PONTIFICIA UNIVERSIDAD CATOLICA DE VALPARAISO

FACULTAD DE INGENIERIA

ESCUELA DE INGENIERIA INFORMATICA

Generación de Casos de Pruebas Unitarios para Java basados en la Técnica de McGregor & Sykes

DANIELLA ANDDREA ROJAS PACHECO

Informe Final del Proyecto para optar al Titulo Profesional de

Ingeniero Civil en Informática

Diciembre 2005

Profesor Guía :

Jorge Bozo Parraguez

Dedicada a Dios y a mi Madre, por ser mi Fuerza y Fortaleza en este largo camino. Y a querido hijo, por ser mi inspiración.

Daniella Rojas Pacheco

Agradecimientos

A Dios por darme la oportunidad de crecer como persona.

A mi Familia por apoyarme en este proyecto de vida.

A la Universidad por haberme acogido y haber sido mi segundo hogar.

A mis profesores, por entregarme las armas para enfrentarme al mundo laboral.

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, porque 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 esta técnica y en *Diseño por Contrato* con *JML (*Lenguaje de Modelamiento de Java*)*, se diseña y construye un prototipo funcional de una herramienta que genera asistidamente casos de prueba unitarios

para variables enteras, 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.

Abstract

The unitary tests are important to diminish the tests in the later phases, but little the time and costs are practiced by that these represent, because the cases of tests are generated habitually of manual form. In the case of systems constructed under paradigm *Object Oriented*, the unitary tests are centered in the class and one of the techniques used is the technique of *McGregor and Sykes*. Being based on this technique and *Design by Contract* with *JML (*Java Modeling Language*)*, a functional prototype of a tool is designed and constructed that semi-automatically generates unitary cases of test for integer variables, going to the programming language *Java* and using *JUnit* for the execution of such.

Key words: Unitary Test, Test Cases, Technique of McGregor and Sykes, JML.

Capítulo 1:
Introducción

La *Prueba de Software* es un arte entre las actividades de verificación y validación [1] [11], esto porque se concentra entre un 40 a un 75 por ciento del tiempo de las fases de desarrollo y mantención del ciclo de vida del software, y por ser una actividad que necesita creatividad por parte del testeador [14]. Se llama *Prueba* al proceso de ver si existe diferencia entre el comportamiento esperado (cumpliendo los requerimientos) y el comportamiento que es observado realmente. Para ello se utilizan *casos de prueba*, que son el conjunto de entrada y resultados esperados para un componente con el propósito de causar fallas y detectar defectos [14].

Una de las fases de prueba poco practicada hoy en día es la ***Unitaria*** [20]. Preferentemente son los mismos desarrolladores de la *unidad* [1] los que realizan estas pruebas, porque tienen un mayor conocimiento sobre éstas y por ende pueden realizar mejores casos de prueba para testearla. Llámese *unidad* a la ***función*** en caso de programas estructurados o la *clase* en el caso de programas Orientados a Objetos (OO). Pero existe un problema en este punto ¿quién asegura que las pruebas son realizadas correctamente? La duda surge porque la construcción de una unidad es un proceso creativo y la prueba es un proceso destructivo, por lo que los desarrolladores pueden caer en el pensamiento de no *poder* encontrar errores en la unidad, por la ansiedad de querer ver que su creación está correcta. Esto es parte de los problemas de la psicología de las pruebas [5]. Si las pruebas de la unidad las realiza un agente externo al grupo de desarrolladores se puede evitar este problema, pero los costos y el tiempo incrementarían porque el agente debe entender lo que hace la unidad y como lo hace, y aún así estaría la duda si los casos de prueba son buenos para testear. Es por eso la necesidad de la automatización en este nivel, para que los desarrolladores realicen las pruebas y no caigan en la problemática descrita, disminuyendo los costos de la fase.

Pero en los programas OO existen diferencias en los niveles de prueba, por el concepto de ***clase***. Ya en principio de los años ’90, existe una gran preocupación al darse cuenta que las técnicas para probar sistemas estructurados no eran efectivas en programas OO, por el poco cubrimiento de código y de errores producidos por la herencia y el polimorfismo, que escapaban de los rangos de los criterios que se aplicaban hasta entonces [21].

Hay herramientas que apoyan las pruebas Unitarias para programas OO, sobre todo para aplicaciones construidas con el lenguaje ***Java***, pero en su mayoría carecen de una parte del proceso, que es la generación de los casos de prueba. Es en ese punto donde se aloja el desarrollo de este trabajo: *la generación de casos de prueba Unitarias para programas construidos con Java.*

Para la generación de casos de prueba existen diversos enfoques, y uno de ellos es el de Caja Negra, por la capacidad de encontrar errores en la funcionalidad de los sistemas. La técnica de McGregor y Sykes [12] es una de las técnicas que se basa en este enfoque, a nivel unitario, utilizando para ello las pre y post condiciones de las clases.

Este documento se organiza de la siguiente forma: una introducción, la cual contempla algunas definiciones básicas y el alcance de la propuesta, incluyendo los objetivos por lograr. A continuación, se presenta el capítulo 2 con la selección de las técnicas para generar casos de prueba, en las que se encuentra la técnica de Diseño por Contrato y la técnica de Mcgregor y Sykes, y se explica como éstas fueron adaptadas para lograr una automatización. En el siguiente capítulo se narra el análisis, donde se modela los casos de uso y el modelo de clases del prototipo, y el diseño que contempla estudio de las etapas más complejas a modelar, las cuales son la aplicación de la técnica de McGregor y Sykes y la derivación posterior de los casos de prueba, identificando los métodos para lograr la implementación del prototipo. Al llegar al capítulo 4 se muestra el comportamiento del prototipo. En en capítulo 5 se realiza una experimentación con 2 casos de estudio, para determinar que tan efectiva es la herramienta para generar casos de pruebas que detecten errores. Por último se presentan las conclusiones obtenidas de este trabajo.

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 [12,14]. Por lo tanto, se considera prueba unitaria a la prueba de métodos (prueba intra-método e inter-método) y prueba intra-clase [15,19].

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 Orientadas a Objetos

Al conocer los problemas que hay para determinar los niveles de prueba, se puede observar que también es complicado decidir que enfoque de prueba realizar y cual de sus técnicas aplicar, puesto que, no se pueden ocupar directamente las técnicas conocidas en el enfoque estructural, sobre todo en la fase unitaria, quien es la que presenta mayor problema ya que es la primera actividad que se realiza para validar un programa y por estar directamente relacionada con la clase. Para ellos se han establecido diversas adaptaciones de las técnicas conocidas y otras creadas que cubran las expectativas de la prueba unitaria.

1.1.1.1 Enfoque de Caja Negra

Las pruebas de caja negra son aplicables a la prueba unitaria tanto para sistemas estructurados como Orientados a Objeto, y en este último, en todas las divisiones (prueba a nivel de *Método* y de *Intra-Clase*) [14,18].

La propuesta realizada de Offut e 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* [14].

Otra de las propuestas realizada por *McGregor y Sykes* [12,19] trata específicamente en 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.

Por otra parte, *Robert Binder* propone una técnica llamada *N+* [1,19] y se divide en 2 pasos:

1. Generar casos de prueba cumpliendo con el criterio *All Round-Trip Path* (Todos los Caminos Ida y Vuelta).

2. Generar casos de prueba que cubra el criterio *Sneak Path* (Caminos Errados).

1.1.1.2 Enfoque de Caja Blanca

Harrold y Rothermel [15] 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 [6]. Para los métodos individuales en una clase, y los métodos que envían mensajes a otros métodos en la clase, la técnica propuesta es similar a las técnicas de prueba existentes del flujo de datos.

1.1.1.3 Técnicas Basadas en Errores: Criterio de Análisis de Mutantes

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 casos de prueba capaz de revelar las diferencias de comportamiento existente entre el programa y sus mutantes [15]. Cuando el mutante presenta un comportamiento diferente del programa se dice que es un *mutante muerto*. De lo contrario, si para todo el conjunto de prueba el comportamiento del mutante es el mismo que del programa entonces se habla que el mutante es un *mutante vivo*, y se debe analizar para verificar si él es

equivalente al programa original o si el conjunto de prueba debe ser mejorado para matar al mutante vivo. Los operadores de mutación, son las reglas que definen las alteraciones que deben ser aplicadas en el programa original y así generar los mutantes. Los operadores representan una implementación de un modelo de defectos y al utilizarlos valida si el programa contiene o no el tipo de error modelado por ellos.

Un punto importante que se debe destacar, es la medida objetiva que arroja el criterio y que guarda relación con la adecuación de los casos de prueba utilizados para validar los programas. La métrica utilizada es la ***Calificación de Mutación (Mutation Score*)**, que relaciona el número de mutantes muertos con el número de mutantes generados. La calificación se calcula de la siguiente manera:

$$MS = \frac{dm - em}{m} \qquad (1.1)$$

Donde:

***MS*:** Mutation Score

***dm*:** número de mutantes muertos por los casos de pruebas en el programa.

***m*:** número total de mutantes generados del programa.

***em*:** número de mutantes equivalentes generados del programa.

La *Calificación de Mutación* varía es el intervalo de 0 a 100, mientras mayor sea la calificación más adecuado es el conjunto de casos de prueba para el programa que está siendo probado. La fórmula depende tanto de los casos de prueba que son capaces de matar mutantes, como de la capacidad de reconocer los mutantes equivalentes.

1.1.2 Herramientas que apoyan las Pruebas Unitarias para *Java*

Algunas de las herramientas que apoyan las pruebas unitarias para *Java* 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 [20] 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, que mide la cantidad de mutantes muertos por los datos de prueba, es decir, indica el porcentaje de errores detectados por los casos de prueba. Un mutante representa un error sembrado. La desventaja de la herramienta es que no detecta los mutantes equivalentes.

- **TCAT/java:** es una herramienta [21] 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 1.1.

Tabla 1.1 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		✓	✓

1.1.3 Una Herramienta “ideal” para 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. Esta etapa se realiza, en la mayoría 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.

Estos requerimientos se encuentran esquematizados en el diagrama de casos de uso mostrado en la ilustración 1.1.

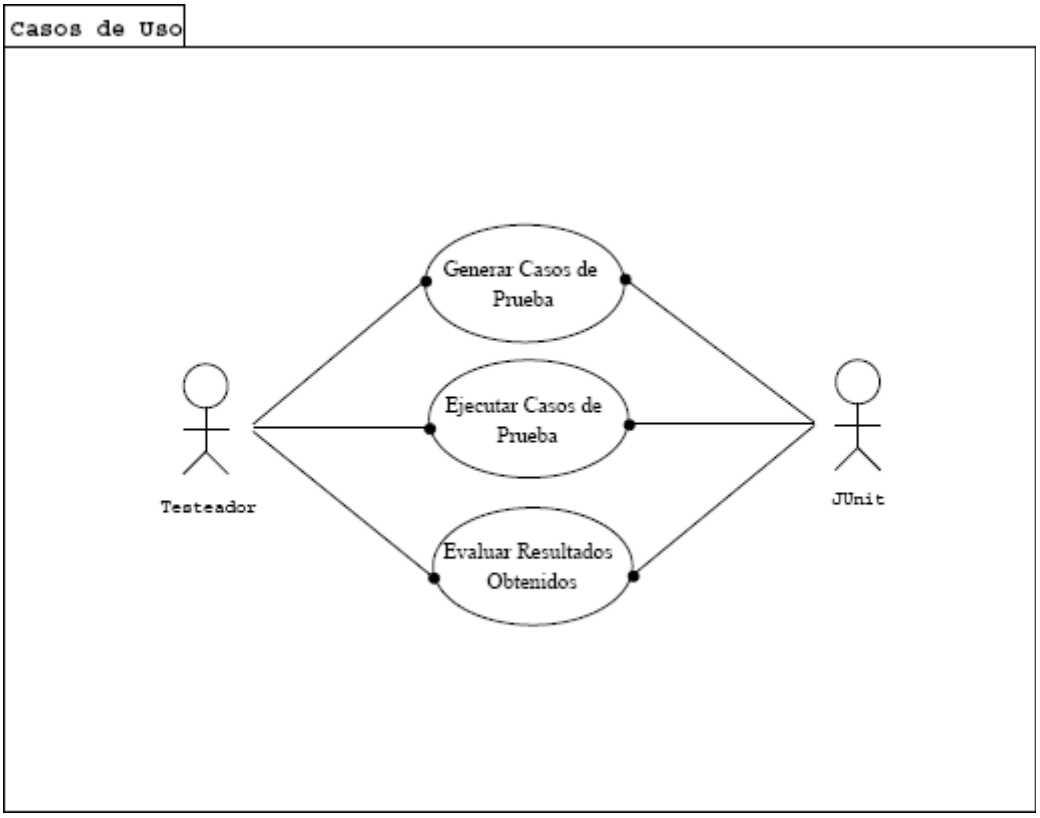

Ilustración 1.1 Diagrama de Caso de Uso de una herramienta “Ideal”

Las herramientas vistas no cubren por completo tales requerimientos, 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. *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. Esto lleva a la necesidad de una herramienta que genere casos de prueba. Por lo tanto, esta tesis se basa en *la generación asistida de casos de prueba para la Fase Unitaria*.

Los puntos que se da prioridad son:

La generación de casos de prueba para las clases a nivel de método (prueba intra-método e inter-método), y así validar el funcionamiento de los métodos por separado y en conjunto como parte de un objeto. Los casos de prueba generados por la herramienta deben validar que se cumplan los requerimientos de la clase, por lo que deben estar acordes con la especificación de ésta.

Se dio mayor énfasis a los casos de prueba que validan a nivel intra-método e inter método.

El lenguaje que se adoptó para realizar las pruebas es *Java*, por el auge que tiene en la implementación de aplicaciones desarrolladas bajo el paradigma OO. *Java* maneja el uso de excepciones en las clases, pero éstas no serán probadas, sólo se generaron casos de prueba que validan la funcionalidad de la clase y sus métodos.

La eficacia de los casos de prueba generados asistidamente fue evaluada principalmente por el criterio de *Análisis de Mutantes*, porque entrega una medida de adecuación de los casos de prueba utilizados para la validación de la clase bajo prueba. Para poder medir la efectividad de los casos de prueba, el prototipo sólo se genera casos de prueba para ***variables enteras tipo primitivo int*** **de** ***Java*** y así poder tener una base para las evaluaciones.

1.2 Objetivos

A partir de la propuesta de proyecto se desprenden los siguientes objetivos generales y específicos:

1.2.1 Objetivo General

El Objetivo General es:

Diseñar e implementar un prototipo funcional de una herramienta que asista la generación de casos de prueba unitarios para variables enteras en aplicaciones Java.

1.2.2 Objetivos Específicos

Los objetivos específicos son:

1. Establecer los requerimientos que debe cumplir una herramienta que realice pruebas unitarias.

2. Adaptar las técnicas de caja negra que se utilizarán para la construcción de los casos de pruebas para que sean programables.

3. Diseñar la herramienta según los requerimientos y la técnica para construir casos de prueba.

4. Implementar un prototipo funcional para medir la efectividad en la construcción de los casos de prueba

5. Medir la efectividad de los casos de prueba generados mediante Análisis de Mutantes

1.3 Trabajos Relacionados con la Generación de Casos de Prueba

El Grupo de Ingeniería de Software (Gris), de la Universidad de la República de Uruguay, dirigidos por el profesor Diego Vallespir [19], están trabajando en el desarrollo de una

herramienta que genera automáticamente casos de prueba para objetos y la ejecución de éstos. Se basa en las técnicas de Robert Binder (N+) y de McGregor y Sykes [1, 12]. Utiliza como lenguaje de especificación OCL y lenguaje natural. Además necesita traspasar el diagrama de estados de la clase a formato XML, según el estándar IEEE. Está orientado para cualquier lenguaje de programación. Hasta el momento, la herramienta genera casos de prueba para variables enteras. Esta herramienta fue llamada JPrePost, pero actualmente el proyecto se llama JFing.

Otro trabajo de investigación y desarrollo es el realizado por el Departamento de Ciencias de Computación de la Universidad de Texas, dirigido por Yoonsik Cheon, Myoung Yee Kim, y Ashaveena Perumandla [3]. Este proyecto utiliza algoritmo genético para la generación de los datos de prueba y hace uso de las especificaciones escritas en JML para determinar los resultados esperados. La idea básica para generar los datos de prueba consiste en producir una población inicial aleatoriamente, y realizando operaciones de cruzas y mutaciones, para crear mejores datos de prueba que encuentren mayor cantidad de errores. La medida de calidad o *fitness value* corresponde a alguna medida de cobertura de prueba, ya sea de caja negra o caja blanca, y así conseguir la mejor solución posible. El objetivo es que una población converja a una solución global. Inicialmente se produce una población aleatoria y se calcula el *fitness* de cada solución. Según el *fitness* se establece quienes serán los padres. Los padres se combinan para formar la descendencia usando al operador de cruce que encadena los genes de dos cromosomas. El objetivo del cruce es producir descendencia que combina los rasgos más buenos de ambos padres y así obtener una mejor descendencia. Varios miembros de la descendencia son mutados para producir diversidad dentro de población. Una nueva generación se selecciona de la descendencia y la población anterior. El proceso se repite hasta que se llegue a la condición de término definida.

En este capítulo se presentó la problemática a solucionar, en conjunto con algunas definiciones básicas necesarias para la compresión del documento. Además se establecieron los objetivos que moverán el desarrollo de este trabajo.

Capítulo 2:
Selección de Técnicas para Generar Casos de Prueba

El criterio que se ocupó para la elección de la técnica, guarda relación con las herramientas vistas anteriormente. *TCAT/java*, por ejemplo, genera los casos de prueba, los cuales ejecutan y determina la cobertura de los casos de prueba, pero sólo trabaja con enfoque de caja blanca. Por otra parte, *MuJava* se basa en análisis de mutantes y no genera los casos de pruebas. Estas características llevaron a seleccionar una técnica de caja negra y así complementar estas herramientas. En este trabajo se utilizará un enfoque de *Caja Negra*, utilizando para ello lo conocido como *Diseño por Contrato*.

2.1 Diseño por Contrato

El ***Diseño por Contrato*** (Design By Contract - DBC) es un método para el desarrollo de software. La principal idea es que una clase y sus clientes tienen un contrato el uno con el otro [9].

Si bien las ***precondiciones*** y de ***postcondiciones*** se usan como una manera formal para especificar los requerimientos de un software en la etapa de diseño, lo nuevo de *DBC* es que la especificación se realiza en el código ejecutable. El contrato señala las obligaciones que poseen tanto el cliente como el desarrollador, independiente de cómo las desarrollen [1]. Las obligaciones asociadas al cliente se señalan mediante las *precondiciones*; las obligaciones del desarrollador se señalan mediante las *postcondiciones*. Las *precondiciones* y las *postcondiciones* deben especificarse para cada método de la clase. Además se utiliza la llamada *invariante de clase*, que permite manifestar las restricciones de la clase, independiente del estado en que se encuentre el objeto instanciado de la clase.

El *DBC*, además de permitir dar una mayor claridad en la especificación de comportamiento de una clase, permite una mayor facilidad para realizar pruebas unitarias,

puesto que señala los resultados esperados de los métodos asociados a la clase bajo prueba [9].

Una de las técnicas de prueba que se puede aplicar utilizando *DBC* es la técnica de ***McGregor y Sykes*** [12], la cual genera los casos de prueba de las especificaciones de las clases y por ende, se basa en los requerimientos de la clase. Está técnica genera metacasos de prueba mediante dos tablas de contribuciones, una de precondiciones y la otra de postcondiciones, formando así los pares que señalan el requisito que deben cumplir los datos de entrada y cual son los resultados esperados para tales datos de entrada al ejecutarse el método, y derivando de éstos pares los casos de prueba. A continuación una descripción de la técnica.

2.2 Técnica de McGregor y Sykes

Se trata específicamente de la derivación de casos de prueba mediante un conjunto de reglas extraídas desde las precondiciones y postcondiciones de los métodos, teniendo en cuenta los conectores lógicos de *implicancia*, *conjunción*, *disyunción* y *negación*, y las sentencias condicionales *"si entonces sino"*, que están presente en las mismas. [12, 19]. Para utilizar este método se asume que la clase que se va a probar tiene una completa y correcta especificación, y que va ser probada dentro de los contextos de los modelos. Además se asume que la especificación está expresada en un lenguaje de especificación como es OCL (Object Constraint Language) o un lenguaje natural, y/o como diagrama de transición de estados. Se prefiere más la especificación formal para generar casos de prueba. La idea es identificar las reglas 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 dirigir esas reglas. Desde las reglas, se crean los casos de prueba con los valores de salida específicos, incluyendo los valores típicos y la frontera, y se determina las salidas correctas. Finalmente se agregan los casos de prueba para ver que pasa cuando una precondición es violada. Para identificar las reglas de los casos de prueba generales de precondiciones y de postcondiciones, se puede analizar cada uno de los conectivos lógicos de una condición de

OCL y se puede listar los casos de prueba que son los resultados de la estructura de esa condición.

Las reglas se presentan en las tablas 2.1 y 2.2, las que resultan de las expresiones lógicas de las precondiciones y postcondiciones.

Tabla 2.1 Contribución a partir de las precondiciones según McGregor y Sykes

Expresión Lógica	Contribución
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 2.2 Contribución a partir de las postcondiciones según McGregor y Sykes

Expresión Lógica	Contribución
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

Usando estas tablas se pueden elaborar los casos de prueba mínimos necesarios para probar los métodos (sin considerar los casos de clases de equivalencias y valores límites), usando todas las combinaciones de precondiciones y postcondiciones. Para hacerlo se deben seguir los siguientes pasos:

1. Identificar una lista de contribuciones de la precondición especificada en la entrada en la tabla 2.1 que iguale a la forma de la precondición.

2. Identificar una lista de contribuciones de la postcondición especificada en la entrada en la tabla 2.2 que iguale a la forma de la postcondición.

3. Formar los metacasos de prueba haciendo todas las posibles combinaciones de entradas desde las listas de contribuciones. Un camino es sustituir cada restricción de entrada desde la primera lista por cada ocurrencia de PRE en la segunda lista.

4. Eliminar algunas condiciones generadas por la tabla que no son significativas.

2.3 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 restricciones usado para describir expresiones acerca de modelos *UML*, pero que no es ejecutable y generalmente no es usada en el código fuente [16,17]. 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 de modelado que sirva para especificar directamente la clase en la implementación, adaptando la técnica al lenguaje.

La primera alternativa ya está siendo utilizada en una herramienta llamada *JPrePost*, la cual utiliza *OCL* adaptando la nomenclatura [19]. 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 y Sykes* son las siguientes:

1. **La técnica de *McGregor y Sykes* puede aplicarse a cualquier lenguaje de especificación:** esto es porque al describir los requisitos que debe cumplir la clase bajo prueba (Class Under Test – CUT) se señala que la clase debe estar especificada en un lenguaje como lo es OCL, en lenguaje natural y/o en diagramas de transición de estados. Por lo que deja abierto la posibilidad de utilizar cualquier lenguaje de especificación, con la

tarea de adaptar la técnica a dicho lenguaje. Además, se asume que la clase se encuentra correctamente especificada.

2. **Es un lenguaje de especificación para *Java*:** estas especificaciones se introducen en el código fuente de Java, completando la descripción de los métodos y clases [2].

3. **Existencia de las cláusulas para precondición y postcondición:** *JML* dispone de dos cláusulas para especificar las precondiciones y las postcondiciones de un método, las cuales son ***requires*** y ***ensures***. La cláusula **requires** indica la condición que se debe cumplir para que un método pueda ejecutarse, es decir, corresponde a la aserción de precondición. La cláusula **ensures** indica la condición que se debe cumplir al acabar la ejecución de un método, es decir, corresponde a la aserción de postcondición. Estas cláusulas se deben colocar antes de la declaración del método que se va a especificar.

4. **Existencia de la cláusula para la invariante de clase:** la invariante de clase es un predicado que expresa ciertas propiedades que son ciertas durante toda la vida de las instancias de la clase, como se mencionó anteriormente. En realidad esta condición debe ser cierto solamente en aquellos momentos en que el objeto es estable, es decir, no se encuentra ejecutando algunos de sus métodos (que probablemente estarán modificando su estado). Para especificar la invariante se ocupa la cláusula ***invariant***. Puede incluirse en cualquier parte del código fuente de la clase, siempre que no sea dentro de algún método.

5. **Existencia de operadores lógicos equivalentes a *OCL*:** los operadores lógicos utilizados en *JML* son equivalentes a las de *OCL*, y que son utilizadas en Java. Estas expresiones se muestran en la tabla 2.3.

Tabla 2.3 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	---

Estas similitudes justifican el uso de *JML* con la técnica de *McGregor y Sykes*. Además, *JML* tiene otras expresiones y palabras reservadas, las cuales se utilizarán para evaluar las expresiones de postcondiciones como lo muestra la tabla 2.4 [2,9].

Tabla 2.4 Expresiones y palabras reservadas de *JML*

Palabras Reservadas	Descripción
\result	Representa al valor devuelto por el método, esto se utiliza porque las variables retornadas porlos métodos son en su mayoría variables locales.
\old	Corresponde al valor que tenía la expresión antes de ejecutar el método.
(* *)	Son marcas para señalar los comentarios en JML, los que se introducen entre estas marcas. Para unirlas a la expresión lógica se utiliza el operador lógico "&&", y el comentario se evalúa como verdadero.
behavior	Señala el comportamiento del método. Existe la cláusula *normal_behavior* que señala el comportamiento cuando no se eleva una excepción, y *excepcional_behavior*, para señalar el comportamiento cuando se eleva una excepción
also	Señala que un método tiene comportamientos alternativos.
spec_public	Especifica las variables miembros de la clase

En *JML* las especificaciones son escritas en comentarios con anotación especial, los cuales comienzan con el caracter @. Además es posible utilizar los métodos ***equals()*** y ***length()*** que son propios de Java.

JML tiene más opciones de uso, pero se señalan aquellos aspectos considerados para ocupar la técnica de *McGregor y Sykes*.

Para entender lo descrito anteriormente, se puede ver el uso de las aserciones propias de *JML*, utilizando *DBC* en el código fuente de la clase ***Ejemplo*** de la ilustración 2.1.

```
import java.io.*;

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

    private /*@ spec_public @*/int max;

    //@public invariant true;

    public void Ejemplo(){ this.max = 0;}

    /*@(* method getMax(): return int *);
    @public behavior
    @requires   true;
    @ensures \result == max  ;
    @*/
    public int getMax(){ return max;}

    /*@(* method setMax(max int) *);
    @public behavior
    @requires true;
    @ensures this.max == max;
    @*/
    public void setMax(int max){ this.max = max;}

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

Ilustración 2.1 Clase *Ejemplo*, especificada con cláusulas de *JML*

Al realizar la adaptación de la técnica de *McGregor y Sykes* [12] se efectuaron los cambios a las tablas de contribuciones, mostrados en las tablas 2.5 y 2.6. En la tabla 2.2 de contribuciones de precondiciones propuesta por *McGregor y Sykes* [12] se puede observar que la cláusula ***implies*** tiene dos derivaciones en *OCL*, por lo que se tuvo que crear la cláusula para el operador de *consecuencia* Además en *JML* no existen diferencias entre los operadores de *disyunción* y *disyunción excluyente* como en *OCL*, puesto que se señalan con la mismo operador. La situación de exclusión de la disyunción se trata implícitamente en *JML* como en *Java* y en otros lenguajes de programación. Y por último, la cláusula ***if-then-else***, si bien existe una sentencia equivalente en *JML*, es poco utilizada por el uso de la

cláusula ***also***, que permite especificar varios comportamientos. Al igual que en el caso de la tabla de contribuciones de precondiciones, se establecen diferencias entre las tablas de contribuciones de postcondiciones [16, 17].

Tabla 2.5 Contribución a partir de las precondiciones adaptado a *JML*

Expresión Lógica	Contribución
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: 1 y 2 representan componentes en una expresión JML	

Tabla 2.6 Contribución a partir de las postcondiciones adaptado a *JML*

Expresión Lógica	Contribución
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)

Al aplicar las tablas de contribuciones al ejemplo anterior se derivan los metacasos de prueba señalados en la ilustración 2.2.

```
Clase: Ejemplo
Método: Maximo(int x, int y)
1.-(true, \result >= x && \result >= y && \result == x)
2.-(true, \result >= x && \result >= y && \result == y)
3.-(true, \result >= x && \result >= y && \result == x && \result == y)
```

Ilustración 2.2 Aplicación de la técnica de *McGregor y Sykes* sobre la clase *Ejemplo*

Con estos metacasos se pueden generar los casos de prueba donde los datos de entrada pueden ser cualquier par de números enteros y según estos datos, el resultado debe ser mayor o igual a cualquiera de los 2 números ingresados y que además sea igual a uno o el otro (*metacasos de prueba 1 y 2*), o bien, a ambos números, es decir, que ambos números sean iguales (*metacaso de prueba 3*). Un ejemplo es mostrado en la ilustración 2.3.

```
Datos de entrada: (3,5)
Resultado esperado: 5
```

Ilustración 2.3 Caso de prueba generado a partir de los metacasos de prueba de la clase *Ejemplo*

Los datos **(3,5)** cumplen con metacaso de prueba, ya que los datos de entrada es un par de números enteros (esto es porque la primera componente es **True**) y el resultado esperado es **5**, uno de los números de entrada que resulta ser el mayor de los 2.

Esta técnica es realizada de forma manual, pero es susceptible a 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. Los datos de entrada y los resultados esperados que son generados deben cumplir con los requerimientos extraídos de la CUT.

3. Debe existir alguna herramienta que permita ejecutar los casos de prueba, por ejemplo *JUnit*, y para ello la CUT la clase con los casos de prueba deben estar compilados.

En este capítulo se estudió cada una de las técnicas y la forma que éstas pueden ayudar a automatizar la etapa de generación de casos de prueba. Para ello se realizó una adaptación, integrando cada una en la derivación de los metacasos de prueba. Esto permite determinar el análisis y diseño del prototipo de la herramienta propuesta en el objetivo principal.

Capítulo 3:
Análisis y Diseño del Prototipo Funcional de la Herramienta

Para la construcción del prototipo funcional, se modeló el comportamiento de la clase y se determinó como estas funciones debían implementarse, identificando las clases participantes con sus respectivos atributos y métodos. Estas clases se diseñaron mediante el análisis de cada uno de los casos de uso, realizando un estudio y experimentación de cada una etapas de la creación de los casos de prueba.

3.1 Caso de Uso Extendido de la Herramienta

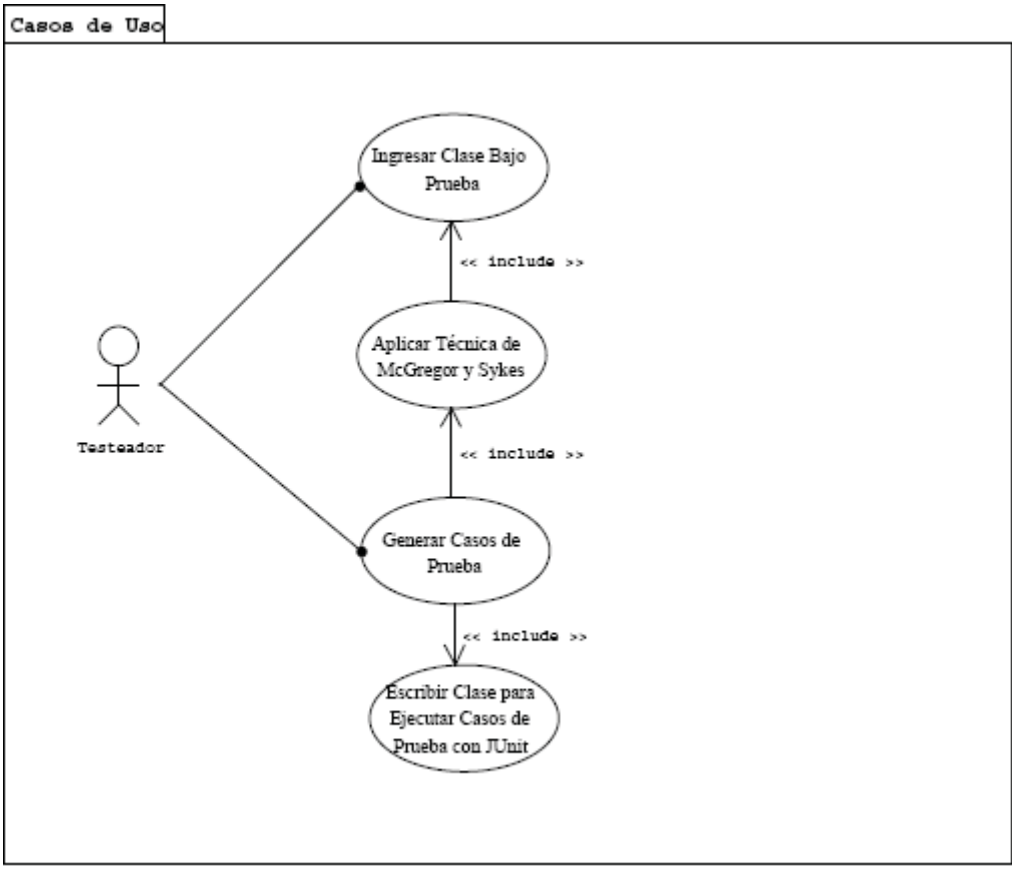

Ilustración 3.1 Diagrama de Caso de Uso de Generación de Casos de Prueba

3.1.1 Actor: Testeador

3.1.1.1 Propósito

Corresponde al usuario que efectúa las pruebas.

3.1.2 Caso de Uso: Ingresar Clase Bajo Prueba

3.1.2.1 Propósito

Ingreso del nombre de la clase bajo prueba para la identificación de las especificaciones en *JML*.

3.1.2.2 Eventos / Interacción con Interfaz de Usuario

Curso Básico

1. El testeador ingresa el nombre de la clase bajo prueba que se desea probar.

2. El sistema busca el código fuente de la clase bajo prueba en la carpeta predefinida.

3. El sistema recorre el código fuente de la clase bajo prueba para buscar las especificaciones que formarán los requerimientos de los casos de prueba.

3.1.2.3 Precondiciones

Curso Básico

1. El código fuente de la clase bajo prueba debe encontrarse en la carpeta predefinida por la aplicación.

2. La clase debe estar correctamente especificada, como lo especifica el *Diseño por Contrato*.

3.1.2.4 Postcondiciones

Curso Básico

1. Entrega las especificaciones necesarias para formar los requerimientos para construir los casos de prueba para la clase bajo prueba.

3.1.2.5 Flujo de Excepción

Curso Básico

1. Si el código fuente de la clase bajo prueba no existe en la carpeta predefinida entonces el sistema da aviso al testeador sobre la situación y se aborta la ejecución del sistema.

2. Si el código fuente de la clase bajo de prueba no está especificada entonces el sistema da aviso al testeador y se aborta la ejecución del sistema.

3. Si el código fuente de la clase bajo prueba no se encuentra especificada en *JML* entonces el sistema da aviso al testeador y se aborta la ejecución del sistema.

3.1.3 Caso de Uso: Aplicar Técnica de McGregor y Sykes

3.1.3.1 Propósito

Detección de las especificaciones de cada método y de cada invariante de la clase para producir los requerimientos los casos de prueba.

3.1.3.2 Eventos / Interacción con Interfaz de Usuario

Curso Básico

1. El sistema recibe el nombre del archivo con las especificaciones

2. El sistema lee las invariantes de clase.

3. Si existe invariante de clase entonces el sistema guarda las invariantes en el archivo predefinido para los requerimientos de casos de prueba, sino el sistema guarda las

invariantes en el archivo predefinido para los requerimientos de casos de prueba con valor verdadero.

4. Mientras existan especificaciones de métodos hacer

4.1 El sistema lee las especificaciones del método
4.2 El sistema aplica la técnica de *McGregor y Sykes*
4.3 El sistema guarda los requerimientos de prueba generados mediante la técnica en un archivo predefinido.

3.1.3.3 Precondiciones

Curso Básico

1. La especificación debe estar en una carpeta predefinida, ésta es creada por el sistema.

2. Debe cumplir con el estándar predefinido para su utilización en la creación de los requerimientos de casos de prueba.

3.1.3.4 Postcondiciones

Curso Básico

1. Entrega de requerimientos de Prueba para la construcción de los casos de prueba para la clase.

3.1.4 Caso de Uso: Generar Casos de Prueba

3.1.4.1 Propósito

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*.

3.1.4.2 Eventos / Interacción con Interfaz de Usuario

Curso Básico

1. El sistema recibe el nombre del archivo con los requerimientos de casos de prueba de cada método y las invariantes de clase.

2. Mientras existan requerimientos de casos de prueba para métodos hacer

2.1 El sistema lee los requerimientos de casos de prueba
2.2 El sistema genera los datos de entrada del método, según su tipo, cumpliendo con los requerimientos de los casos de prueba, siendo éstos los datos de prueba.
2.3 El sistema genera los datos de salida del método, cumpliendo con los requerimientos de los casos de prueba, siendo éstos los resultados esperados.
2.4 El sistema guarda los casos de prueba generados mediante los requerimientos de casos de prueba en un archivo predefinido.
2.5 El sistema guarda la invariante de clase para validar su cumplimiento.

3.1.4.3 Precondiciones

Curso Básico

1. La especificación debe estar en una carpeta predefinida, ésta es creada por el sistema.

2. Debe cumplir con el estándar predefinido para su utilización en la generación de los casos de prueba.

3.1.4.4 Postcondiciones

Curso Básico

1. Entrega los casos de prueba necesarios para construir la clase *Test* para la clase bajo prueba.

3.1.5 Caso de Uso: Escribir Clase para Ejecutar Casos de Prueba con JUnit

3.1.5.1 Propósito

Escritura de los casos de prueba generados para la CUT en el formato de las clases de prueba ejecutadas por *JUnit*.

3.1.5.2 Eventos / Interacción con Interfaz de Usuario

Curso Básico

1. El sistema recibe el nombre del archivo con los casos de prueba de cada método y la invariante de clase.

2. El sistema construye una clase llamada Test<<nombre de la clase bajo prueba>> que hereda de la clase *TestCase* de *JUnit*.

3. Mientras existan casos de prueba para métodos de la clase hacer

3.1 El sistema lee la invariante de clase
3.2 El sistema ingresa la invariante de clase en un método heredado de la clase *TestCase* de *JUnit*.
3.3 El sistema lee los casos de prueba
3.4 El sistema ingresa los casos de prueba en los métodos heredados de la clase *TestCase* de *JUnit*.

4. El sistema ingresa la invariante de clase en un método heredado de la clase *TestCase* de *JUnit*.

3.1.5.3 Precondiciones

Curso Básico

1. La especificación debe estar en una carpeta predefinida, ésta es creada por el sistema.

2. Debe cumplir con el estándar predefinido para su utilización en la construcción de la clase *TestClase.java*.

3.1.5.4 Postcondiciones

Curso Básico

1. Entrega la clase *TestClase.java* para ser ejecutada en *JUnit* y así detectar los errores existentes en la clase según los requerimientos de ésta y probar el cumplimiento de la invariante de clase cada vez que se ejecuta algún método de la clase.

3.2 Diagrama de Clase

En la construcción de la herramienta se crearon las siguientes clases, las cuales se presentan mediante las vistas que se presentan a continuación.

3.2.1 Vista: Crear Directorio de CPruebaJ

Corresponde a las clases que permiten crear el directorio donde se depositan los archivos generados por la herramienta (ilustración 3.2).

- **Clase *CarpCPruebaJ*:** Crea el Directorio de la herramienta para depositar cada uno de los artefactos creados para la creación de los casos de prueba. Además obtiene el Path para que funcione la herramienta.

- **Clase *DirectCPruebaJ*:** Corresponde al comando para crear automáticamente el directorio de la herramienta. Crea el directorio raíz y las carpetas predefinidas para los artefactos creados por la herramienta.

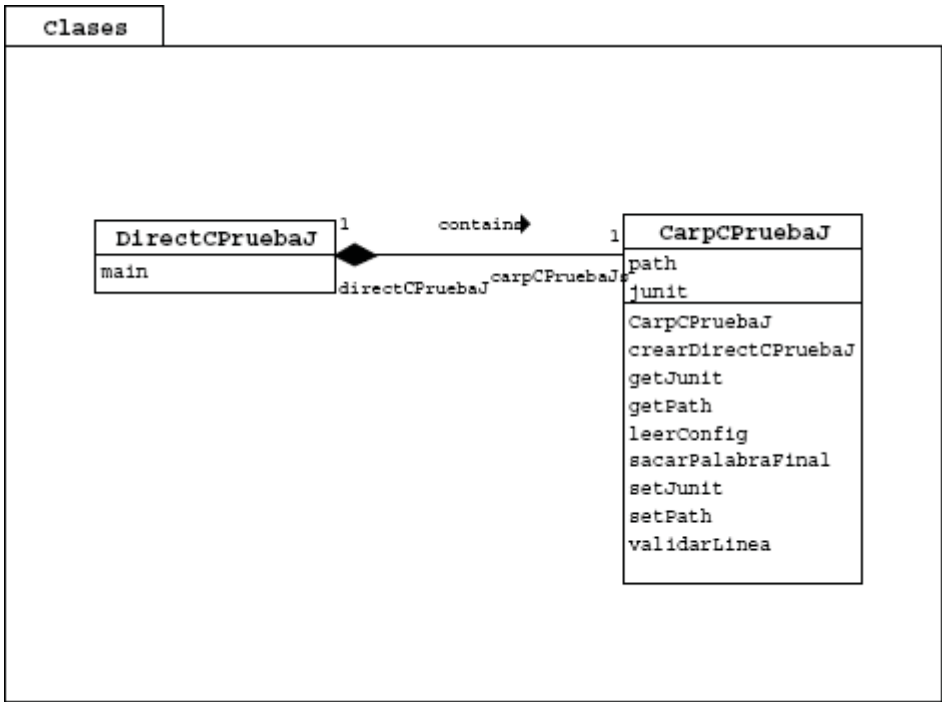

Ilustración 3.2 Vista: Crear Directorio de CPruebaJ

3.2.2 Vista: Crear Casos de Prueba

Corresponde a las clases que permiten crear los casos de pruebas (ilustración 3.3).

- **Clase *Asercion*:** Corresponde a la clase que captura las aserciones de la CUT, hechas en *JML*.

- **Clase *CrearCasosPrueba*:** Corresponde al comando para crear automáticamente los casos de prueba a partir del código fuente de la clase bajo prueba.

- **Clase *TestCase*:** Corresponde a la clase que construye los casos de prueba a partir de los *metacasos de prueba*.

- **Clase *CarpCPruebaJ*:** Crea el Directorio de la herramienta para depositar cada uno de los artefactos creados para la creación de los casos de prueba. Además obtiene el Path para que funcione la herramienta.

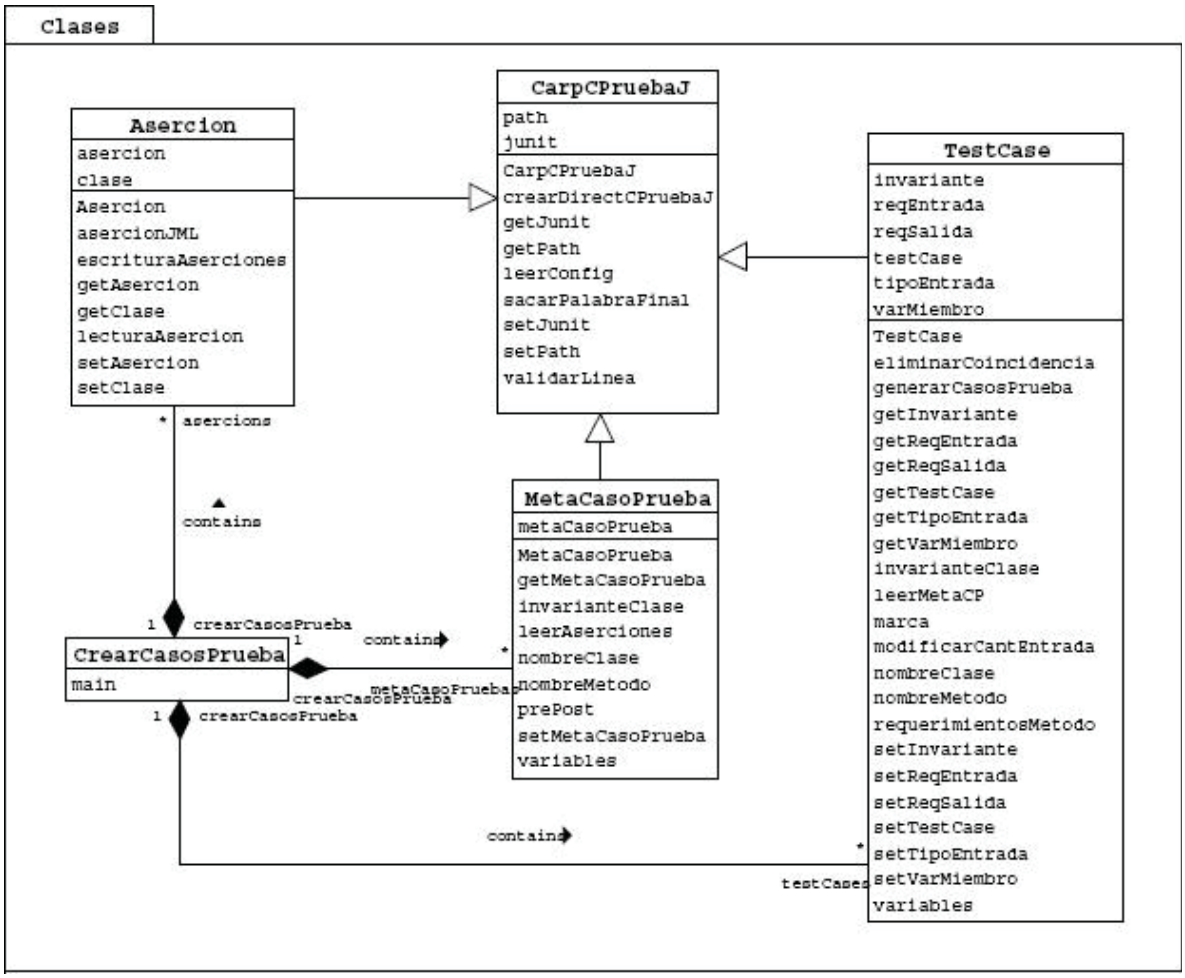

Ilustración 3.3 Vista: Crear Casos de Prueba

3.2.3 Vista: Simular Técnica de McGregor y Sykes

Corresponde a las clases que permiten emular el comportamiento de la Técnica de McGregor y Sykes, generando los *metacasos de prueba* (ilustración 3.4).

- **Clase *PrePostCondicion*:** Simula el comportamiento de la Técnica funcional de *McGregor y Sykes*, utilizando las aserciones de precondición y postcondición otorgadas por *JML*.

- **Clase *MetaCasoPrueba*:** Corresponde a la clase que crea los *metacasos de prueba*, utilizando para ello la técnica de *McGregor y Sykes*.

Ilustración 3.4 Vista: Simular Técnica de *McGregor y Sykes*

3.2.4 Vista: Generar Casos de Prueba

Corresponde a las clases que permiten generar, a partir de los *metacasos de prueba*, los casos de prueba para una clase determinada (ilustración 3.5)

- **Clase *DatosDePrueba*:** Guarda los datos de prueba de un método bajo prueba. Estos se derivan de las variables forman parte de las precondiciones y/o postcondiciones del método al que se esta generando los casos de prueba.

- **Clase *DatosEntradaEnteros*:** Corresponde al generador de datos de prueba, que nacen a partir de los *metacasos de prueba*.

- **Clase *ResultadosEsperados*:** Guarda los resultados esperados de un caso de prueba para un método.

- **Clase *TestCase*:** Corresponde a la clase que construye los casos de prueba a partir de los *metacasos de prueba*.

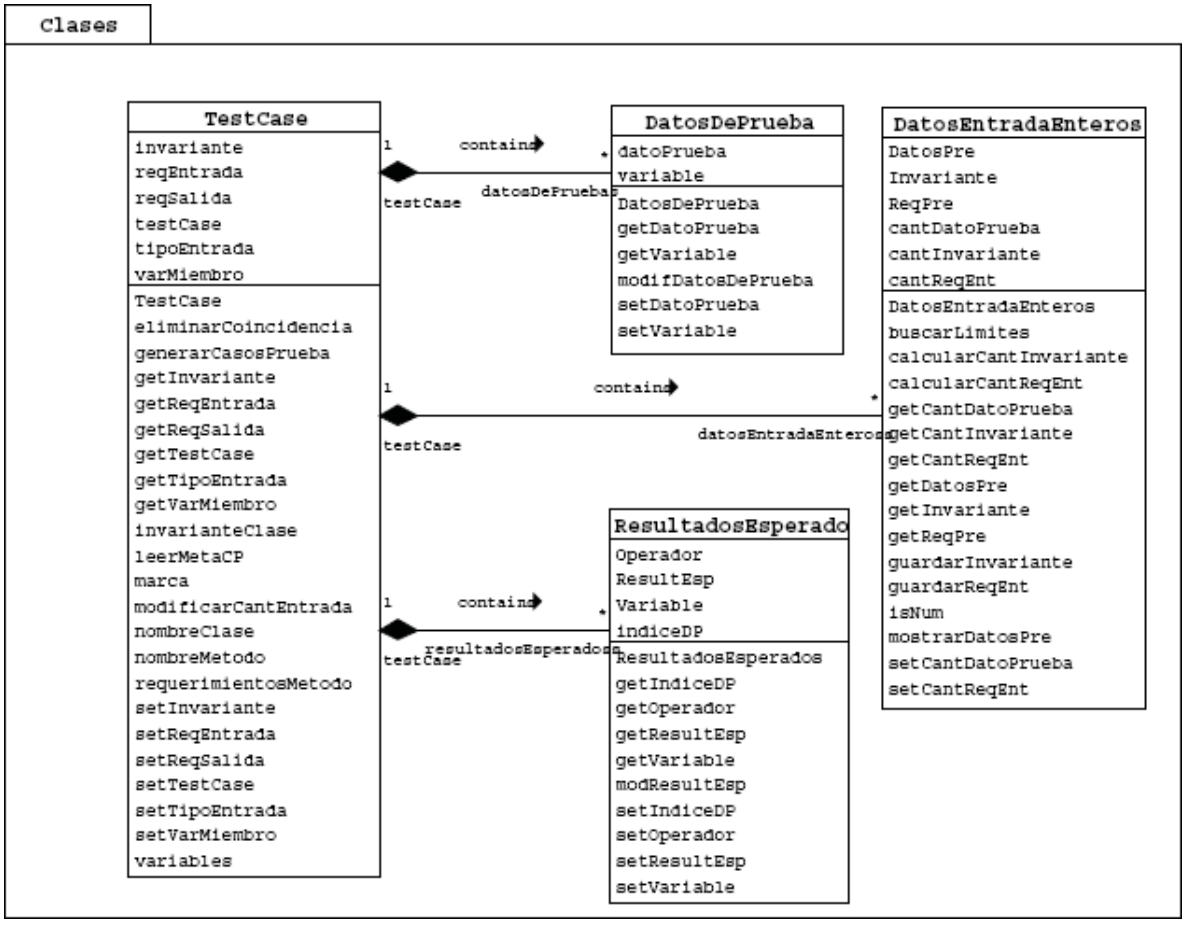

Ilustración 3.5 Vista: Generar Casos de Prueba

3.2.5 Vista: Crear Clase Test para JUnit

Corresponde a las clases que construyen, a partir de los casos de prueba, la clase que permite ejecutar estos casos de prueba (ilustración 3.6).

- **Clase *ClaseJunit*:** Genera la clase JUnit con los casos de prueba generados previamente, con la herramienta.

- **Clase *CrearTestClase*:** Corresponde al comando para crear automáticamente la clase *Test* con estándar *JUnit*. Esta clase contiene los casos de pruebas generados para la clase bajo prueba, con el fin de ejecutar estos casos de prueba con la herramienta *JUnit*.

- **Clase *CarpCPruebaJ*:** Crea el Directorio de la herramienta para depositar cada uno de los artefactos creados para la creación de los casos de prueba. Además obtiene el Path para que funcione la herramienta.

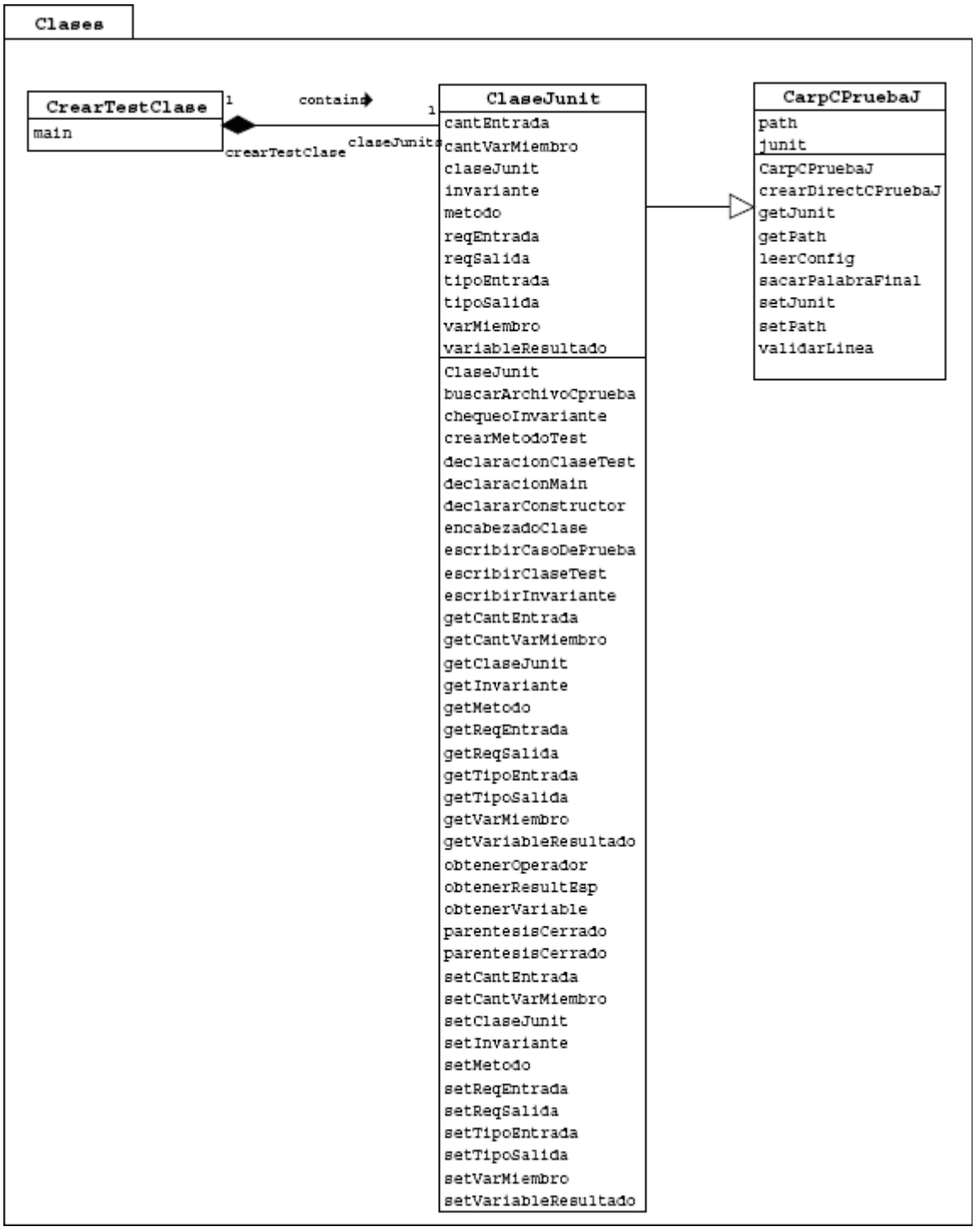

Ilustración 3.6 Vista: Crear Clase *Test* para *JUnit*

3.2.6 Vista: Compilar Clases de la Herramienta

Corresponde a las clases que permiten compilar las clases participantes en la generación de los casos de prueba, que son la CUT y la clase generada por la herramienta para ejecutar dichos casos de prueba (ilustración 3.7).

- **Clase *Compilacion*:** Permite establecer la compilación de la clase bajo prueba y la clase generada por la herramienta.

- **Clase *Compilador*:** Compilador de *Java*.

- **Clase *CompilarCUT*:** Corresponde al comando para compilar automáticamente la clase bajo prueba y la clase generada por la herramienta con estándar *JUnit*.

- **Clase *CarpCPruebaJ*:** Crea el Directorio de la herramienta para depositar cada uno de los artefactos creados para la creación de los casos de prueba. Además obtiene el Path para que funcione la herramienta.

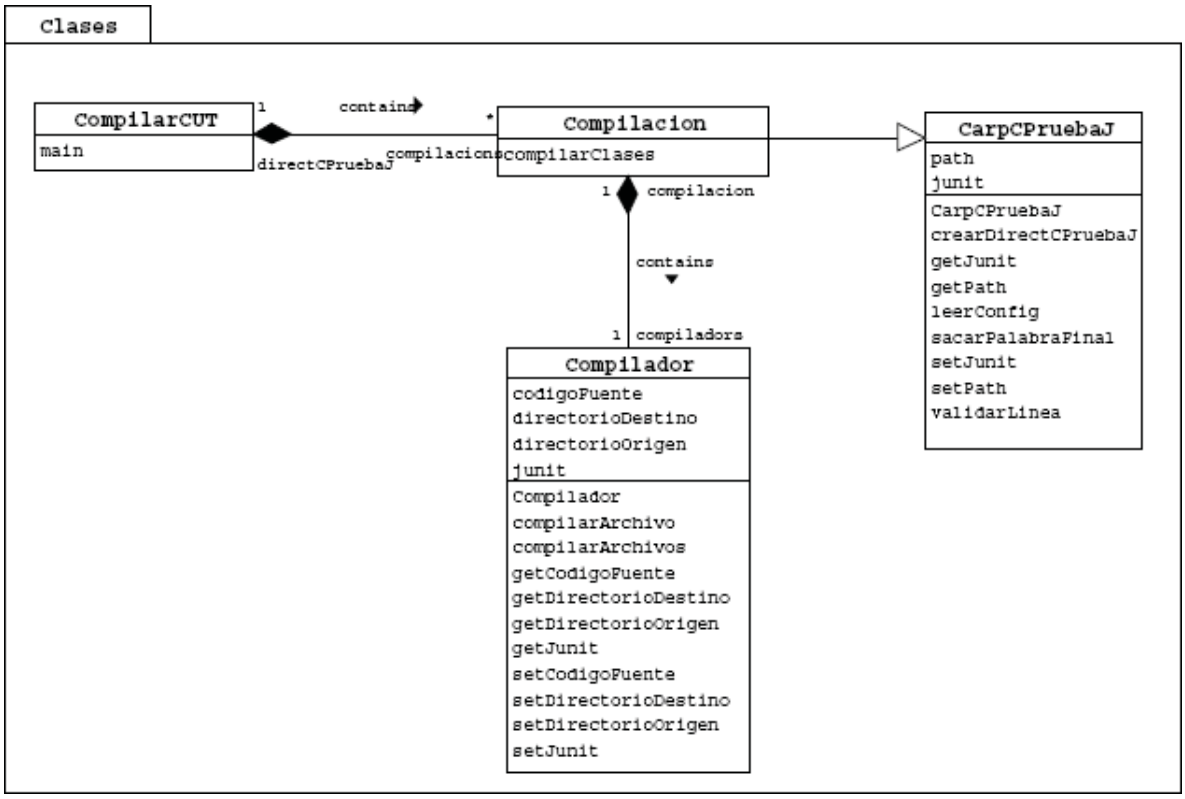

Ilustración 3.7 Vista: Compilar Clases de la Herramienta

3.3 Derivación de Casos de Prueba

La derivación de los casos de prueba no es una tarea trivial. Las funcionalidades críticas en la derivación fueron la aplicación de la técnica de *McGregor y Sykes* para conseguir los metacasos de prueba y luego la obtención de los casos de prueba a partir de estos metacasos de prueba.

3.3.1 Obtención de los Metacasos 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. Para poder automatizar la técnica se tomó las siguientes consideraciones:

1. Se establecen las precondiciones, las cuales se identifican por la cláusula ***requires***.

2. Se analizan las expresiones lógicas, según la tabla de contribución.

3. Según las contribuciones asociadas, se forma la primera componente de los requerimientos.

4. Se establecen las postcondiciones, las cuales se identifican por la cláusula ***ensures***.

5. Se analizan las expresiones lógicas, según la tabla de contribución.

6. Según las contribuciones asociadas, se forma la segunda componente de los requerimientos.

7. Por cada componente de precondición se asigna una componente de postcondición.

Al realizar el estudio para automatizar esta técnica, se consiguió la cantidad de requerimientos de entrada y de salida de un metacaso de prueba 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, así como la expresión sin operadores lógicos (según la tabla 2.5), el número de requerimientos de entrada o de salida para los metacasos de prueba es uno. En cambio, al estar una expresión compuesta por los operadores de disyunción e implicancia, la cantidad de requerimientos está dada por:

$$\sum_{n=1}^{m} \binom{m}{n} \qquad (3.1)$$

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 2.5 y 2.6. Esto se cumple tanto para las expresiones con paréntesis y sin ellos.

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

$$A \Rightarrow B \equiv \neg A \vee B \qquad (3.2)$$

3.3.2 Derivar los 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 de los metacasos de prueba. 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* [7], puesto que los errores se concentran en los límites de los datos participantes. Así los datos de prueba se conforman por valores límites, extraídos de los metacasos de prueba y un valor medio. La derivación de los casos no se concentra en validar que el método haga lo que no tiene que hacer, por lo que es decisión del testeador ingresar este tipo de casos, cumpliendo el formato de los casos de prueba. 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 los atributos que participan en los metacasos de prueba.

Sea ***PM*** el conjunto de los parámetros de un método. Su cardinalidad se encuentra en el rango de **[0, *n*]**:

$$(\#PM \in [0, n])(n \in \mathrm{N}) \qquad (3.3)$$

El rango comienza en **0** porque hay métodos que no tienen parámetros de entrada.

Sea ***AC*** el conjunto de los atributos de la clase participantes en los metacasos de prueba. Su cardinalidad se encuentra en el rango de **[0, *n*]**:

$$(\#AC \in [0, n])(n \in \mathrm{N}) \qquad (3.4)$$

El rango comienza en **0** porque no siempre aparecen atributos en los metacasos prueba.

Como en los metacasos de prueba siempre son participes al menos uno de los conjuntos, Se define como ***VDP*** - *Variables de Datos de Prueba* - a la ***unión*** de los conjuntos ***PM*** y ***AC***:

$$VDP = AC \cup PM \qquad (3.5)$$

La cardinalidad de ***VDP*** está dada:

$$\#VDP = \#AC + \#PM - \#(AC \cap PM) \qquad (3.6)$$

Se denota el conjunto **VPD** como:

$$VDP = \{vdp_1, vdp_2, ..., vdp_i, ..., vdp_m\} \qquad (3.7)$$

Sea ***RE*** el conjunto de resultados esperados derivados de un metacasos de prueba. Su cardinalidad se encuentra en el rango de **[1, *n*]**:

$$(\#RE \in [1, n]) \qquad (3.8)$$

El rango comienza en **1** porque un método siempre debe realizar una acción, ya sea comportándose como función o procedimiento.

La herramienta es capaz de crear ***dp*** *datos de prueba* para una variable ***vdp*** perteneciente ***VDP***. Sea *a* y *b* perteneciente a ***Z***, se considera la tabla 3.1 para establecer la cantidad de datos de prueba.

Tabla 3.1 Cantidad de datos de prueba según expresión de metacaso de prueba

Expresión	Cantidad de datos de Prueba (*dp*)
vdp < a	3
vdp <= a	4
a < *vdp* < *b*	3
a < vdp <= b	4
a <= vdp <= b	5
vdp == a	1
vdp != a	2

Si la cardinalidad de *VDP* es *m*, es decir, existen *m* variables de datos de pruebas, y por cada variable $vdp_i \in VDP$, se crean *dp* datos de prueba, entonces la cantidad de casos de prueba que genera la herramienta es:

$$\prod_{vdp_i = vdp_1}^{vdp_m} (dp_{vpd_i}) \times \#RE \qquad (3.9)$$

Esta cantidad de casos de prueba está dada para un metacaso de prueba generado por la herramienta.

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

Debe quedar claro que no necesariamente todos los casos de prueba pueden ser utilizados, eso dependerá de la especificación de los métodos de la clase. La herramienta puede crear casos de prueba que cumplan con la especificación, pero no cumplan con la funcionalidad del método; es decisión del testeador dejar o no este tipo de casos de prueba. Además la herramienta permite el ingreso de los propios casos de prueba.

Para poder ejecutar los casos de prueba, *CPruebaJ* genera la clase con formato ***JUnit***.

Como es un prototipo de herramienta, la generación de casos de prueba es sólo para clases que trabajen con variables enteras (tipo primitivo *int*). También no elimina los casos de prueba que puedan repetirse.

La herramienta genera los casos de prueba, pero se requiere de la asistencia de programador/testeador para verificar que estos casos de prueba sirvan para encontrar errores. Aún así, el tiempo en probar una clase se considera menor en comparación al tiempo que se utiliza para aplicar la técnica, generar los casos de prueba y luego ejecutarlos con alguna herramienta.

En consecuencia, se han establecido cada caso de uso que representa la funcionalidad del prototipo, estableciendo además, las clases necesarias con sus atributos y métodos. Estas clases se derivaron a partir del comportamiento observado en cada etapa detectada, tanto en el ingreso de la CUT, la generación de los metacasos de prueba a partir de la técnica de McGregor y Sykes y la generación de los casos de prueba con la ayuda de la técnica de Valores Límites.

Capítulo 4 :

CPruebaJ - Una Herramienta para Generar Casos de Prueba

4.1 Documentos y Directorio Generados por la Herramienta

El prototipo de la herramienta fue llamada ***CPruebaJ***, es decir, ***Casos de Prueba para Java.*** Este prototipo crea varios artefactos para generar los casos de pruebas. Esto hace el uso de la herramienta más fácil y rápida de comprender.

4.1.1 Directorio CPruebaJ

Para la correcta utilización de la herramienta es necesario establecer un directorio que guarde cada artefacto generado por ésta, el cual consiste en las carpetas de la ilustración 4.1.

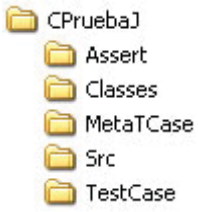

Ilustración 4.1 Directorio *CPruebaJ*

- **CPruebaJ**: corresponde al directorio raíz de la herramienta, por lo que lleva su nombre. La herramienta trabaja bajo este directorio, por lo que debe ser creada antes de ocuparla.

- **Src**: en este directorio debe colocarse el código fuente de la CUT, de donde serán extraídos los requerimientos de los casos de prueba. Además, en esta carpeta se guarda la clase que genera la herramienta para ejecutar los casos de prueba de la CUT.

- **Classes**: en este directorio se guarda la CUT compilada y la clase con los casos de prueba de la CUT. Esto permite asegurar que al ejecutar los casos de prueba *JUnit* encintrará la CUT compilada.

- **Assert**: 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 metacasos de prueba de la CUT, 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 metacasos de prueba.

4.1.2 Comandos de CPruebaJ

Para cada una de las funciones que cumple la herramienta hay un comando asociado, los cuales se describen a continuación:

- ***DirectCPruebaJ***: Permite crear automáticamente el directorio de la herramienta. Este comando se ejecuta de la siguiente manera:

```
C:\java DirectCPruebaJ
```

- ***CrearCasosPrueba***: Permite crear los casos de prueba de la CUT. Para ello, es necesario que el código fuente de la CUT se encuentre en la carpeta ***Src***. Este comando se ejecuta de la siguiente manera:

```
C:\java CrearCasosPrueba <<Nombre de la CUT>>
```

- ***CrearTestClase***: Permite crear la clase que ejecutará los casos de prueba generados por la herramienta para la CUT. Este comando se ejecuta de la siguiente manera:

```
C:\java CrearTestClase <<Nombre de la CUT>>
```

- ***CompilarCUT***: Permite compilar la clase CUT, que se encuentra en la carpeta *Src*. Si ya se ha generado la clase que contiene los casos de prueba, entonces el comando compilará ambas clases, dejando la compilación en la carpeta *Classes*. Este comando se ejecuta de la siguiente manera:

```
C:\java CompilarCUT <<Nombre de la CUT>>
```

Para utilizar los comandos de la herramienta, debe crearse previamente una carpeta llamada ***Temp*** en ***C:***, en la cual se debe colocar el archivo ***CPJ.config***, que *contiene el Path del directorio CPruebaJ*, que por defecto es ***C:\CPruebaJ.*** Este Path *puede ser modificado por el testeador.*

4.1.3 Documentos Generados por la Herramienta

Los documentos que genera la herramienta, y que utiliza son:

- ***Assert***: Es un archivo con extensión ***.txt*** que guarda las cláusulas de la especificación de la CUT, es decir las precondiciones, postcondiciones y las invariantes de clase. El formato que sigue es el siguiente:
  1. El encabezado del documento corresponde al nombre de la clase, tal como es especificado en el código de la CUT.
  2. Las especificaciones de las variables miembro también son guardadas en el

documento, si existen.
3. Seguido de las variables miembro es especificado las invariantes de clase.
4. La especificación de los métodos es guardada en el documento, lo que contempla el nombre del método con sus argumentos de entrada y el tipo de dato que devuelve (según sea el caso) y sus precondiciones y postcondiciones.

- ***MetaTCase***: es un archivo con extensión ***.txt*** que guarda los requerimientos de casos de prueba de la CUT, derivados de las precondiciones, postcondiciones y las invariantes de clase, según la adaptación de la Técnica de McGregor y Sykes, que se encuentran en el documento ***Assert***. El formato que sigue es el siguiente:
  1. El encabezado del documento corresponde al nombre de la clase, tal como es especificado en el código de la CUT.
  2. Las variables miembro también son guardadas en el documento, si existen.
  3. La invariante de clase es guardada después de las variables miembro.
  4. Para cada método de la clase especificado en el documento ***Assert***, se procede a guardar los requerimientos de casos de prueba generados con la herramienta.

- ***TestCase***: es un archivo con extensión ***.txt*** que guarda los datos de prueba y los resultados esperados de la CUT, derivados de los requerimientos de los casos de prueba, que se encuentran en el documento ***MetaTCase***. El formato que sigue es el siguiente:
  1. El encabezado del documento corresponde al nombre de la clase, tal como es especificado en el código de la CUT.
  2. Las variables miembro también son guardadas en el documento, si existen
  3. Se guarda la invariante de clase
  4. Para cada método de la CUT se realiza guarda:
     a. Nombre del método
     b. Por cada requerimiento de casos de prueba:
        i. Se guarda el requerimiento de casos de prueba

ii. Las variables correspondiente a los datos de entrada y los resultados

iii. Datos de entrada generados por la herramienta que cumplan con las precondiciones

iv. Resultados esperados generados por la herramienta que cumplan con las postcondiciones

- ***Test***: es un archivo con extensión .java que guarda la clase ejecutada por JUnit con los casos de prueba de las CUT, que se encuentran en el documento ***TestCase***. El formato que sigue es el siguiente:

1. El encabezado guarda la importación del framework JUnit:

```
import junit.framework.*
```

2. El nombre de la clase ***Test*** que hereda de la clase *TestCase*, propia de JUnit:

```
public class TestEjemplo extends TestCase
```

3. Un constructor con un llamado a **super(*String*)**:

```
public TestEjemplo (String mayor){
        super(mayor);
}
```

4. Para cada método de la CUT, se construyen métodos que contienen instancias de la CUT, con lo cual se utiliza los métodos de la clase TestCase para validar los casos de prueba generados con la herramienta, llamando a los métodos de la CUT, ingresando los datos de entrada y entregando el resultado esperado.

4.1.4 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.

Se programador/testeador debe declarar métodos que permitan el acceso a todos los atributos, para proceder a una buena prueba. Estos métodos corresponden a ***set()*** y ***get()***, los cuales deben ser públicos. Además señala una buena forma de administrar el encapsulamiento. Si bien existen libros que señalan a estos métodos como privados, se sigue la recomendación de la guía de *JUnit*, *Pragmatic Unit Testing* que dice: ***"…es probablemente mejor romper el encapsulamiento para trabajar con código probado que trabajar con un buen encapsulamiento con código no probado"*** [7]. Éstos deben declararse de la siguiente manera:

- Para asignar un valor a una variable miembro se utiliza el método ***set*: *set***<<Nombre de Variable Miembro>>. La variable miembro debe comenzar con mayúscula.

- Para obtener el valor de una variable miembro se utiliza el método ***get*: *get***<<Nombre de Variable Miembro>>. La variable miembro debe comenzar con mayúscula.

Como ejemplo se puede utilizar la especificación dada en la clase *Ejemplo* de la Ilustración 2.1.

En este capítulo se mostró cada uno de los formatos necesarios para los artefactos que genera la herramienta, así como el estándar que debe seguir el programador para que los casos de pruebas sean generados exitosamente. Además, se muestra la forma que la herramienta opera en cada una de las etapas, es decir, como genera los casos de prueba de la CUT, como construye la clase que contiene los casos de prueba para ser ejecutados y como compila la CUT en conjunto con la clase de *JUnit*. Con el término de este capítulo se da paso para probar la efectividad de los casos de prueba generados asistidamente.

Capítulo 5 :
Resultados Obtenidos con *CPruebaJ*

Para verificar que tan efectivos son los casos de prueba que es capaz de generar el prototipo, se escogieron dos casos de estudios, con diferente funcionalidad y complejidad. Para cada caso de estudio se genera casos de pruebas, con los respectivos artefactos asociados. Una vez creados los casos de prueba y la clase para ejecutarlos, se realiza las pruebas de efectividad, la cual se hizo con la técnica de prueba ***Análisis de Mutantes,*** aplicando la métrica ***Calificación de Mutación*** (fórmula 1.1). Para automatizar la creación de los mutantes se utilizó la herramienta ***MuJava***. También se utilizó esta herramienta y *JUnit* para ejecutar los casos de pruebas generados por *CPruebaJ* sobre los mutantes generados por *MuJava*.

A continuación se describe cada caso de estudio y sus respectivos resultados.

5.1 Caso de Estudio: Clase Ejemplo

La funcionalidad de la clase *Ejemplo* (Ilustración 2.1) consiste básicamente en determinar quién, entre dos números enteros, es el mayor. Para ello consta de un atributo y tres métodos, los cuales son:

- *Atributo* ***max****:* atributo que almacena el valor del número mayor encontrado.

- *Método* ***setMax(int max)***: método que permite otorgar el valor al atributo ***max***, con el valor del parámetro ***max***.

- *Método* ***getMax ()***: método que permite obtener el valor del atributo ***max***.

- *Método* ***maximo(int x, int y)***: método que permite determinar el valor mayor entre dos números enteros, los cuales están almacenados en los parámetros ***x*** e ***y***.

Al generar los casos de prueba de la clase *Ejemplo*, *CPruebaJ* construyó los siguientes artefactos:

- ***AssertEjemplo*:** contiene todas las aserciones descritas en *JML* de la clase (ilustración 5.1).

```
clase  Ejemplo
variable int max
invariante  true
metodo  getMax(): return int
requires   true
ensures \result == this.max
metodo  setMax(max int)
requires true
ensures this.max == max
metodo  maximo(x int, y int): return int
requires true
ensures \result >= x && \result >= y && (\result == x || \result == y)
```

Ilustración 5.1 AssertEjemplo

- ***MetaTCaseEjemplo***: contiene los metacasos de prueba de la clase *Ejemplo*, derivados a partir de las aserciones almacenadas en *AssertEjemplo* (ilustración 5.2).

```
clase  Ejemplo
variable int max
invariante  true
metodo  getMax(): return int
rqcp: true , \result == this.max
metodo  setMax(max int)
rqcp: true , this.max == max
metodo  maximo(x int, y int): return int
rqcp: true , \result >= x && \result >= y &&\result == x&&\result == y
rqcp: true , \result >= x && \result >= y &&\result == x
rqcp: true , \result >= x && \result >= y &&\result == y
```

Ilustración 5.2 MetaTCaseEjemplo

- ***TestCaseEjemplo***: contiene los casos de pruebas generados para la clase *Ejemplo*. Estos se crearon a partir de los metacasos de pruebas almacenados en *MetaTCaseEjemplo* (ilustración 5.3).

```
clase  Ejemplo
variable int max
invariante  true

metodo  getMax(): return int

rqcp: true , \result == this.max
V-R: max; resultados
cp0:Integer.MIN_VALUE;\result == Integer.MIN_VALUE
cp1:Integer.MAX_VALUE;\result == Integer.MAX_VALUE
cp2:0;\result == 0
#Fin casos de prueba

metodo  setMax(max int)

rqcp: true , this.max == max
V-R: max; resultados
cp0:Integer.MIN_VALUE;this.max == Integer.MIN_VALUE
cp1:Integer.MAX_VALUE;this.max == Integer.MAX_VALUE
cp2:0;this.max == 0
#Fin casos de prueba

metodo  maximo(x int, y int): return int

rqcp: true , \result >= x && \result >= y &&\result == x&&\result == y
V-R: x,y; resultados
cp0:Integer.MIN_VALUE,Integer.MIN_VALUE;\result >= Integer.MIN_VALUE
cp2:Integer.MIN_VALUE,Integer.MIN_VALUE;\result == Integer.MIN_VALUE
cp16:Integer.MAX_VALUE,Integer.MAX_VALUE;\result >= Integer.MAX_VALUE
cp18:Integer.MAX_VALUE,Integer.MAX_VALUE;\result == Integer.MAX_VALUE
cp32:0,0;\result >= 0
cp34:0,0;\result == 0

rqcp: true , \result >= x && \result >= y &&\result == x
V-R: x,y; resultados
cp3:Integer.MAX_VALUE,Integer.MIN_VALUE;\result >= Integer.MAX_VALUE
cp4:Integer.MAX_VALUE,Integer.MIN_VALUE;\result >= Integer.MIN_VALUE
cp5:Integer.MAX_VALUE,Integer.MIN_VALUE;\result == Integer.MAX_VALUE
```

Ilustración 5.3 Extracto de TestCaseEjemplo

Para verificar que se han generado la cantidad de metacasos de prueba y respectivamente los casos de pruebas se muestra el análisis de los artefactos *MetaTCaseEjemplo* y *TestCaseEjemplo.*

5.1.1 Metacasos de Prueba de *Ejemplo*

El encabezado (azul) es estándar para todos los documentos generados por la herramienta. En las especificaciones del método *maximo* (Ilustración 5.3) se observa la existencia tanto del operador *de conjunción* como del operador de *disyunción* en las postcondiciones y sólo la existencia de la cláusula *true* en las precondiciones. Por lo tanto, las precondiciones generan *un metacaso de prueba de entrada*. Por otra parte, para las postcondiciones, la herramienta genera una mayor cantidad de metacasos de prueba de salida, según el número de expresiones unidas por él. Aplicando la fórmula 3.1, se obtiene la siguiente cantidad de metacasos de prueba de salida:

$$\textit{Metacasos de Prueba de Salida} \longrightarrow \sum_{n=1}^{2} \binom{2}{n} = 3$$

Realizando las combinaciones entre las precondiciones y las postcondiciones, se generan 3 metacasos de prueba para los casos de prueba:

Metacasos de prueba de Entrada x Metacasos de prueba de Salida = Total Metacasos de prueba
1 x 3 = 3

Este resultado coincide con el número de metacasos de prueba generados por *CPruebaJ* (Ilustración 5.3, rojo). Además los metacasos de prueba cumplen con la técnica de *McGregor y Sykes.*

5.1.2 Casos de Prueba de *Ejemplo*

Con respecto a la generación de casos de prueba, se utiliza como se mencionó, además de la técnica de **McGregor** y **Sykes**, la técnica de caja negra ***Análisis Valores Límites.*** Estos casos de prueba se generan para todos los metacasos de pruebas derivados. Si bien genera un buen número de casos de prueba, es necesaria la asistencia del testeador. Los casos de prueba serán más exactos mientras mejor esté especificada la clase. En este caso, las aserciones del método ***maximo*** no señalan específicamente quien es el máximo entre los dos números (x e y), pero da las directrices para determinar el resultado esperado según la entrada. La herramienta construyó una gran cantidad de casos de prueba para cada

metacaso de prueba, y se dejaron aquellos casos que cumplen con la funcionalidad del método. Aplicando la fórmula 3.9, se obtiene la cantidad de casos prueba generada para cada metacaso de prueba en el método *máximo,* indicada en la tabla 5.1.

Tabla 5.1 Total de casos de prueba para el método *maximo*

Metacaso de Prueba	#VDP	dp_x	dp_y	#RE	N°Casos de Prueba
true ,\result >= x && \result >= y && \result == x && \result == y	2	3	3	4	36
true, \result >= x && \result >= y &&\result == x	2	3	3	3	27
true, \result >= x && \result >= y &&\result == y	2	3	3	3	27
Total de casos de Prueba					90

Esto da la suma de 90 casos de prueba para el método *maximo*. Pero la herramienta no es capaz de eliminar aquellos casos de prueba repetidos, puesto que es sólo un prototipo. Además se produjeron casos que cumplían con los requerimientos de casos de prueba, pero no cumplía con la funcionalidad del método bajo prueba, por lo que se eliminaron estos casos. Es posible dejarlos pero, insertando un mensaje para señalar que el caso no debe pasar la prueba. Por lo que se seleccionaron 24 casos de pruebas de los generados.

Para el caso de estudio se genera la clase *TestEjemplo* para ejecutar los casos de pruebas, la cual tiene la estructura mostrada en la ilustración 5.4.

```
import junit.framework.*;
public class TestEjemplo extends TestCase{
        public TestEjemplo(String prueba){ super(prueba); }

        public void testgetMax(){
                Ejemplo Obj = new Ejemplo();
                int aux = 0;
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(Integer.MIN_VALUE);
                assertEquals(Integer.MIN_VALUE,Obj.getMax());
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(Integer.MAX_VALUE);
                assertEquals(Integer.MAX_VALUE,Obj.getMax());
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(0);
                assertEquals(0,Obj.getMax()); }

        public void testsetMax(){
                Ejemplo Obj = new Ejemplo();
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(Integer.MIN_VALUE);
                assertEquals(Integer.MIN_VALUE,Obj.getMax());
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(Integer.MAX_VALUE);
                assertEquals(Integer.MAX_VALUE,Obj.getMax());
                assertTrue("Chequeo Invariante",true);
                Obj.setMax(0);
                assertEquals(0,Obj.getMax()); }

        public void testmaximo(){
                Ejemplo Obj = new Ejemplo();
                int aux = 0;
                assertTrue("Chequeo Invariante",true);
                assertTrue(Obj.maximo(Integer.MIN_VALUE,Integer.MIN_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));
```

Ilustración 5.4 Extracto de clase TestEjemplo

5.1.3 Validación de los Casos de Prueba con Análisis de Mutantes

Para validar los casos de prueba, y así certificar su eficacia para encontrar errores en la clase, se utilizó la herramienta ***MuJava***. Ésta generó mutantes de clase y de método, cuya cantidad y tipo se muestra en el gráfico de barras (ilustración 5.5). Los tipos de mutantes se clasifican por el operador aplicado en la mutación. La descripción de cada operador de mutación aplicado es descrita en cada análisis.

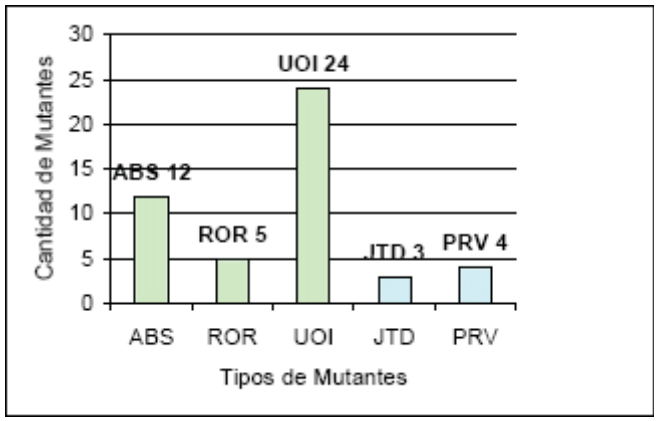

Ilustración 5.5 Mutantes de la Clase *Ejemplo*

5.1.3.1 Mutantes generados a nivel de método

Descripción de Mutantes

Los mutantes generados a nivel de método son los mostrados en el gráfico de barras en verde, y se clasifican según el operador de mutación aplicado, los cuales son:

- **ABS**: Este operador de mutación realiza un cambio de valor de una variable por otro valor. Para la clase *Ejemplo*, se generan 12 mutantes mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.6.

```
19      return this.max;
20    }
21
22    public void setMax( int max )
23    {
24      this.max = max;
25    }
26
27    public int maximo( int x, int y )
28    {
29      int z = 0;
30      if (x >= y) {
31        z = x;
32      } else {
33        z = y;
```

```
19      return this.max;
20    }
21
22    public void setMax( int max )
23    {
24      this.max = 0;
25    }
26
27    public int maximo( int x, int y )
28    {
29      int z = 0;
30      if (x >= y) {
31        z = x;
32      } else {
33        z = y;
```

Ilustración 5.6 Extracto del código original de Clase Ejemplo y de un Mutante ABS

A la izquierda de la ilustración 5.6 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 24). En el código original se declara el atributo **max** con el valor del parámetro ***max***. En el mutante se setea el atributo **max** con el valor ***0***.

- **ROR**: Este operador de mutación reemplaza un operador de comparación por otro. Para la clase *Ejemplo*, se generan 5 mutantes mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.7.

```
24      this.max = max;
25    }
26
27    public int maximo( int x, int y )
28    {
29      int z = 0;
30      if (x >= y) {
31        z = x;
32      } else {
33        z = y;
34      }
35      return z;
36    }
37
```

```
24      this.max = max;
25    }
26
27    public int maximo( int x, int y )
28    {
29      int z = 0;
30      if (x <= y) {
31        z = x;
32      } else {
33        z = y;
34      }
35      return z;
36    }
37
```

Ilustración 5.7 Extracto del código original de Clase Ejemplo y de un Mutante ROR

A la izquierda de la ilustración 5.7 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 30). El operador de mutación cambia el operador de comparación ***mayor o igual*** del código original por ***menor o igual***.

- **UOI**: Este operador de mutación reemplaza un operador de comparación por otro. Para la clase *Ejemplo*, se generan 24 mutantes mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.8.

```
25    }
26
27    public int maximo( int x, int y )
28    {
29       int z = 0;
30       if (x >= y) {
31          z = x;
32       } else {
33          z = y;
34       }
35       return z;
36    }
37
38 }
```

```
25    }
26
27    public int maximo( int x, int y )
28    {
29       int z = 0;
30       if (x >= ++y) {
31          z = x;
32       } else {
33          z = y;
34       }
35       return z;
36    }
37
38 }
```

Ilustración 5.8 Extracto del código original de Clase Ejemplo y de un Mutante UOI

A la izquierda de la ilustración 5.8 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 30). El operador de mutación agrega el operador de ***suma unario*** al parámetro y.

Ejecución de los casos de pruebas generados por CPruebaJ en los Mutantes

Al ejecutar los casos de pruebas generados por CPruebaJ en cada mutante, se obtuvo los resultados de la tabla 5.2.

Tabla 5.2 Resultados de casos de prueba sobre los mutantes a nivel de método de la clase *Ejemplo*

Descripción	ABS	ROR	UOI	Totales
Mutantes Vivos	0	0	8	**8**
Mutantes Muertos	12	4	16	**32**
Mutantes Equivalentes	0	1	0	**1**
Total Mutantes Generados	12	5	24	**41**
Calificación de Mutación	**100%**	**100%**	**67%**	**80%**

Como muestra los resultados, se genera un mutante equivalente, el cual se comporta de una manera similar al código original.

Al ver los resultado nace la pregunta sobre cuáles mutantes no fueron cubiertos por los casos de prueba. Todos estos fueron del tipo **UOI**, los cuales insertan *operadores*

aritméticos unarios al código fuente. Al ver el próximo caso de estudio se puede responder con claridad esta pregunta. Aún así, el resultado es muy satisfactorio, ya que se consiguió cubrir un ***80%*** de los errores insertados por los mutantes a nivel de método.

5.1.3.2 Mutantes generados a nivel de clase

Descripción de Mutantes

Los mutantes generados a nivel de clase son los mostrados en el gráfico de barras en azul, y se clasifican según el operador de mutación aplicado, los cuales son:

- **JTD**: Este operador de mutación borra los usos de la palabra clave ***this***. Para la clase *Ejemplo*, se generan 3 mutantes mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.9.

```
14      this.max = 0;
15  }
16
17  public int getMax()
18  {
19      return this.max;
20  }
21
22  public void setMax( int max )
23  {
24      this.max = max;
25  }
26
27  public int maximo( int x, int y )
28  {
```

```
14      this.max = 0;
15  }
16
17  public int getMax()
18  {
19      return this.max;
20  }
21
22  public void setMax( int max )
23  {
24      max = max;
25  }
26
27  public int maximo( int x, int y )
28  {
```

Ilustración 5.9 Extracto del código original de Clase Ejemplo y de un Mutante JTD

A la izquierda de la ilustración 5.9 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 24). En el código mutado se muestra el atributo **max** sin la palabra reservada ***this***.

- **PRV**: Este operador de mutación reemplaza una variable de por otra, ya sea un parámetro o un atributo de la clase. Para la clase *Ejemplo*, se generan 4 mutantes

mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.10.

```
25    }                                    25    }
26                                         26
27    public int maximo( int x, int y )    27    public int maximo( int x, int y )
28    {                                    28    {
29       int z = 0;                        29       int z = 0;
30       if (x >= y) {                     30       if (x >= y) {
31          z = x;                         31          z = y;
32       } else {                          32       } else {
33          z = y;                         33          z = y;
34       }                                 34       }
35       return z;                         35       return z;
36    }                                    36    }
37                                         37
38 }                                       38 }
```

Ilustración 5.10 Extracto del código original de Clase Ejemplo y de un Mutante PRV

A la izquierda de la ilustración 5.10 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 31). El operador de mutación cambia variable asigna a **z**, de **x** a **y**.

Ejecución de los casos de pruebas generados por CPruebaJ en los Mutantes

Al ejecutar los casos de pruebas generados por *CPruebaJ* en cada mutante, se obtuvieron los resultados de la tabla 5.3.

Tabla 5.3 Resultados de casos de prueba sobre los mutantes a nivel de clase de la clase *Ejemplo*

Descripción	JTD	PRV	Totales
Mutantes Vivos	0	0	**0**
Mutantes Muertos	1	4	**5**
Mutantes Equivalentes	2	0	**2**
Total Mutantes Generados	3	4	**7**
Calificación de Mutación	**100%**	**100%**	**100%**

Si bien el objetivo se enfoca en los mutantes de método, es interesante ver que sucede con los mutantes a nivel de clase y como se comportan los casos de prueba frente a este tipo de

errores. El resultado obtenido es completamente exitoso, aunque sean pocos los mutantes generados a nivel clases.

5.2 Caso de Estudio: Clase *PilaEntera*

La clase *PilaEntera* es una implementación particular de una pila. Para ello consta de dos atributos y siete métodos, los cuales son:

- *Atributo* ***elemento***: atributo que contabiliza los elementos almacenados en la pila.

- *Atributo* ***suma***: atributo que almacena la suma los valores de los elementos ingresados a la pila.

- *Método* ***PilaEntera()***: método constructor de la clase, asignando a cada atributo el valor inicial.

- *Método* ***setSuma(int suma)***: método que permite otorgar el valor al atributo ***suma***, con el valor del parámetro ***suma***.

- *Método* ***getSuma()***: método que permite obtener el valor del atributo ***suma***.

- *Método* ***setElemento(int elemento)***: método que permite otorgar el valor al atributo *elemento*, con el valor del parámetro ***elemento***.

- *Método* ***getElemento()***: método que permite obtener el valor del atributo ***elemento***.

- *Método* ***push(int num)***: método que permite agregar elementos a la pila. Sólo puede ingresar un entero mayor o igual que cero y menor que cien. Este método hace que la pila aumente en uno la cantidad de elementos que contiene y que aumente el miembro de la clase *suma*, en una cantidad igual al entero ***num***.

- Método ***pop()***: método que permite sacar un elemento de la pila. Devuelve un entero seteado en ***cien*** en caso que el acumulador sea mayor o igual que ***cien***, en caso contrario devuelve el valor del acumulador. Además, resta al acumulador el entero devuelto.

El código de la clase Pila se encuentra en la ilustración 5.11.

El comportamiento de la clase depende de la máxima cantidad de elementos que la pila acepta. Es por ello que la invariante exige que la cantidad de *elementos* sea *mayor o igual* a *cero* y *menor o igual* a *cien*. Además obliga a que la *suma* de los elementos ingresados sea siempre *mayor o igual* a *cero*.

CPruebaJ generó los siguientes artefactos:

- ***AssertPilaEntera*****:** contiene todas las aserciones descritas en *JML* de la clase (ilustración 5.12).

- ***MetaTCasePilaEntera*****:** contiene los metacasos de prueba de la clase *PilaEntera*, derivados a partir de las aserciones almacenadas en *AssertPilaEntera* (ilustración 5.13).

- ***TestCasePilaEntera*****:** contiene los casos de pruebas generados para la clase *PilaEntera*. Estos se crearon a partir de los metacasos de pruebas almacenados en *MetaTCasePilaEntera* (ilustración 5.14).

Para verificar que se han generado la cantidad de metacasos de prueba y respectivamente los casos de pruebas, se muestra el análisis de los artefactos *MetaTCasePilaEntera* y *TestCasePilaEntera*.

```
import java.io.*;

public class PilaEntera{
    //@(*class PilaEntera*);

    private /*@ spec_public @*/int suma;
    private /*@ spec_public @*/int elemento;

    //@public invariant suma >= 0 && elemento >=0 && elemento <=1000;

    public void PilaEntera()

    /*@(* method getSuma(): return int *);
    @public behavior
    @requires   true;
    @ensures \result == this.suma  ;
    @*/
    public int getSuma(){ return this.suma;}

    /*@(* method getElemento(): return int *);
    @public behavior
    @requires   true;
    @ensures \result == this.elemento  ;
    @*/
    public int getElemento(){ return this.elemento; }

    /*@(* method setSuma(suma int) *);
    @public behavior@requires suma >= 0;
    @ensures this.suma == suma;
    @*/
    public void setSuma(int suma){ this.suma = suma;}

    /*@(* method setElemento(elemento int) *);
    @public behavior
    @requires elemento >=0 && elemento <=1000;
    @ensures this.elemento == elemento;
    @*/
    public void setElemento(int elemento){ this.elemento = elemento;}

    /*@(* method push(num int) *);
    @public behavior
    @requires num >= 0 && nun < 100 && elemento < 1000;
    @ensures suma == \old(suma)+num && elemento == \old(elemento)+1;
    @*/
    public void push(int num){
        suma = suma + num;
        elemento = elemento + 1;
    }

    /*@(* method pop():return int *);
    @public behavior
    @requires elemento >=1 && suma >= 100;
    @ensures  \result == 100 && suma == \old(suma)-100 && elemento==\old(elemento) - 1;
    @also
    @requires elemento >=1 && suma < 100;
    @ensures  \result == \old(suma) && suma == 0 && elemento == \old(elemento) - 1;
    @*/
    public int pop(){
        int result;
        if(suma >= 100){
            suma = suma - 100;
            result = 100;
        }else{
            result = suma;
            suma = 0;
        }

        elemento = elemento - 1;
        return result;
    }
}
```

Ilustración 5.11 Clase *PilaEntera*

```
clase  PilaEntera
variable int suma
variable int elemento
invariante  suma >= 0 && elemento >=0 && elemento <=1000
metodo  getSuma(): return int
requires   true
ensures \result == this.suma
metodo  getElemento(): return int
requires   true
ensures \result == this.elemento
metodo  setSuma(suma int)
requires suma >= 0
ensures this.suma == suma
metodo  setElemento(elemento int)
requires elemento >=0 && elemento <=1000
ensures this.elemento == elemento
metodo  push(num int)
requires num >= 0 && num < 100 && elemento < 1000
ensures suma == \old(suma)+num && elemento == \old(elemento)+1
metodo  pop():return int
requires elemento >=1 && suma >= 100
ensures  \result == 100 && suma == \old(suma)-100 &&
elemento==\old(elemento) - 1
requires elemento >=1 && suma < 100
ensures  \result == \old(suma) && suma == 0 && elemento ==
\old(elemento) - 1
```

Ilustración 5.12 AssertPilaEntera

```
clase  PilaEntera
variable int suma
variable int elemento
invariante  suma >= 0 && elemento >=0 && elemento <=1000
metodo  getSuma(): return int
rqcp: true , \result == this.suma
metodo  getElemento(): return int
rqcp: true , \result == this.elemento
metodo  setSuma(suma int)
rqcp: suma >= 0 , this.suma == suma
metodo  setElemento(elemento int)
rqcp: elemento >=0 && elemento <=1000 , this.elemento == elemento
metodo  push(num int)
rqcp: num >= 0 && num < 100 && elemento < 1000 , suma ==suma+num &&
elemento ==elemento+1
metodo  pop():return int
rqcp: elemento >=1 && suma >= 100 , \result == 100 && suma ==suma-100
&& elemento==elemento- 1
rqcp: elemento >=1 && suma < 100 , \result ==suma&& suma == 0 &&
elemento ==elemento- 1
```

Ilustración 5.13 MetaTCasePilaEntera

```
clase  PilaEntera
variable int suma
variable int elemento
invariante  suma >= 0 && elemento >=0 && elemento <=1000

metodo  getSuma(): return int

rqcp: true , \result == this.suma
V-R: suma; resultados
cp0:0;\result == 0
cp1:1;\result == 1
cp2:Integer.MAX_VALUE;\result == Integer.MAX_VALUE
cp3:1073741823;\result == 1073741823
#Fin casos de prueba

metodo  getElemento(): return int

rqcp: true , \result == this.elemento
V-R: elemento; resultados
cp0:0;\result == 0
cp1:1;\result == 1
cp2:1000;\result == 1000
cp3:999;\result == 999
cp4:499;\result == 499
#Fin casos de prueba

metodo  setSuma(suma int)

rqcp: suma >= 0 , this.suma == suma
V-R: suma; resultados
cp0:0;this.suma == 0
cp1:1;this.suma == 1
cp2:Integer.MAX_VALUE;this.suma == Integer.MAX_VALUE
```

Ilustración 5.14 Extracto de TestCasePilaEntera

5.2.1 Metacasos de Prueba de *PilaEntera*

En las especificaciones del método *pop()* (ilustración 5.13), por ejemplo, se observa la existencia de la palabra reservada *also*, la cual indica que depende de las precondiciones que se cumplan, es cómo se comporta el método. Es por ello que la herramienta crea *dos* tipos de metacasos de prueba. La cantidad total de metacasos de prueba para *pop()* son:

2(Metacasos de prueba de Entrada x Metacasos de prueba de Salida) = Total Metacasos de prueba
2(1 x 1) = 2

Este resultado coincide con el número de metacasos de prueba generados por CPruebaJ (ilustración 5.13, rojo). Además los metacasos de prueba cumplen con la técnica de McGregor y Sykes.

5.2.2 Casos de Prueba de *PilaEntera*

Gracias a que las especificaciones eran más exactas, la herramienta construyó una gran cantidad de casos de pruebas directamente aplicables a la clase. Aplicando la fórmula 3.9, se obtiene la cantidad de casos prueba generada para cada metacaso de prueba en el método *pop()*, indicada en la tabla 5.4.

Tabla 5.4 Total de casos de prueba para el método *pop*

Metacaso de Prueba	#VDP	dp_{suma}	$dp_{elemento}$	#RE	N° Casos de Prueba
elemento >=1 && suma >= 100 , \result == 100 && suma ==suma-100 && elemento==elemento- 1	2	4	5	3	60
elemento >=1 && suma < 100 , \result ==suma&& suma == 0 && elemento ==elemento- 1	2	4	5	3	60
Total de casos de Prueba					120

Esto da la suma de *120* casos de prueba para el método *pop*. En este caso de estudio, no fue necesario eliminar casos de prueba, por la buena especificación que existía para la clase.

Para el caso de estudio se genera la clase *TestPilaEntera* para ejecutar los casos de pruebas, la cual tiene la estructura que se muestra en la ilustración 5.15.

```
import junit.framework.*;
public class TestPilaEntera extends TestCase{

	public TestPilaEntera(String prueba){
		super(prueba);
	}

	public void testgetSuma(){
		PilaEntera Obj = new PilaEntera();
		int aux = 0;
		assertTrue("Chequeo Invariante",Obj.getSuma() >= 0 &&
	Obj.getElemento() >=0 && Obj.getElemento() <=1000);
		Obj.setSuma(0);
		assertEquals(0,Obj.getSuma());
		assertTrue("Chequeo Invariante",Obj.getSuma() >= 0 &&
Obj.getElemento() >=0 && Obj.getElemento() <=1000);
		Obj.setSuma(1);
		assertEquals(1,Obj.getSuma());
		assertTrue("Chequeo Invariante",Obj.getSuma() >= 0 &&
Obj.getElemento() >=0 && Obj.getElemento() <=1000);
		Obj.setSuma(Integer.MAX_VALUE);
		assertEquals(Integer.MAX_VALUE,Obj.getSuma());
		assertTrue("Chequeo Invariante",Obj.getSuma() >= 0 &&
Obj.getElemento() >=0 && Obj.getElemento() <=1000);
		Obj.setSuma(1073741823);
		assertEquals(1073741823,Obj.getSuma());
	}
	public void testgetElemento(){

		PilaEntera Obj = new PilaEntera();
		int aux = 0;
		assertTrue("Chequeo Invariante",Obj.getSuma() >= 0 &&
Obj.getElemento() >=0 && Obj.getElemento() <=1000);
```

Ilustración 5.15 Extracto de clase TestPilaEntera

5.2.3 Validación de los Casos de Prueba con Análisis de Mutantes

Al igual que el caso de estudio anterior, se utilizó la herramienta *MuJava*. Ésta generó mutantes de clase y de método que se explicaron en el caso anterior, cuyo tipo y cantidad se muestra en el gráfico de barra (ilustración 5.16). Además se describe el operador AOR que utlizó *MuJava* para generar mutantes para la clase.

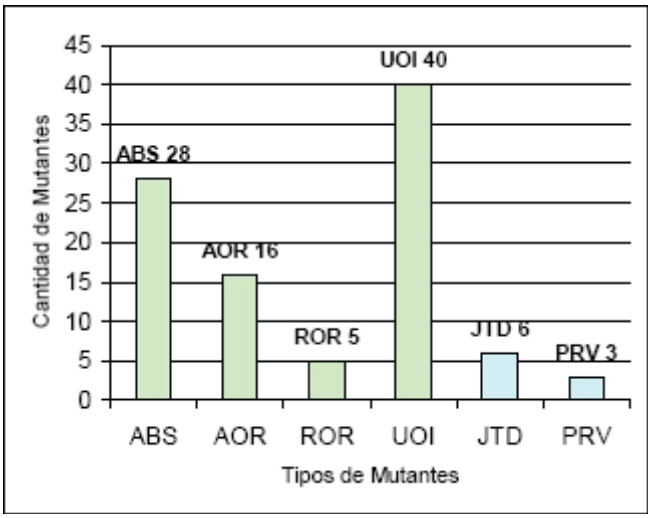

Ilustración 5.16 Mutantes de la Clase *PilaEntera*

5.2.3.1 Mutantes generados a nivel de método

Descripción de Mutantes

Al igual que el anterior, se crearon los mismos tipos de mutantes, y además se agregó a la lista otro tipo de mutante:

- **AOR**: Este operador de mutación reemplaza los operadores aritméticos básicos por otros operadores aritméticos. Para la clase *PilaEntera*, se generan 16 mutantes mediante este operador. Un ejemplo de mutante correspondiente al operador descrito se encuentra en la ilustración 5.17.

A la izquierda de la ilustración 5.17 se encuentra una parte del código de la clase en estudio. A la derecha se encuentra el mutante generado y se destaca en rojo donde mutó el código con el operador (línea 43). El operador de mutación reemplaza el operador aritmético de suma por el operador aritmético de división.

```
38  }                                          38  }
39                                             39
40  public void push( int num )                40  public void push( int num )
41  {                                          41  {
42     suma = suma + num;                      42     suma = suma + num;
43     elemento = elemento + 1;                43     elemento = elemento / 1;
44  }                                          44  }
45                                             45
46  public int pop()                           46  public int pop()
47  {                                          47  {
48     int result;                             48     int result;
49     if (suma >= 100) {                      49     if (suma >= 100) {
50        suma = suma - 100;                   50        suma = suma - 100;
51        result = 100;                        51        result = 100;
52     } else {                                52     } else {
```

Ilustración 5.17 Extracto del código original de Clase *PilaEntera* y de un Mutante AOR

Ejecución de los casos de pruebas generados por CPruebaJ en los Mutantes

Al ejecutar los casos de pruebas generados por CPruebaJ en cada mutante, se obtuvieron los resultados de la tabla 5.5.

Tabla 5.5 Resultados de casos de prueba sobre los mutantes a nivel de método de la clase *PilaEntera*

Descripción	ABS	AOR	ROR	UOI	Totales
Mutantes Vivos	0	0	0	17	**17**
Mutantes Muertos	28	16	4	40	**71**
Mutantes Equivalentes	0	0	1	0	**1**
Total Mutantes Generados	28	16	5	23	**89**
Calificación de Mutación	**100%**	**100%**	**100%**	**58%**	**81%**

Como se observa en los resultados, se genera un mutante equivalente, el cual se comporta de una manera similar al código original. Al igual que el caso de estudio anterior se dio un porcentaje del tipo de mutante **UOI** que no fue cubierto por la herramienta. Aún así, el resultado obtenido fue satisfactorio, ya que se logró un alto porcentaje de efectividad de los casos de prueba.

5.2.3.2 Mutantes generados a nivel de clase

Descripción de Mutantes

Se crearon los mismos tipos de mutantes a nivel de clase que el caso de estudio anterior, dando como resultado 9 mutantes (6 del tipo **JTD** y 3 del tipo **PRV**).

Ejecución de los casos de pruebas generados por CPruebaJ en los Mutantes

Al ejecutar los casos de pruebas generados por *CPruebaJ* en cada mutante, se obtuvieron los resultados de la tabla 5.6.

Se identificaron los mutantes equivalentes y con aquellos mutantes no equivalentes se obtuvo un resultado completamente exitoso.

Tabla 5.6 Resultados de casos de prueba sobre los mutantes a nivel de clase de la clase *Ejemplo*

Descripción	JTD	PRV	Totales
Mutantes Vivos	0	0	**0**
Mutantes Muertos	2	3	**5**
Mutantes Equivalentes	4	0	4
Total Mutantes Generados	6	3	**9**
Calificación de Mutación	**100%**	**100%**	**100%**

5.2.4 Análisis de Resultados

Como se pudo observar, la cobertura de errores de los casos de prueba para ambos casos fue del orden del *80% a nivel de método* y *100% a nivel de clase*. Esto revela una alta capacidad de los casos de prueba generados por *CPruebaJ* para encontrar errores. Pero quedó una pregunta pendiente al ver los resultados: *¿Qué tipo de mutantes no fueron cubiertos?* Al realizar el segundo caso de estudio, se mostró que los casos de prueba tenían el mismo comportamiento, aun cuando las clases fueran diferentes en comportamiento, complejidad y especificación. Al revisar cada uno de los mutantes no cubiertos de cada caso de estudio - todos del tipo **UOI** - se llegó a la conclusión de que estos errores no pueden ser cubiertos porque la funcionalidad de la clase no cambia aún con esos errores. Para demostrar lo anterior se tomó un mutante de los *no* cubiertos de cada clase de estudio y se observó su código (ilustraciones 5.18 y 5.19).

En cada uno de los casos de verificó que el ***operador aritmético unario insertado*** en el código original no afecta el resultado de obtenido ya que ***opera después de asignar el valor***

en la variable retornada, aunque se aplique *directamente* a un atributo, (*línea 24* del mutante de la clase *Ejemplo*) o **parámetro** (*línea 42* del mutante de la clase *PilaEntera*). Además, si afecta a un parámetro, este tipo de variable no se vuelve a utilizar dentro del método. Es así que los resultados retornados por los métodos no son alterados por estos errores. Es recomendable utilizar alguna herramienta de caja blanca o inspección de código para detectar previamente este tipo de errores.

```
19      return this.max;
20  }
21
22  public void setMax( int max )
23  {
24      this.max = max;
25  }
26
27  public int maximo( int x, int y )
28  {
29      int z = 0;
30      if (x >= y) {
31          z = x;
32      } else {
33          z = y;
```

```
19      return this.max;
20  }
21
22  public void setMax( int max )
23  {
24      this.max = max--;
25  }
26
27  public int maximo( int x, int y )
28  {
29      int z = 0;
30      if (x >= y) {
31          z = x;
32      } else {
33          z = y;
```

Ilustración 5.18 Mutantes no cubiertos por *CPruebaJ* de la clase *Ejemplo*

```
37      this.elemento = elemento;
38  }
39
40  public void push( int num )
41  {
42      suma = suma + num;
43      elemento = elemento + 1;
44  }
45
46  public int pop()
47  {
48      int result;
49      if (suma >= 100) {
50          suma = suma - 100;
51          result = 100;
```

```
37      this.elemento = elemento;
38  }
39
40  public void push( int num )
41  {
42      suma = suma + num--;
43      elemento = elemento + 1;
44  }
45
46  public int pop()
47  {
48      int result;
49      if (suma >= 100) {
50          suma = suma - 100;
51          result = 100;
```

Ilustración 5.19 Mutantes no cubiertos por *CPruebaJ* de la clase *PilaEntera*

En este capítulo se mostró el resultado de la utilización de la herramienta para generar casos de prueba y la efectividad de éstos para detectar errores sembrados por los operadores

de mutación aplicados. En resumen, a partir de una buena elección y estudio de técnicas de caja negra, la adaptación de las mismas para ser automatizadas, la identificación de cada una de las tareas de la herramienta y por consiguiente, la construcción del prototipo, se pudo obtener los resultados esperados, permitiendo demostrar que es posible semi-automatizar de alguna manera la etapa de generación de casos de prueba y que esta automatización es efectiva para detectar una gran cantidad de errores como demostraron los resultados obtenidos.

Conclusiones

Tal como el presente trabajo lo demuestra, se logró conseguir la adaptación de la técnica de *McGregor y Sykes*, utilizando *DBC* y *JML*. La generación de los metacasos de prueba no fue una tarea trivial, puesto que se necesitaba de un buen manejo de la lógica proposicional y de la experimentación de la técnica para analizar su comportamiento frente diversos escenarios. Así se pudo determinar el número de metacasos de prueba, dando una idea muy cercana de la cantidad de casos de prueba que pueden generar *CPruebaJ*.

Una de las decisiones claves para el buen término del prototipo, fue el establecer un estándar para el programador/testeador como una guía de buenas prácticas a la hora de programar. Esto ayudó a automatizar gran parte de la funcionalidad. Esta tarea fue posible gracias a la riqueza de lenguaje de *JML*, ya que tiene muchas formas de denotar las especificaciones de la clase, muy similares a otros lenguajes de especificación como *OCL*. Con la definición del estándar y el lenguaje con el cual escribirá los requerimientos del método, el programador tendrá que tener claro cual es la funcionalidad de la clase a implementar. Esto obliga también a los analistas a que realicen una buena toma de requerimientos en las etapas de análisis y diseño, para que las especificaciones sean más cercanas al comportamiento que se desea lograr en la clase.

Los documentos generados por *CPruebaJ* también cumplen con un estándar, para que sean de mayor comprensión para el testeador y sea más fácil lograr una automatización en las etapas que se requieren.

CPruebaJ genera actualmente el directorio de la herramienta, los artefactos antes señalados (especificación, metacasos de prueba, casos de prueba y clase ejecutada por *JUnit*) y cumple con la construcción de los metacasos de prueba 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 eficaces a la hora de cubrir 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.

Otro logro importante fue dar la funcionalidad de compilación de la CUT y la clase con los casos de prueba desde la herramienta, evitando la tarea de tener que copiar las clases compiladas a la carpeta predefinida, disminuyendo el tiempo de prueba para el programador/testeador y otorgando un orden a los artefactos generados.

Utilizando la técnica de *Análisis de Mutantes* se pudo validar los casos de prueba generados por *CPruebaJ*, utilizando para ello dos casos de estudio de diferente complejidad. Con esta experimentación se pudo concluir que los casos de pruebas son eficaces para cualquier tipo de clase. Además se mostró que mientras más cercanos son las especificaciones, mejores casos de prueba puede crear la herramienta. Fue interesante descubrir que hay errores que no pueden ser cubiertos por técnicas de caja negra y revela la necesidad de aplicar diversos enfoques de prueba para detectarlos.

Otro punto importante a destacar es el hecho que la herramienta genere la clase con los casos de prueba para ser ejecutada por *JUnit*, reduciendo considerablemente el tiempo para que estos sean ejecutados, y por ende reduce el tiempo de pruebas unitarias. Es así que los costos de enseñar a los programadores utilizar la herramienta y especificar los requerimientos en *JML*, es altamente recompensado con el tiempo y dinero que se ahorra al no contratar un externo para que haga las pruebas unitarias. Se debe pensar que basta que se le enseñe una vez al programador a utilizar *CPruebaJ*, y éste lo ocupará en todos los proyectos que se le presenten. En cambio, sería necesario contratar a un externo, enseñarle como funcional la aplicación y que es lo que debe probar siempre que se comience con otro proyecto.

Basados en los resultados obtenidos en la construcción y evaluación de la herramienta, se puede presentar como trabajos futuros el elaborar una técnica que permita identificar los casos de prueba repetidos y evitar al programador/testeador tener que indagar en los casos de prueba para eliminarlos. Otro trabajo que se puede enfrentar es adaptar la herramienta

con alguna técnica de caja blanca, que cubra aquellos errores no fueron identificados con enfoque de caja negra. La ampliación del prototipo a tipos de datos *double, float, string*, también sería una forma de incrementar la funcionalidad de la herramienta, para que en un fututo logre trabajar con objetos.

Se debe recordar que *CPruebaJ* es un complemento para las herramientas existentes, con la finalidad de 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.

[3] **Cheon Yoonsik, Yee Kim Myoung y Perumandla Ashaveena**: A Complete Automation of Unit Testing for Java Programs, Department of Computer Science, University of Texas at El Paso, EEUU. Febrero 2005. <http://www.dc.fi.udc.es/ai/tp/practica/jml/jmlintro.html>

[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] **Glenford J. Myers**: The art of Software Testing. Wiley, New York, 1979.

[6] **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, 1994, pages 154-163.

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

[8] **Labiche Yvan**: On Testing Object-Oriented Programs. LIS – Aerospatiale LAAS – CNRS

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

[10] **Ma Yu-Seung, Offutt Jeff, and Rae Kwon Yong**: MuJava: An Automated Class Mutation System. Department of Information and Software Engineering, George Mason University, E.E.UU, 2004.

[11] **Maldonado J.C., Barbosa E., y otros**: Introdução ao Teste de Software, Instituto de Ciencias Matemáticas y de Computación – ICMC-USP, San Carlos, Brasil, Versión 2004-01.

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

[13] **Pezzè M. y Young M.**: Testing Object Oriented Software. Proceedings of the 26th International Conference on Software Engineering (ICSE'04), 2004 IEEE.

[14] **Pressman Roger S.**: Ingeniería del Software. McGraw Hill, Quinta Edición, 2002.

[15] **Rizzo Vizenci Auri M**: Orientación a Objeto, Definição, Implementação e Análise de Recursos de Teste e Validação, Tesis presentada al Instituto de Ciencias Matemáticas y de Computación – ICMC-USP, San Carlos, Brasil, 2004.

[16] **Rojas Daniella, Bozo Jorge, Rusu Cristian**: 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, México 2006.

[17] **Rojas Daniella, Bozo Jorge**: Generación Casos de Prueba Unitarios para Java basados en la Técnica de McGregor y Sykes. Proceedings de la Conferencia Latinoamericana de Informática CLEI 2006, p. 87, Chile 2006.

[18] **Somerville, I.** : Ingeniería de Software, Adisson Wesley, 2000.

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

[20] ***: 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/>

[21]***: 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>

Lista de Abreviaturas o Siglas

ABS: Reemplazo del valor de una variable.

AOR: Reemplazo de Operador Aritmético Binario.

CUT: Class Under Test – Clase Bajo Prueba.

DBC: Design by Contract – Diseño por Contrato.

JML: Java Modeling Language – Lenguaje de Modelamiento de Java.

JTD: Eliminación de la palabra reservada *this.*

OCL: Object Constraint Language – Lenguaje de Restricciones de Objeto.

OO: Object Oriented – Orientado a Objeto.

PRV: Paso por referencia de una variable del mismo tipo.

ROR: Reemplazo de operador de comparación.

UML: Unified Modeling Language – Lenguaje de Modelamiento Unificado.

UOI: Inserción de Operador Aritmético Unario.

Anexo A: Instalación de CPruebaJ

Para el uso *CPruebaJ*, se deben realizar los siguientes pasos:

1. Instalar *Java*, una versión superior o igual ***j2sdk1.4***.

2. Agregar la librería ***tools.jar***, ubicada en el directorio *lib* del *Java* instalado, al CLASSPATH:

Ilustración A.1 *Librería tools.jar* en el CLASSPATH

3. Agregar el *Path* de la ubicación del ***jar*** *CPruebaJ* en el CLASSPATH

Ilustración A.2 Herramienta *CPruebaJ* en CLASSPATH

4. Agregar el *Path* de la ubicación del ***jar*** de *JUnit* en el CLASSPATH

Ilustración A.3 Herramienta *JUnit* en CLASSPATH

Anexo B: Manual de Usuario

Para utilizar la herramienta se debe seguir los siguientes pasos:

1. Crear una carpeta temporal ***Temp*** en la raíz ***C:***\, y en ella guardar el archivo de configuración el cual contiene la ubicación del directorio de *CPruebaJ*. Este archivo se llama ***CPJ.config***.

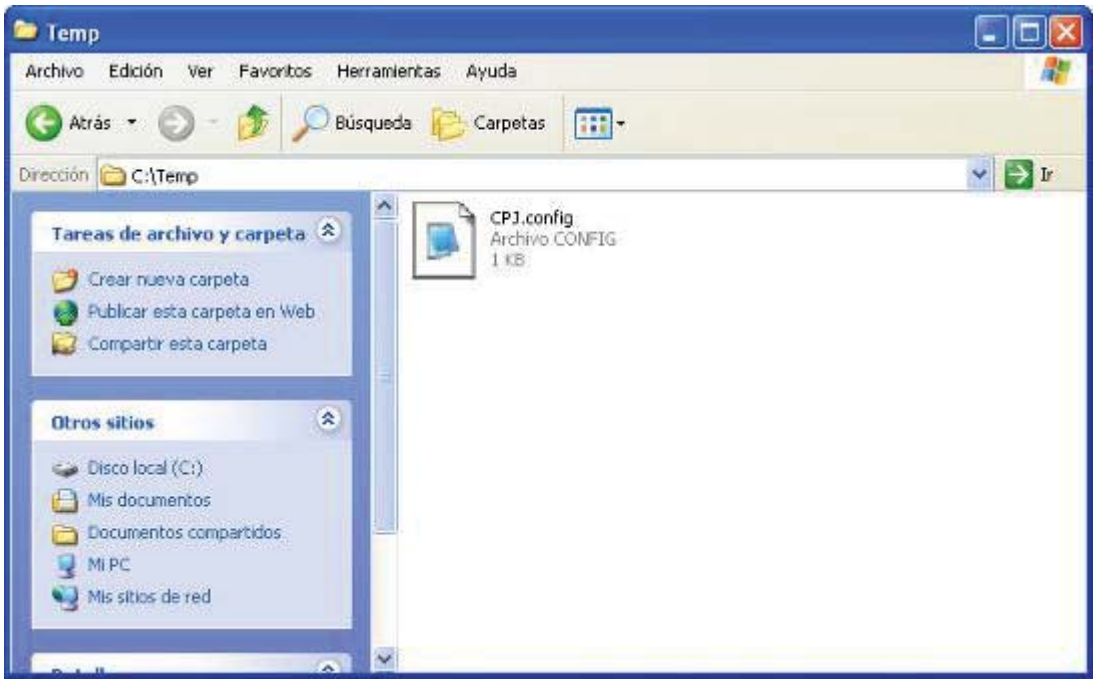

Ilustración B.1 *CPJ.config*

El archivo contiene la siguiente estructura:

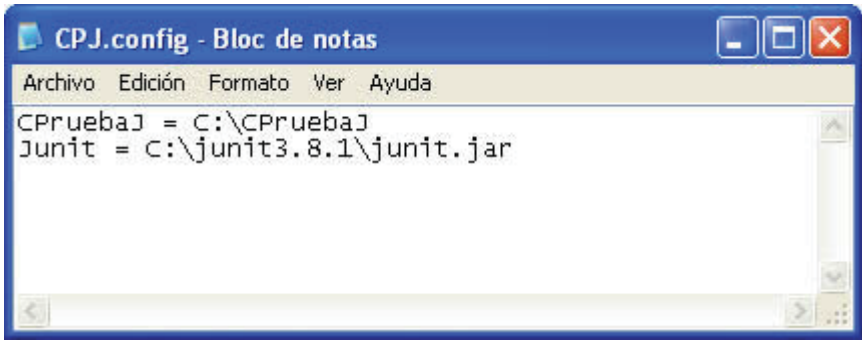

Ilustración B.2 Estructura *CPJ.config*

En este archivo se encuentra la ubicación del ***jar*** de *CPruebaJ* y el ***jar*** de *JUnit*. Estos valores están seteados por defecto, pero deben asignarse según la ubicación real de cada herramienta.

2. Crear el directorio de *CPruebaJ*. Este directorio se puede crear con el comando: **java tesis.DirectCPruebaJ**. Este comando creará la siguiente estructura:

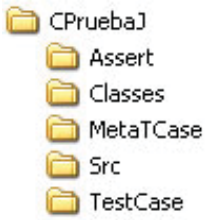

Ilustración B.3 Directorio *CPruebaJ*

- **CPruebaJ**: corresponde al directorio raíz de la herramienta

- **Src**: directorio se debe guardar el código fuente de la Clase Bajo Prueba

- **Classes**: directorio donde se guarda la Clase Bajo Prueba compilada y la clase con los casos de prueba.

- **Assert**: contiene el archivo que guarda las cláusulas de la especificación de la Clase Bajo Prueba.

- **MetaTCase**: contiene el archivo que guarda los metacasos de prueba de la Clase Bajo Prueba.

- **TestCase**: contiene el archivo que guarda los datos de prueba y los resultados esperados de la Clase Bajo Prueba.

3. Guardar el código fuente de la clase que se requiere probar en la carpeta ***Src***.

4. Ejecutar el comando java tesis.CrearCasosPrueba **<<nombre de la clase bajo prueba>>** para crear los casos de prueba para la clase.

5. Una vez creados los casos de prueba, el testeador debe revisarlos para asegurar que estos casos de pruebas son aptos para probar la clase.

6. Generar la clase para ejecutar los casos de prueba con el comando **java tesis.CrearTestClase <<nombre de la clase bajo prueba>>**.

7. Compilar las clases (CUT y/o la clase con los casos de prueba) con el comando **java tesis.CompilarCUT <<nombre de la clase bajo prueba>>**.

8. Ejecutar la clase de prueba con *JUnit*.