

1) Explique las similitudes entre las pruebas de componentes mediante mutación de interfaces y las pruebas de servicios web mediante perturbación de datos.

2) Suponga que tenemos un sistema orientado a objetos como el de la figura siguiente, y que queremos probar el correcto funcionamiento de este sistema ejecutando los servicios que ofrece la clase *C1*. Razone si tendría utilidad inyectar fallos en las clases interiores (*C2* a *C5)* para realizar las pruebas.



# Capítulo 7. PRUEBAS DE ARTEFACTOS DIVERSOS

Como se mencionó en el primer capítulo, se aplican diferentes técnicas de prueba a cada tipo de producto software. En este capítulo se mencionan algunas técnicas de prueba para artefactos distintos del código fuente.

## 1. *Testing* basado en requisitos

Diversos estudios han mostrado que el coste de corregir un error crece más que linealmente conforme se avanza en las diversas fases del ciclo de vida. Así, si corregir un error en la fase de requisitos cuesta 1 unidad monetaria, corregir el mismo error en la codificación puede ser 10 veces más costoso, y hasta 100 veces más cuando el producto ya se encuentra en explotación.

Un estudio citado por Mogyorodi [33] indica que el origen del 56% de los errores de los proyectos software se encuentra en la fase de requisitos. De este 56%, aproximadamente la mitad se debe a requisitos mal redactados, ambiguos, poco claros o directamente incorrectos, mientras que la otra mitad se debe a requisitos que, simplemente, no se encontraban en la especificación.

Con el fin de arrastrar el menor número posible de errores hacia fases posteriores del desarrollo, deben verificarse los cuatro principales atributos de calidad de los requisitos:

- Compleción: los requisitos deben especificar completa y claramente el problema.
- Corrección: no debe haber ambigüedad en ningún requisito. Cualquier requisito no debe tener más que una interpretación.
- Consistencia: no debe haber requisitos contradictorios.
- Factibilidad: los requisitos deben ser factibles e implementables.

La metodología RBT [33] (*Requirements-Based Testing*) pretende, a partir de una revisión inicial de los requisitos, derivar el número mínimo de casos de prueba necesario para cubrir el 100% de los requisitos funcionales. RBT consta de los siguientes 12 pasos:

1) **Validación de los requisitos con respecto a sus objetivos**. A partir del plan del proyecto, se comparan sus objetivos (en donde se describe *por qué* se desarrolla el proyecto) con los requisitos (en los que se especifican *qué* funcionalidades va a servir el sistema que se va a desarrollar). Si ese *qué* no se corresponde con el *por qué*, los requisitos no cumplirán los objetivos del proyecto y éste fracasará. Pueden detectase, además, requisitos (*qués)* que no encuentren sus correspondientes *por qués*: estos requisitos no forman parte del alcance del proyecto y pueden ser eliminados.

2) **Descripción de los requisitos mediante casos de uso.** Al describir los requisitos mediante casos de uso, y éstos mediante escenarios, se puede comprobar si los requisitos se encuentran satisfechos por los diferentes escenarios de los casos de uso. En caso de que no, se habrá encontrado que la especificación de requisitos está incompleta.

3) **Revisión de la ambigüedad.** Se revisa la redacción de los requisitos con objeto de eliminar palabras, frases o construcciones ambiguas y conseguir que el requisito pueda ser probado (Figura 35).

**Descripción inicial de un requisito:**
Cuando el cajero automático sea forzado, éste enviará una alerta al departamento de TIC. En el caso de que el cajero sea abierto sin llave y sin introducir el código de seguridad, el cajero alertará inmediatamente al departamento TIC con objeto de que se tome la acción correspondiente.

**El requisito no puede ser probado porque contiene ambigüedades, algunas de las cuales son:**

1) Tipo de alerta que debe enviar el cajero al departamento TIC.
2) Qué significa que el cajero es "forzado".
3) ¿Es lo mismo que el cajero sea forzado y que es abierto sin llave y sin introducir el código de seguridad?
4) ¿Qué ocurre si se utiliza la llave y se introduce un código incorrecto, o al revés?
5) ¿Cuál es la acción "correspondiente"?

**Figura 35. Revisión de las ambigüedades en un requisito**

4) **Revisión de los requisitos por parte de expertos en el dominio,** de modo que se revise la corrección y compleción de los requisitos.

5) **Creación de una tabla de decisión.** Los requisitos se traducen a tablas de decisión, que permiten encontrar y resolver problemas de sinonimia, aclarar precedencias entre requisitos, explicitar información que permanecía más o menos oculta, y comentar el proceso de pruebas de integración.

Supongamos un sistema bancario en el que hay una función de retirada de efectivo en la que se hace una comprobación del saldo de la cuenta y del crédito concedido a dicha cuenta. La función conoce el *importe* que se desea retirar, el *saldo* de la cuenta y el *crédito* concedido (el *crédito* es un número negativo, que representa el saldo negativo que se autoriza alcanzar a este cliente). Las posibles acciones son las siguientes:

- Autorizar la retirada del efectivo y enviar una carta.
- Autorizar la retirada del efectivo.
- Suspender la cuenta y enviar una carta.

La función tiene los siguientes requisitos:

- Si *saldo-importe>=0,* se autoriza la retirada de efectivo.
- Si *saldo-importe<0 ^ saldo-importe>=crédito,* se autoriza la retirada y se envía una carta.
- Si *saldo-importe<0 ^ saldo-importe<crédito,* se suspende la cuenta y se envía una carta.

Del enunciado se extraen 3 efectos:

- E1: autorizar la retirada de efectivo.
- E2: suspender la cuenta.
- E3: enviar la carta.

Estos efectos se consiguen a partir de las siguientes causas:

- C1: *saldo-importe>=0*.
- C2: *saldo-importe>=crédito*.

A continuación, se construye la tabla de decisión:

| | **Reglas** | | | |
|---|---|---|---|---|
| | 1 | 2 | 3 | 4 |
| C1: *saldo-importe>=0* | 0 | 0 | 1 | 1 |
| C2: *saldo-importe>=crédito* | 0 | 1 | 0 | 1 |
| E1: autorizar retirada | 0 | 1 | - | 1 |
| E2: suspender la cuenta | 1 | 0 | - | 0 |
| E3: enviar la carta | 1 | 1 | - | 0 |

En la tabla de decisión, la regla 3 es infactible pues se corresponde con causas mutuamente excluyentes (C1 y C2). Por las propiedades de reducción de tablas de decisión, podría reducirse el número de reglas. Para la función descrita tendríamos que escribir tres casos de prueba (uno por

**Figura 36. Construcción de la tabla de decisión**

6) **Revisión de la consistencia de la tabla de decisión,** lo que permitirá obtener los casos de prueba que cubran el 100% de los requisitos funcionales.

7) **Revisión de los casos de prueba por parte de los autores de los requisitos,** de modo que si se encuentra algún problema con algún caso de prueba, los requisitos asociados con el caso de prueba podrán ser corregidos y los casos de prueba rediseñados.

8) **Validación de los casos de prueba con los expertos en el dominio**, con objeto de encontrar posibles problemas en los requisitos de los cuales proceden.

9) **Revisión de los casos de prueba por parte de los desarrolladores,** de modo que éstos conozcan mejor los requisitos del sistema que tienen que construir y puedan orientar la construcción hacia la superación de estos casos de prueba.

10) **Utilización de casos de prueba en las revisiones del diseño.** Con esta tarea, se puede comprobar si el diseño que se está realizando satisfará los casos de prueba y, por tanto, los requisitos. Igualmente, podría llegarse a la conclusión de que alguno de los requisitos no sea factible.

11) **Utilización de casos de prueba en las revisiones del código.** Puesto que cada fragmento de código sirve un fragmento de los requisitos, los casos de prueba pueden utilizarse para comprobar que, efectivamente, el código se comporta como se esperaba.

12) **Verificación del código contra los casos de prueba.** En este último paso, se construyen casos de prueba ejecutables a partir de los casos de prueba de alto nivel. Cuando todos los casos ejecutables se ejecuten satisfactoriamente contra el código, puede decirse que se ha verificado el 100% de la funcionalidad.

## 2. Secuencias de métodos

En el contexto de las pruebas de sistemas orientados a objetos, Kirani y Tsai [34] consideran la *secuencia* como un concepto fundamental para la prueba de clases. Una secuencia representa el orden correcto en el que los métodos públicos de una clase pueden ser invocados. De acuerdo con estos autores, una de las aplicaciones de las secuencias es la prueba de clases a partir de su especificación como una máquina de estados.

Así, puede utilizarse la especificación del comportamiento de una cierta clase en forma de máquina de estados para generar casos de prueba, ejecutarlos y medir nuevos criterios de cobertura. En el ejemplo de la Figura 37, se conseguiría una cobertura completa de estados con la secuencia *m1.m2.m3.m4*, mientras que deberían ejecutarse otras secuencias para lograr cobertura de transiciones o cobertura de caminos. A partir de aquí surgen con facilidad otros criterios de cobertura, como la ejecución de cada camino cero veces, una vez y más de una vez, cobertura de guardas, de acciones, etc.

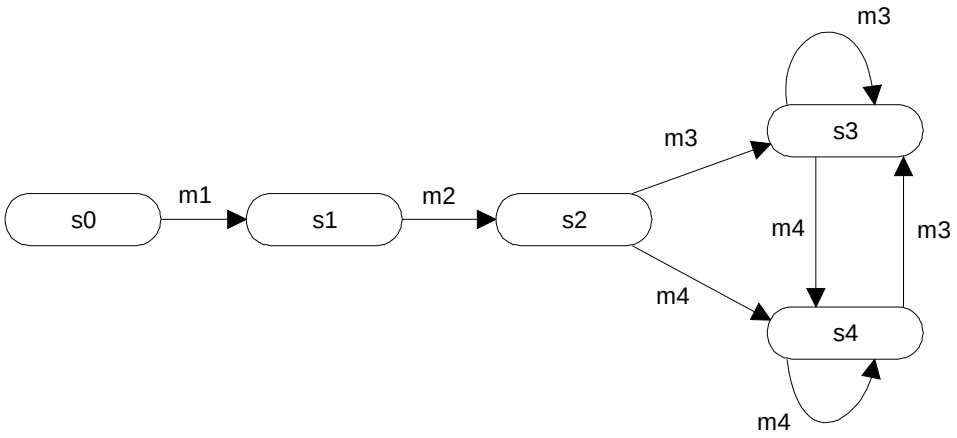



**Figura 37. Máquina de estados para una cierta clase**

# 3. Especificaciones formales o semiformales

## 3.1. Propuesta de Tse y Xu

Tse y Xu [35] utilizan la especificación formal de una clase para obtener el espacio de estados de la clase y generar casos de prueba.

De manera general, una clase puede especificarse utilizando combinadamente dos formas de representación: una *capa funcional*, en la que se representan los valores abstractos de los objetos de la clase, y una *capa de restricciones*, en la que se anotan la clase y sus operaciones con precondiciones, postcondiciones e invariantes.

La Figura 38 muestra las dos capas de una clase *Account*, que podría representar una cuenta bancaria, con un determinado lenguaje de especificación (aunque lo cierto es que existen muchos más para representar lo mismo, como se verá más adelante en, por ejemplo, la Figura 40).

Capa funcional de una clase

```
module ACCOUNT is
    including MONEY .
    sort AccountSort .

    var X : AccountSort .
    var M : Money .
    var N : Money .

    let creditLimit = 1000 .

    op empty : -> AccountSort .
    op creditS : AccountSort Money -> AccountSort .

    op balance : AccountSort -> Money .
    eq balance(empty) = 0 .
    eq balance(creditS(X, M)) = M + balance(X) .

    op charge : AccountSort Money -> Money .
    eq charge(empty, N) = 0 .
    ceq charge(creditS(X, M), N) = N * 0.05
        if N < 200 and balance(creditS(X, M)) < 0 .
    ceq charge(creditS(X, M), N) = 10
        if N >= 200 and balance(creditS(X, M)) < 0 .
    ceq charge(creditS(X, M), N) = 0
        if balance(creditS(X, M)) >= 0 .

    op debitS : AccountSort Money -> AccountSort .
    eq debitS(empty, N) = creditS(empty, - N) .
    eq debitS(creditS(X, M), N) =
        creditS(X, (M - N - charge(creditS(X, M), N))) .
endmodule
```

Capa de restricciones de la misma clase

```
class Account is
    based on sort AccountSort .

    invariant {balance(self) >= - creditLimit} .

    constructor Account()
    ensure {balance(result) == balance(empty)} .

    method credit(M : Money)
    require {M > 0} .
    ensure {balance(post-self) ==
        balance(creditS(pre-self, M))} .

    method debit(M : Money)
    require {M > 0 and balance(debitS(pre-self, M)) >= -
        creditLimit} .
    ensure {balance(post-self) ==
        balance(debitS(pre-self, M))} .

    method findBalance() : Money
    ensure {result == balance(pre-self)
        and balance(post-self) == balance(pre-self)} .
endclass
```

**Figura 38. Capas funcional y de restricciones de una clase**

La capa funcional incluye la descripción de los tipos importados (*including MONEY)*, el nombre del tipo (*sort AccountSort*), las posibles variables que se utilicen a continuación (*var X, M, N)* y un conjunto de cláusulas que especifican los resultados de las operaciones:

- *op* representa la signatura de la operación. Por ejemplo: *op charge : AccountSort Money -> Money* indica que la operación *charge* toma un parámetro de tipo *AccountSort* y otro de tipo *Money* y que devuelve un *Money*.
- *eq* representa propiedades de las operaciones descritas mediante ecuaciones. Por ejemplo, *eq balance(empty) = 0* indica que el resultado de ejecutar la operación *balance* (en inglés, "saldo") sobre una cuenta creada con la operación *empty* (en inglés, "vacía"), es cero; *eq debitS(empty, N) = creditS(empty, - N)* indica que sacar *N* euros de una

cuenta vacía (cuyo saldo es cero) es lo mismo que sacar los mismos *N* euros a crédito.

- *ceq* representa propiedades de las operaciones descritas mediante ecuaciones condicionales. Por ejemplo, *ceq charge(creditS(X, M), N) = N * 0.05 if N < 200 and balance(creditS(X, M)) < 0* indica que sacar *N* euros de una cuenta *X* de la que se han sacado a crédito *M* euros supone el cobro de un *5%* de comisión sobre el importe sacado, si éste es menor que *200* euros y el saldo de la cuenta es negativo.

La capa de restricciones del caso de la figura incluye, por ejemplo, una invariante que indica que el saldo de la cuenta debe ser siempre superior al crédito concedido (*invariant {balance(self) >= - creditLimit}*); una postcondición sobre el constructor *Account* que denota que el saldo de una cuenta recién creada es cero (*constructor Account() ensure {balance(result) == balance(empty)}*), y una precondición para la operación *credit(M : Money)* que expresa que el importe que se saca a crédito debe ser positivo (*require { M>0 }).*

A partir de estas consideraciones, los autores realizan una partición del espacio de estados de la clase, obteniendo estados *abstractos* por cada término booleano de la capa funcional. A partir, por ejemplo, de la propiedad *eq balance(empty)=0* podrían obtenerse los estados en que la cuenta tiene saldo negativo, saldo cero y saldo positivo. Combinando además estos estados con la invariante de la capa de objeto (*invariant {balance(self) >= - creditLimit}*), se podrían obtener los siguientes cinco subestados:

- *balance(self) < -creditLimit* ← inalcanzable debido a la invariante
- *balance(self) = -creditLimit*
- *0> balance(self) > -creditLimit*
- *balance(self) = 0*
- *balance(self) > 0*

Puesto que el valor de *creditLimit* es –1000, los estados concretos son los siguientes (nótese la adición de un estado inicial adicional, en el que el objeto se encuentra antes de ser creado; igualmente podrían crearse estados finales si la clase tuviera destructores):

$S_0$= { no creado }; $S_1$= { b==1000 }; $S_2$= { -1000<b<0}; $S_3$= { b==0}; $S_4$={ b>0}

Obtenidos los estados, se genera el conjunto de transiciones, formadas por una llamada a un método y una posible condición de guarda, obtenida de las precondiciones del método. De este modo, del anterior conjunto de estados y del ejemplo de la Figura 38 se obtiene la máquina de estados mostrada en la Figura 39.

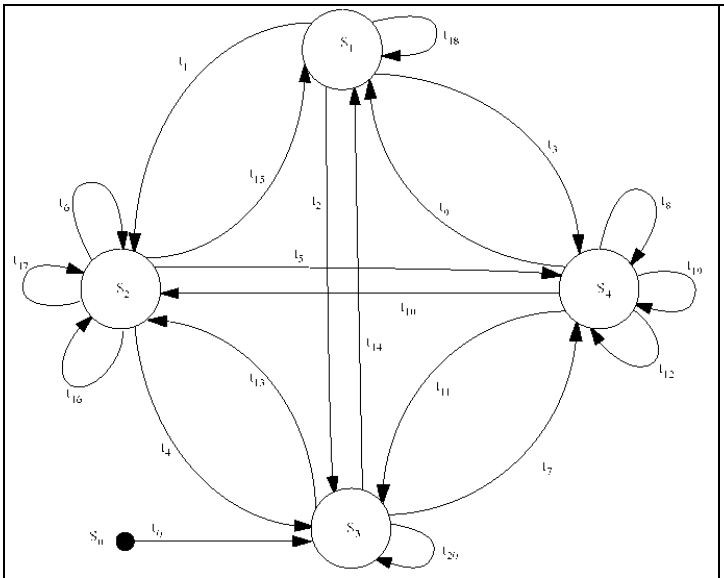



**Figura 39. Máquina de estados obtenida, tomada de**

La máquina de estados se procesa para generar secuencias de métodos que logren diferentes criterios de cobertura, como cobertura de estados, transiciones y caminos.

## 3.2. Método ASTOOT

Doong y Frankl [36] utilizan especificaciones algebraicas de las clases para la generación y ejecución de casos de prueba en su método ASTOOT (*A Set of Tools for Object-Oriented Testing*). Una especificación algebraica consta de una parte sintáctica y otra semántica:

- La sintáctica incluye los nombres y signatura de las operaciones.
- La semántica incluye una lista de axiomas que describen la relación entre funciones, utilizándose en muchos casos axiomas de reescritura para esta descripción. Así, dos secuencias de operaciones $S_1$ y $S_2$ son equivalentes si se pueden usar los axiomas como reglas de reescritura para transformar $S_1$ en $S_2$.

La siguiente figura, tomada de , muestra la especificación algebraica de una cola de prioridad. De acuerdo con esta figura, la secuencia *create.add(5).delete(3)* es equivalente a la secuencia *create.add(5)*, pues puede aplicarse dos veces el sexto axioma.

```
type Priority_Queue
syntax
   create: - > Priority_Queue;
   add: Priority_Queue x Integer
        - > Priority_Queue;
   delete: Priority_Queue - > Priority_Queue;
   empty: Priority_Queue - > Boolean;
   largest: Priority_Queue - > Integer;
   eqn: Priority_Queue x Priority_Queue
        - > Boolean;
declare
   A, B: Priority_Queue;
   x, y: Integer;
semantics
   1: empty(create) - > true;
   2: empty(add(A,x)) - > false;
   3: largest(create) - > - ∞;
   4: largest(add(A,x)) - >
        if x > largest(A) then x
        else largest(A);
   5: delete(create) - > create;
   6: delete(add(A,x)) - >
        if x > largest(A) then A
        else add(delete(A),x);
   7: eqn(A,B) - >
        if empty(A) and empty(B) then true
        else if (empty(A) and not empty(B)) or
                (not empty(A) and empty(B))
        then false
        else if largest(A) = largest (B)
        then eqn(delete(A),delete(B))
        else false
end
```

**Figura 40. Especificación algebraica de una cola de prioridad**

Para estos autores, dos objetos $O_1$ y $O_2$ de clase *C* son *observacionalmente equivalentes* si y sólo si:

(1) *C* es una clase primitiva (entero, real...), $O_1$ y $O_2$ tienen valores idénticos.

(2) *C* no es una clase primitiva y, para cualquier secuencia *S* de operaciones de *C* que devuelven un objeto de clase *C'*, $O_1.S$ es observacionalmente equivalente a $O_2.S$ como objeto de clase *C'*.

En otras palabras, cuando es imposible distinguir $O_1$ de $O_2$ usando las operaciones de *C*. Si se dispusiera de una cantidad de tiempo infinita para comprobar si dos objetos son observacionalmente equivalentes, podría utilizarse el siguiente procedimiento para comprobar la corrección de la clase *C*:

- Sea *U* el conjunto de tuplas *($S_1$, $S_2$, etiqueta)*, donde $S_1$ y $S_2$ son secuencias de mensajes, y *etiqueta* es un texto que vale "equivalente" o "no equivalente".
- Para cada tupla de *U*, enviar las secuencias $S_1$ y $S_2$ a $O_1$ y $O_2$ y comprobar si ambos objetos son observacionalmente equivalentes.
- Si todas las equivalencias observacionales se comprueban de acuerdo con las etiquetas, entonces la implementación de *C* es correcta, e incorrecta en caso contrario.

La "equivalencia observacional" de dos objetos puede resolverse con un método tipo "equals"; sin embargo, la finitud del tiempo para realizar las pruebas es irresoluble, por lo que sus autores la han adaptado en la familia de herramientas *ASTOOT*. El componente de generación de casos de prueba toma la descripción sintáctica y semántica de la clase que se va a probar (Figura 40) y la traduce a una representación arborescente. Con esta representación, lee una secuencia de operaciones suministrada por el usuario y le aplica una serie de transformaciones para obtener secuencias de operaciones equivalentes. Las operaciones incluidas en las secuencias son simbólicas, en el sentido de que carecen de parámetros reales. Por último, y teniendo en cuenta la asunción explicada más arriba, la comprobación de la corrección de la clase la realizan pasando valores reales a los parámetros de los mensajes contenidos en las secuencias de cada tupla.

### 3.3. Obtención automática de especificaciones algebraicas

Henkel y Diwan [37] han desarrollado una herramienta que obtiene de manera automática especificaciones algebraicas a partir de clases Java. Comienzan obteniendo una lista de secuencias de operaciones válidas mediante llamadas sucesivas a los constructores de la clase y a algunos de sus métodos, pasando valores adecuados a los parámetros de estas operaciones. Si alguna de las se-

cuencias lanza una excepción, se deshace la última operación o se prueba con un nuevo valor del argumento (Figura 41).

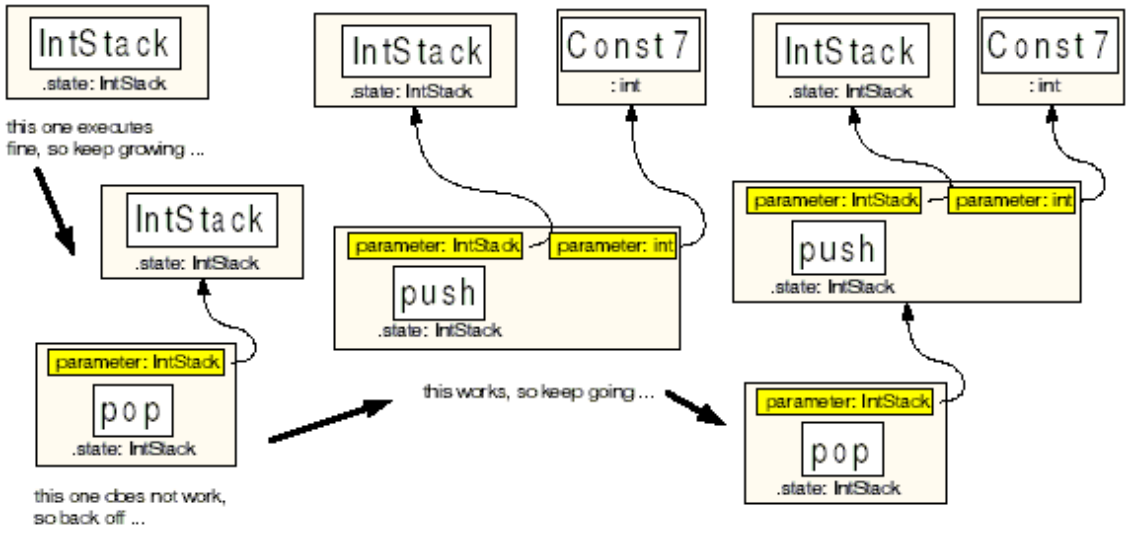



**Figura 41. Generación incremental de términos aplicada a la clase *IntStack* (Pila de enteros), tomada de**

Generadas las secuencias, obtienen ecuaciones de igualdad del estilo de las mostradas en la Figura 38 y en la Figura 40, pero que incluyen valores reales en lugar de simbólicos:

- Ecuaciones de igualdad de secuencias diferentes, como por ejemplo:

  *pop(push(IntStack().state, 4).state).state=IntStack().state*

- Ecuaciones de igualdad de secuencias con constantes, como:

  *pop(push(IntStack().state, 4).state).retval=4*

A partir de las ecuaciones se generan los axiomas, que constan de dos secuencias y una serie de variables cuantificadas. Así, de la ecuación de la izquierda, se obtiene el axioma de la derecha:

| | |
|---|---|
| *IntAdd(size(IntStack().state).retval,1).retval = size(push(IntStack().state, 3).state).retval* | *∀s:IntStack, ∀i:int IntAdd(size(s).retval,1).retval = size(push(s, I).state).retval* |

**Figura 42. Ejemplo de obtención de un axioma a partir de una ecuación**

Muchos de los axiomas generados son redundantes, por lo que con posterioridad se detectan y se eliminan aplicando reglas de reescritura, para lo que primero deben identificarse éstas de entre el conjunto de axiomas. Se considera que un axioma es una regla de reescritura si: (1) el lado izquierdo y el derecho tienen diferente longitud; y (2) las variables libres que aparecen en lado más

corto son un subconjunto de las variables libres que aparecen en el lado derecho.

## 4. Pruebas a partir de máquinas de estado

Las máquinas de estado han sido fuente frecuente de trabajos de investigación en el área de pruebas de software. Una máquina de estados es un grafo dirigido cuyos nodos representan los estados en los que puede encontrarse una instancia, y cuyos arcos representan transiciones entre dichos estados. Las transiciones, normalmente, se disparan por la ejecución de una operación de la instancia.

El uso de las máquinas de estado para describir casos de prueba (Sección 2, página 78) se puede completar con la definición de criterios de cobertura específicos para máquinas de estado. A continuación se describen los criterios propuestos por Offutt et al. [38].

### 4.1. 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.

### 4.2. Cobertura de predicados

Los predicados son las expresiones booleanas que etiquetan las guardas (condiciones en las transiciones de las máquinas de estado). Formalmente, un *predicado* es una expresión booleana compuesta de *cláusulas* y cero o más *operadores booleanos*. A su vez, una cláusula es una expresión booleana formada por expresiones booleanas combinadas con operadores no booleanos (por ejemplo, operadores relacionales). Por último, una expresión booleana es una expresión cuyo valor puede ser *true* o *false*. Esto se ilustra en la Tabla 11.

| | |
|---|---|
| Expresión booleana | *a*<br>*b* |
| Cláusula | *a>b*<br>*a<b* |
| Predicado | *(a>b) and (b<c)* |

**Tabla 11. Expresión booelana, cláusula y predicado (*a*, *b* y *c* son valores booleanos que pueden valer *true* o *false*)**

La cobertura de predicados se basa en el hecho de que cada cláusula debería ser probada independientemente, como si su valor no estuviese afectado por el resto de cláusulas. En otras palabras, deben considerarse las diferentes cláusulas de un predicado capaces de determinar el valor de todo el predicado:

> *Una cláusula* $c_i$ *determina un predicado si un cambio en el valor de verdad de* $c_i$ *cambia el valor de todo el predicado.*

Un predicado es cubierto completamente por un conjunto de casos de prueba cuando se cubren todas la cláusulas que lo componen. Más formalmente:

> *Para cada predicado* P *en cada transición y cada cláusula* $c_i$ *de* P*, un conjunto de casos de prueba* T *satisface el criterio de cobertura de predicados si* T *incluye casos tales que cada cláusula* $c_i$ *en* P *toma los dos valores (true y false).*

Supongamos que deseamos obtener casos de prueba para verificar este criterio con el predicado *(A or B) and C.* Para satisfacer este requisito, podemos construir la tabla de verdad para ese predicado (Tabla 12) y tomar aquellas combinaciones de *A, B* y *C* que hagan a *A true* una vez y *false* otra, a *B true* una vez y *false* otra, y a C *true* una vez y *false* otra: o sea, que podríamos reducir el conjunto de casos a los que aparecen resaltados en la tabla.

| A | B | C | A or B | (A or B) and C |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**Tabla 12. Tabla de verdad para *(A or B) and C***

### 4.3. 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 43, 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).*

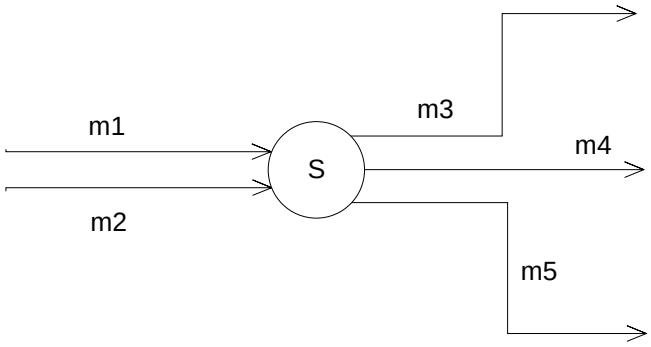



**Figura 43. Un fragmento de una máquina de estados**

## 4.4. 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 previamente ha seleccionado el ingeniero de pruebas.

## 5. Obtención de casos de prueba a partir de casos de uso

Basanieri et al. [39] utilizan diagramas de casos de uso y de secuencia para derivar casos de prueba desde las etapas iniciales del desarrollo de un sistema orientado a objetos. Tras realizar una serie de análisis y adaptaciones de los diagramas, aplican los siguientes pasos:

- Definir el conjunto de secuencias de mensajes *MS* a partir de los diagramas de secuencia. Cada secuencia comienza con un mensaje *m* sin predecesor (habitualmente, un mensaje enviado al sistema por un actor) y el conjunto de mensajes cuya ejecución dispara *m* (aquellos cuyo inicio está en el foco de control en que termina *m*).
- Analizar subcasos, que básicamente consiste en construir varias secuencias a partir de las posibles instrucciones condicionales que anotan los diagramas de secuencia.
- Identificar los conjuntos de valores de prueba, que se construyen a partir de los tipos de los parámetros de los métodos y del análisis de los diagramas de clases del sistema.
- Seleccionar, para cada mensaje de la secuencia, las situaciones relevantes en que el mensaje puede ocurrir y, para cada valor de prueba, valores válidos que puedan incluirse en el mensaje.
- Eliminar valores contradictorios o poco significativos, para lo que los autores sugieren utilizar clases de equivalencia.
- Obtener *procedimientos de prueba,* que puede hacerse automáticamente con los datos obtenidos en los pasos anteriores.

Así, a partir del diagrama de secuencia mostrado en la Figura 44, se identifican las siguientes secuencias de mensajes:

*MS_1: 1.start(), 1.1.open()*
*MS_2: 2.enterUserName(String)*
*MS_3: 3.enterPassword(String)*
*MS_4: 4.loginUser(), 4.1.validateuserIDPassword(String, String)*
*4.2.setupSecurityContext(), 4.2.1.new UserID()*
*4.3.closeLoginSelection()*

El diagrama incluye sin embargo una instrucción condicional, por lo que la secuencia *MS_4* puede dividirse en dos:

*MS_4.1: 4.loginUser(), 4.1.validateuserIDPassword(String, String)*
*4.2.setupSecurityContext(), 4.2.1.new UserID()*
*MS_4.2: 4.loginUser(), 4.1.validateuserIDPassword(String, String)*
*4.3.closeLoginSelection()*

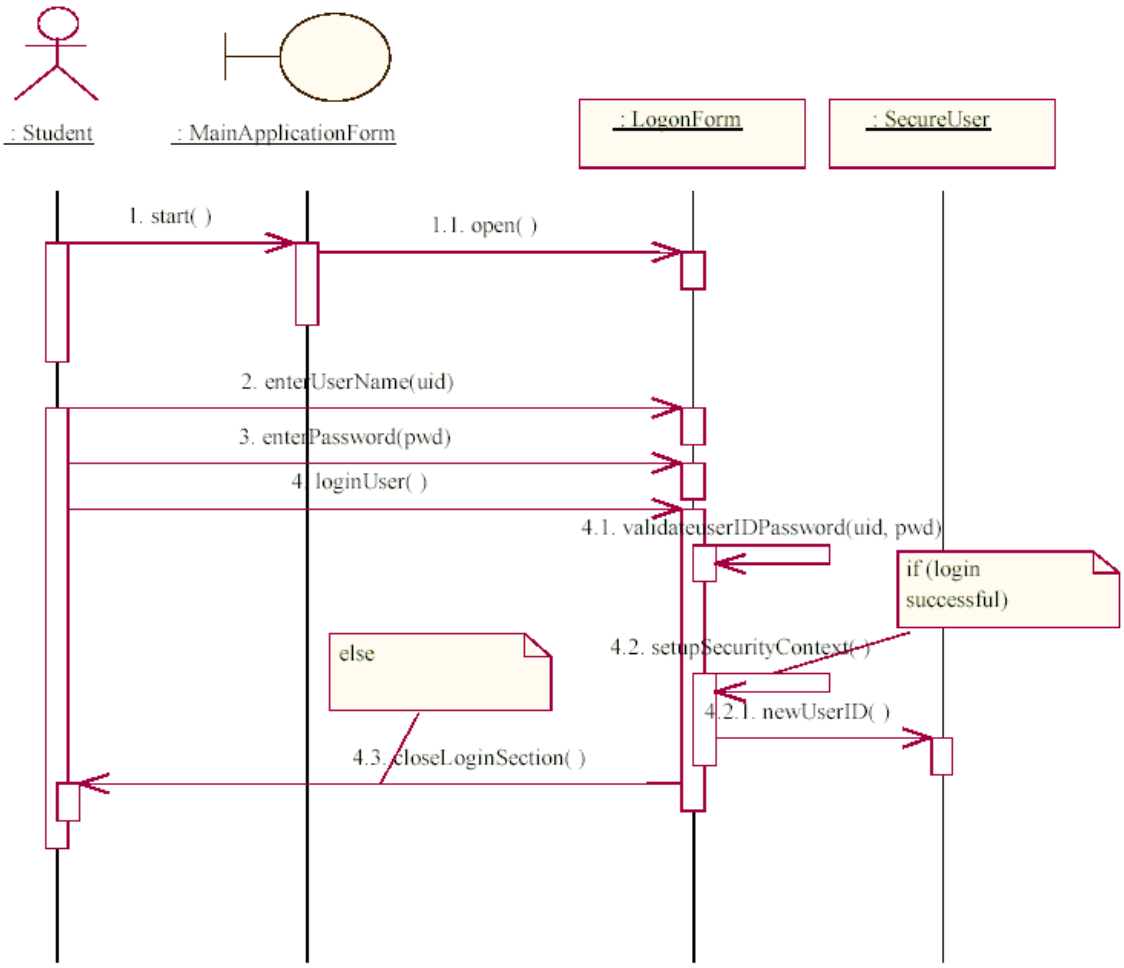



**Figura 44. Diagrama de secuencia de ejemplo, tomado de**

## 6. Diseño de clases orientados a la facilidad de pruebas

Baudry et al. analizan la complejidad de diagramas de clases UML con respecto a su facilidad de prueba, con el objetivo de proponer una serie de guías de diseño que permitan diseñar sistemas orientados a objeto fáciles de probar o, al menos, probables "en tiempo finito". El estudio se basa fundamentalmente en medidas de acoplamiento, que viene a ser el grado en que unos elementos dependen de otros; así, se asume que a mayor acoplamiento (tanto directo como indirecto), mayor dificultad para probar el diseño.

Estos autores dan las siguientes definiciones:

**Relación de uso directa:** la clase *A* usa la clase *B* cuando existe una asociación desde *A* hacia *B*, o bien cuando hay una dependencia con el estereotipo <<*usa*>> desde *A* hacia *B*. Este conjunto de relaciones se llama *SDU (Set of Direct Usage relationship)*. Nótese que una asociación no dirigida representa realmente dos relaciones de este conjunto.

**Relación de uso transitiva:** es el conjunto formado por la clausura transitiva de *SDU*.

**Relación de uso transitiva real:** si existe una relación transitiva entre las clases *A, B* y, además, el código fuente obtenido del diagrama de clases permite la interacción entre dos instancias de *A* y de *B,* entonces se dice que dicha relación transitiva es real.

**Interacción de clases (interacción potencial):** siendo *A, B* dos clases, ocurre si y sólo si $A\ R_i\ B$ y $B\ R_i\ A$, siendo *i* distinto de *j*. [La interacción es *potencial* porque tal vez no llegue físicamente a producirse en la ejecución del sistema].

**Interacción entre objetos (interacción real):** siendo *A, B* dos clases, ocurre si y sólo si (1) $A\ R_i\ B$ y $B\ R_i\ A$, siendo *i* distinto de *j*, y (2) $R_i$ y $R_j$ son relaciones transitivas reales. [La interacción es *real* porque se produce la interacción entre ambos objetos en la ejecución del sistema obtenido del diseño].

El **criterio de prueba** propuesto es el siguiente: *para cada interacción entre clases, o bien se obtiene un caso que prueba la interacción entre objetos, o bien se obtiene un informe mostrando que tal interacción no es factible.*

Puesto que la tarea de producir casos de prueba o informes es imposible si el número de interacciones es muy alto, es preciso obtener diseños que disminuyan el número de interacciones, de modo que se haga factible así la prueba completa del diseño de clases.

Antes de someterlo al criterio de pruebas, el diseño de clases debe ser evaluado para conocer el número de interacciones entre clases, modificándolo si es muy alto y tal mejora es posible, o rechazándolo directamente en otro caso. La mejora del diseño puede lograrse reduciendo el acoplamiento o anotándolo con restricciones que eviten la codificación de interacciones entre objetos propensas a error. Los autores proponen la anotación utilizando los estereotipos <<*create*>> (para indicar que la clase *A* crea instancias de la clase *B*),

<<*use_consult*>> (para indicar que la clase *A* sólo utiliza métodos tipo *get* de *B*) y <<*use_def*>> (para indicar que la clase *A* modifica el estado de las instancias de *B*). Estos estereotipos se utilizan en el proceso de construcción del *Grafo de Dependencias de Clases* (en inglés *Class Dependency Graph,* o *CDG)*, que representa las relaciones transitivas entre clases junto a las relaciones de herencia e implementación. La siguiente figura muestra algunos ejemplos sobre la obtención del *CDG*, en la que aparecen algunas etiquetas añadidas a los nodos del *CDG* para representar el tipo de relación existente.

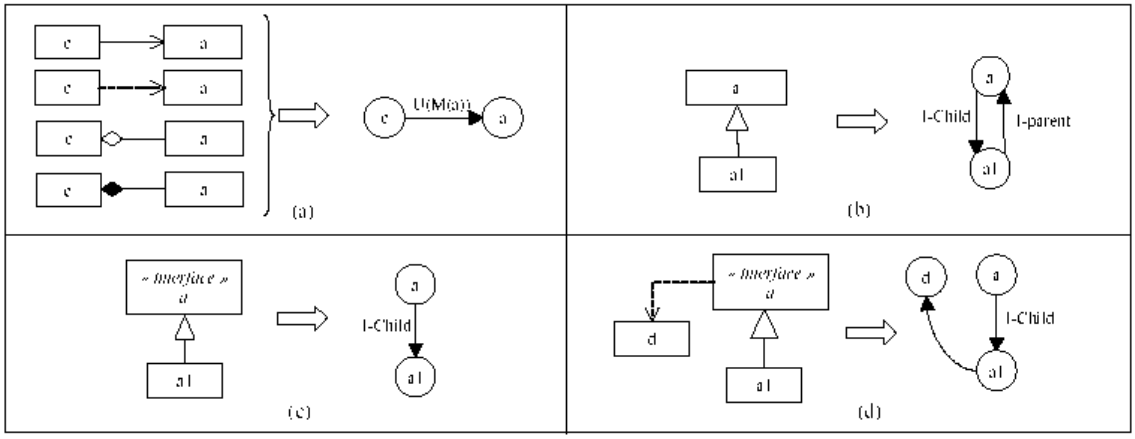



**Figura 45. Algunos ejemplos de obtención del CDG, tomada de**

A partir del *CDG* pueden calcularse medidas como la **complejidad de una interacción,** definida en función de la complejidad de los diferentes caminos por los que puede irse desde un objeto hasta otro:

$$complejidad(CI) = \sum_{i=1}^{nbPaths}((complejidad(P_i) \cdot \sum_{j>i} complejidad(P_j))$$

**Ecuación 5. Complejidad de una interacción**

La **complejidad de un camino en una interacción** es necesaria para calcular el valor de la Ecuación 5, y se define como el productorio de la complejidad asociada a cada jerarquía cruzada por la interacción (parámetro *IH)*:

$$complejidad(P) = \prod_{i=1}^{nbCrossed} complejidad(IH, P)$$

**Ecuación 6. Complejidad de un camino en una interacción**

Obviamente, es preciso poder calcular la **complejidad de un camino que pasa por una jerarquía de herencia:**

$$complejidad(IH, P) = \sum_{i=1}^{nbDP} complejidad(dp_i)$$

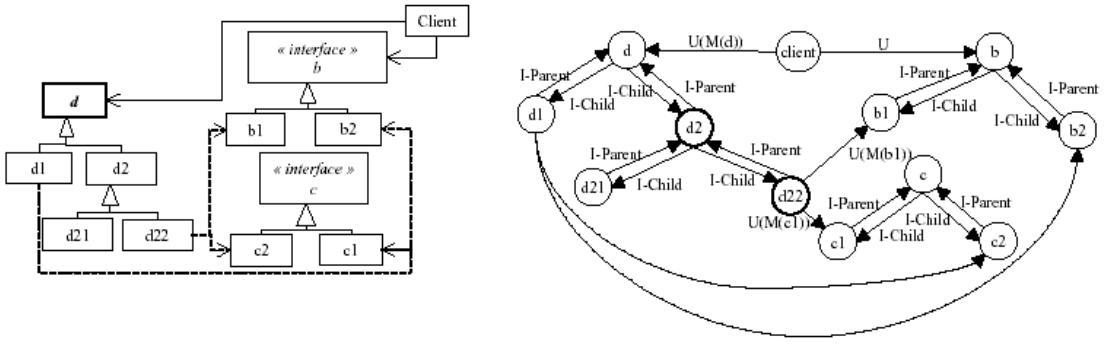
**Ecuación 7. Complejidad de un camino que pasa por una jerarquía de herencia**

Por último, la **complejidad de un camino de descendientes** es el número de interacciones potenciales entre las clases que hay en ese camino. El caso peor se daría cuando todas las clases estuvieran relacionadas con todas las clases, lo que supone un valor máximo de *n·(n-1)*. De manera general, la complejidad de un camino se corresponde con la Ecuación 8, en donde *h* representa la altura del camino:

$$comlejidad(dp) = h \cdot (h-1)$$

**Ecuación 8. Complejidad de un camino de descendientes**

En la siguiente figura, tomada de , la complejidad del camino que va desde *client* hasta *b1* pasando por la relación de herencia de la clase *d* es *1+3*(3-1)+1: 1* por la relación de uso de *client* con respecto a *d; 3*(3-1)* por el camino de descendientes de la jerarquía *d*, y otra vez *1* por la relación de uso desde *d22* a *b1*.




**Ecuación 9. Ejemplo para el cálculo**

## 7. Criterios de cobertura de pruebas para diseños UML

Andrews at al. [40] proponen varios criterios de cobertura para las pruebas de diferentes diagramas UML.

Para **diagramas de clases**, proponen los siguientes:

- *AEM (Association-end multiplicity):* dado un conjunto de pruebas *T* y un modelo *SM, T* debe causar que se cree cada par de multiplicidades representativo en las asociaciones de *SM*. Así, si existe una asociación cuya multiplicidad es, en un extremo, *p..n,* debería instanciarse la aso-

ciación con *p* elementos (valor mínimo), *n* elementos (valor máximo) y con uno o más valores en el intervalo *(p+1, n-1)*.

- *GN (Generalization):* dado un conjunto de pruebas *T* y un modelo *SM, T* debe conseguir que se cree cada relación de generalización de *SM*.
- *CA (Class attribute):* dado un conjunto de pruebas *T,* un modelo *SM* y una clase *C, T* debe conseguir que se creen conjuntos de valores representativos de los diferentes atributos de la clase *C*. El conjunto de *valores representativos* se consigue en tres pasos: (1) crear valores representativos para cada atributo, para lo que pueden usarse clases de equivalencia; (2) calcular el producto cartesiano de estos valores; (3) eliminar los conjuntos de valores inválidos, considerando el dominio del problema, posibles restricciones que anoten el diagrama, etc.

Para **diagramas de interacción**, los criterios propuestos son:

- *Cobertura de condiciones*: dado un conjunto de casos de prueba *T* y un diagrama de interacción *D*, *T* debe conseguir que cada condición del diagrama se evalúe a *true* y a *false*.
- Cobertura completa de predicados (*FP*: *full predicate coverage*): cada cláusula de cada condición debe evaluarse a *true* y a *false*.
- Cobertura de mensajes *(EML: each message on link):* cada mensaje del diagrama debe ejecutarse al menos una vez.
- Cobertura de caminos (*AMP: all message paths):* todos los posibles caminos de ejecución deben ejecutarse.
- Cobertura de colecciones (*Coll: Collection coverage):* el conjunto de casos de prueba debe probar cada interacción con colecciones al menos una vez.

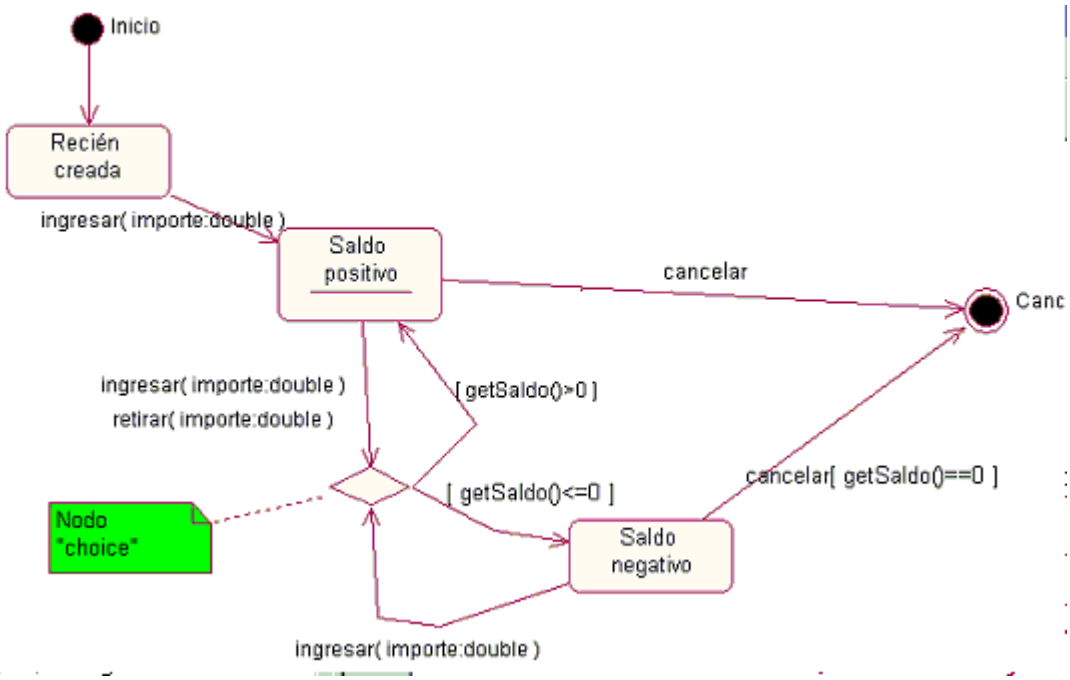
## 8. Revisiones e inspecciones de código fuente

Las revisiones e inspecciones de código fuente son una técnica para la detección manual de errores en el código . Se trabaja bajo el principio de que “cuatro ojos ven más que dos”, de tal manera que el método de trabajo consistirá, básicamente, en pasar el código escrito por un programador a un tercero o grupo de terceros, que tratará de encontrar posibles errores, faltas de adecuación al estilo de codificación utilizado por la organización, etc.

Para ello suelen utilizarse listas de comprobación (*checklists)*, que enumeran defectos y en los que el revisor anota su presencia o ausencia. En  puede consultarse una lista de comprobación de errores típicos del lenguaje Java, entre los que se incluyen el control de overflows y underflows, restricción adecuada del acceso a los miembros de las clases, control de apertura y cierre de ficheros, etc.

## 9. Ejercicios

1) Escriba casos de prueba que cumplan los criterios de cobertura mencionados en la Sección 4 para la siguiente máquina de estados, que representa el comportamiento de una cuenta corriente:




2) Escriba el código correspondiente a la máquina de estados del ejercicio anterior. Sométalo a los casos de prueba que ha propuesto. Analice si hay algún tipo de relación entre la cobertura de la máquina de estados y la del código fuente.

# Capítulo 8. MÉTRICAS PARA EL PROCESO DE PRUEBAS

En este capítulo se presentan algunas métricas importantes para mantener bajo control el proceso de pruebas.

## 1. Introducción

Como es bien sabido, existen multitud de métricas para medir diferentes atributos del software, como el número de líneas de código, número de puntos-función, complejidad ciclomática, complejidad de datos, etc., que pueden posteriormente ser utilizadas para la realización de predicciones sobre futuros proyectos de desarrollo o mantenimiento, asignación de recursos. De esta forma, el equipo de desarrollo es capaz de conocer las posibles desviaciones en el proyecto.

Esta misma filosofía puede también aplicarse al proceso de pruebas y, de hecho, también para él se han definido algunas métricas. Además de las importantes medidas de cobertura, ya discutidas en capítulos anteriores, algunas métricas son las siguientes:

1) Número de casos de prueba.
2) Número de casos de prueba/LOC
3) Número de casos de prueba/Número de requisitos
4) LOC de casos de prueba/LOC
5) Número de asserts/LOC
6) Número de casos de prueba planificados, ejecutados y pasados.
7) Coste (en unidades monetarias o en tiempo) de las diferentes actividades del proceso de pruebas.
8) Número de defectos encontrados en la fase de pruebas.
9) Diferentes medidas de obertura.

A continuación se revisan algunas métricas importantes para el proceso de pruebas.

## 2. Porcentaje de detección de defectos

El objetivo de la fase de pruebas no es demostrar que el programa es correcto, sino encontrar los errores que haya en él, de tal manera que dudaremos de la fiabilidad de un software en el que no se han encontrado errores.

El porcentaje de detección de defectos (PDD) es el cociente de dividir el número de defectos encontrados en la fase de pruebas por el número total de defectos presentes en el software. Así, si el software objeto de la prueba tiene 100 defectos y encontramos 50, PDD=0,5.

El número total de defectos sólo puede ser creciente, por lo que el PDD será, en todo caso, decreciente. Para estimar el número total de defectos es necesario disponer de datos históricos de anteriores proyectos de desarrollo.

## 3. Porcentaje de defectos corregidos

El porcentaje de defectos corregidos (PDC) es el resultado de dividir el número de defectos corregidos entre el número total de defectos. Así, pues, PDC≤PDD.

## 4. Medida del retorno de la inversión (ROI)

A continuación se reproduce un ejemplo del libro de Fewster y Graham [41].

Supongamos que el proceso de pruebas en una empresa supone 10.000 €/año, y que gracias a este proceso el PDD es del 70%. Asumamos también que el coste de corregir un defecto antes de la entrega es de 100 €, y de 1000 € si se corrige después. Supongamos también que se introducen 1000 defectos en un sistema de tamaño grande. Puesto que PDD=70%, se habrán encontrado 700 defectos antes de la entrega, lo que supone un coste de 700x100=70.000 €.

En un momento dado, la empresa puede decidir la inversión de 10.000 € más en pruebas (mediante, por ejemplo, cursos de formación como éste, adquisición de herramientas, etc.), con el fin de intentar elevar el PDD desde el 70% actual hasta el 90%. ¿Compensará la inversión realizada? Para ello calculamos el ROI a un año tal y como se muestra en la siguiente tabla.

| | **Proceso actual** | **Proceso mejorado** |
|---|---|---|
| Coste del testing | 10000 | 20000 |
| PDD | 70% | 90% |
| Defectos encontrados en las pruebas | 700 | 900 |
| Coste de corregir en las pruebas | 70000 | 90000 |
| Defectos encontrados después de las pruebas | 300 | 100 |
| Coste de corregir después de las pruebas | 300000 | 100000 |
| Coste total | 380000 | 210000 |
| Beneficio | | 170000 |
| Inversión realizada | | 10000 |
| ROI (beneficio/inversión) | | 1700% |

**Tabla 13. Cálculo del ROI para un proyecto**

# Capítulo 9. REFERENCIAS


1. ISO/IEC, ISO/IEC 12207. International Standard. Software Life Cycle Processes. International Standard Organziation/International Electrotechnical Committee, 1995. Geneve.
2. Karlström D, Runeson P and Nordén S. *A minimal test practice framework for emerging software organizations.* Software Testing, Verification and Reliability, 2005. **15**(3): p. 145-166.
3. Giraudo G and Tonella P. *Designing and conducting an empirical study on test management automation.* Empirical Software Engineering, 2003. **8**(1): p. 59-81.
4. Runeson P, Andersson C and Höst M. *Test processes in software product evolution -a qualitative survey on the state of practice.* Journal of Software Maintenance and Evolution: Research and Practice, 2003. **15**(1): p. 41-59.
5. Meudec C. *ATGen: automatic test data generation using constraint logic programming and symbolic execution.* Software Testing, Verification and Reliability, 2001. **11**(2): p. 81-96.
6. Ball T, Hoffman D, Ruskey F, Webber R and White L. *State generation and automated class testing.* Software Testing, Verification and Reliability, 2000. **10**: p. 149-170.
7. Myers B. *The Art of Software Testing*. 1979: John wiley & Sons.
8. Cornett S. *Code Coverage Analysis*. 2004.
9. IEEE Standard for Software Unit Testing. Institute of Electrical and Electronics Engineers, 1987.
10. Pressman RS. *Ingeniería del Software, un enfoque práctico (3ª Edición)*. 1993: McGraw-Hill.
11. Gallagher L, Offutt J and Cincotta A. *Integration testing of object-oriented components using finite state machines.* Software Testing, Verification and Reliability, 2006. **16**.
12. Offutt AJ. *A practical system for mutation testing: help for the common programmer*. 12th International Conference on Testing Computer Software. 1995.
13. Offutt AJ, Rothermel G, Untch RH and Zapf C. *An experimental determination of sufficient mutant operators.* ACM Transactions on Software Engineering and Methodology, 1996. **5**(2): p. 99-118.
14. Polo M, Piattini M and Tendero S. *Integrating techniques and tools for testing automation.* Software Testing, Verification and Reliability, 2007. **17**(1): p. 3-39.
15. Ma Y-S, Offutt J and Kwon YR. *MuJava: an automated class mutation system.* Software Testing, Verification and Reliability, 2005. **15**(2): p. 97-133.
16. Grindal M, Offutt AJ and Andler SF. *Combination testing strategies: a survey.* Software Testing, Verification and Reliability, 2005. **15**: p. 167-199.
17. Hoffman D, Strooper P and White L. *Boundary values and automated component testing.* Software Testing, Verification and Reliability, 1999. **9**(1): p. 3-26.
18. Harrold M, Gupta R and Soffa M. *A methodology for controlling the size of a test suite.* ACM Transactions on Software Engineering and Methodology, 1993. **2**(3): p. 270-285.

19. Jeffrey D and Gupta N. *Test suite reduction with selective redundancy*. International Conference on Software Maintenance. 2005. Budapest (Hungary): IEEE Computer Society.
20. Tallam S and . NG. *A concept analysis inspired greedy algorithm for test suite minimization*. 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2005.
21. Heimdahl M and George D. *Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing*. 19th IEEE International Conference on Automated Software Engineering. 2004.
22. McMaster S and Memon A. *Call Stack Coverage for Test Suite Reduction*. 21st IEEE International Conference on Software Maintenance. 2005. Budapest (Hungary).
23. Shiba T, Tsuchiya T and Kikuno T. *Using artificial life techniques to generate test cases for combinatorial testing*. 28th Annual International Computer Software and Applications Conference. 2004. Hong Kong, China: IEEE Computer Society Press.
24. Malaiya Y. *Antirandom testing: Getting the most out of black-box testing*. International Sym-posium on Software Reliability Engineering (ISSRE 95). 1995. Toulouse, France: IEEE Computer Society Press, Los Alamitos, CA.
25. Lapierre S, Merlo E, Savard G, Antoniol G, Fiutem R and Tonella P. *Automatic Unit Test Data Generation Using Mixed-Integer Linear Programming and Execution Trees*. International Conference on Software Maintenance. 1999. Oxford, England.
26. Díaz E, Tuya J and Blanco R. *Pruebas automáticas de cobertura de software mediante una herramienta basada en Búsqueda Tabú*. VIII Jornadas de Ingeniería del Software y Bases de Datos. 2003. Alicante, Spain.
27. Baudry B, Fleurey F, Jézéquel J-M and Traon YL. *Automatic test case optimization: a bacteriologic algorithm.* IEEE Software, 2005: p. 76-82.
28. Kanstrén T. *Towards a deeper understanding of test coverage.* Journal of Software Maintenance and Evolution: Research and Practice, 2008. **20**(1): p. 59-76.
29. Rumbaugh J, Jacobson I and Booch G. *The Unified Modelling Language Reference Manual*. Segunda edición ed, ed. A.-W.O.T. Series. 2005: Addison-Wesley.
30. Edwards SH. *A framework for practical, automated black-box testing of component-based software.* Software Testing, Verification and Reliability, 2001(11): p. 97-111.
31. Ghosh S and Mathur AP. *Interface mutation.* Software Testing, Verification and Reliability, 2001(11): p. 227-247.
32. Offutt J and Xu W. *Generating Test Cases for Web Services Using Data Perturbation*. Workshop on Testing, Analysis and Verification of Web Services. 2004. Boston, Massachusetts.
33. Mogyorodi GE. *What is Requirement-Based Testing?* CrossTalk: The Journal of Defense Software Engineering, 2003(march): p. 12-15.
34. Kirani S and Tsai WT. *Method sequence specification and verification of classes.* Journal of Object-Oriented Programming, 1994. **7**(6): p. 28-38.
35. Tse T and Xu Z. *Test Case Generation for Class-Level Object-Oriented Testing*. 9th International Software Quality Week. 1996. San Francisco, CA.

36. Doong RK and Frankl PG. *The ASTOOT approach to testing object-oriented programs.* ACM Transactions on Software Engineering and Methodology, 1994. **3**(2): p. 101-130.
37. Henkel J and Diwan A. *Discovering Algebraic Specifications from Java Classes*. 17th European Conference on Object-Oriented Programming (ECOOP). 2003: Springer.
38. Offutt AJ, Liu S, Abdurazik A and Amman P. *Generating test data from state-based specifications.* Software Testing, Verification and Reliability, 2003(13): p. 25-53.
39. Basanieri F, Bertolino A and Marchetti E. *The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects*. 5th International Conference on The Unified Modeling Language. 2002: Springer-Verlag. LNCS.
40. Andrews A, France R and Ghosh S. *Test adequacy criteria for UML design models.* Software Testing, Verification and Reliability, 2003(13): p. 95-127.
41. Fewster M and Graham D. *Software Test Automation*. 2000: Addison-Wesley.