Informe Técnico – Technical Report
DPTOIA-IT-2001-001
Noviembre, 2001

VERIFICACIÓN CON XML

Pedro Jesús Vázquez Escudero
María N. Moreno García
Francisco J. García Peñalvo

Departamento de Informática y Automática
Universidad de Salamanca

Revisado por:

Dr. Miguel Ángel Laguna Serrano

Departamento de Informática

Universidad de Valladolid

mlaguna@infor.uva.es

Dr. José Rafael García-Bermejo Giner

Departamento de Informática y Automática

Universidad de Salamanca

Aprobado en el Consejo de Departamento de 29 de octubre de 2001

Información de los autores:

Pedro Jesús Vázquez Escudero

Estudiante de doctorado del Departamento de Informática y Automática

Universidad de Salamanca

pedro.vazquez.e@terra.es

Dra. María N. Moreno García

Área de Lenguajes y Sistemas Informáticos

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 – Salamanca

mmg@usal.es

Dr. Francisco José García Peñalvo

Área de Ciencia de la Computación e Inteligencia Artificial

Departamento de Informática y Automática

Facultad de Ciencias - Universidad de Salamanca

Plaza de la Merced S/N – 37008 – Salamanca

fgarcia@usal.es

Este trabajo ha sido parcialmente financiado por el proyecto “DOLMEN” de la CICYT, TIC2000-1673-C06-05, Ministerio de Ciencia y Tecnología.

Resumen

Este documento presenta un marco de trabajo para la verificación automática de programas en entornos de sistemas orientados a objetos. El núcleo del sistema lo constituye un repositorio de clases en formato XML. Un *parser* o analizador sintáctico analiza el código fuente en el lenguaje seleccionado, extrae de él la información relevante del programa y alimenta el repositorio. El lenguaje XML es utilizado como metalenguaje para la creación de un árbol de sintaxis abstracto que lo independiza del lenguaje de programación utilizado. El repositorio sirve para la creación de casos de prueba. El uso de patrones de prueba facilitará la generación automática de módulos de prueba que serán posteriormente alimentados por los casos de prueba seleccionados. Las ventajas de utilizar XML como metalenguaje se pueden observar en la extensibilidad del entorno de trabajo: basta con añadir un *parser* para otro lenguaje y el sistema completo será utilizable para dicho lenguaje. No será necesario modificar las herramientas creadas para el sistema. Es más, con un *parser* UML es posible subir hasta la fase de diseño, anticipando la creación de casos de prueba a dicha etapa del ciclo de vida.

Abstract

This document presents a framework for automated software testing in object oriented systems environments. The heart of the system consists of an XML-classes repository. A parser or syntactic analyzer inspects the source code of the language of choice and extracts the relevant information into the repository. XML language is used as metalanguage in order to create the syntax abstract tree that is independent of the programming language. The repository can be used for test generation purposes. The test patterns usage will facilitate the automatic generation of test modules that will then be seeded by the selected test cases. The advantages of using XML as metalanguage are the extensibility of the framework: just adding a parser for another language the whole system will be available for this language. No changes are needed in the tools created for the system. Further more, a UML parser should make a step into the design phase, anticipating the test case creation to this phase of the software life cycle.

Tabla de Contenidos

1. Introducción

Este documento presenta un marco de trabajo para la verificación automática de programas en entornos de sistemas orientados a objetos.

La representación del código fuente de un programa mediante un fichero de texto plano obliga a la realización de un *análisis sintáctico* posterior para hacer visible la estructura del mismo en actividades como la traducción a código ejecutable. Herramientas de ingeniería de software tales cómo generadores de casos de prueba, mecanismos de obtención de métricas, etc. necesitan conocer dicha estructura y es norma habitual que cada herramienta incorpore un proceso de análisis sintáctico.

La *representación externa* del *árbol de sintaxis abstracta* del programa en un lenguaje estandarizado permitiría ahorrar recursos de máquina al evitar la repetición de tareas como el análisis sintáctico y facilitaría la cooperación entre las herramientas de distintos fabricantes, además de no ligar la herramienta al lenguaje para el que se haya creado. Se propone la utilización de XML (*eXtensible Markup Language*) [Bray et al., 2000] como lenguaje de soporte a las distintas herramientas de Ingeniería de Software, y en particular a la verificación automática de programas.

El documento se organiza como sigue:

En primer lugar se pasa revista a las principales técnicas de prueba del software, la prueba estructural o de Caja Blanca y la prueba funcional o de Caja Negra.

A continuación se describen distintos entornos de verificación automática de programas y distintas herramientas de ayuda basadas en XML.

Se propone el uso de patrones de prueba para la generación automática de módulos de prueba, lo que separaría el código real del programa del código de prueba, además de ahorrar recursos de codificación. Estos módulos de prueba se alimentarían de la base de datos de casos de prueba para su ejecución resultando un proceso totalmente automático.

Por último, se propone la creación de un repositorio de módulos en formato XML, auténtico núcleo del sistema. Un *parser* o analizador sintáctico analiza el código fuente en el lenguaje seleccionado y extrae de él la información relevante del programa y alimenta el repositorio. El lenguaje XML es utilizado como metalenguaje para la creación de un árbol de sintaxis abstracto que lo hace independiente del lenguaje de programación utilizado. El repositorio sirve para la creación de casos de prueba.

2. Técnicas de prueba del software

2.1. Objetivos de la prueba del software

La prueba del software es una fase clave del ciclo de vida de un producto. En ella se determina la disposición del producto para ser entregado basándose en unos criterios preestablecidos de corrección y calidad. La actividad de verificación del software debe por tanto ser cuidadosamente planificada al objeto de poder asegurar que la fase de prueba garantiza esos niveles de corrección y calidad.

Antes de continuar, conviene definir los dos términos principales relacionados con la fase de prueba [Boehm, 1981]:

- Verificación: responde a la pregunta ¿se está construyendo el producto correctamente?
- Validación: responde a la pregunta ¿se está construyendo el producto correcto? Es decir, ¿se ajusta a los requisitos del cliente?

La prueba confirma los niveles de calidad alcanzados en todas las etapas del ciclo de vida mediante la aplicación adecuada de métodos y herramientas.

Las estrategias actuales de verificación y validación (V&V) se aplican a todas las etapas del ciclo de vida. Por ejemplo, siguiendo a Boehm, [Boehm, 1984] los cuatro criterios básicos para la V&V de las especificaciones de requisitos y de diseño son:

- Compleción: una especificación es completa si todas sus partes están presentes y cada parte está completamente desarrollada.
- Consistencia: interna (las especificaciones no deben entrar en conflicto unas con otras) y externa (las especificaciones no deben entrar en conflicto con especificaciones o entidades externas) y trazabilidad (los detalles de las especificaciones han de tener claros antecedentes en especificaciones anteriores o en los objetivos del sistema).
- Factibilidad: una especificación es factible si los beneficios del sistema exceden sus costes.
- Verificabilidad: una especificación es verificable si se puede identificar una técnica económicamente factible que permita determinar si un producto software satisfará o no una especificación.

En este trabajo nos centraremos en la prueba de programas y más concretamente de sistemas orientados a objetos.

Glen Myers [Myers, 1979] establece tres reglas principales que sirven como objetivos de prueba:

1. La prueba es un proceso de ejecución de un programa con la intención de descubrir un error.
2. Un buen caso de prueba es aquél que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
3. Una prueba tiene éxito si descubre un error no detectado hasta entonces.

La prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el software.

Leyendo estas reglas se pone de manifiesto el carácter destructivo del proceso de prueba. El desarrollador debe cambiar de punto de vista, eliminando todas las nociones previas que tenga sobre la posible corrección del software que ha desarrollado.

Otra característica deseable del proceso de prueba es que sea capaz de encontrar diferentes clases de errores en la mínima cantidad de tiempo posible y con el mínimo esfuerzo.

Existen dos enfoques a la hora de probar un producto software que se presentarán en los dos apartados siguientes. No son auto-excluyentes y de hecho son complementarios.

2.2. Prueba de la Caja Blanca

También llamada estructural o a pequeña escala. Usa la estructura de control del diseño del programa para derivar los casos de prueba, es decir, se aprovecha del conocimiento de la

estructura interna del mismo. Mediante los métodos de prueba de la caja blanca se pueden derivar casos de prueba que:

1. Garanticen que se ejercitan por lo menos una vez todos los *caminos independientes* de cada módulo.
2. Se ejercitan todas las *decisiones lógicas* en sus caras verdadera y falsa.
3. Se ejecutan todos los *bucles* en sus límites y con sus límites operacionales.
4. Se ejercitan las *estructuras de datos internas* para asegurar su validez.

Una justificación para la realización de este tipo de prueba en vez de invertir el tiempo asegurando que se han alcanzado los requisitos funcionales del programa (prueba de la caja negra) la encontramos en [Jones, 1981]:

1. Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa. Se cometen más errores al diseñar o implementar funciones no habituales.
2. A menudo se cree que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma regular.
3. Los errores tipográficos son aleatorios.

2.2.1. Notación de grafo de flujo

Antes de describir las diferentes estrategias de pruebas estructurales se definen los conceptos sobre grafos en que se basan las mismas.

- Grafo de flujo: representan el flujo de control de un programa y ayudan en la obtención de conjuntos de prueba. Está formado por un conjunto de nodos y arcos.
- Nodos: cada nodo del grafo representa una o más sentencias del programa.
- Arcos: son líneas dirigidas que unen dos nodos. Los arcos entre nodos representan flujo de control. Un arco debe terminar en un nodo.
- Región: es un área rodeada por nodos y arcos.
- Nodo predicado: es un nodo que contiene una condición.
- Caminos independientes: es un camino que introduce por lo menos un nuevo conjunto de sentencias de proceso (debe moverse por lo menos por un arco nuevo en el camino).
- Medida de la complejidad ciclomática *V(G)* [McCabe, 1976]: da un valor límite para el número de pruebas que se deben diseñar y ejecutar para garantizar que se cubren todas las sentencias del programa. Este límite es el *número ciclomático*, que mide el número de caminos linealmente independientes para un grafo de flujo dado, y que viene dado por la fórmula:

$$V(G) = 1 + d$$

  donde *d* es el número de nodos predicado del grafo G.
- Conjunto básico: colección de caminos que garantizan la ejecución por lo menos una vez de todas las sentencias del programa.

La figura 1 muestra un ejemplo de grafo de flujo con nodos predicado, con V(G) = 1 + 3 = 4.

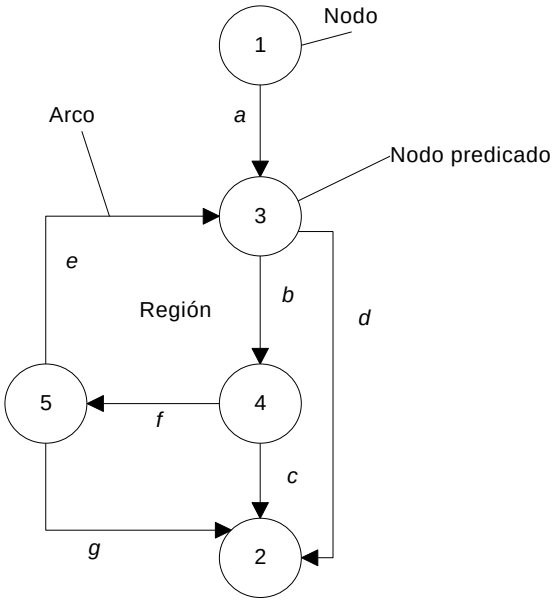

Figura 1. Grafo de flujo

2.2.2. Matrices de grafos

Son útiles para mecanizar la obtención del grafo de flujo y la determinación de un conjunto básico de caminos:

- En cada celda se indica la arista que conecta dos nodos dados.
- La representación en forma de matriz de conexiones reemplaza el identificador de la arista por un 1.
- Peso de enlace: da información adicional sobre la matriz de conexión. De forma sencilla un peso 1 indica que existe una conexión y un peso 0 indica que no existe conexión.

La figura 2 muestra la matriz de grafo y la matriz de conexiones correspondiente al grafo de flujo de la figura 1.

	NODO	1	2	3	4	5
N	**1**			*a*		
O	**2**					
D	**3**		*d*		*b*	
O	**4**		*c*			*f*
	5		*g*	*e*		

NODO	1	2	3	4	5
1			*1*		
2					
3		*1*		*1*	
4		*1*			*1*
5		*1*	*1*		

Figura 2. Matriz de grafo y matriz de conexiones

2.2.3. Estrategias de pruebas estructurales

A continuación se describen brevemente las diferentes estrategias de pruebas estructurales o de Caja Blanca [Fenton y Pfleeger, 1997]:

- *Cobertura de sentencias*: consiste en seleccionar casos de prueba que garanticen que cada sentencia o secuencia de sentencias sin puntos de decisión ha sido probada al menos una vez.
- *Cobertura de ramas*: consiste en seleccionar un conjunto de caminos de tal forma que cada rama del programa cae en al menos un camino. Un 100% de cobertura de ramas garantiza un 100% de cobertura de sentencias, pues toda sentencia está en alguna rama.
- *Cobertura de caminos*: es la estrategia de prueba de Caja Blanca más exhaustiva y consiste en seleccionar casos de prueba de tal forma que cada camino posible del programa es ejecutado al menos una vez. Resulta imposible llevar a la práctica esta estrategia debido a que, por un lado, la existencia de un simple bucle podría implicar la existencia de un número infinito de caminos y, por otro lado, pueden existir caminos inalcanzables para cualquier valor de la entrada.

  Debido a estos problemas se han sugerido otras estrategias que intentan eliminarlos:

  - *Prueba del camino simple*: consiste en la ejecución de todos los caminos simples (un camino simple es aquél que no contiene la misma rama más de una vez) [Prather, 1987].
  - *Prueba estructurada de McCabe, o de caminos linealmente independientes*: consiste en la ejecución de todos los caminos linealmente independientes a través del grafo de flujo del programa [McCabe, 1976]. McCabe define el concepto de número ciclomático, el cuál es igual al número de casos de prueba requeridos para satisfacer la estrategia.

La obtención del *índice de efectividad de las pruebas* permite conocer el grado de extensión en que los casos de prueba satisfacen una estrategia de prueba particular para un programa dado y un conjunto de casos de prueba. Siendo *T* una estrategia de prueba para cubrir una clase de objetos (como caminos, caminos simples, caminos linealmente independientes, ramas o sentencias), el *índice de efectividad de las pruebas* se define como la proporción entre el número de objetos probados al menos una vez y el número total de objetos.

Los dos apartados siguientes muestran con un mayor grado de profundidad las estrategias de prueba del camino básico y la prueba de bucles.

2.2.3.1. Prueba del Camino Básico

Prueba de Caja Blanca definida por McCabe [McCabe, 1976]. Este método permite al diseñador derivar una medida de la complejidad lógica de un programa y usarla como una guía para definir un conjunto básico de caminos de ejecución. Se garantiza que los casos de prueba que ejercitan el conjunto básico ejecutan todas las sentencias del programa al menos una vez.

La derivación de casos de prueba se hace siguiendo los siguientes pasos:

1. Derivar un grafo de flujo partiendo del diseño o del código fuente.
2. Determinar la complejidad ciclomática de este grafo de flujo.
3. Determinar un conjunto básico de caminos linealmente independientes.
4. Preparar casos de prueba que fuercen la ejecución de cada camino del conjunto básico.

2.2.3.2. Prueba de bucles

Los posibles errores en las construcciones de bucles son:

- De iniciación
- De indexación o incremento
- En los límites

Esta técnica se centra exclusivamente en la validez de las construcciones de bucles. Se definen cuatro clases de bucles diferentes y un conjunto de pruebas para cada tipo de bucle:

- Bucles simples: donde *n* es el máximo número de pasos permitidos por el bucle:
  1. Saltar el bucle completamente.
  2. Pasar una sola vez por el bucle.
  3. Hacer *m* pasos por el bucle con $m < n$.
  4. Hacer $n - 1$, n y $n + 1$ pasos por el bucle.
- Bucles anidados:
  1. Comenzar en el bucle más interior. Disponer todos los demás bucles en sus valores mínimos.
  2. Realizar pruebas de bucle simple con el bucle más interior mientras se mantienen los bucles exteriores con valores mínimos. Añadir pruebas de fuera de rango o de valores excluidos.
  3. Progresar hacia fuera realizando pruebas para el siguiente bucle, manteniendo los demás bucles exteriores en valores mínimos y los demás bucles anidados con valores “típicos”.
  4. Continuar hasta que todos los bucles hayan sido probados.
- Bucles concatenados: pueden ser probados como bucles simples si cada bucle es independiente de los otros. Si no es así, se puede utilizar el método de bucles anidados.
- Bucles no estructurados: deben rediseñarse, no probarse.

2.3. Prueba de la Caja Negra

También llamada funcional o a gran escala. Se ignora la estructura de control concentrándose en los requisitos funcionales del sistema ejercitando todos ellos. Las pruebas se diseñan para responder a las siguientes preguntas:

1. ¿Cómo se prueba la validez funcional?
2. ¿Qué clases de entrada compondrán buenos casos de prueba?
3. ¿Es el sistema particularmente sensible a ciertos valores de entrada?
4. ¿De qué forma están aislados los límites de una clase de datos?
5. ¿Qué volúmenes y niveles de datos tolerará el sistema?
6. ¿Qué efectos sobre la operación del sistema tendrán combinaciones específicas de datos?

Mediante las técnicas de prueba de la caja negra se derivan casos de prueba que satisfacen los siguientes criterios [Myers, 1979]:

1. Casos de prueba que reducen, en un coeficiente que es mayor que uno, el número de casos de prueba adicionales que se deben diseñar para alcanzar una prueba razonable.
2. Casos de prueba que dicen algo sobre la presencia o ausencia de clases de errores en lugar de un error asociado solamente con la prueba en particular que se encuentre disponible.

Los errores, que este tipo de prueba intenta encontrar, son de alguna de las siguientes categorías:

1. Funciones incorrectas o ausentes.
2. Errores de interfaz.
3. Errores en estructuras de datos o acceso a bases de datos externas.
4. Errores de rendimiento.
5. Errores de iniciación y de terminación.

La prueba de la Caja Blanca debería realizarse al principio del proceso de prueba, mientras que la prueba de la Caja Negra debería realizarse en posteriores etapas.

Se resumen a continuación los métodos presentados en [Pressman, 1998].

2.3.1. Partición equivalente

Divide el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba. La partición equivalente intenta definir un caso de prueba que descubra clases de errores y por tanto reduzca el número de casos de prueba necesarios. Está basado en una evaluación de las clases de equivalencia para una condición de entrada. Las clases de equivalencia se pueden definir de acuerdo con las siguientes directrices:

1. Si una condición de entrada especifica un rango, se definen una clase de equivalencia válida (dentro del rango) y dos inválidas (por debajo del rango y por encima del rango).
2. Si una condición de entrada requiere un valor específico, se definen una clase de equivalencia válida (el valor específico) y dos inválidas (por debajo del valor y por encima del valor).

3. Si una condición de entrada especifica un miembro de un conjunto, se definen una clase de equivalencia válida (dentro del conjunto) y una inválida (fuera del conjunto).
4. Si una condición de entrada es lógica, se definen una clase válida (cierto) y una inválida (falso).

De igual forma se construyen las particiones de equivalencia para los valores del dominio de salida.

2.3.2. Análisis de valores límite

Complementa la partición equivalente dado que selecciona casos de prueba en los "bordes" de una clase. Además de centrarse en las condiciones de entrada, también deriva casos de prueba para el dominio de salida. Las directrices para derivar casos de prueba son:

1. Para una condición de entrada que especifica un rango limitado por los valores *a* y *b*, los casos de prueba deben incluir los valores *a* y *b* y los valores justo por debajo de *a* y justo por encima de *b*.
2. Para una condición de entrada que especifica un número de valores, los casos de prueba deben incluir el valor mínimo, el máximo y los valores justo por debajo del mínimo y justo por encima del máximo.
3. Aplicar las directrices 1 y 2 a las condiciones de salida.
4. Si las estructuras de datos internas tienen límites preestablecidos, debe diseñarse un caso de prueba que ejercite la estructura en sus límites.

2.3.3. Técnicas de grafos de causa-efecto

Proporcionan una concisa representación de las condiciones lógicas y sus correspondientes acciones. Sigue cuatro pasos:

1. Se listan para un módulo las causas (condiciones de entrada) y los efectos (acciones), asignando un identificador a cada uno de ellos.
2. Se desarrolla un grafo de causa-efecto.
3. Se convierte el grafo en una tabla de decisión.
4. Las reglas de la tabla de decisión se convierten a casos de prueba.

3. Entornos de verificación automática de programas

3.1. Introducción

A continuación se describen brevemente distintos entornos de trabajo para pruebas unitarias de programas indicando, a juicio de los autores, las principales ventajas e inconvenientes de cada entorno. De entre la gran cantidad de entornos de trabajo disponibles se han seleccionado aquellos que son de código abierto, lo que facilita su conocimiento y mejora.

Se incluye un apartado de herramientas de ayuda complementarias tales como sistemas de compilación masivos o de ejecución automática de los programas tras la compilación.

3.2. Descripción de entornos

3.2.1. JUnit

3.2.1.1. Introducción

JUnit [Gamma y Beck, 2001] es un entorno sencillo para pruebas unitarias de programas **Java** [Gosling et al., 2000]. Facilita la creación de casos de prueba y colecciones de casos de prueba, su ejecución y posterior verificación de resultados.

El método de trabajo consiste en ejecutar las pruebas cada vez que se lanza una compilación. Para ello **JUnit** permite la ejecución de las pruebas de forma automática y la comprobación de los resultados.

Las pruebas pueden ser agrupadas en colecciones (*suites*) que pueden contener casos de prueba y otras colecciones. De esta manera se puede construir una jerarquía de casos de prueba para efectuar pruebas de regresión con todos los casos de prueba de una sola vez.

Construir un caso de prueba consiste en escribir un método que ejercita el código que se quiere probar y definir el resultado esperado. El entorno de trabajo facilita los mecanismos adecuados para la realización de esta tarea de una forma cómoda.

Como su propio nombre indica está diseñado para la ejecución de pruebas unitarias, es decir para probar bloques de código desde el punto de vista de su arquitectura interna y por tanto es un entorno pensado para el desarrollador.

3.2.1.2. Diseño de JUnit

JUnit está diseñado alrededor de dos *patrones de diseño* [Gamma et al, 1995] principales: el patrón *Command* y el patrón *Composite*:

- `TestCase` es un objeto de tipo *comando*. Cualquier clase que contenga métodos de prueba ha de ser una subclase de la clase `TestCase. Un TestCase` puede definir cualquier número de métodos `testxxx()`. Para comparar el valor obtenido en la prueba con el valor esperado se invoca al método `assert()` en cualquiera de sus variantes pasando una expresión `boolean` que devuelve `true` para indicar si la prueba es correcta. Las subclases `TestCase` que contengan los métodos `testxxx()` pueden usar los métodos `setUp()` y `TearDown()` a su conveniencia para iniciar y destruir respectivamente los objetos a probar.
- Las instancias `TestCase` pueden ser *compuestas* en jerarquías `TestSuite` que invoquen de forma automática todos los métodos `testxxx()` definidos en cada instancia `TestCase`. Una `TestSuite` es una composición de otras pruebas, que pueden ser tanto instancias de tipo `TestCase` como de tipo `TestSuite`, permitiendo construir colecciones de colecciones de casos de pruebas con un nivel de profundidad arbitrario, y ejecutar todos los casos de prueba de forma automática devolviendo un único estado: prueba pasada con éxito o prueba con fallo.

En la figura 3 se muestra el diseño de **JUnit** incluyendo los patrones de diseño utilizados por los autores [Gamma y Beck, 2000/1]:

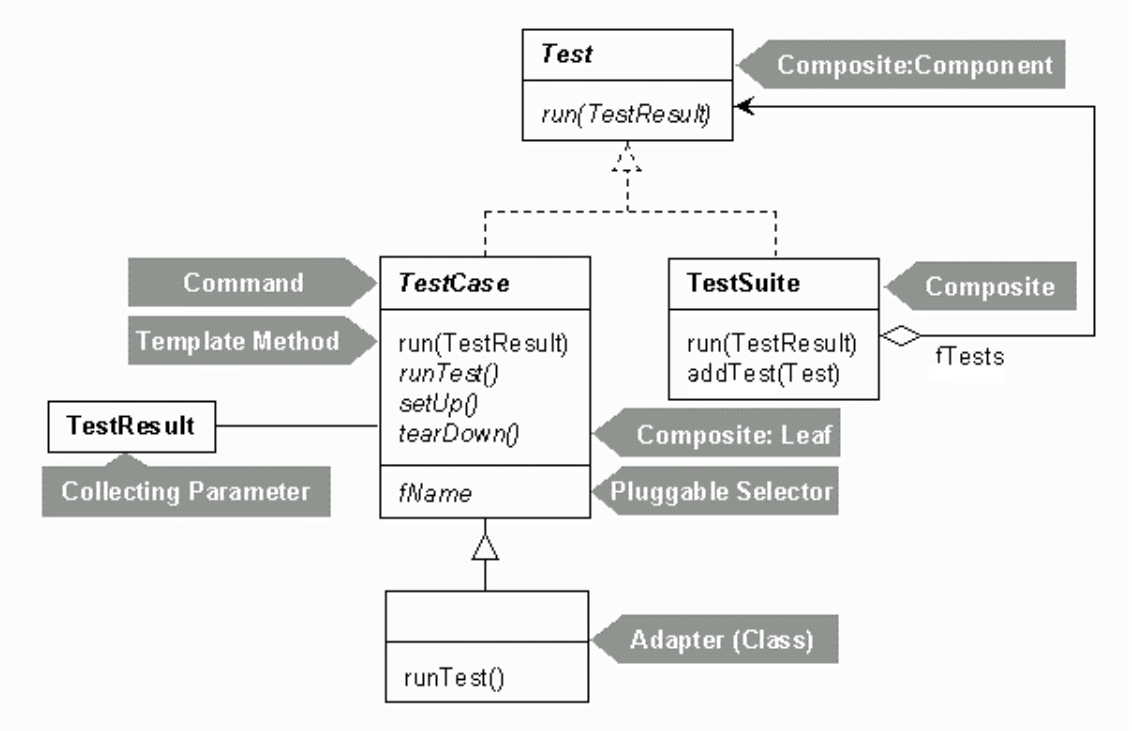

Figura 3. JUnit: Diseño [Gamma y Beck, 2000/1]

3.2.1.3. Construcción y ejecución de casos de prueba

El criterio utilizado es construir casos de prueba que ejerciten el comportamiento del componente que tenga el máximo potencial de fallo, para maximizar el rendimiento del tiempo dedicado a las pruebas.

Para construir un caso de prueba se siguen los siguientes pasos:

1. Definir una subclase de `TestCase`.
2. Sobrecargar el método `setUp()` para iniciar los objetos a probar.
3. Sobrecargar el método `TearDown()` para destruir los objetos a probar.
4. Definir uno o más métodos `testxxx()` que ejerciten los objetos a probar.
5. Definir un método `suite()` que cree una `TestSuite` conteniendo todos los métodos `testxxx()` de `TestCase`.
6. Definir un método `main()` que ejecute `TestCase`.

Para construir una colección de casos de prueba (`TestSuite`) se siguen los siguientes pasos:

1. Definir una subclase de `TestCase`.
2. Definir un método *factory* `suite()` que crea `TestSuite` conteniendo todas las instancias de `TestCase` y `TestSuite` contenidas en `TestSuite`.
3. Definir un método `main()` que ejecute `TestSuite`.

Para ejecutar tanto los casos de prueba como las colecciones de casos de prueba, **JUnit** dispone de dos interfaces, una en modo gráfico y otra en modo texto. Ambas informan del número de casos de prueba que se han ejecutado, cuántos errores o fallos se han producido y

devuelve un estado de finalización. Resulta esencial que la interfaz de usuario para la ejecución de los casos de prueba sea sencilla para que las pruebas se puedan ejecutar rápidamente.

3.2.1.4. Interacción del código de programa con el código de los casos de prueba

Estas son las recomendaciones a seguir para organizar los casos de prueba respecto al código de programa a probar:

1. Crear casos de prueba en el mismo paquete que el código a probar. Si se quiere evitar la combinación del código de aplicación con el código de prueba, se recomienda crear una réplica paralela de la estructura de directorios de la aplicación que contenga el código de prueba.
2. Por cada paquete **Java** de la aplicación definir una clase `TestSuite` que contenga todos los casos de prueba para la verificación del código del paquete. Esto permite la ejecución de una colección de casos de prueba en cualquier nivel de abstracción del programa.
3. Definir clases `TestSuite` similares que creen colecciones de casos de prueba a mayor nivel y menor nivel en los otros paquetes (y sub-paquetes) de la aplicación.
4. Asegurarse que el proceso de construcción incluye la compilación de todos los casos de prueba y colecciones de casos de prueba, para garantizar que las pruebas se ejecutan con los últimos casos de prueba generados.

3.2.1.5. Ejemplo

Para ilustrar la forma de trabajar con **JUnit** se muestra un ejemplo de aritmética con monedas de distintos países [Gamma y Beck, 2000/2]. Se difiere la conversión de una moneda a otra mediante la aplicación del tipo de cambio, por lo que las operaciones válidas serán la adición o substracción entre monedas del mismo tipo. Cuando se deban añadir o substraer monedas de distintos tipos se hace creando una bolsa de monedas que contendrá el acumulado de cada tipo de moneda.

La clase `Money` representa un valor de dinero en una única moneda.

```
class Money {
    private int fAmount;
    private String fCurrency;
    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency= currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }
}
```

La suma de dinero representado en la misma moneda sería:

```
public Money add(Money m) {
    return new Money(amount()+m.amount(), currency());
}
```

El primer caso de prueba consistiría en sumar, por ejemplo, 12 francos suizos y 14 francos suizos. Esto probaría el método `add()` de la clase `Money`:

```
public class MoneyTest extends TestCase {
    //…
    public void testSimpleAdd() {
        Money m12CHF= new Money(12, "CHF");  // (1)
        Money m14CHF= new Money(14, "CHF");
        Money expected= new Money(26, "CHF");
        Money result= m12CHF.add(m14CHF);    // (2)
        assert(expected.equals(result));     // (3)
    }
}
```

(1) Crea los objetos implicados en la prueba.

(2) Ejercita la prueba.

(3) Verifica el resultado comparándolo con el esperado.

El segundo caso de prueba verifica si dos objetos son iguales:

```
public void testEquals() {
    Money m12CHF= new Money(12, "CHF");
    Money m14CHF= new Money(14, "CHF");

    assert(!m12CHF.equals(null));
    assertEquals(m12CHF, m12CHF);
    assertEquals(m12CHF, new Money(12, "CHF")); // (1)
    assert(!m12CHF.equals(m14CHF));
}
```

En la verificación se sobrecarga el método `equals()` definido en `Object` que compara si dos objetos son iguales añadiendo la comprobación que considera que dos objetos `Money` son iguales si tienen el mismo valor pero no son el mismo objeto:

```
public boolean equals(Object anObject) {
    if (anObject instanceof Money) {
        Money aMoney= (Money)anObject;
        return aMoney.currency().equals(currency())
            && amount() == aMoney.amount();
    }
    return false;
}
```

El método `assert()` se hereda de `TestCase` y dispara un fallo que es registrado por **JUnit** cuando el argumento no es cierto. Se define el método `assertEquals()` por lo habitual de las pruebas comparando por igualdad.

Se puede evitar la duplicación de código en los casos de prueba sobrecargando el método `setUp()` para iniciar los objetos con sus valores y sobrecargando el método `tearDown()` para destruir los objetos y finalizar la prueba:

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f14CHF;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }
}
```

De esta forma los dos casos de prueba se simplifican quedando así:

```
public void testEquals() {
    assert(!f12CHF.equals(null));
    assertEquals(f12CHF, f12CHF);
    assertEquals(f12CHF, new Money(12, "CHF"));
    assert(!f12CHF.equals(f14CHF));
}

public void testSimpleAdd() {
    Money expected= new Money(26, "CHF");
    Money result= f12CHF.add(f14CHF);
    assert(expected.equals(result));
}
```

La ejecución de los casos de prueba puede hacerse de dos formas: *estática* o *dinámica*. En la estática se sobrecarga el método `runTest()` heredado de `TestCase` y se invoca al caso de prueba deseado:

```
TestCase test= new MoneyTest("simple add") {
    public void runTest() {
        testSimpleAdd();
    }
};
```

La forma dinámica usa el mecanismo de *reflexión* para implementar `runTest()`. Asume que el nombre de la prueba es el nombre del método del caso de prueba a invocar. Es más compacta pero menos segura:

```
TestCase test= new MoneyTest("testSimpleAdd");
```

Siempre que se tengan más de un caso de prueba se puede agruparlos en una colección para que sea ejecutada de una sola vez. Esto se hace definiendo un método llamado `suite()`, que es como un método `main()` especializado en ejecutar pruebas. Este sería su aspecto:

Forma estática:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(
        new MoneyTest("money equals") {
            protected void runTest() { testEquals(); }
        }
    );

    suite.addTest(
        new MoneyTest("simple add") {
            protected void runTest() { testSimpleAdd(); }
        }
    );
    return suite;
}
```

Forma dinámica:

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

En la forma dinámica simplemente se le pasa el nombre de la clase que contiene las pruebas y **JUnit** extrae los métodos de prueba automáticamente.

Para el tercer caso de prueba se utilizan bolsas de monedas que pueden almacenar simultáneamente dinero en diferentes monedas y han de permitir aritmética entre todas las combinaciones posibles de `Money` y `MoneyBag`:

```
class MoneyBag {
    private Vector fMonies= new Vector();

    MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }

    MoneyBag(Money bag[]) {
        for (int i= 0; i < bag.length; i++)
            appendMoney(bag[i]);
    }
}
```

El método `appendMoney()` añade una clase de moneda a la lista y se asegura de sumar los valores de aquellas monedas de la misma clase. En este caso se crean dos bolsas de monedas:

```
protected void setUp() {
    f12CHF= new Money(12, "CHF");
    f14CHF= new Money(14, "CHF");
    f7USD=  new Money( 7, "USD");
    f21USD= new Money(21, "USD");
    fMB1= new MoneyBag(f12CHF, f7USD);
    fMB2= new MoneyBag(f14CHF, f21USD);
}
```

La prueba de igualdad quedaría así:

```
public void testBagEquals() {
    assert(!fMB1.equals(null));
    assertEquals(fMB1, fMB1);
    assert(!fMB1.equals(f12CHF));
    assert(!f12CHF.equals(fMB1));
    assert(!fMB1.equals(fMB2));
}
```

Y el método `add()`:

```
public Money add(Money m) {
    if (m.currency().equals(currency()) )
        return new Money(amount()+m.amount(), currency());
    return new MoneyBag(this, m);
}
```

Debido a las dos representaciones `Money` y `MoneyBag` se introduce la interfaz `IMoney` para ocultar esta diferencia en el código cliente:

```
interface IMoney { public abstract IMoney add(IMoney aMoney); //… }
```

Un caso de prueba para la adición sería:

```
public void testMixedSimpleAdd() {
    // [12 CHF] + [7 USD] == {[12 CHF][7 USD]}
    Money bag[]= { f12CHF, f7USD };
    MoneyBag expected= new MoneyBag(bag);
    assertEquals(expected, f12CHF.add(f7USD));
}
```

Y una colección de casos de prueba:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testBagEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    suite.addTest(new MoneyTest("testMixedSimpleAdd"));
    suite.addTest(new MoneyTest("testBagSimpleAdd"));
    suite.addTest(new MoneyTest("testSimpleBagAdd"));
    suite.addTest(new MoneyTest("testBagBagAdd"));
    return suite;
}
```

Para saber qué ocurre si se substraen todas las monedas de una clase de una bolsa, en lugar de recorrer el código fuente del programa y repasarlo se puede simplemente escribir un nuevo caso de prueba:

```
public void testSimplify() {
    // {[12 CHF][7 USD]} + [-12 CHF] == [7 USD]
    Money expected= new Money(7, "USD");
    assertEquals(expected, fMS1.add(new Money(-12, "CHF")));
}
```

El sistema devuelve un fallo indicando que se tiene una bolsa de monedas a pesar de que sólo queda una clase de moneda en ella. Sólo queda corregir el problema y volver a pasar la prueba:

```
public IMoney addMoney(Money m) {
        return (new MoneyBag(m, this)).simplify();
}
public IMoney addMoneyBag(MoneyBag s) {
        return (new MoneyBag(s, this)).simplify();
}
private IMoney simplify() {
  if (fMonies.size() == 1)
       return (IMoney)fMonies.firstElement()
  return this;
}
```

3.2.1.6. Valoración

Como ventajas principales de **JUnit** se encuentran las siguientes:

- Con **JUnit** los casos de prueba para un método se escriben inmediatamente después de haber escrito dicho método.
- La duplicidad en el código de prueba es evitada nada más detectarse (ver en los ejemplos la introducción de `setUp()`).
- El método `suite()` permite la ejecución de nuevos casos de prueba conjuntamente con los antiguos, pudiendo elegir siempre entre la ejecución de un caso de prueba individual o una colección de ellos.

- Ayuda en la costumbre de escribir primero el caso de prueba y luego trabajar en la implementación del código (ver ejemplo de substracción de todas las monedas de una clase).

Una característica destacable en el diseño de **JUnit** es el uso exhaustivo de patrones de diseño, que suele ser habitual en entornos de trabajo maduros. Esto hace que sea más fácil de usar, pero normalmente implica que sea más difícil de modificar. Lo opuesto sucede en entornos de trabajo inmaduros, con un uso muy escaso de patrones de diseño.

Como principal inconveniente destacar las pocas características funcionales disponibles, aunque en su descargo indicar que pueden utilizarse características adicionales en extensiones al paquete, entre las que se encuentra `TestDecorator`, que permite la ejecución de código adicional antes y después de la ejecución de cada caso de prueba.

3.2.2. Entorno de pruebas unitarias de BruceEckel

3.2.2.1. Introducción

Bruce Eckel [Eckel, 2000] intenta conseguir un entorno de pruebas unitarias más sencillo que el de **JUnit**. Plantea el mismo mecanismo de automatización: construir pruebas integradas dentro del código que se crea, y ejecutar dichas pruebas como una tarea añadida cada vez que se compila. Sería como una forma de extender el compilador buscando también errores semánticos.

Insiste en la necesidad de escribir los casos de prueba antes que el código alcanzando, según él, los siguientes beneficios:

1. Describir lo que se supone que debe hacer el código.
2. Facilitar un ejemplo de cómo debería ser usado el código, mostrando todas las llamadas a métodos importantes.
3. Facilitar un camino para verificar cuando el código está terminado (cuando todas las pruebas se ejecutan correctamente).

Haciéndolo así, la prueba se convierte en una herramienta de desarrollo, no sólo en una fase de verificación que puede saltarse.

3.2.2.2. Descripción del entorno de trabajo

El proceso de construcción (*build*) consistirá en la compilación de los programas y la ejecución de pruebas unitarias. Las pruebas se integran en los archivos `makefile` y si su proceso `make` asociado ejecuta todas las pruebas con éxito, el sistema será validado, en caso contrario, los mensajes de error mostrarán las pruebas que fallaron.

La clase `UnitTest` se utiliza para crear los casos de prueba que han de ser subclases de `UnitTest`. Se comienza creando una clase `static` más interna dentro de la clase que se quiere probar. El código de prueba tiene acceso completo a la estructura interna de la clase que está siendo probada, dado que las clases interiores tienen acceso automático a todos los elementos de las clases exteriores, incluso a aquellos que son `private`:

```
//: com:bruceeckel:test:UnitTest.java
// The basic unit testing class
package com.bruceeckel.test;
import java.util.ArrayList;

public class UnitTest {
  static String testID;
  static ArrayList errors = new ArrayList();
```

```
  // Override cleanup() if test object
  // creation allocates non-memory
  // resources that must be cleaned up:
  protected void cleanup() {}
  // Verify the truth of a condition:
  protected final void assert(boolean condition){
    if(!condition)
      errors.add("failed: " + testID);
  }
}
```

Los métodos de prueba usan el método `assert()` para validar expresiones. Si la condición es cierta, la prueba tiene éxito.

Se definen los objetos de prueba creándolos como miembros de clase ordinarios de la clase de prueba, y se crea un nuevo objeto de la clase de prueba por cada método de prueba. De esta forma se evitan los posibles efectos laterales entre pruebas distintas. Esto es una diferencia respecto de **JUnit** que usa los métodos `setUp()` y `tearDown()`.

Se muestra un ejemplo con dos casos de prueba:

```
//: c02:TestDemo.java
// Creating a test
import com.bruceeckel.test.*;

public class TestDemo {
  private static int objCounter = 0;
  private int id = ++objCounter;
  public TestDemo(String s) {
    System.out.println(s + ": count = " + id);
  }
  public void close() {
    System.out.println("Cleaning up: " + id);
  }
  public boolean someCondition() { return true; }
  public static class Test extends UnitTest {
    TestDemo test1 = new TestDemo("test1");
    TestDemo test2 = new TestDemo("test2");
    public void cleanup() {
      test2.close();
      test1.close();
    }
    public void testA() {
      System.out.println("TestDemo.testA");
      assert(test1.someCondition());
    }
    public void testB() {
      System.out.println("TestDemo.testB");
      assert(test2.someCondition());
      assert(TestDemo.objCounter != 0);
    }
  }
}
```

Al ejecutarlo se obtiene la siguiente salida, siendo la mayoría líneas de traza de la ejecución:

```
test1: count = 1
```

```
test2: count = 2
TestDemo.testA
Cleaning up: 2
Cleaning up: 1
test1: count = 3
test2: count = 4
TestDemo.testB
Cleaning up: 4
Cleaning up: 3
```

La ejecución se realiza utilizando la clase `RunUnitTests` que provee el método `main()` adecuado para ello, haciendo uso significativo de la *reflexión*: el programa busca la clase de prueba y ejecuta cada prueba:

1. Sólo busca las clases más interiores declaradas en la clase actual derivadas de `UnitTest.`
2. Comprueba que sean `static`.
3. Sólo los métodos `public` sin argumentos y que devuelven tipos `void` serán considerados código de prueba.
4. Se crea una instancia del objeto de prueba y es limpiado por cada prueba.

```
//: com:bruceeckel:test:RunUnitTests.java
// Discovering the unit test
// class and running each test.
package com.bruceeckel.test;
import java.lang.reflect.*;
import java.util.Iterator;

public class RunUnitTests {
  public static void
  require(boolean requirement, String errmsg) {
    if(!requirement) {
      System.err.println(errmsg);
      System.exit(1);
    }
  }
  public static void main(String[] args) {
    require(args.length == 1,
      "Usage: RunUnitTests qualified-class");
    try {
      Class c = Class.forName(args[0]);
      // Only finds the inner classes
      // declared in the current class:
      Class[] classes = c.getDeclaredClasses();
      Class ut = null;
      for(int j = 0; j < classes.length; j++) {
        // Skip inner classes that are
        // not derived from UnitTest:
        if(!UnitTest.class.
            isAssignableFrom(classes[j]))
          continue;
        ut = classes[j];
        break; // Finds the first test class only
      }
      // If it found an inner class,
```

```
  // that class must be static:
  if(ut != null)
    require(
      Modifier.isStatic(ut.getModifiers()),
      "inner UnitTest class must be static");
  // If it couldn't find the inner class,
  // maybe it's a regular class (for black-
  // box testing:
  if(ut == null)
    if(UnitTest.class.isAssignableFrom(c))
      ut = c;
  require(ut != null,
    "No UnitTest class found");
  require(
    Modifier.isPublic(ut.getModifiers()),
    "UnitTest class must be public");
  Method[] methods = ut.getDeclaredMethods();
  for(int k = 0; k < methods.length; k++) {
    Method m = methods[k];
    // Ignore overridden UnitTest methods:
    if(m.getName().equals("cleanup"))
      continue;
    // Only public methods with no
    // arguments and void return
    // types will be used as test code:
    if(m.getParameterTypes().length == 0 &&
       m.getReturnType() == void.class &&
       Modifier.isPublic(m.getModifiers())) {
         // The name of the test is
         // used in error messages:
         UnitTest.testID = m.getName();
         // A new instance of the
         // test object is created and
         // cleaned up for each test:
         Object test = ut.newInstance();
         m.invoke(test, new Object[0]);
         ((UnitTest)test).cleanup();
    }
  }
} catch(Exception e) {
  e.printStackTrace(System.err);
  // Any exception will return a nonzero
  // value to the console, so that
  // 'make' will abort:
  System.exit(1);
}
// After all tests in this class are run,
// display any results. If there were errors,
// abort 'make' by returning a nonzero value.
if(UnitTest.errors.size() != 0) {
  Iterator it = UnitTest.errors.iterator();
  while(it.hasNext())
    System.err.println(it.next());
  System.exit(1);
}
```

```
  }
}
```

El criterio para identificar los casos de prueba es el siguiente: se le puede dar el nombre que se quiera a las clases que representan los casos de prueba. El único requisito es que sean subclases de `UnitTest`. Sólo los métodos `public` que no tienen argumentos y devuelven `void` serán tratados como casos de prueba.

Se permite incluir cualquier código de impresión en la clase de prueba, a diferencia de **JUnit** que para poder hacerse debía utilizarse una extensión (`TestDecorator`) del paquete principal.

Se puede añadir un caso de prueba a una clase derivada de una que ya tiene una clase de prueba:

```
//: c02:TestDemo2.java
// Inheriting from a class that
// already has a test is no problem.
import com.bruceeckel.test.*;

public class TestDemo2 extends TestDemo {
  public TestDemo2(String s) { super(s); }
  // You can even use the same name
  // as the test class in the base class:
  public static class Test extends UnitTest {
    public void testA() {
      System.out.println("TestDemo2.testA");
      assert(1 + 1 == 2);
    }
    public void testB() {
      System.out.println("TestDemo2.testB");
      assert(2 * 2 == 4);
    }
  }
}
```

Las pruebas de Caja Negra tratan a la clase a probar como una caja impenetrable. Sólo pueden acceder a las porciones `public` de la clase. Esta prueba se corresponde más a la prueba funcional. Para realizar pruebas de Caja Negra se debe crear la clase de prueba como una *clase global* en lugar de una clase interior. Las clases de prueba de Caja Negra se situarán en un directorio separado de la clase que se está probando. Las demás reglas son las mismas. Las pruebas de Caja Negra sirven como buen ejemplo de programación, ya que muestran el código que se debería escribir para usar la clase a probar. Se muestra un ejemplo:

```
//: c02:testable:Testable.java
package c02.testable;

public class Testable {
  private void f1() {}
  void f2() {} // "Friendly": package access
  protected void f3() {} // Also package access
  public void f4() {}
}
```

Aquí la prueba de Caja Negra está en su propio paquete estando únicamente disponibles los métodos públicos:

```
//: c02:test:BlackBoxTest.java
```

```
import c02.testable.*;
import com.bruceeckel.test.*;

public class BlackBoxTest extends UnitTest {
  Testable tst = new Testable();
  public void test1() {
    //! tst.f2(); // Nope!
    //! tst.f3(); // Nope!
    tst.f4(); // Only public methods available
  }
}
```

3.2.2.3. Valoración

Se trata de un entorno más sencillo que **JUnit**. No incluye la agrupación de casos de prueba en colecciones (`TestSuite`), por lo que la ejecución de agrupaciones de casos de prueba a voluntad puede ser más costosa, aunque se basa en la inclusión de la ejecución de las pruebas en los archivos `makefile`, por lo que en este caso sólo se ejecutarán los casos de pruebas realmente necesarios, es decir, únicamente los que afecten a módulos que se hayan modificado.

3.3. Patrones de diseño de pruebas

3.3.1. Patrones de diseño

El concepto de *patrones de diseño* se ha heredado del campo de la arquitectura. En 1977, Christopher Alexander [Alexander et al., 1977] estableció un nuevo enfoque afirmando que "el mismo conjunto de leyes determina la estructura de una ciudad, de un edificio o de una habitación". Su solución consiste en identificar problemas típicos y sus soluciones adecuadas mediante *patrones*: cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de su solución, de forma que puede utilizarse continuamente sin hacer dos veces lo mismo.

Este concepto se empezó a aplicar en el contexto de la Ingeniería de Software y más concretamente en el enfoque de programación orientada a objetos, buscando soluciones generales y por tanto reutilizables a problemas similares. Se trata de reutilizar las *ideas* pensadas por una persona para solucionar un problema, llevando el concepto de reutilización a un nivel de abstracción mayor que el de la mera reutilización del código de programación: al nivel del diseño orientado a objetos.

En [Gamma et al., 1995] se define un patrón de diseño como: "una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular." Centrándose en el mundo del diseño software clasificaron gran cantidad de patrones según su utilidad:

- Patrones sobre creación: sobre cómo crear instancias de objetos: factoría abstracta, *builder*, prototipo, *singleton*.
- Patrones estructurales: sobre cómo agrupar y organizar objetos: *bridge*, *adapter*, *proxy*, *facade*, *flyweight*.
- Patrones de comportamiento: sobre cómo se relacionan los objetos entre sí en tiempo de ejecución: *command*, *interpreter*, *iterator*, *mediator*, *observer*.

Definieron una estructura básica o plantilla (*pattern template*) para describir las partes de un patrón de diseño:

- Nombre y clasificación: el nombre ha de ser representativo e identificarlo unívocamente.
- Problema: descripción del problema que soluciona.
- Motivación: ilustra un escenario de aplicación del patrón mostrando previamente una solución sin aplicarlo para observar las ventajas de hacerlo.
- Aplicabilidad: situaciones en las que se puede aplicar.
- Estructura: Descripción gráfica de la estructura del patrón.
- Participantes: clases y objetos involucrados.
- Colaboraciones: describe la comunicación entre objetos.
- Ejemplo: ejemplo de aplicación. Ha de ser lo suficientemente simple para que sea comprendido rápidamente y lo suficientemente completo como para poder explicar todas las características del patrón con dicho ejemplo.
- Patrones relacionados.

En el caso del lenguaje **Java**, la mayoría de patrones se pueden encontrar en las clases base.

Otra de las ventajas del uso de patrones de diseño además de la reutilización es en la ayuda en la documentación de un sistema.

Los problemas principales del trabajo con patrones de diseño es que debido a su grado de abstracción y generalidad muchos patrones no son comprensibles y que las soluciones a veces no son evidentes.

3.3.2. Diseño de pruebas basado en patrones

3.3.2.1. Plantilla de patrones de diseño de pruebas

Siguiendo la técnica de los patrones de diseño, Robert V. Binder [Binder, 2000] define la siguiente plantilla de patrones de diseño de pruebas (*Test Design Pattern Template*):

- Nombre: una palabra o frase que identifica el patrón y sugiere su enfoque general.
- Objetivo: ¿Qué problema de diseño de pruebas resuelve este patrón? ¿Cuál es la estrategia de pruebas? Dar una descripción muy breve.
- Contexto: ¿Bajo qué circunstancias se aplica este patrón? ¿A qué clase de entidades software? ¿A qué alcance? Esta sección se corresponde con las de “motivación”, “aplicabilidad” de los patrones de diseño.
- Modelo de Fallo: ¿Qué clase de defectos busca este patrón? El fallo debe alcanzarse a partir de los datos de entrada de la prueba y el estado del sistema, debe producir resultados incorrectos y ha de ser propagado a la salida de tal forma que sea observable por la persona que realiza la prueba.
- Estrategia: esta sección aclara como se ha de generar e implementar la colección de casos de prueba. Tiene cuatro subsecciones obligatorias:
  - Modelo de pruebas: define una representación de las responsabilidades y/o implementación que son el objetivo del diseño de pruebas.
  - Procedimiento de prueba: define un algoritmo, técnica o heurística por la que se generan los casos de prueba del modelo.

  - Comprobación: define el algoritmo, la técnica o heurística por la cual los resultados actuales de un caso de prueba se evalúan como pasado con éxito/ no pasado.
  - Automatización: discute los enfoques automáticos para la generación de casos de prueba, ejecución de pruebas y evaluación de la ejecución de las pruebas. Se suele mostrar con ejemplos.
- Criterio de entrada: lista de precondiciones para una prueba efectiva y eficiente con este patrón. Una implementación no ha de ser probada hasta que no esté *lista* para ser probada (*test-ready*[1]).
- Criterio de salida: criterio objetivo que ha de alcanzarse para dar por completa la prueba con este patrón. Por ejemplo, cobertura de sentencias a nivel de método.
- Consecuencias: prerrequisitos generales, costes, beneficios, riesgos y consideraciones en el uso de este patrón.
- Usos conocidos: ¿cuáles son los usos conocidos de este patrón de diseño? ¿Cuáles son los usos conocidos de los modelos de prueba y estrategias incorporadas en este patrón? ¿Cuáles son la eficiencia y la efectividad de este patrón o estrategias similares establecidas por estudios empíricos?
- Patrones relacionados: patrones de diseño similares o complementarios.

Esta plantilla de patrones de diseño de pruebas se centra en los aspectos fundamentales del diseño de pruebas: elección de la estrategia de pruebas apropiada al caso particular, tipos de defectos que se buscan en la fase de pruebas, cómo construir una colección de casos de prueba, la automatización de la prueba, los criterios de entrada y salida y las ventajas e inconvenientes.

3.3.2.2. Uso de patrones de diseño de pruebas

El uso de patrones de diseño de pruebas no modifica sustancialmente las tareas básicas de los procesos de prueba. Los pasos básicos para utilizar un patrón de pruebas son los siguientes:

1. En las fases iniciales del proceso de desarrollo, seleccionar los patrones de pruebas que se correspondan con el campo de aplicación y la estructura del sistema en desarrollo y sus partes.
2. Desarrollar el modelo de pruebas para la implementación a probar.
3. Instrumentar el trabajo de desarrollo y prueba de tal manera que la colección de casos de prueba sea aplicada a una implementación que haya alcanzado el umbral mínimo de operatividad tal y como se haya establecido en el criterio de entrada.
4. Generar la colección de casos de prueba aplicando el procedimiento de pruebas al modelo de pruebas.
5. Desarrollar la implementación de la automatización de las pruebas.

[1] El término *Test-ready* fue acuñado por Bob Poston y significa que el diseño de pruebas o la ejecución de las pruebas pueden comenzar o continuar sin interferencias debido a especificaciones y/o implementaciones incompletas o a la ausencia de ellas. La Evaluación de la Disposición para las Pruebas (*Test Readiness Assessment*) determina la disposición de un producto para ser probado y es una actividad definida por el SEI en el marco del CMM [Paulk et al., 1993].

6. Ejecutar y evaluar las pruebas. Si no se alcanza la cobertura recomendada, revisar la colección de casos de prueba.

3.3.2.3. Clasificación de patrones de diseño de pruebas

En la Tabla 1 se muestra la clasificación propuesta en [Binder, 2000] de patrones de diseño de pruebas.

Scope	**Pattern Name**	**Intent**
Method Scope	Category-Partition	Design a test suite based on input/output analysis.
	Combinational Function Test	Design a test suite for behaviors selected by combinational logic.
	Recursive Function Test	Design a test suite for a recursive method.
	Polymorphic Message Test	Design a test suite for a client of polymorphic server.
Class Scope	Invariant Boundary Test	Identify test vectors based on the class invariant.
	Non-modal Class Test	Design a test suite for a class without sequential constraints.
	Modal Class Test	Design a test suite for a class with sequential constraints.
	Quasi-Modal Class Test	Design a test suite for a class with content-determined sequential constraints.
Class Scope Integration	Small Pop	Order of code/test at method/class scope.
	Alpha-Omega Cycle	Order of code/test at method/class scope.
Flattened Class Scope	Polymorphic Server Test	Design a test suite to check LSP compliance of a polymorphic server hierarchy.
	Modal Hierarchy Test	Design a test suite for a hierarchy of modal classes.
Reusable Components	Abstract Class Test	Develop and test an implementation of an interface.
	Generic Class Test	Develop and test an implementation of a parameterized class.
	New Framework Test	Develop and test a demo application of new framework.
	Popular Framework Test	Test changes to widely used framework.
Subsystem	Class Association Test	Design a test suite based on class associations.
	Round-trip Scenario Test	Design a test suite for aggregate state-based behavior.
	Controlled Exception Test	Design a test suite to verify exception handling.
	Mode Machine Test	Design a test suite based on sequentially-constrained stimulus-response scenarios.
Integration	Big Bang Integration	Try everything at the same time.
	Bottom up Integration	Integration by dependencies.
	Top Down Integration	Integration by control hierarchy.
	Collaboration Integration	Integration by cluster scenarios.
	Backbone Integration	Hybrid integration of subsystems.
	Layer Integration	Integration for layered architecture.
	Client/Server Integration	Integration for client/server architecture.
	Distributed Services Integration	Integration for distributed architecture.
	High Frequency Integration	Build and test at frequent, regular intervals.
Application Scope	Extended Use Case Test	Develop testable use cases, design a test suite to cover application input-output relationships.
	Covered in CRUD	Exercise all basic operations.
	Allocate Tests by Frequency	Allocate system test effort to maximize operational reliability.
Regression Test	Retest All	Rerun all tests.
	Retest Risky Use Cases	Rerun tests of risky code.
	Retest Profile	Rerun tests by frequency of use.
	Retest Changed Code	Rerun tests for code that depends on changes.
	Retest Within Firewall	Rerun tests for code that is impacted by changes.

Tabla 1: Patrones de Diseño de Pruebas [Binder, 2000].

La Tabla 2 muestra una clasificación de patrones de diseño para la automatización de las pruebas.

Capability	Pattern Name	Intent
Built-in Test	Percolation	Perform automatic verification of super/subclass contracts.
Test Cases	Test Case/TestSuite Method	Implement a test case or a test suite as a method.
	Catch All Exceptions	Test driver generates and catches IUT's exceptions.
	Test Case /Test Suite Class	Implement test case or test suite as an object of class TestCase.
Test Control	Server Stub	Use a stub implementation of a server object for greater control.
	Server Proxy	Use a proxy implementation of a server object for greater control.
API/Class Drivers	TestDriver Super Class	Use an abstract superclass for all test drivers.
	Percolate the Object Under Test	Pass the object under test to driver.
	Symmetric Driver	Driver hierarchy is symmetric to hierarchy of classes under test.
	Subclass Driver	Driver is a subclass.
	Private Access Driver	Driver uses encapsulation avoiding features.
	Test Control Interface	Driver uses interface extension features.
	Drone	Driver is a mixin.
	Built-in Self Test	Driver is implemented as part of an application class.
Test Execution Control	Command Line Test Bundle	Code and build a test executable to be run from a command line or console.
	Incremental Testing Framework	Test suites are based on a simple framework that supports incremental development.
	Fresh Objects	Test environment with registration by and interface to built-in test in all application objects.

Tabla 2: Patrones de Diseño para la Automatización de las Pruebas [Binder, 2000].

Por último en la Tabla 3 se muestra una clasificación de patrones para los resultados esperados para un caso de prueba, denominados por el autor *Test Oracle Micro-Patterns*.

Oracle Patterns (micro-pattern schema)		
Approach	**Pattern Name**	**Intent**
Judging	Judging	The tester evaluates pass/no-pass by looking at the output on a screen, a listing, using a debugger, or another suitable human interface.
Pre-Specification	Solved Example	Develop expected results by hand or obtain from a reference work.
	Simulation	Generate exact expected results with a simpler implementation of the IUT (e.g., a spreadsheet.)
	Approximation	Develop approximate expected results by hand or with a simpler implementation of the IUT.
	Parametric	Characterize expected results for a large number of items by parameters
Gold Standard	Trusted System	Run new test case against a trusted system to generate results.
	Parallel Testing	Run the same live inputs into the IUT and a trusted system. Compare the output.
	Regression Testing	Run an old test suite against a partially new system.
	Voting	Compare the output of several versions of the IUT.
Organic	Smoke Test	Use the basic operability checks of the run time environment.
	Reversing	Reverse the IUT's transformation.
	Built-in Test	Don't develop expected results. Implement assertions that define valid and invalid results.
	Executable Specification	Actual input values and output values are used to instantiate the parameters of an executable specification. A specification checker will reject an instantiation which is inconsistent, indicating incorrect output.
	Built-in Check	Compare expected and actual total, checksum, or similar encoding.
	Generated Implementation	Generate a new implementation from a specification, compare output from the IUT and the generated IUT for the same test case.
	Different But Equivalent	Generate message sequences that are different but should have the result; run on separate objects and compare for equality.

Tabla 3: Patrones de Diseño para los valores esperados de los casos de prueba [Binder, 2000].

3.3.2.4. Ejemplo de patrones de diseño de pruebas

Se selecciona como ejemplo el patrón “Partición por Categoría” (*Partition-Category*), un patrón de pruebas sobre métodos. El objetivo del patrón es diseñar una colección de casos de prueba basada en análisis de entrada/salida. Se basa en la teoría de las *clases de equivalencia*, introducida en los apartados 2.3.1. y 2.3.2. del presente trabajo. En dichos apartados se describen los conceptos de *Partición equivalente* y *Análisis de valores límite* relacionados con este patrón. El patrón define una forma sistemática de dividir el dominio de entrada:

NOMBRE: Partición por Categoría

CONTEXTO: Validar un método no sencillo que no es una función combinacional ni recursiva. Es un método para generar y especificar casos de pruebas funcionales o de caja negra.

MODELO DE FALLO: Fallos relacionados con valores límite implementados incorrectamente.

ESTRATEGIA:

1. Especificar `fv`, `fo`, `fe`. Precondiciones y Postcondiciones.
    - `fv`: Parametros x Estado Abstracto -/-> Resultados
    - `fo`: Parametros x Estado Abstracto -/-> Estado Abstracto
    - `fe`: Parametros x Estado Abstracto -/-> Excepcion
    - Precondicion: `pre(m) = pre(fv)` ó `pre(fo)` ó `pre(fe)`
    - Postcondicion: `post(m) = {pre(fi)} m {post(fi)}` con `i=v,o,e`
2. Identificar las variables de decisión y los resultados.
3. Identificar los valores límite para cada variable de decisión (Invariante).
4. Identificar los puntos dentro del dominio y fuera del dominio de esos valores límite.
5. Construir la matriz de pruebas de dominio (Tabla de decisión), donde:
    - Si son pocas variables y valores límite se pueden hacer todas las combinaciones entre las variables, eliminando las que no se puedan dar.
    - Si no, aplicar el criterio 1x1.

CRITERIO DE SELECCIÓN 1x1:

1. Un punto válido y un punto inválido por cada relación condicional (>, <, >=, <=).
2. Un punto válido y dos inválidos por cada condición de igualdad.
3. Un punto válido y uno inválido por cada tipo no escalar (booleanos, complejos, cadenas de caracteres).
4. Construir la matriz de pruebas de dominio, donde sólo un punto válido o inválido aparecerá en cada caso de prueba.

MATRIZ DE PRUEBAS DEL DOMINIO:

La matriz de pruebas del dominio o tabla de decisión se representa de la siguiente manera:

VARIABLE	VALOR	CASO 1	CASO 2	.....	CASO n
X_1	Válido				
	Inválido				
	Representativo				
.....					
X_n	Válido				
	Inválido				
	Representativo				
ACCIONES (Resultado esperado)				.....	

Tabla 4: Matriz de pruebas del dominio [Binder, 2000].

4. XML como soporte para la verificación automática de programas

4.1. Introducción

Una computadora sólo es capaz de ejecutar órdenes en el código binario que comprende. Para que la escritura de órdenes fuera más fácilmente realizable por las personas se idearon los *lenguajes de programación de alto nivel* que mediante una serie de reglas gramaticales y sintácticas generalmente más sencillas que las de un lenguaje natural abstraían el código binario final en instrucciones secuenciales generalmente escritas en inglés con una variedad mayor o menor de operadores, operandos, bucles y condiciones dependiendo del lenguaje utilizado. Debido a que un programa de estas características no era ejecutable directamente en la computadora, se construyeron herramientas de traducción (compiladores e intérpretes) que basándose en las reglas gramaticales y sintácticas de cada lenguaje particular, leen ficheros de texto plano y traducen estas instrucciones en código binario ejecutable directamente en la computadora.

La base de estos programas la constituyen las *expresiones regulares y las gramáticas*. Las expresiones regulares describen combinaciones de caracteres que forman *tokens* o elementos del lenguaje, y las gramáticas describen cómo se pueden combinar los *tokens* para formar construcciones de más alto nivel que constituirán las sentencias del lenguaje. Existen utilidades como **Lex** (para la parte léxica) y **Yacc** (para la parte sintáctica) que permiten automatizar las tareas necesarias de reconocimiento del lenguaje.

Mediante este procedimiento el compilador obtiene un *árbol de sintaxis abstracta* para el programa dado que refleja de forma más directa su estructura, y permite la traducción a código binario y la optimización de dicha traducción de forma más sencilla que si se hiciera directamente del fichero de texto plano original, aunque éste sea más cercano al lenguaje natural y por tanto más fácil de leer y entender por las personas.

La principal limitación de la representación del código fuente mediante un fichero de texto plano es la necesidad de la realización de un *análisis sintáctico* posterior para hacer visible la estructura del programa y de esta manera accesible a otras herramientas de ingeniería de

software tales cómo generadores de casos de prueba, obtención de métricas, etc. Esta aproximación implica la realización del análisis sintáctico en cada herramienta a utilizar.

Además, la inclusión de un analizador en una herramienta liga a la misma al lenguaje de programación específico, reduciendo su generalidad. Al no haber una estandarización en la representación externa estructurada del programa fuente la cooperación entre las distintas herramientas se hace difícil.

4.2. XML para representar código fuente

XML es un lenguaje de marcas extensible estandarizado (*eXtensible Markup Language*) [Bray et al., 2000] basado en el lenguaje de marcas generalizado estándar SGML (*Standard Generalized Markup Language*) [ISO, 1986] que siendo más sencillo que este, mantiene la compatibilidad. A diferencia de HTML (*HyperText Markup Language*) [W3C, 1999/5] permite la utilización de etiquetas de marcas definidas por el usuario para adaptarse mejor al contenido del documento a manejar. Otra diferencia principal entre XML y HTML es que el primero está orientado al contenido del documento mientras que el segundo está más orientado a la presentación.

Un ejemplo sencillo de representación de un automóvil en formato XML podría ser el siguiente:

```
<Automovil>
  <Numero de chasis> HGJY3256 </Numero de chasis>
  <Marca> Audi </Marca>
  <Modelo> A2 </Modelo>
  <Propietario>
        <Nombre> Luis Rodriguez </Nombre>
        <Direccion> La Platina, 2 </Direcion>
        <Codigo Postal> 37009 </Codigo Postal>
        <Localidad> Salamanca </Localidad>
        <Provincia> Salamanca </Provincia>
  </Propietario>
</Automovil>
```

Esta representación sólo contiene información acerca de estructura y contenido de los datos, a diferencia de su equivalente en HTML, en el que sólo se expresa cómo serán mostrados los datos en pantalla:

```
<H1> Automovil </H1>
<H2> Numero de chasis: HGJY3256 </H2>
<H2> Marca: Audi <H2>
<H2> Modelo: A2 <H2>
<H2> Propietario: <H2>
<p>  Luis Rodriguez
     La Platina, 2
      37009 Salamanca (Salamanca)
</p>
```

En el caso XML, la apariencia de los datos se deja que la decida la aplicación que leerá el documento.

Estableciendo una analogía respecto al caso de representar programas de ordenador, si bien la representación en texto plano puede resultar suficiente para ser leída e interpretada por una persona en la mayoría de los casos, la representación XML es mucho más precisa y aprovechable para su uso por herramientas de ingeniería de software pues pone a disposición de éstas hasta el más mínimo detalle de su estructura y contenido.

Greg J. Badros ha diseñado el lenguaje de marcas **JavaML** [Badros, 2000] para la representación de código fuente Java. En la figura 4 se muestra la representación XML elegida por el autor de entre las numerosas posibles para este pequeño programa Java de ejemplo:

```
import java.applet.*;
import java.awt.*;
public class FirstApplet
             extends Applet {
  public void paint(Graphics g) {
       g.drawString("FirstApplet", 25, 50);
  }
}
```

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
3
4 <java-source-program name="FirstApplet.java">
5  <import module="java.applet.*"/>
6  <import module="java.awt.*"/>
7  <class name="FirstApplet" visibility="public">
8      <superclass class="Applet"/>
9      <method name="paint" visibility="public" id="meth-15">
10       <type name="void" primitive="true"/>
11       <formal-arguments>
12         <formal-argument name="g" id="frmarg-13">
13           <type name="Graphics"/></formal-argument>
14       </formal-arguments>
15       <block>
16         <send message="drawString">
17          <target><var-ref name="g" idref="frmarg-13"/></target>
18           <arguments>
19             <literal-string value="FirstApplet"/>
20             <literal-number kind="integer" value="25"/>
21             <literal-number kind="integer" value="50"/>
22           </arguments>
23         </send>
24       </block>
25     </method>
26   </class>
27 </java-source-program>
```

Figura 4: FirstApplet.java en formato JavaML.

JavaML incluye un *unparser* XML que ha sido añadido al compilador **Jikes** de IBM [IBM, 1999]. El método `XMLUnparse` analiza programas escritos en **Java** y construye su equivalente XML siguiendo el DTD de JavaML.

4.3. Extensiones de XML

El *World Wide Web Consortium* (W3C) ha desarrollado nuevos estándares que mejoran y extienden la especificación XML. De especial utilidad para el desarrollo de herramientas de Ingeniería de Software son las siguientes:

- La Especificación de Enlazado XML: **Xlink** [W3C, 1999/1] y la Especificación de punteros Extendidos: *XPointer* [W3C, 1998/1], que permiten manejar enlaces bidireccionales y multicaminos, pueden ser de utilidad, por ejemplo, para construir un sistema de navegación entre árboles de clases de un paquete de software, facilitando la comprensión de las relaciones de herencia entre clases.

- El Lenguaje de Hojas de Estilo Extensible: **XSL** [W3C, 1999/2] sirve para obtener vistas o representaciones de un documento XML para ser leído por una persona. En el caso que se presenta, por ejemplo, se puede utilizar para obtener código fuente **Java** a partir de su especificación en el lenguaje **JavaML**, o para obtener un informe del número de casos de prueba ejecutados y los resultados de dichas pruebas. La especificación XSLT [W3C, 1999/4] o de Transformaciones XSL permite transformar un documento XML en otro. Esta especificación se podría utilizar, por ejemplo, para transformar un diseño UML [Booch et al., 1999] en clases **Java** escritas en **JavaML**.
- El Lenguaje de Consultas XML: **XML-QL** [W3C, 1998/2] permite la extracción y transformación de datos de documentos XML semi-estructurados. Se basa en dos construcciones principales: un patrón para identificar los elementos a extraer y una regla de construcción que especifica qué transformaciones realizar con los elementos extraídos.
- El Modelo de Objetos de Documento: **DOM** [W3C, 1999/3] define la forma de acceder y modificar los elementos de documentos XML. Sería la forma natural de representar en memoria los árboles de sintaxis abstracta de los programas.

5. Herramientas de ayuda para entornos de verificación con XML

5.1. Casos de prueba escritos en XML: JXUnit

Los datos de prueba son la información fundamental con la que se trabaja en cualquier clase de prueba. Desde la prueba de un método hasta las pruebas de aceptación por parte del cliente, en las pruebas están involucrados los datos de entrada, los datos de salida esperados y la comparación entre los datos esperados y los efectivamente obtenidos.

JXUnit [JXUnit, 2001] es una herramienta de código abierto basada en *scripts* que permite definir casos de prueba en formato XML, separando los datos de prueba del código de prueba. Utiliza el entorno de pruebas **JUnit** para la ejecución de los casos de prueba. A continuación se explica de forma condensada el funcionamiento de **JXUnit**.

5.1.1. El lenguaje de marcas: JXU

Una prueba se divide en una serie de pequeños pasos:

1. Crear los datos de prueba.
2. Ejecutar el código de prueba.
3. Comparar los resultados contra los valores esperados.

Se definen propiedades de la prueba que serán utilizadas para pasar los datos entre los pasos de prueba. Una propiedad cualquiera tendrá un nombre y un valor. Se utiliza el lenguaje de marcas XML, Java XML Unit (JXU) para describir casos de prueba y la serie de pasos de prueba a efectuar con un caso de prueba. Un caso de prueba descrito con **JXUnit** tendría un aspecto similar a este:

```
<jxu>
        <set name="input"
                value="123"/>
        <eval stepClass="DummyTestStep"/>
        <isEqual name="output"
```

```
                value="123"
                message="Output was not 123"/>
</jxu>
```

Este ejemplo crea una propiedad llamada `input` con el valor "`123`". Ejecuta "`DummyTestStep`" y comprueba que se haya creado la propiedad llamada `output` y que además sea igual a "`123`". Si esto no es cierto, la prueba falla con el mensaje "`Output was not 123`".

5.1.2. El entorno de pruebas: JXU y QJML

El entorno global de pruebas gira alrededor de la clase `net.sourceforge.jxunit.JXTestCase`. Esta clase extiende a la clase del entorno **JUnit** `junit.framework.TestCase` y tiene una clase `suite` que es llamada por **JUnit**.

Una segunda clase, `net.sourceforge.jxunit.JXProperties` es la clase contenedora de las propiedades de los casos de prueba. Esta clase extiende a la clase `java.util.HashMap`.

Los archivos JXU son convertidos en una colección de objetos por cada paso de prueba (*TestStep*) mediante las clases siguientes:

```
set       net.sourceforge.jxunit.JXTestSet
eval      net.sourceforge.jxunit.JXEval
isEqual   net.sourceforge.jxunit.JXIsEqual
```

Estas clases implementan la interfaz `net.sourceforge.jxunit.JXTestStep`:

```
package net.sourceforge.jxunit;

public interface JXTestStep
{
        public void eval(JXTestCase testCase)
             throws Throwable;
}
import net.sourceforge.jxunit.*;
```

La clase que ejecuta el caso de prueba también implementa `JXTestCase`. El caso del ejemplo sería el siguiente:

```
public class DummyTestStep implements JXTestStep
{
        public void eval(JXTestCase testCase)
             throws Throwable
        {
             JXProperties properties=testCase.getProperties();
             Object data=properties.get("input");
             properties.put("output",data);
        }
}
```

La relación entre el archivo JXU y las clases **JXUnit** se describe mediante un documento escrito en el lenguaje de marcas **QJML** que será tratado en mayor profundidad en el apartado 5.2. Aquí se muestra un ejemplo de cómo se describe la relación entre el elemento `eval` y la clase `JXEval`:

```
<bean tag="eval">
        <rem>Create and run a test step.</rem>
        <implements>testStep</implements>
```

```
<targetClass>net.sourceforge.jxunit.JXEval</targetClass>
        <attributes>
                <item coin="stepClass">
                        <field name="stepClass"/>
                </item>
        </attributes>
</bean>

<text tag="stepClass">
    <rem>The fully qualified class name of the test step</rem>
</text>
```

QJML también permite simplificar el trabajo con estructuras de datos complejas transformando en objetos las representaciones hechas en XML mediante las reglas definidas en el documento **QJML** correspondiente. He aquí un ejemplo:

```
<dataList>
        <dataItem>abc<dataItem>
        <dataItem>def<dataItem>
        <dataItem>g<dataItem>
        <dataItem>hi<dataItem>
</dataList>
```

Este sería el archivo QJML:

```
<qjml root="dataList">
        <bean tag="dataList">
                <rem>A list of String objects</rem>
                <targetClass>java.util.ArrayList</targetClass>
                <elements>
                     <item coin="dataItem" repeating="true">
                             <identity kind="list"/>
                     </item>
                </elements>
        </bean>
        <text tag="dataItem">
                <rem>A simple text value</rem>
        </text>
</qjml>
```

Los archivos QJML han de ser transformados en su forma compilada QIML. El archivo `jxu` correspondiente que relaciona el archivo que contiene la estructura de datos y la forma de convertirlos en objetos de prueba sería:

```
<jxu>
        <set name="input"
                file="myData.xml" schema="mySchema.qiml"/>
        <save name="input"
                file="genData.xml" schema="mySchema.qiml"/>
        <eval stepClass="DummyTestStep"/>
        <ifEqual converse="true"
                     name="output" schema="mySchema.qiml"
                     file="genData.xml">
                <save name="output" schema="mySchema.qiml"
                      file="genData_.xml"/>
                <fail>Dummy Test Failure: genData.xml</fail>
</jxu>
```

Los pasos son los siguientes:

1. El paso `set` convierte el contenido del archivo `myData.xml` en un objeto llamado `input` a partir de las reglas contenidas en `mySchema.qiml`.
2. El paso `save` crea el archivo `genData.xml` a partir del objeto de entrada para asegurarse que la comparación final se efectúa en la forma exacta en que el objeto `dataList` es convertido a XML.
3. El paso `eval` ejecuta la prueba llamando a `DummyTestStep`.
4. El paso `isEqual` primero convierte el objeto de salida en una cadena XML a partir de las reglas contenidas en `mySchema.qiml` y después compara dicha cadena con el contenido del archivo `genData.xml`.

El paso `step` también se puede utilizar para conservar una copia de los datos de salida, que será de especial utilidad si la prueba falla, como se muestra en el ejemplo anterior.

5.1.3. El contexto de pruebas: JXUC

Lo visto en el apartado anterior sirve para ejecutar un caso de pruebas. El contexto de pruebas permite la ejecución de las pruebas más de una vez, usando múltiples hilos de ejecución y/o con múltiples archivos de pruebas. El contexto se define en un archivo llamado `test.jxuc` que debe residir en el mismo directorio que el archivo `test.jxu`. A continuación se muestra un ejemplo:

```
<jxuc>
        <directoryScan dir="testDirectory">
                <includeFiles regexp=".txt$"/>
                <excludeFiles regexp="_.txt$"/>
        </directoryScan>
</jxuc>
```

Los nombres de los archivos de pruebas a incluir o a excluir son filtrados mediante expresiones regulares, y por cada uno de los seleccionados es ejecutada una vez la prueba definida en el archivo `test.jxu`.

5.1.4. Valoración

JXUnit se centra en separar los datos de prueba del código de prueba. Las ventajas principales de este enfoque son:

- Los datos de prueba pueden ser modificados sin necesidad de modificar el código.
- Los datos pueden ser validados, para reducir el número de “fallos falsos”.
- Los datos pueden ser generados de forma externa (que será nuestro caso, obteniéndolos del repositorio de programas XML) o capturados de un proceso de producción.

5.2. Quick

Quick [Quick, 2001] es una utilidad para generar y procesar archivos en formato XML. Estructuras arbitrarias de objetos pueden ser convertidas en árboles de elementos XML. Recíprocamente, documentos XML pueden ser convertidos en estructuras de objetos. **Quick** es un sistema de modelado de datos para Java. Soporta la *herencia*, incluyendo elementos *interface* y *abstractos*. Permite al programador un control bastante alto sobre la generación de código

fuente mediante archivos de configuración. **Quick** trabaja con componentes **Java** (**Java Beans**[2]) y editores de propiedades de componentes, que permiten el uso de tipos de datos de usuario (clases **Java**) durante el proceso de atributos XML y elementos simples con contenido de texto.

Quick proporciona también un entorno de trabajo (el paquete *Open Conversion Model* descrito en el siguiente apartado) para realizar transformaciones de datos simples y complejas.

El lenguaje de marcas *Quick Java Markup Language* (QJML) se utiliza en el entorno **JXUnit** para describir la relación entre el archivo JXU y las clases **JXUnit**.

5.2.1. Componentes de Quick

Quick consta de los siguientes modelos de datos: *QJML, QDML, QIML, OCM* y utilidades de conversión entre los distintos modelos.

Quick Java Markup Language (QJML) es un lenguaje de marcas XML para describir modelos **Java** y cómo convertir dicho modelo del formato XML en objetos Java. Elementos de QJML son: clases Java, interfaces, campos y propiedades de componentes. QJML también puede ser usado para generar clases Java y partiendo del grafo de un objeto extraer los datos para crear un documento XML.

Quick Data Markup Language (QDML) es una forma simplificada de QJML del que se han eliminado todas las referencias a clases interfaces, campos y propiedades. Soporta la herencia de **Java**. Describe un modelo de datos Java sin atarse a una implementación particular. Esto es de particular utilidad para el intercambio de datos en internet en el que el emisor y el receptor comparten el modelo de datos pero no necesariamente una misma implementación.

Quick Internal Markup Language (QIML) es la forma compilada de QJML. Los elementos de un documento QIML se usan para configurar los objetos del motor de **Quick** que a su vez se usan para convertir los documentos XML en objetos Java y viceversa.

Open Conversion Model (OCM) es un sistema de composición de elementos para definir transformaciones de datos complejas. Utiliza el patrón de diseño *tree-factory* [JXQuick, 2001] para encapsular la creación de árboles de objetos. La *factoría* centraliza el problema de la creación de nuevos tipos de objetos, simplificando la creación de los mismos. En primer lugar el documento es convertido en un árbol de objetos factoría que son usados para generar el contexto operativo para las transformaciones de datos. En segundo lugar, utiliza el árbol de objetos del modelo de datos para crear los objetos de control.

OCM también se utiliza para escribir algunas de las utilidades de **Quick**.

Por último, **Quick** incorpora un conjunto de utilidades de conversión entre los distintos modelos de datos que manipula. Esta es una lista de algunas de ellas:

- `qjml2java`: convierte un archivo QJML a un conjunto de archivos en código java
- `qjml2qiml`: convierte un archivo QJML en un archivo QIML
- `dtd2qdml`: convierte un DTD XML en un esquema QDML
- `qdml2dtd`: convierte un esquema QDML en un DTD XML
- `qdml2qjml`: convierte un esquema QDML en un esquema QJML
- `qjml2qdml`: convierte un esquema QJML en un esquema QDML
- `dtd2xml`: convierte un archivo DTD a un archivo XML

[2] Pequeños programas de aplicación que tienen una función específica [SUN, 1997].

5.2.2. Ejemplo en QJML

A continuación se presenta un ejemplo con dos componentes **Java**, uno de los cuales extiende el otro:

```
public class A
{
        public String s="";
        public int i=0;
}

public class B extends A
{
        public float f=0.0;
}
```

Esta es su correspondiente representación en QJML:

```
<bean tag="a">
        <targetClass>A</targetClass>
        <attributes>
                <item coin="sam">
                        <field name="s">
                <item>
                <item coin="index">
                        <field name="i">
                <item>
        </attributes>
</bean>

<text tag="sam"/>
<text tag="index" type="int"/>

<bean tag="b">
        <extends>a</extends>
        <targetClass>B</targetClass>
        <attributes>
                <item coin="fred">
                        <field name="f">
                <item>
        </attributes>
</bean>

<text tag="fred" type="float"/>
```

Y este es un ejemplo del elemento *b* en XML:

```
<b sam="hi!" index="24" fred=".3"/>
```

5.2.3. Valoración

Quick representa una forma sencilla de transformar archivos XML en una estructura de objetos mediante el esquema de relación QJML obteniendo clases de forma automática que pueden convivir con aquéllas que se hayan creado a mano. Tiene una API muy sencilla con unos pocos métodos estáticos lo que facilita su uso.

Se pone énfasis en la importancia de diseñar modelos de datos en XML, en vez de derivarlos del diseño del programa, independizándolos de las implementaciones concretas, muy útil para el intercambio de datos entre emisores y receptores de datos heterogéneos, por ejemplo,

en internet. **Quick** se puede utilizar para validar la estructura y contenido de un documento XML.

Además, facilita la conversión de documentos XML en estructuras de objetos específicos de aplicación y viceversa, lo que puede ser aprovechado, por ejemplo, por otras herramientas como **JXUnit** para la transformación de casos de prueba formulados en XML en objetos para ejecutar dichos casos de prueba. Se facilita así el nexo de unión entre los casos de prueba y el código de prueba que puede separarse en archivos distintos, con las ventajas ya señaladas en el apartado dedicado a **JXUnit**.

5.3. Automatización de compilaciones y ejecuciones de casos de prueba: Ant

Un aspecto importante del proceso de pruebas unitarias es lograr el automatismo de las mismas. Una de las mejores formas de conseguirlo es integrar los procesos de compilación de programas y ejecución de casos de prueba en un único proceso de construcción (*build*). La utilidad **Ant** [Davis, 2000] facilita esta integración mediante la creación de un fichero de definición/configuración en formato XML, que sustituye al tradicional archivo *Makefile* basado en secuencias de comandos de *shell*. El mecanismo es similar al de la utilidad *make* que se basa en árboles de objetivos a construir y las dependencias existentes entre ellos. El fichero de configuración contiene el árbol de objetivos. Cada objetivo contiene las tareas a ser ejecutadas para obtenerlo. A continuación se muestra un ejemplo de fichero de configuración en formato XML:

```
<project name="Sample.Project" default="runtests" basedir=".">
    <property name="app.name"   value="sample" />
    <property name="build.dir"  value="build/classes" />

    <target name="JUNIT">
       <available property="junit.present"
                   classname="junit.framework.TestCase" />
    </target>

    <target name="compile" depends="JUNIT">
            <mkdir dir="${build.dir}"/>
            <javac srcdir="src/main/" destdir="${build.dir}" >
                <include name="**/*.java"/>
            </javac>
    </target>

    <target name="jar" depends="compile">
            <mkdir dir="build/lib"/>
            <jar jarfile="build/lib/${app.name}.jar"
                 basedir="${build.dir}" includes="com/**"/>
    </target>

    <target name="compiletests" depends="jar">
            <mkdir dir="build/testcases"/>
            <javac srcdir="src/test" destdir="build/testcases">
                <classpath>
                   <pathelement location="build/lib/${app.name}.jar" />
                   <pathelement path="" />
                </classpath>
                <include name="**/*.java"/>
            </javac>
    </target>
```

```
<target name="runtests" depends="compiletests" if="junit.present">
        <java fork="yes" classname="junit.textui.TestRunner"
            taskname="junit" failonerror="true">
            <arg value="test.com.company.AllJUnitTests"/>
            <classpath>
               <pathelement location="build/lib/${app.name}.jar" />
               <pathelement location="build/testcases" />
               <pathelement path="" />
               <pathelement path="${java.class.path}" />
            </classpath>
        </java>
</target>
</project>
```

Cada fichero de configuración XML contiene un proyecto. Cada elemento `task` puede tener un atributo `id` con un valor único para poder ser referenciado posteriormente.

Un proyecto tiene tres atributos:

- `name`: nombre del proyecto.
- `default`: el objetivo por defecto cuando no se especifica ninguno.
- `basedir`: directorio base desde el que se hacen todos los cálculos de caminos.

Cada proyecto define uno o más objetivos (*targets*). Un objetivo es un conjunto de tareas que se quieren ejecutar. Un objetivo puede depender de otros. Por ejemplo, si se tiene un objetivo para compilar y otro para crear un entregable con el resultado de la compilación, sólo se podrá construir el entregable si previamente se ha compilado, es decir, el objetivo entregable depende del objetivo a compilar. **Ant** resuelve estas dependencias mediante el atributo `depends`. El atributo `depends` sólo especifica el orden en que los objetivos serán ejecutados, no el hecho de que el objetivo que contiene el atributo deba ejecutarse. Si en un atributo `depends` hay más de un objetivo, se intentarán ejecutar en el orden en que aparecen (es decir, de izquierda a derecha). Por ejemplo:

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="B"/>
<target name="D" depends="C,B,A"/>
```

Si queremos ejecutar *D*, las dependencias indican que se ejecutará primero *A*, luego *B* y luego *C*, dado que *C*, que aparece primero en la lista de dependencias de *D*, depende de *B* y *B* a su vez depende de *A*.

Un objetivo se ejecuta una sola vez. Además, se puede establecer como condición para la ejecución el hecho de que se cumpla una condición:

```
<target name="build-module-A" if="module-A-present"/>
<target name="build-own-fake-module-A" unless="module-A-present"/>
```

Si no hay establecidas condiciones de ejecución, el objetivo se ejecutará siempre.

Una tarea (*Task*) es una porción de código que puede ser ejecutada. Puede tener muchos atributos. El valor de un atributo puede contener referencias a una propiedad. Estas referencias serán resueltas antes de que la tarea se ejecute. La estructura para la definición de una tarea es como sigue:

```
<name attribute1="value1" attribute2="value2" ... />
```

donde `name` es el nombre de la tarea, `attributeN` es el nombre del atributo y `valueN` es el valor para dicho atributo. Existen un conjunto de tareas internas predefinidas.

Un proyecto puede tener un conjunto de propiedades. Cada propiedad tiene un nombre y un valor. Las propiedades pueden ser usadas como valor de los atributos de tareas (se encierra entre llaves el nombre de la propiedad y se precede del símbolo `$`). Existe un conjunto de propiedades internas predefinidas.

Existen otra serie de características que no se detallan aquí, tales como el paso de argumentos desde línea de órdenes, filtros, referencias, etc.

Cada tarea del árbol de objetivos es ejecutada por un objeto que implementa una *interfaz*: `Task`.

Ant se integra con una serie de entornos de desarrollo, entre ellos, **VisualAge for Java** [IBM, 2001], **JBuilder** [Borland, 2001], **NetBeans Forte** [Sun, 2001], etc.

La principal ventaja de este enfoque es la independencia del sistema operativo, facilitando el desarrollo de software sobre múltiples plataformas. Como contrapartida, se pierde la potencia inherente de ciertos comandos de *shell* específicos de algunos sistemas operativos.

6. Trabajo adicional

En apartados anteriores se describen herramientas que permitirían obtener un repositorio de programas en formato XML, un entorno sencillo para pruebas unitarias, la definición de las pruebas unitarias en formato XML y la automatización de su ejecución. En este apartado se propone un marco de trabajo con un repositorio de programas en XML y pasos adicionales para completar un entorno totalmente automático de verificación de programas.

6.1. Repositorio XML

El objetivo es la generación y ejecución de casos de prueba de la Caja Blanca o pruebas unitarias de forma totalmente automática. Se crearán herramientas que accedan al repositorio de programas en formato XML y obtengan un número suficiente de casos de prueba que garanticen, por ejemplo, que todos los métodos de una clase son ejercitados. Además, el automatismo de la fase de ejecución de los casos de prueba generaría un informe de los casos de prueba ejecutados, cuáles de ellos han resultado satisfactorios y cuáles han fallado.

Un esquema de esta concepción sería el de la figura 5:

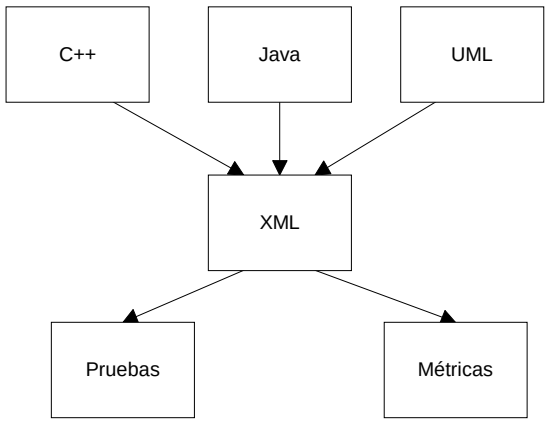

Figura 5: Repositorio para verificación con XML.

Se propone la representación en lenguaje XML del contenido de los programas escritos en cualquier lenguaje de programación, obteniendo una representación externa del árbol de sintaxis abstracta y haciendo por tanto visible para otras herramientas dicha representación, en concreto para la verificación automática de programas. Una representación externa de este tipo ahorraría costes de computación, pues hace innecesaria la obtención del árbol de sintaxis abstracta por cada herramienta, realizándose esta tarea una sola vez. Además, facilitaría la cooperación entre las distintas herramientas, hecho que sería casi imposible si se trabajase con herramientas propietarias.

Esto sería también aplicable a lenguajes de modelado tales como UML (*Unified Modeling Language*) [Booch et al., 1999] lo que ampliaría el alcance del estudio al nivel de diseño orientado a objetos.

Los programas convertidos a formato XML constituyen así un repositorio con la información relevante de los ficheros en texto plano originales y dicha información podrá ser usada para los más diversos propósitos por herramientas de ingeniería de software creadas al efecto.

Se propone **Java** como el lenguaje ideal para la realización de un prototipo para la verificación automática de programas debido a sus características que lo hacen sintácticamente muy limpio.

La extensibilidad del sistema se puede realizar de forma fácil sin más que añadir un analizador sintáctico para el lenguaje de nuestra elección que convierta a formato XML los ficheros de texto plano de dicho lenguaje. Esto permitiría el aprovechamiento de las herramientas directamente sin cambios, pues éstas leen de la estructura XML creada y no del código fuente original.

La separación de los datos de prueba del código de prueba permite dar un paso más en la automatización de las pruebas unitarias mediante la generación de forma externa de datos de prueba, obteniéndolos del repositorio de programas XML. Se puede desarrollar una herramienta que permita elegir el tipo de datos de prueba a obtener: de cobertura de sentencias, de valores límite, etc. Pudiendo esto ser configurado a voluntad.

Otro camino a seguir sería la generación automática de código de prueba basándose en patrones de prueba y en los casos de prueba generados en el paso anterior guardados en formato XML.

La obtención de métricas acerca del código fuente también se facilita una vez que éste es convertido a formato XML. Herramientas disponibles para este lenguaje como *ltxml* de la Universidad de Edimburgo [UELTG, 2001] pueden ser el punto de partida para un trabajo en profundidad en este sentido. Ltxml consiste en un conjunto integrado de herramientas que proporcionan dos vistas de un fichero XML. Una como un conjunto plano de elementos de marcas y texto, y otra como una secuencia de elementos XML estructurados en árbol, pudiéndose mezclar ambas. Además, un lenguaje de consulta permite seleccionar fácil y rápidamente aquellas partes de interés de un documento XML. Una utilidad como `sgcount` lista las clases de elementos que aparecen en un documento XML indicando el número de veces que aparecen, por ejemplo, definiciones de métodos de clases, de referencias a variables, etc. obteniéndose más información acerca del contenido de un programa que la típica medida del número de líneas de código.

Jdepend [Clark, 2001] analiza las interdependencias entre paquetes generando métricas de calidad de diseño enfocando las métricas en términos de dependencia aferente y eferente, es decir, de acoplamiento entre clases.

6.2. Prototipo de entorno de verificación automática de programas con XML

En la figura 6 se representa de forma gráfica un prototipo para el entorno de verificación automática de programas con XML propuesto.

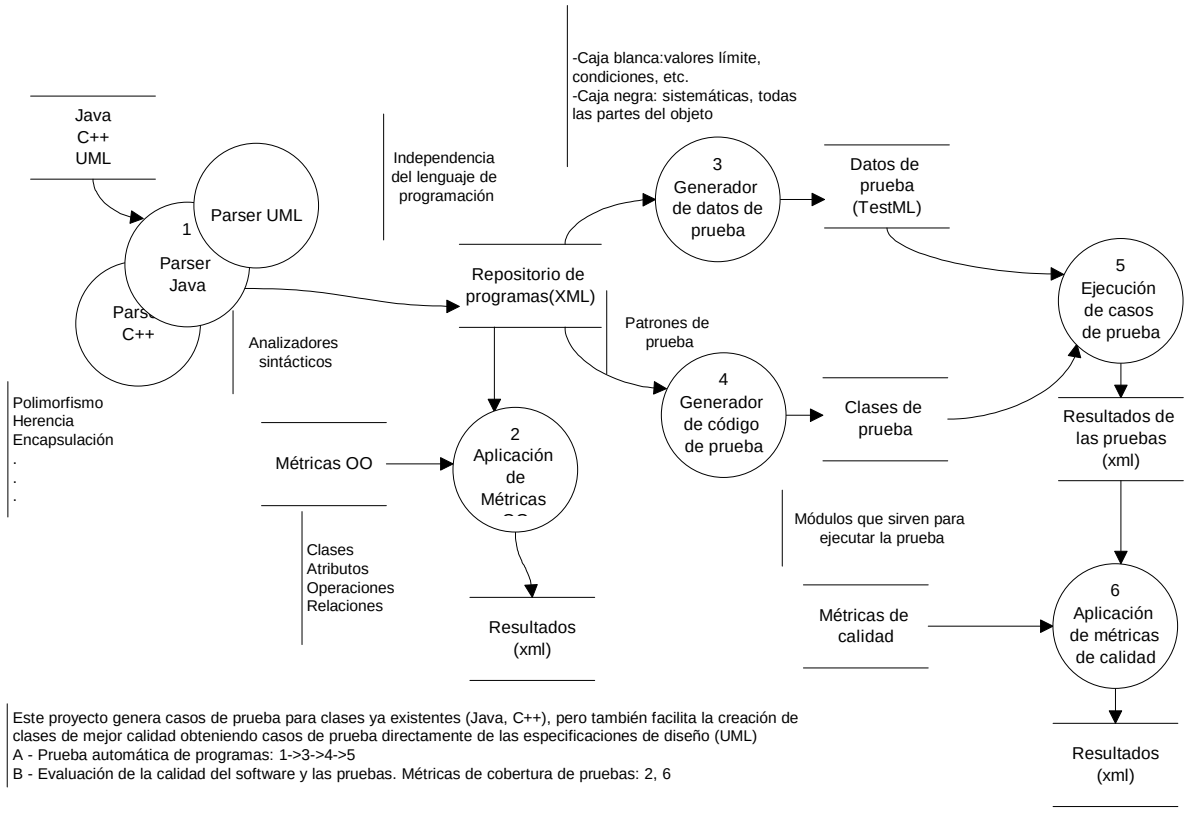

Figura 6: Prototipo para entorno de verificación automática de programas con XML

7. Conclusiones

El enfoque orientado a objetos se presenta, por sus características específicas, como el ideal para el desarrollo de herramientas de prueba unitaria del software. Separando el comportamiento de los objetos de la interfaz gráfica se facilita la automatización de las pruebas y su chequeo, lo que elimina gran parte de lo tedioso que normalmente resulta esta fase del ciclo de desarrollo. Conceptos como el desarrollo evolutivo se basan en la escritura de una pequeña parte de código y su correspondiente prueba, alcanzando con este enfoque la prueba unitaria su máximo significado.

La introducción de un lenguaje de marcas como XML para la representación del código fuente de los programas, los casos de prueba y en la configuración de los procedimientos de compilación hacen que las fases de desarrollo, compilación y prueba se integren mejor, al tiempo que facilita la incorporación de herramientas de ayuda potentes, flexibles y reutilizables, con el valor añadido de la independencia de la plataforma para la que sean desarrolladas inicialmente.

El uso de XML como substrato común para todas las herramientas facilita la inclusión de automatismos en todas las etapas de la prueba: generación automática de casos de prueba a partir del código fuente, generación automática de código de prueba, ejecución automática de los casos de prueba y obtención de informes de los resultados de las pruebas.

Las pruebas de software están íntimamente ligadas con la calidad del producto software desarrollado. Es por esto que también resulta de interés aprovechar el repositorio de programas XML para la obtención de métricas de calidad.

El lenguaje **Java**, por su sintaxis relativamente simple y clara se presenta como el mejor candidato para el desarrollo de estas herramientas. Trabajos posteriores permitirán su extensión a otros lenguajes orientados a objetos, o incluso a lenguajes de modelado como UML.

El lenguaje XML se convierte de esta manera en el formato universal para la representación de información de las distintas herramientas de Ingeniería de software, facilitando la comunicación entre ellas, la reutilización de la información y la ampliación de los entornos con menor esfuerzo del habitual.

8. Bibliografía

[Alexander et al., 1977] Alexander, C., Ishikawa, S. y Silverstein M., *A Pattern Language*. Oxford University Press, 1977.

[Badros, 2000] Badros, Greg J., JavaML: *A Markup Language for Java Source Code*. Dept. Of Computer Sciece and Engineering, University of Washington, Seattle, WA, 2000. (http://www.cs.washington.edu/research/constraints/web/badros-javaml-www9.pdf).

[Binder, 2000] Binder, R.V., *Testing Object-Oriented Systems: Models, Patterns and Tools*. Addison-Wesley, 2000.

[Boehm, 1981] Boehm, B., *Software Engineering Economics*,. Prentice Hall, 1981, p. 37.

[Boehm, 1984] Boehm, B., *Verifying and Validating Software Requirements and Design Specifications*. IEEE Software, January 1984, pp. 75-88.

[Booch et al., 1999] Booch, G., Rumbaugh, J. y Jacobson, I., *El Lenguaje Unificado de Modelado: Guía de usuario*. Addison Wesley, 1999.

[Borland, 2001] Borland Software Corporation, JBuilder, Version 5, 2001. (http://www.borland.com/jbuilder).

[Bray et al., 2000] Bray, T., Paoli, J., Sperberg-McQueen y Maler, E. C. M., *Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation*, 6 October 2000. (http://www.w3.org/TR/REC-xml).

[Clark, 2001] Clark, M., *Jdepend*. Clarkware Consulting, Inc. January 25, 2001.

[Canna, 2001] Canna, J., *Testing, fun? Really? Using unit and functional tests in the development process*. IBM developerWorks, March 2001.

[Davis, 2000] Davis, M., *Incremental development with Ant and Junit. Using unit test to improve your code in small steps*, IBM developerWorks, 2000. (ftp://www6.software.ibm.com/software/developer/library/j-ant.pdf).

[Eckel, 2000] Eckel, B., *Thinking in Patterns, chapter 2: Unit Testing*. Libro electrónico, versión preliminar 2000. (http://www.bruceeckel.com/).

[Fenton y Pfleeger, 1997] Fenton, N. E. y Pfleeger, S.L., *Software metrics. A rigorous and practical approach*, PWS Pub., 1997.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., y Vlissides, J., "*Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995

[Gamma y Beck, 2000/1] Gamma, E. y Beck, K., *JUnit A Cook's Tour*, 2000

[Gamma y Beck, 2000/2] Gamma, E. y Beck, K., *JUnit Test Infected: Programmers Love Writing Tests*, 2000.

[Gamma y Beck, 2001] Gamma, E. y Beck, K., *JUnit, 2001*. (http://www.junit.org).

[Gosling et al., 2000] Gosling, J., Joy, B., Steele, G., y Bracha, G. The Java Language Specification, Second Edition, Addison-Wesley, 2000.
(ftp://ftp.javasoft.com/docs/specs/langspec-2.0.pdf).

[IBM, 1999] IBM Corporation. *Jikes Java Compiler*, 1999.
(http://www.alphaworks.ibm.com/tech/Jikes).

[IBM, 2001] IBM Corporation. *Visual Age for Java*, Version 4.0, 2001. (http://www-4.ibm.com/software/ad/vajava).

[ISO, 1986] ISO. *Standard Generalized Markup Language (SGML). ISO 8879*, 1986. (http://www.iso.ch/cate/d16387.html).

[Jones, 1981] Jones, T.C., *Programming Productivity: Issues for the 80s*, IEEE Computer Society Press, 1981.

[JXQuick, 2001] JXQuick, *Data-Centric programming with Java and XML: The Open Conversion Model*, SourceForge, 2001. (http://quickutil.sourceforge.net/view//OCM).

[JXUnit, 2001] JXUnit, *Building Suites of Test Data with XML*. Release 2.0.0. 18 April 2001. (http://www.jxunit.sourceforge.com).

[McCabe, 1976] McCabe, T. *A Software Complexity Measure*, IEEE Trans. Software Engineering, vol.2 December 1976, pp. 308-320.

[Myers, 1979] Myers, G., *The Art of Software Testing*, Wiley, 1979.

[Paulk, et al., 1993] Paulk, M.C., Curtis, B., Chrissis, M.B., y Weber, C.V., *Capability Maturity Model for Software*, Version 1.1. CMU/SEI-93-TR-24. Pittsburgh: Software Engineering Institute, 1993.

[Prather, 1987] Prather, R.E., *On hierarchical software metrics*, Software Engineering Journal, 2(2), pp. 42-5, 1987

[Pressman, 1998] Pressman, R.S., *Software Engineering, A Practical Approach*, Mc Graw Hill, 1998.

[Quick, 2001] *Quick*. Release 4.0.5, SourceForge, 3rd May 2001.

[Sun, 1997] Sun Microsystems Inc., *JavaBeans™ API specification*, Version 1.01, Sun Microsystems Inc., 1997. (http://java.sun.com/products/javabeans/docs/beans.101.pdf).

[Sun, 2001] Sun Microsystems Inc., *NetBeans Forte for Java*, Version 3, 2001 (http://www.sun.com/forte/ffj).

[UELTG, 2000] University of Edinburgh Language Technology Group. *LT XML version 1.2*, Septiembre 2000. (http://www.ltg.ed.ac.uk/software/xml/xmldoc/xmldoc.html).

[W3C, 1998/1] World Wide Web Consortium, *Xml extended pointer specification (xpointer)*, W3C Recommendation, June 1998. (http://www.w3.org/TR/1998/WD-xptr).

[W3C, 1998/2] World Wide Web Consortium, *XML-QL: A query language for XML*, W3C Note, 19-August-1998. (http://www.w3.org/TR/1998/NOTE-xml-ql-19980819).

[W3C, 1999/1] World Wide Web Consortium, *Xml linking specification (xlink)*, W3C Recommendation, June 1999. (http://www.w3.org/TR/WD-xlink).

[W3C, 1999/2] World Wide Web Consortium, *Extensible style language (xsl)*, W3C Recommendation, June 1999. (http://www.w3.org/TR/WD-xsl).

[W3C, 1999/3] World Wide Web Consortium, *Document object model (dom)*, W3C Recommendation, June 1999. (http://www.w3.org/TR/PR-DOM-Level-1).

[W3C, 1999/4] World Wide Web Consortium, *XSL Transformations (XSLT)*, , Version 1.0, W3C Recommendation, 16-November-1999. (http://www.w3.org/TR/xslt).

[W3C, 1999/5] World Wide Web Consortium, *HyperText Markup Language (HTML) 4.01*, Version 4.01, W3C Recommendation, 24 December 1999. (http://www.w3.org/TR/1999/REC-html401-19991224).