Generación Casos de Prueba Unitarias para Java Basadas en la Técnica de McGregor y Sykes

Daniella Rojas Pacheco
Pontificia Universidad Católica de Valparaíso
Escuela de Ingeniería Informática
Valparaíso - Chile
daniella.rojas.p@mail.ucv.cl

Jorge Bozo Parraguez.
Pontificia Universidad Católica de Valparaíso
Escuela de Ingeniería Informática
Valparaíso - Chile
jbozo@ucv.cl

Resumen

Las pruebas unitarias son importantes para disminuir las pruebas en las fases posteriores, pero son poco practicadas por el tiempo y costos que éstas representan, ya que los casos de pruebas se generan habitualmente de forma manual. En el caso de sistemas construidos bajo paradigma Orientado a Objetos, las pruebas unitarias se centran en la clase y una de las técnicas utilizada es la "técnica de McGregor y Sykes". Basándose en ésta y además, en diseño por contrato y JML, se diseña y se construye un prototipo funcional de una herramienta que genere asistidamente casos de prueba unitarios, dirigiéndose al lenguaje de programación Java y utilizando JUnit para la ejecución de los mismos.

Palabras claves: Prueba Unitaria, Casos de Prueba, Técnica de McGregor y Sykes, JML.

1. Introducción

La *Prueba* es el proceso de ver si existe diferencia entre el comportamiento esperado (cumpliendo los requerimientos) y el comportamiento que es observado realmente. Los *casos de prueba* son el conjunto de entrada y resultados esperados para un componente con el propósito de causar fallas y detectar defectos [12].

Para la generación de casos de prueba existen diversos enfoques, y uno de ellos es el de Caja Negra. Una de las técnicas que se basa en este enfoque es la técnica de McGregor y Sykes, dirigida a los sistemas Orientados a Objetos (OO). Esta técnica es utilizada para diseñar y construir un prototipo funcional de una herramienta que genera asistidamente casos de prueba. Se apoya en el Diseño por Contrato [10] y JML [2,8].

En la primera sección de este trabajo, se muestra brevemente el contexto de la herramienta y se justifica la creación de la misma, señalando los objetivos que se desean alcanzar. La segunda sección describe las técnicas utilizadas para la creación de la herramienta. Posteriormente, se muestra la etapa actual de construcción de la herramienta llamada *CPruebaJ* -Casos de Prueba para Java - junto con los resultados obtenidos hasta el momento. Por último se señalan las conclusiones.

1.1. Contexto de la Propuesta

Existen diferencias que dificultan las actividades de prueba de programas OO, sin poder realizarlas como las actividades de prueba de programas estructurados. Según Robert Binder [1], estas discrepancias se presentan porque los programas OO consideran a la *clase* como la unidad más pequeña para ser testeada, ya que no se puede probar más de una operación a la vez (la visión convencional de la unidad de prueba), pero sí como parte de una clase. Además la clase es considerada la unidad fundamental en la programación OO, en cambio en los programas estructurados, la parte más pequeña es la función [9,12]. Por lo tanto, se considera prueba unitaria a la prueba de métodos (prueba intra-método e inter-método) y prueba intra-clase [1,14].

Diversos enfoques son utilizados en las pruebas unitarias, así como herramientas que han sido construidas como apoyo para esta fase.

1.1.1. Enfoques de Prueba en OO

a. Enfoque de Caja Negra

La propuesta realizada por Offut y Irvine [11], es aplicar a la prueba OO el Método de Particionamiento de Categoría (Category-Partition Method), el cual clasifica las operaciones de las clases basadas en la función que cada una lleva a cabo. Existen otros métodos de

particionamiento al nivel de clase, los cuales son: Particionamiento basado en estados y Particionamiento basado en atributo [12].

La propuesta realizada por *McGregor y Sykes* [9,14] trata específicamente en la derivación de casos de prueba mediante un conjunto de reglas de derivación de estos casos a partir de las precondiciones y postcondiciones de los métodos.

La propuesta de Robert Binder se llama N+ [1,14] y se divide en 2 pasos:

1. Generar casos de prueba cumpliendo con el criterio *All Round-Trip path*.
2. Generar casos de prueba que cubra el criterio *Sneak path*.

b. Enfoque de Caja Blanca

Harrold y Rothermel [5] comentan que los criterios de flujos de datos utilizados en la prueba de programas estructurados pueden ser utilizados, tanto para la prueba de intra-método, como prueba inter-método dentro de una clase. Pero estos criterios no consideran las interacciones del flujo de datos que se presentan cuando los usuarios de una clase invocan secuencias de métodos en un orden arbitrario.

c. Técnicas Basadas en Errores: Criterio de Análisis de Mutantes

Si bien, esta técnica es utilizada para probar programas estructurados, se ha adaptado para lenguajes orientados a objetos, como es Java [7].

Éste es un criterio que utiliza un conjunto de programas ligeramente modificados (mutantes) obtenido a partir de un determinado programa. El objetivo es encontrar un conjunto de prueba capaz de revelar las diferencias de comportamiento existente entre el programa y sus mutantes [3].

1.1.2. Herramientas que Existen como Apoyo a las Pruebas Unitarias en OO

Algunas de las herramientas que apoyan las pruebas unitarias son:

- **JUnit:** es una herramienta de código abierto [4], específicamente un framework, desarrollado por Erich Gamma y Kent Beck. Permite las Ejecución automática de los casos de prueba, comparando el resultado esperado con el resultado real, señalando cualquier diferencia entre ambos.
- **MuJava:** es una herramienta de código abierto [15] que fue creada por Yu Seung Ma, Yong Rae Kwon y Jeff Offut. Realiza pruebas basadas en errores, específicamente, basándose en la técnica de Análisis de Mutantes, generando automáticamente los mutantes de la clase bajo prueba. Además calcula el *mutants score* de los datos de prueba. Desgraciadamente, no elimina los mutantes equivalentes.
- **TCAT/java:** es una herramienta [16] que hace pruebas de caja blanca, particularmente de criterio de Flujo de Control. Fue creada para plataforma Windows por TestWorks, pero ha sido extendida para Unix.

La comparación de estas herramientas se encuentra en la tabla 1:

Tabla 1: Características presentadas por las herramientas de apoyo a la OO

Criterios	*JUnit*	*MuJava*	*TCAT/java*
Enfoque de caja blanca			✓
Enfoque de caja negra			
Enfoque basado en errores		✓	
Generación automática de casos de prueba			✓
Ejecución automática de casos de prueba	✓	✓	✓
Cobertura de casos de prueba		✓	✓

1.2. Problemática y Objetivos

No todas las herramientas vistas generan casos de prueba, como JUnit que sólo ejecuta y evalúa los casos de prueba. MuJava, por su parte, tampoco genera los casos de prueba, pero a diferencia de JUnit, utiliza un enfoque basado en errores para determinar la cobertura de los casos de prueba. Esto lleva a la necesidad de una herramienta que genere casos de prueba. TCAT/java, por otro lado, genera los casos de prueba, los ejecuta y determina la cobertura de éstos, pero sólo trabaja con enfoque de caja blanca. Con estos antecedentes se formulan los siguientes objetivos:

- Diseñar y construir un prototipo funcional de una herramienta que asista la generación de casos de prueba para Pruebas Unitaria en Software desarrollados con enfoque OO.
- Medición de la efectividad de los casos de prueba generados mediante análisis de mutantes
- Selección y utilización de una técnica de caja negra y así complementar esta herramienta.

2. Herramienta "Ideal" que Colabore con las Pruebas Unitaria

Una herramienta "ideal" que apoye las pruebas unitarias debe ser capaz de cubrir los siguientes requerimientos:

1. **Generación de Casos de Prueba**: es la etapa en la cual se generan los casos de prueba mediante los enfoques conocidos para la Clase Bajo Prueba (CUT). Esta etapa se realiza la mayoría de los casos de forma manual por el testeador, pero es posible automatizarla.
2. **Ejecución de Casos de Prueba**: en esta etapa se procede a probar el programa con los casos de prueba, obteniendo así los resultados reales. Esta etapa es realizada preferentemente de forma automática
3. **Evaluación de Resultados Obtenidos**: Se determina si pasa o no la prueba, comparando los resultados reales obtenidos de la etapa anterior y los resultados esperados en la generación de casos de prueba (oráculo). Esto se puede realizar de forma manual o automática.

Como se mencionó anteriormente, se desea cubrir el primer requerimiento, y para ello se selecciona una técnica de caja negra.

3. Selección de Técnicas que Permitan Generar Casos de Prueba

3.1. Diseño por Contrato

El Diseño por Contrato (*Design By Contract - DBC*) es un método para desarrollo de software. La principal idea es que una clase y sus clientes tienen un contrato el uno con el otro [8,10]. Las obligaciones asociadas al cliente se señalan mediante las precondiciones; las obligaciones del desarrollador se señalan mediante las postcondiciones, las cuales, deben especificarse para cada método de la clase. Una de las técnicas de prueba que se puede aplicar utilizando DBC es la técnica de McGregor y Sykes [9].

3.2. Técnica de McGregor y Sykes

La técnica *McGregor y Sykes* deriva los casos de prueba mediante un conjunto de reglas, construidas a partir de las precondiciones y postcondiciones de los métodos, teniendo en cuenta los operadores lógicos (conjunción, disyunción, disyunción excluyente, implicancia y negación), además de las sentencias condicionales ("si entonces sino"), que están presentes en las mismas [9,14]. Se asume que la especificación está expresada, ya sea, en un lenguaje de especificación, como es Object Constraint Language (OCL) [9], en un lenguaje natural, y/o en un diagrama de transición de estados. La idea es identificar las reglas o requerimientos para los casos de prueba para todas las posibles combinaciones de situaciones en la cual una precondición puede llevarse a cabo y las postcondiciones pueden ser logradas. Entonces se crean casos de prueba que puedan cumplir estos requerimientos. En las tablas 2 y 3 [9] se muestran las expresiones lógicas que componen las precondiciones y las postcondiciones y los respectivos requerimientos para los casos de prueba:

Tabla 2: Requerimientos de casos de prueba a partir de las precondiciones según McGregor y Sykes

Expresión Lógica	Requerimientos de casos de prueba
True	(true, Post)
1	(1, Post)
Not 1	(not 1 , Post)
1 and 2	(1 and 2, Post)
1 or 2	(1, Post) (2, Post) (1 and 2, Post)
1 xor 2	(1 and not 2, Post) (not 1 and 2, Post)
1 implies 2	(not 1, Post) (2, Post) (not 1 and 2, Post)
If 1 then 2 Else 3 endif	(1 and 2, Post) (not 1 and 3, Post)
Nota: 1, 2 y 3 representan componentes en una expresión OCL	

Tabla 3: Requerimientos de casos de prueba a partir de las postcondiciones según McGregor y Sykes

Expresión Lógica	Requerimientos de casos de prueba
1	(Pre, 1)
1 and 2	(Pre, 1 and 2)
1 or 2	(Pre, 1) (Pre, 2) (Pre, 1 and 2)
1 xor 2	(Pre, 1 and not 2) (Pre, not 1 and 2)
1 implies 2	(Pre, not 1 or 2)
If 1 then 2 Else 3 endif	Se omiten los casos de prueba por simplicidad

3.3. Adaptación de la Técnica de McGregor y Sykes

Si bien la técnica de McGregor y Sykes puede aplicarse sobre DBC, este método está enfocado especialmente para OCL, que es un lenguaje de especificación usado para describir expresiones acerca de los modelos de UML. Por lo tanto, para utilizar esta técnica sobre DBC existen dos alternativas:

1. utilizar directamente OCL sobre la implementación de la clase adoptando cierta nomenclatura para la detección de las aserciones de precondiciones y postcondiciones, ó

2. utilizar otro lenguaje que sirva para especificar directamente la clase en la implementación, adaptando la técnica al lenguaje de especificación.

La primera alternativa ya está siendo utilizada en una herramienta llamada JPrePost, la cual utiliza OCL adaptando una nomenclatura [14]. Por lo que se optó por la segunda alternativa, usando para ello JML (*Java Modeling Language*) [2]. Las razones para la elección de JML para adaptar la técnica de McGregor son las siguientes:

1. La técnica de McGregor y Sykes puede aplicarse a cualquier lenguaje de especificación.
2. JML es un lenguaje de especificación para Java [2].
3. JML tiene cláusulas que señalan las precondiciones y las postcondiciones (*requires* y *ensures*).
4. JML tiene una cláusula para la invariante de clase.
5. JML tiene operadores lógicos equivalentes a OCL los que se visualizan en la siguiente tabla (tabla 4).

Tabla 4: Comparación entre los operadores lógicos de OCL y JML

Operador Lógico de OCL	Operador Lógico de JML
Not	!
And	&&
Or	||
Xor	||
Implies	==>
---	<==
If –then- else	*Ausente*

JML tiene otras formas de expresión las que se utilizarán para evaluar las expresiones de postcondiciones como son [2,8]: **\result, \old, (*y*), behavior, also, spec_public, etc**. Las especificaciones son escritas en comentarios con una anotación especial, los cuales comienzan con el caracter "@".

Al realizar la adaptación de la técnica de McGregor y Sykes se efectuaron los siguientes cambios a las tablas de contribuciones (tabla 5 y 6):

Tabla 5: Requerimientos de casos de prueba a partir de las precondiciones adaptado a JML

Expresión Lógica	Requerimientos de casos de prueba
True	(true, ensures)
1	(1, ensures)
! 1	(!1 , ensures)
1 && 2	(1 && 2, ensures)
1 || 2	(1, ensures) (2, ensures) (1 && 2, ensures)
1 ==> 2	(!1, ensures) (2, ensures) (!1 && 2, ensures)
1 <== 2	(1, ensures) (!2, ensures) (1 && !2, ensures)
Nota: 1y 2 representan componentes en una expresión JML	

Tabla 6: Requerimientos de casos de prueba a partir de las postcondiciones adaptado a JML

Expresión Lógica	Requerimientos de casos de prueba
1	(requires, 1)
1 && 2	(requires, 1 && 2)
1 || 2	(requires, 1) (requires, 2) (requires, 1 && 2)
1 ==> 2	(requires, !1 || 2)
1 <== 2	(requires, 1 || !2)

Esta técnica es realizada de forma manual, pero es susceptible de ser automatizada, y para ello se deben tomar en cuenta lo siguiente:

1. Los requerimientos son extraídos a partir de la CUT, por lo que se debe contar con el código fuente.
2. La clase debe estar previamente compilada.
3. Los datos de entrada y los resultados esperados son generados aleatoriamente, cumpliendo con los requerimientos extraídos de la CUT.
4. Debe existir alguna herramienta que permita ejecutar los casos de prueba (Ejemplo. JUnit).

4. Herramienta para Generar Casos de Prueba: *CPruebaJ*

Para automatizar la mayor parte de la funcionalidad *Generar Casos de Prueba*, es necesario considerar desde el ingreso de la clase hasta el momento que se entregan los casos de prueba en un formato adecuado para JUnit. Estas funcionalidades se describen a continuación:

- **Ingreso de Clase bajo Prueba**: Ingreso del nombre de la clase bajo prueba para la identificación de las especificaciones en JML.
- **Aplicación de Técnica de McGregor y Sykes**: Detección de las especificaciones de cada método y de cada invariante de la clase para producir los requerimientos los casos de prueba.
- **Generación de Casos de Prueba**: Generación de los datos de prueba y los resultados esperados según los requerimientos de los casos de prueba obtenidos mediante la técnica de McGregor y Sykes.
- **Escritura de TestClase para JUnit**: Escritura de los casos de prueba generados para la CUT en el formato de las clases de prueba ejecutadas por JUnit

4.1. Documentos y Directorio Generados por la Herramienta

En la primera etapa de la herramienta se contaba con una estructura de directorio para dejar cada uno de los documentos generados de la herramienta [13]. En el avance del prototipo se mejoró tal estructura, para incorporar la funcionalidad de compilación en conjunto de la CUT y de la clase ejecutada por JUnit, consiguiendo los siguientes beneficios:

- Asegurar la existencia de la clase ejecutable (.class) de ambas para realizar la ejecución de las pruebas
- Generar los casos de pruebas en etapas previas a la construcción (análisis y diseño).

La nueva estructura está compuesta por:

- **CPruebaJ**: Es el directorio raíz de la herramienta, por lo que lleva su nombre. La herramienta trabaja bajo este directorio, y debe ser creado antes de ocuparla.
- **Src:** en este directorio debe colocarse el código fuente de la CUT y la clase ejecutada por JUnit, de donde serán extraídos los requerimientos de los casos de prueba.
- **Classes:** En este directorio se recepciona la CUT compilada. Esto es para asegurar a la herramienta que la CUT se encuentra compilada para ser ejecutada.

Nota: es responsabilidad del testeador colocar los archivos con extensión "java" y "class" en los directorios correspondientes para que la herramienta funcione correctamente.

- **Asert**: Contiene el archivo que guarda las cláusulas de la especificación de la CUT, es decir, las precondiciones, postcondiciones y las invariantes de clase.
- **MetaTCase**: Contiene el archivo que guarda los requerimientos de casos de prueba de la CUT (metacasos de prueba), derivados de las precondiciones, postcondiciones y las invariantes de clase, según la adaptación de la Técnica de McGregor y Sykes.
- **TestCase**: Contiene el archivo que guarda los datos de prueba y los resultados esperados de la CUT, derivados de los requerimientos de los casos de prueba.

4.2. Estándar para el Programador/Testeador

El testeador debe procurar dar la base para que la formación de los casos de prueba sea exitosa. Para ello, debe cumplir con un estándar a la hora de programar. Por lo tanto, el programador/testeador debe:

- Especificar en comentario de JML el nombre de la clase, después de la declaración del encabezado de la clase.
- Especificar las variables miembros con JML, si éstas existen.
- Especificar la invariante. Si no existe invariante, especificarla como "`true`".
- Especificar el nombre de los métodos de la CUT en comentarios. Esto se hace antes de declarar la clase.
- Especificar la precondición. Si no existe precondición, especificarla como "`true`".
- Especificar la postcondición.

4.3. Diagrama de Clase del Prototipo de la Herramienta "CPruebaJ"

Para la construcción de CPruebaJ se estableció el siguiente diagrama de clases (Figura 1):

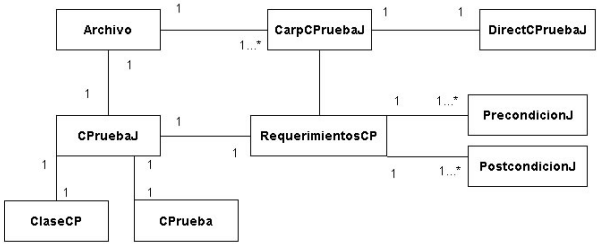

Figura 1: Diagrama de Clase de CPruebaJ

- **Clase *Archivo***: esta clase está destinada al manejo de la CUT, tanto del código fuente como del archivo generado en la compilación, verificando la existencia de estos archivos.
- **Clase *CarpCPruebaJ*:** Esta clase es la encargada de crear el directorio CPruebaJ con los directorios asociados.
- **Clase *DirectCPruebaJ***: Esta clase ejecuta la clase CarpCPruebaJ, siendo el comando para crear el directorio CPruebaJ.
- **Clase *RequerimientosCP***: Esta clase construye los requerimientos de los casos de prueba, a partir de la técnica de McGregor y Sykes. Para ello, extrae las especificaciones de la CUT.
- **Clase *PrecondicionJ***: Esta clase forma parte de la construcción de los requerimientos de los casos prueba, derivando los requerimientos de las precondiciones de la CUT. Se basa en la tabla de contribución propuesta por McGregor y Sykes, adaptadas a JML.
- **Clase *PostcondicionJ***: Esta clase forma parte de la construcción de los requerimientos de los casos prueba, derivando los requerimientos de las postcondiciones de la CUT. Se basa en la tabla de

contribución propuesta por McGregor y Sykes, adaptadas a JML.

- **Clase *CPruebaJ***: Esta clase ejecuta las clases Archivo y RequerimientosCP, siendo el comando para generar los casos de prueba y cada documento asociado.
- **Clase *CPrueba*:** forma los datos de entrada y los resultados esperados
- **Clase *ClaseCP***: forma la clase con el esqueleto de JUnit.

4.4. Construcción de los Requerimientos de Casos de Prueba

La construcción de los requerimientos de los casos de prueba está basada, como se mencionó anteriormente, en la técnica de McGregor y Sykes. La herramienta es capaz de generar los requerimientos de casos de prueba según las expresiones lógicas existentes tanto en las precondiciones y como en las postcondiciones.

Al realizar el estudio para automatizar esta técnica, se consiguió la cantidad de requerimientos que pueden ser formados según el operador lógico que componga la expresión. En el caso que la expresión lógica sea formada por los operadores de *negación* y *conjunción* (*y)*, así como la expresión sin operadores lógicos (según la tabla 5), el número de requerimientos para los casos de prueba es uno. En cambio, al estar una expresión compuesta por los operadores de *disyunción* (*o)* e *implicancia*, la cantidad de requerimientos está dada por:

$$\text{Cantidad de Requerimientos de Casos de Prueba} = \sum_{n=1}^{m} \binom{m}{n}$$

Donde **m** es el número de expresiones unidas con el operador y **n** es la cantidad de combinaciones originadas por el operador según las tablas 5 y 6. Esto se cumple tanto para las expresiones con y sin paréntesis.

Para derivar los requerimientos de los casos de prueba a partir de la *implicancia* se utilizó la equivalencia con el operador de disyunción, dada por:

$$A \rightarrow B = \neg A \vee B$$

4.5. Generación de Casos de Prueba

Para construir los casos de prueba se revisan tanto los requerimientos de los datos de prueba como los requerimientos de los resultados esperados. Se utiliza, además, otra técnica de caja negra para producir los casos, como es *Análisis de Valores Límite*, siguiendo las recomendaciones de la guía de JUnit [6], puesto que los errores se concentran en los límites de los datos participantes. Los datos de prueba lo conforman los datos derivados del conjunto de las variables de entrada del método bajo prueba y los datos derivados del conjunto de las variables miembros que participan en los requerimientos de los casos de prueba.

Sea VE el *conjunto de las variables de entrada* (parámetros) de un método. Sea VM el *conjunto de las variables miembros* (atributos) participantes en los requerimientos de casos de prueba. Se define como VDP (*Variables de Datos de Prueba*) a la unión de los conjuntos VE y WM. Sea RE el *conjunto de resultados esperados* derivados de un requerimiento de casos de prueba. El número de casos de prueba producidos por la herramienta está dado por:

$$\text{Cantidad de Casos de Prueba} = \left(\sum_{i \in VDP}^{n \in VDP} dp_i \right) \times \#RE$$

Donde dp_i es la cantidad de datos de prueba para la variable i, con $i \in VDP$ comenzando por la primera variable hasta n, que representa la última variable del conjunto VDP. Esta cantidad está dada para un requerimiento de casos de prueba.

Se le suma los casos de prueba generados desde los requerimientos, la validación de la invariante, la cual actúa como un resultado esperado frente la ejecución de cada método, dando una visión de cómo se debe comportar la clase al momento de ser instanciada y estando recibiendo mensajes.

Para probar la CUT con los casos de prueba, CPruebaJ permite crear la clase que los ejecuta con formato JUnit.

No necesariamente todos los casos de prueba pueden ser utilizados, eso dependerá de la especificación de los métodos de la clase. Además la herramienta permite el ingreso de los propios casos de prueba. En esta etapa es necesaria la *asistencia del testeador*, para eliminar aquellos casos de prueba que se alejan de la funcionalidad del método y para agregar los casos de prueba que él estime necesarios.

Como la experimentación del prototipo de la herramienta es con fines académicos, sólo se genera casos de prueba para el tipo de dato primitivo *int*, con el propósito de verificar la eficacia de éstos para detectar errores.

5. Resultados Obtenidos

5.1. Ejecución de la Herramienta

Para ver los resultados conseguidos con la herramienta, se utiliza una clase llamada *Ejemplo*, simple en su estructura, y que contiene el método *maximo()*, el cual

tiene como requerimiento determinar el número mayor entre dos enteros, y los métodos *setMax()* y *getMax()* para el acceso a la variable miembro *max*. Una parte de su implementacón y especificación están dadas en la Figura 2. Las especificaciones están destacadas con color verde.

La clase cumple con los requisitos mínimos para la elaboración de los casos de prueba, como b requiere la herramienta. Al ejecutar CPruebaJ se obtienen, hasta el momento, los siguientes documentos:

- **AsertEjemplo**: que guarda las especificaciones de la clase Ejemplo.
- **MetaTCaseEjemplo**: que contiene todos los requerimientos de los casos de prueba.
- **TestCaseEjemplo**: que guarda los casos de prueba generados mediante los requerimientos de casos de prueba.

```
public class Ejemplo{
    //@(*class Ejemplo*);

    private /*@ spec_public @*/int Max;
    //@public invariant true;
...

    /*@(* method maximo (int x , int y): return int *);
    @public behavior
    @requires true ;
    @ensures \result >= x && \result >= y && (\result ==
x || \result == y);
    @*/
    public static int maximo(int x, int y){
    int z = 0;
    if(x >= y)
        z = x;
    else
        z = y;
    return z;
    }
...
}
```

Figura 2: Clase Ejemplo

El contenido del primer documento no será mostrado por la simpleza de éste. Los meta casos de prueba del método *maximo()* guardados en *MetaTcaseEjemplo* se muestra de manera más comprensible para el lector en la tabla 7.

Tabla 7: Requerimientos de casos de prueba generados por CPruebaJ para el método *maximo()*

Requerimientos de Entrada	Requerimientos de Salida
true	result >= x && result >= y && result == x && result == y
true	result >= x && result >= y && result == x
true	result >= x && result >= y && result == y

En las especificaciones de la clase Ejemplo (Figura 2) se puede ver la existencia tanto del operador *de conjunción* (&&) como del operador de *disyunción* (||) en las postcondiciones y sólo de la cláusula *true* en las precondiciones en el método *maximo()*. En el caso de las precondiciones se genera un requerimiento. Por otra parte, en las postcondiciones, el primer operador genera un requerimiento, en cambio, el segundo genera una mayor cantidad de requerimientos, según el número de expresiones unidas por él. Al aplicar la primera fórmula, se obtiene una cantidad de 3 requerimientos de casos de prueba (1 requerimiento derivado de la precondición por 3 requerimientos derivados de las postcondiciones). Los requerimientos cumplen con la técnica de McGregor y Sykes.

Con respecto al tercer documento, muestra un extracto de los casos de pruebas derivados de los requerimientos de casos de prueba del método *maximo()*:

Tabla 8: Extracto de prueba generados por CPruebaJ para el método *maximo()*

Datos de Prueba		Resultado Esperado
x	y	result
MIN_VALUE	MIN_VALUE	result >= MIN_VALUE
MIN_VALUE	MIN_VALUE	result == MIN_VALUE
MAX_VALUE	MAX_VALUE	result >= MAX_VALUE
MAX_VALUE	MAX_VALUE	result == MAX_VALUE
0	0	result >= 0
0	0	result == 0
MAX_VALUE	MIN_VALUE	result >= MAX_VALUE
MAX_VALUE	MIN_VALUE	result >= MIN_VALUE
MAX_VALUE	MIN_VALUE	result == MAX_VALUE
NOTA: MIN_VALUE y MAX_VALUE corresponden a las constantes con valor máximo y mínimo para el tipo primitivo *int* en JAVA.		

Es necesaria la asistencia del testeador, para validar los casos de prueba. Los casos de prueba serán más exactos mientras mejor esté especificada la clase. En este caso, la especificación del método *maximo()* no señala específicamente quien es el máximo entre los dos números (x e y), pero da las directrices para saber quien lo es, y cual es el resultado esperado según la entrada. Se crearon una gran cantidad de casos de prueba para cada requerimientos, y se dejaron aquellos casos que cumplen con la funcionalidad del método. El total de casos de prueba para el método *maximo()* es de 90, pero se dejaron 24 porque se produjeron una gran cantidad de casos de prueba repetidos, dada la especificación.

Para crear los casos de prueba para esta clase se utiliza el comando ***java CrearCasosPrueba Ejemplo***. Para crear la clase que ejecutará dichos casos de prueba se usa el comando ***java Crear TestClase Ejemplo***, la que arroja como resultado la siguiente clase:

```
import junit.framework.*;
public class TestEjemplo extends TestCase{
    public TestEjemplo(String prueba){ super(prueba);  }
    …
    public void testmaximo(){
        Ejemplo Obj = new Ejemplo();
        int aux = 0;
        assertTrue("Chequeo Invariante",true);
    assertTrue(Obj.maximo(Integer.MIN_VALUE,Integer.M
IN_VALUE) >= Integer.MIN_VALUE);
        assertTrue("Chequeo Invariante",true);
    assertEquals(Integer.MIN_VALUE,Obj.maximo(Integer.
MIN_VALUE,Integer.MIN_VALUE));
        assertTrue("Chequeo Invariante",true);
    assertTrue(Obj.maximo(Integer.MAX_VALUE,Integer.
MAX_VALUE) >= Integer.MAX_VALUE);
        assertTrue("Chequeo Invariante",true);
    assertEquals(Integer.MAX_VALUE,Obj.maximo(Integer
.MAX_VALUE,Integer.MAX_VALUE));
        assertTrue("Chequeo Invariante",true);
    …
    }
}
```

Figura 3: Extracto de la clase TestEjemplo generados por CPruebaJ para JUnit

5.2. Validación de los Casos de Prueba generados por la Herramienta

Para validar los casos de prueba generados por la herramienta se utilizó la herramienta MuJava [14] para generar los mutantes, aplicando todos los operadores de mutación, tanto de clase como de método. En el proceso de validación se crearon mutantes equivalentes, los cuales fueron excluidos del proceso de evaluación para tener una medida real de los mutantes que podían ser matados por los casos de prueba generados por la herramienta.

La cantidad y porcentaje de mutantes muertos, vivos y equivalentes para cada clase se muestra a continuación:

Tabla 9: Resultado de Ejecución de Casos de Prueba en Mutantes a nivel de Método

	Clase *Pila Entera*		Clase *Ejemplo*	
Mutantes: a nivel Método	**Cant.**	**%**	**Cant.**	**%**
M. Vivos	17	19.10	8	19.51
M. Muertos	71	79.78	32	78.05
M. Equivalentes	1	1.12	1	2.44
Mutants Score		**80.68**		**80**

Tabla 10: Resultado de Ejecución de Casos de Prueba en Mutantes a nivel de Clase

	Clase *Pila Entera*		Clase *Ejemplo*	
Mutantes: nivel Clase	**Cant.**	**%**	**Cant.**	**%**
M. Vivos	0	0	0	0
M. Muertos	5	55.56	5	71.43
M. Equivalentes	4	44.44	2	28.57
Mutants Score		**100**		**100**

Se esperaba contar con mutantes a nivel de método, pero la herramienta generó algunos mutantes a nivel de clases, los cuales fueron muertos en su totalidad. En el caso de los mutantes a nivel de Método, un 80% fueron muertos, indicando que la herramienta es capaz de generar un alto porcentaje de casos de prueba que pueden descubrir errores.

Esta experimentación se replicó en otra clase llamada *PilaEntera*, que contaba con más métodos y que trabajaba con variables enteras. Se generaron un mayor número de casos de prueba y de mutantes para validarlos, consiguiendo resultados muy semejantes a los obtenidos en la experimentación con la clase *Ejemplo*. Los resultados se encuentran en las tablas 9 y 10 junto con los de la clase *Ejemplo*. El resultado de mutantes vivos dio a conocer cuales errores no eran cubiertos por la herramienta, lo cual permite retroalimentación para el prototipo funcional.

5. Conclusiones

Tal como el presente documento lo demuestra, se ha logrado conseguir la adaptación de la técnica de McGregor y Sykes, utilizando DBC y JML. La generación de los requerimientos de los casos de prueba no es una tarea trivial, puesto que se necesita un buen conocimiento de lógica proposicional y del manejo de la técnica misma. El conocer el número de requerimientos para los casos de prueba da una idea muy cercana de la magnitud de casos de prueba que se pueden generar.

Es necesario establecer un estándar para el programador/testeador, puesto que JML tiene muchas otras formas de denotar las especificaciones de la clase, esto no limita al programador a usarlas, siempre que cumpla con los requisitos mínimos de la herramienta, además conlleva al programador a tener claro la funcionalidad de la clase y otorgar más cuidado en las especificaciones de ésta en etapas de análisis y diseño.

Los documentos generados por CPruebaJ también cumplen con un estándar, para que sean de mayor comprensión para el testeador.

CPruebaJ genera actualmente el directorio de la herramienta, los documentos antes señalados (especificación, metacasos de prueba, casos de prueba y clase ejecutada por JUnit) y cumple con la construcción de

los requerimientos utilizando los operadores de *conjunción* (&&), *disyunción* (||) e *implicancia* (==>). Además reconoce la precedencia entre operadores lógicos, el uso de paréntesis y aplica, según sea el caso, la *ley de De Morgan*.

Para la construcción de los casos de prueba fue necesario incorporar otras técnicas como *Análisis de Valores Límites*, lo que permitió generar casos más probables a descubrir errores. Esto se visualiza en los resultados obtenidos con respecto a la cantidad de mutantes muertos versus los mutantes vivos que quedaron después de ejecutar los casos de prueba. Utilizando la técnica de Análisis de Mutantes se pudo validar los casos de prueba y además, se pudo clarificar cuales son las próximas mejoras que se debe realizar sobre la herramienta para que tenga mayor cobertura para descubrir errores.

El hecho que la herramienta genere la clase con los casos de prueba para ser ejecutada por JUnit, reduce considerablemente el tiempo para que estos sean ejecutados, y por ende reduce el tiempo de pruebas unitarias.

Por último, se debe recordar que CPruebaJ es un complemento para las herramientas existentes, para cubrir por completo los requisitos principales de una herramienta que apoye las pruebas unitarias.

Referencias

[1] **Binder Robert:** *Testing Object Oriented Systems: Models, Patterns and Tools*, Addison Wesley, 2000.

[2] **Cabarcos Manuel**: Sitio de Web de la documentación de JML, modificada por última vez en Diciembre de 2003, visitada por última vez el 10 de Julio de 2005. <http://www.dc.fi.udc.es/ai/tp/practica/jml/jmlintro.html>

[3] **DeMillo R. A, Lipton R. J. Sayward F. G**: *Hints on test data selection: Help for the practicing programmer*. IEEE Computer, v.11, n. 4, p 34-43, 1978.

[4] **Gamma Erich**: Sitio Web de la herramienta de prueba JUnit. Visitada por última vez el 11 de Abril de 2005. <http://www.junit.org>

[5] **Harrold M. J. y Rothermel G.:** *Performing Data Flow Testing on Classes*. Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. 154-163, 1994.

[6] **Hunt Andy y Thomas**: *Unit Testing in Java with JUnit*, The Pragmatic Programmers, EEUU, 2003

[7] **Kim Sunwoo, Clark John A y McDermid John A**: *Class Mutation: Mutation Testing for Object-Oriented Programs*, High Integrity Systems Engineering Group, Department of Computer Science, University of York, USA, 2000.

[8] **Leavens Gary y Cheon Yoonsik:** *Design by Contract with JML*, February 8, 2004

[9] **McGregor John D., Sykes David A.**: *A practical guide to testing object-oriented software*, Addison-Wesley, 2001.

[10] **Meyer Bertrand**: *Applying "Design by Contract*, in Computer (IEEE), v. 25, n. 10, p.40-51, 1992.

[11] **Offutt A. J. y Irvine A**: *Testing object-oriented software using the categorypartition method*. In 17th International Conference on Technology of Object- Oriented Languages and Systems, p 293–304, Santa Barbara, CA, 1995.

[12]**Pressman, R. :** *Software Engineering: A Practicioner's Approach*. McGraw-Hill, 2004.

[13] **Rojas Daniella, Bozo Jorge**: *Generación Asistida de Casos de Prueba para Pruebas Unitarias en Sistemas Construidos bajo Paradigma Orientado a Objeto*, Proceedings V Jornada Iberoamericanas de Ingería de Software e Ingeniería del Conocimiento, p. 19-26, 2006.

[14] **Vallespir Diego**: *Generación Automática de Casos de Pruebas Unitarios para Objetos*, Universidad de la República, Grupo de Ingeniería de Software (Gris), Uruguay, 2004.

[15] ***: Sitio Web de la herramienta de prueba Mujava. Visitada por última vez el 6 de junio de 2005. <http://www.isse.gmu.edu/faculty/ofut/mujava/>

[16] ***: Sitio Web de la herramienta de prueba TCAT/java Visitado por última el 4 de Junio de 2005. <http://www.soft.com/TestWorks/Products/Downloads/TCAT/down.tcat.phtml>.