jPET 2.0: Un generador automático de casos de prueba sobre programas Java

Proyecto de Sistemas Informáticos

Memoria presentada por

Álvaro González Escudero

Daniel Álvarez Ramírez

y **Laura Acosta Berrio**

Dirigido por los profesores

Miguel Gómez-Zamalloa Gil

Facultad de Informática

Universidad Complutense de Madrid

Madrid, Junio de 2012

Los abajo firmantes autorizan a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Fdo. Álvaro González Escudero

Fdo. Daniel Álvarez Ramírez

Fdo. Laura Acosta Berrio

Resumen

JPet lleva desarrollándose varios años a fin de convertirlo en una herramienta competitiva en el campo del software testing. Se encarga de obtener casos de prueba (test-cases) de código java que garanticen el recubrimiento óptimo del mismo. En su comienzo la forma en que jPet mostraba la información a los usuarios no era fácil de entender, lo que hacía que no fuera útil durante el desarrollo de software. En la actualidad, se ha solucionado este problema guardando la información necesaria de los casos de prueba en ficheros .xml, mostrando al desarrollador de una manera más gráfica y sencilla el trabajo realizado. La herramienta incorpora un visor en el que se puede comprobar el valor de los datos antes y después de la ejecución del código, así como la posibilidad de ver la traza de un caso de prueba en concreto.

En este proyecto vamos a ampliar jPet añadiéndole la funcionalidad de generar tests en código java a partir de los casos de prueba almacenados en los ficheros .xml. De esta manera, el desarrollador puede verificar el funcionamiento del código que pretende testear. Los tests contienen todo lo necesario para su ejecución en java, pero están escritos a modo de plantilla por lo que, aunque ayudan al desarrollador ahorrándole tiempo, es necesaria su colaboración para que recobren sentido y pasen a ser tests válidos.

Abstract

Jpet has become a competitive tool for software testing over the years. It obtains java code test-cases that ensures an optimal coverage. From its beginnings, showing data and explaining it wasn't an easy task, so Jpet was not useful for software developing. Nowadays, this problem has been solved by saving test-cases relevant data to .xml files, guiding software developers through a more graphic and comprehensive way of the work carried out by Jpet. It has a budget that may be interesting for developers, a graphic interface in which you can check data values before and after running the piece of software tested and see a concrete test-case tracing painted in a glowing green.

So now we are going to expand and continue this ongoing project by adding functionality to generate java code unit tests (junits) using the test-cases above-mentioned. Those tests can run on Java but they are just templates. Although it will certainly help developers saving their time, they need to be accordingly modified to apply.

Índice general

Índice general **5**

1. Introducción **7**
- 1.1. Software Testing 7
- 1.2. Conceptos básicos 8
  - 1.2.1. Tipos de testing 8
  - 1.2.2. Recubrimiento 9
- 1.3. Derivación de casos de prueba 10
- 1.4. Estado del arte y tecnologías relacionadas 12
  - 1.4.1. PEX 12
  - 1.4.2. PET 13
  - 1.4.3. JUnit 14
  - 1.4.4. JUnit Factory 15
  - 1.4.5. Automatic JUnit Test Creator 15
  - 1.4.6. KeY 16
- 1.5. Objetivos 17
- 1.6. Organización de la memoria 18

2. El plug-in de Eclipse para jPET 1.0 .... **19**
- 2.1. Eclipse 19
- 2.2. Ejecución de jPET 20
  - 2.2.1. Enlazando con PET 21
  - 2.2.2. Fichero XML 21
  - 2.2.3. Plugin para Eclipse 22
- 2.3. Preferencias 24
- 2.4. Ejemplo ilustrativo 28

3. Creación de junit **30**
- 3.1. Descripción y funcionamiento 30
- 3.2. Detalles de implementación 32
  - 3.2.1. Formato del fichero junit 32
  - 3.2.2. Herramientas usadas 33
  - 3.2.3. Estructura general 34
- 3.3. Incidencias en el desarrollo 38
- 3.4. Ejemplo de junit 40

4. Junit de tipo paramétrico ... 43
4.1. Incidencias en el desarrollo ... 43
4.2. Extensiones ... 43
4.3. Ejemplo de junit paramétricos ... 45

5. Manual de usuario ... 48
5.1. Requisitos ... 48
5.2. Instalación de jPET ... 48
5.3. Manual de uso ... 50
5.3.1. Ejecución ... 50

6. Conclusiones y mejoras de la aplicación ... 54
6.1. Conclusiones ... 54
6.2. Qué se puede mejorar ... 55
6.3. Posibles ampliaciones ... 56

7. Bibliografía ... 57

8. Apéndice ... 59
A. Especificación de fichero XML ... 59
B. Implementación del método “myEquals” ... 63

1. Introducción

1.1. Software Testing

El software testing, por definición, es una actividad que consiste en probar aplicaciones para certificar su buen funcionamiento. Esto ha cobrado mucha importancia con los años, puesto que los programas ganan en extensión y complejidad a cada día que pasa. Por este motivo, se invierte mucho tiempo y dinero en el proceso de testing, y pese a que nunca se puede garantizar que la aplicación vaya a estar libre de errores, es obligatorio realizar este tipo de pruebas si queremos tener éxito al lanzar software al mercado.

La responsabilidad de crear tests eficientes y funcionales recae necesariamente en el programador, quien debe asegurarse de que se están comprobando la mayor parte posible de caminos, así como que se satisfagan de acuerdo a la especificación del programa a testear. Nada debería de fallar en la ejecución de un código crítico, por eso la correcta verificación de la funcionalidad de un programa gana aún más importancia en aplicaciones en las cuales su correcto funcionamiento resulta vital, como por ejemplo, aquellas que se encargan de la seguridad de sistemas, programas que controlan producciones o sistemas que realizan predicciones o previenen desastres. [1]

La realización de este ciclo del desarrollo tiene que ser constante desde la creación de la aplicación. De lo contrario, el coste de subsanar los errores será mayor cuanto más tarde aparezcan. Resulta por ello especialmente importante tratar de desarrollar técnicas que permitan automatizarlo, al menos parcialmente.

1.2. Conceptos básicos

1.2.1. Tipos de testing

Existen dos tipos de pruebas dependiendo de la manera de evaluar la aplicación, pruebas de caja negra (black-box-testing en inglés) y pruebas de caja blanca (glass-box-testing o white-box testing).

El black-box-testing realiza pruebas no transparentes, sin conocer la implementación de la aplicación ni su funcionamiento interno, haciendo uso sólo de sus requisitos de software. Únicamente produce las salidas esperadas con las entradas que suministremos, para que el desarrollador examine estos resultados y pueda comprobar el funcionamiento, correcto o no.

El glass-box-testing por el contrario, se basa en la implementación del programa. En este caso, se escogen entradas concretas para examinar cada una de las posibles ramas de ejecución del programa y comprobar que devuelve las salidas adecuadas.

También podemos clasificar las pruebas de software dependiendo de la fase de desarrollo o el elemento a probar, algunas son:

- *Intregation testing*: Se realiza para hacer interactuar módulos pequeños entre sí.

- *System testing*: Se realiza cuando queremos comprobar el funcionamiento de un sistema al completo.

- *Acceptance testing:* Se realiza cuando el producto está finalizado, normalmente el cliente, para comprobar que cumple con las especificaciones que se dieron al comenzar el proyecto.

- *Regression testing*: Se realiza periódicamente con el propósito de ver si los nuevos cambios en la aplicación no hacen que funcione de manera incorrecta.

- *Unit testing*: Una unidad se concibe como la parte más pequeña de una aplicación susceptible de ser testeada (métodos, en el caso de lenguajes orientados a objetos). El testing de unidad es la metodología por la cual se testean de forma independiente estas unidades.

El glass-box-testing es generalmente aplicable en integration testing y system testing, aunque principalmente se usa en unit testing. El black-box-testing, por el contrario, suele emplearse con tipos de testing de alto nivel, en los que hacer un test que compruebe algo más que la salida esperada estaría totalmente fuera de lugar. [2]

1.2.2. Recubrimiento

En el contexto de software testing se define el recubrimiento como el criterio para evaluar software e indica hasta qué punto un elemento del programa ha sido probado. Los distintos patrones de recubrimiento se basan en la comprobación de entidades más pequeñas dentro del programa, algunos de ellos son:

a) *Recubrimiento de Funciones*: Comprueba si las subrutinas, procedimientos y funciones han sido testeadas debidamente.

b) *Recubrimiento de Instrucciones*: Comprueba si se han ejecutado todas las instrucciones y sentencias del código.

c) *Recubrimiento de Condiciones*: Comprueba si las condiciones en bucles y en sentencias condicionales han sido evaluadas a cierto o falso.

d) *Recubrimiento Loop-k*: Impone que se hayan realizado k iteraciones, o el máximo número de ellas entre todos los bucles del programa si k es mayor.

1.3. Derivación de casos de prueba

Debido a la naturaleza del proyecto, nuestros tests son un híbrido entre unit testing e integration testing, por lo que vamos a hacer uso del glass-box-testing y nos vamos a centrar en él a partir de ahora.

Definimos un caso de prueba, test-case en ingles, como un elemento que permite verificar el buen funcionamiento de un elemento de software, limitado por unas condiciones o valores iniciales.

Llamamos oráculo a la plantilla que muestra la solución de la comparación de los resultados obtenidos durante la ejecución del programa con los resultados esperados. Resulta imposible para el oráculo suministrar información de casos de prueba para todas las posibles ramas de ejecución del programa.

Los casos de prueba de unidad son generalmente escritos por los programadores o en ocasiones por los "testeadores''. Un caso de prueba de unidad consta básicamente de tres partes:

a) La construcción de los datos de entrada.
b) La llamada al método (o unidad de código) a testear
c) La comprobación del resultado, llevada a cabo por el llamado oráculo.

Las derivaciones de casos de prueba se generan recubriendo el código del software que se nos proporcione. Dependiendo del recubrimiento que se lleve a cabo, podemos medir la efectividad con la que se han llevado a cabo las pruebas.

Normalmente se llevan a cabo ciertos pasos para construir las derivaciones de casos de prueba:

- Usar el diseño o código y dibujar el correspondiente gráfico de flujo

- Determinar la complejidad cíclica del gráfico de flujo dibujado.

- Definir un conjunto básico de caminos independientes.

- Prepara los casos de prueba para que ejecuten cada camino de ese conjunto básico.

Explicado de forma sencilla, el criterio de recubrimiento indica qué puntos del código se quieren alcanzar y el proceso de derivación de casos de prueba consiste en hallar las condiciones que tiene que cumplir la entrada para llegar a dichos puntos.

¿Se podrían automatizar estos casos de prueba? Para conseguir esto, haría falta automatizar tanto la generación de entradas como sus correspondientes salidas. Para las entradas hay distintas alternativas, mientras que las salidas sólo se podrían automatizar en el caso de que hubiera un lenguaje de especificación que aporte información sobre la llamada, algo de lo que no disponemos.

Los sistemas generadores producen datos concretos a partir de las restricciones de entrada mediante resolutores de restricciones. Cada vez es más habitual el uso de generadores aleatorios de entradas, pero normalmente los casos de prueba obtenidos no consiguen un recubrimiento óptimo del código, es decir, muchas veces por culpa de la aleatoriedad de la entrada no conseguimos probar todos los caminos posibles. La única solución sería considerar un número muy grande de entradas diferentes, lo que por contrapartida produciría problemas en el mantenimiento de los bancos de casos de prueba.

Por otra parte, no hay ningún conjunto de pruebas que sean perfectas para todos los programas. Dependiendo del programa o la aplicación que estemos probando, algunas pruebas pueden ser muy beneficiosas y otras incluso perjudiciales para la corrección del código.

Existen dos tipos de generación de entradas:

- Dinámicas: hace una ejecución real del código para obtener los valores.

- Estáticas: no se hace una ejecución propia, si no que se deducen los valores. A este tipo se le llama *ejecución simbólica.*[3]

El enfoque estándar de generación automática de casos de prueba de caja blanca consiste en hacer una ejecución simbólica del programa limitada mediante una estrategia de terminación

determinada. Los sistemas de restricciones obtenidos para cada camino determinan las condiciones que los datos de entrada deben cumplir para recorrer dicho camino, y se pueden entender por tanto como casos de prueba. En jPET se propone un esquema de generación automática de casos de prueba mediante ejecución simbólica en CLP para programas imperativos secuenciales con orientación a objetos, que permite obtener automáticamente casos de prueba de unidad para programas java.[4]

La manera más aceptada a la hora de generar casos de prueba consiste en ejecutar el programa de manera simbólica, obtener los casos paramétricos y, si se quiere, resolverlos para conseguir casos concretos.

Puesto que la ejecución simbólica razona en términos de caminos de ejecución, nos permite obtener un caso de prueba para cada camino de ejecución del programa, pudiendo garantizar así un buen recubrimiento con un número reducido de casos de prueba.

1.4. Estado del arte y tecnologías relacionadas

Existen varias tecnologías que orientan su funcionalidad hacia la generación de casos de prueba, y otras cuya finalidad es la generación automática de código fuente java. A continuación vamos a aproximarnos a algunas de las más importantes. También mencionaremos JUnit y Pet, tecnologías que emplearemos en el desarrollo de nuestra aplicación.

1.4.1. PEX

PEX es una herramienta desarrollada por Microsoft utilizada para generar de manera automática casos de prueba. El sistema se utiliza en la plataforma de desarrollo .NET y es capaz de analizar cualquier programa ejecutado en la máquina virtual de .NET.

PEX tiene una orientación significativamente diferente al de jPET, pues combina la ejecución simbólica, y por tanto paramétrica,

con la ejecución concreta, es decir, la ejecución ordinaria del programa. Utilizando un proceso iterativo, PEX lleva a cabo una ejecución concreta del método seleccionado para su análisis y posteriormente examina la traza de ejecución resultante, buscando ramas no exploradas. Una vez encuentra una rama no explorada, se utiliza la ejecución simbólica y un sistema resolutor de restricciones para poder generar valores concretos y explorar la rama encontrada. El proceso se repetirá hasta que se consiga el recubrimiento buscado. PEX se guía por criterios de recubrimiento y de condiciones. Este sistema puede además verificar aserciones en el código que desempeñar la labor de oráculo para encontrar errores de manera directa.

Al combinar la ejecución simbólica con la concreta se consigue en muchos casos incrementar la escalabilidad y lidiar con situaciones en las que la ejecución no es dependiente sólo del código sino de otros factores externos. Entendemos por factores externos la utilización de librerías nativas, llamadas al sistema operativo o interacciones con el usuario. Al encontrarse frente a alguna de estas situaciones, la ejecución simbólica se encuentra con muchas limitaciones y de forma habitual es inaplicable.

PEX se puede integrar como Add-on en el entorno de desarrollo *Visual Studio*, lo que permite un manejo mucho más sencillo. [5]

1.4.2. PET

Es necesario destacar PET como una herramienta ya existente de gran importancia para el desarrollo de este proyecto, ya que forma el núcleo funcional a partir del cual se van a fundamentar todos los elementos de la aplicación.

Es una herramienta encargada de la generación de casos de prueba. Recibe como entrada un código de bytes procedente de Java y un conjunto de criterios de recubrimiento para obtener como salida los casos de prueba que sirven para garantizar dicho recubrimiento.

Ya que el código de bytes no es más que una representación a muy bajo nivel del programa, toda la información inferida por PET

es muy difícil de interpretar a nivel de usuario, causa que imposibilita su utilización durante el proceso de desarrollo software. [6]

1.4.3. JUnit

JUnit consiste en un conjunto de bibliotecas, desarrolladas por Erich Gamma y Kent Beck, utilizadas en programación java para hacer pruebas unitarias.

Merece una atención especial, no sólo debido a las ventajas que ofrece su uso, sino también a que es el principal pilar sobre el que se apoya este proyecto. Con esta herramienta se puede realizar una ejecución de los métodos de la clase java que se desea probar para comprobar que todos funcionan tal y como se espera de ellas. La abstracción es un punto clave a la hora de realizar tests de esta manera, puesto que JUnit es independiente de la forma en la que esté implementado el método, tan sólo hará las comprobaciones pertinentes comparando si ante una entrada de argumentos, la salida obtenida es la que se esperaba.

También sirve como herramienta para realizar pruebas de regresión, pues permite comprobar si las modificaciones de un programa continúan dando el mismo resultado que antes de la modificación.

JUnit ofrece además diferentes formas de visualizar los resultados de los tests, en función de lo que resulte más interesante para el programador en cada momento. Así, se dispone de los resultados en modo texto, una visualización a través de una interfaz gráfica (AWT o Swing) o como tarea en Ant.

El uso de JUnit está tan extendido y es tan utilizado que muchas de las herramientas de desarrollo más importantes, como NetBeans o Eclipse cuentan con plugins encargados de la generación de las plantillas necesarias para la creación de pruebas de manera automática. Queda en manos del programador el implementar el test que se desea realizar según sus especificaciones. [7]

1.4.4. JUnit Factory

JUnit Factory es un plugin de Eclipse que ofrece un servicio experimental de generación automática de test unitarios en JUnit, desarrollado por Agitar Software.

En programación, un aspecto importante radica en encontrar la forma de modificar código de manera sencilla y segura. Como hemos explicado antes, una gran herramienta para esto es JUnit, sin embargo la tarea de generar cada uno de los test recae en manos del programador.

JUnit Factory encuentra ciertas limitaciones a la hora de generar test unitarios de manera automática debido a que el código no suele estar escrito con la intención de realizar test en mente. En ocasiones no es posible realizar un recubrimiento completo del código debido a la necesidad de los objetos creados de adquirir un determinado estado o de interactuar con un recurso externo como una base de datos.

JUnit Factory genera casos simulados para las clases problemáticas. Cuando la simulación falla, el desarrollador puede mejorar la cobertura de forma manual, en función del comportamiento del método para el que se desea realizar el test.[7]

1.4.5. Automatic JUnit Test Creator

Es otra herramienta que facilita la labor de generar JUnits de manera automática. Es capaz de analizar una clase Java y buscar posibles casos de prueba.

El objetivo del proyecto es examinar el estado actual de la creación de tests unitarios para el lenguaje de programación Java. En particular, analizar ineficiencias a la hora de derivar test JUnit. Esta herramienta permite un proceso de alto nivel para la generación de dichos test.

Además, Automatic JUnit Test Creator ofrece una interfaz gráfica de usuario para facilitar el manejo, así como un asistente para la creación de tests. [8]

1.4.6. KeY

KeY, es un demostrador automático de teoremas para programas Java, basado en la lógica dinámica JavaCard (tecnología que permite ejecutar de forma segura pequeñas aplicaciones Java -applets- en tarjetas inteligentes y similares dispositivos empotrados.). Para la demostración de teoremas, KeY utiliza un cálculo cuyas reglas realizan ejecución simbólica del programa. De manera colateral, la ejecución simbólica se puede utilizar para generar casos de prueba. En este sentido, los casos de prueba que genera Key están basados en verificación (*verification-based test generation*). Combina ejecución simbólica con verificación basada en modelos con precondiciones y postcondiciones.

Esta herramienta es capaz de generar *unit tests* para programas Java y permite visualizar los caminos generados por la ejecución simbólica. [9]

1.5. Objetivos

El objetivo de este proyecto es la ampliación de la herramienta jPet, añadiéndole la funcionalidad de crear casos de prueba de unidad java de forma automática, tanto concretos como paramétricos, a partir de los casos de prueba generados por el motor de jPet.

Se propondrán las técnicas necesarias para producir de forma automática casos de prueba de unidad a partir de los casos de prueba obtenidos. Para ello, diseñaremos un esquema de generación de código de tests de unidad a partir de los casos de prueba concretos obtenidos. Es importante observar que en el contexto de lenguajes con memoria dinámica y orientación a objetos, los datos de entrada podrían ser referencias y por tanto, la entrada debe considerar los conjuntos de objetos alcanzables desde dichas referencias. Asimismo, propondremos una metodología de generación de oráculos que permitan comprobar los tests de unidad generados.

Recientemente se ha propuesto una nueva metodología de testing de unidad, ”Parameterized unit-testing'', en la cual primeramente se escriben tests de unidad paramétricos, y después se producen tests concretos por medio de llamadas con argumentos concretos a los tests paramétricos. Un test de unidad paramétrico describe una clase de equivalencia de entradas que llevarían al programa al mismo camino de ejecución, lo que se ajusta claramente con el concepto de los casos de prueba abstractos obtenidos por nuestro generador de casos de prueba. Estudiaremos las extensiones que serían necesarias en nuestro esquema de generación de código para producir tests de unidad paramétricos, así como oráculos paramétricos.

1.6. Organización de la memoria

En este apartado vamos a hacer un resumen de los temas que aparecerán a continuación.

El capítulo dos lo dedicaremos al plug-in de eclipse. En él hablaremos de las funcionalidades que incorpora y lo ilustraremos con un ejemplo.

En los capítulos tres y cuatro entraremos en detalle sobre la creación de distintos junits, cómo están implementados y los problemas encontrados durante. También mostraremos unos ejemplos para facilitar su comprensión.

En el capítulo cinco facilitaremos un manual de usuario en el que se detallan los pasos a seguir para ejecutar el plug-in.

Por último, el capítulo seis analiza el trabajo realizado, conclusiones y mejoras de la aplicación.

2. El plug-in de Eclipse para jPET 1.0

En este capítulo vamos a mostrar una visión global del plugin jPET. Comentaremos los aspectos que ya existían en la versión 1.0 de jPET.

2.1. Eclipse

Uno de los IDE más utilizados y que más herramientas aporta al programador para el desarrollo de aplicaciones en distintos tipos de lenguaje es Eclipse. Se construye mediante un núcleo preparado para trabajar conjuntamente con diferentes plugins, es decir, ofrece soporte para recibir módulos que implementan funcionalidades añadidas a la aplicación.

Gracias a su arquitectura, se ofrece la posibilidad de desarrollar plugins que sólo dependan del núcleo de Eclipse o a su vez, plugins que dependan de otros plugins. De esta forma se obtiene un modelo de capas conseguido gracias a los llamados *puntos de extensión*, que permiten la interconexión de los plugins unos con otros para, entre otras cosas, ofrecer añadidos a la interfaz, procesar resultados obtenidos por el plugin anfitrión, etc.

Cuando se desarrolla un plugin para Eclipse es de gran importancia definir correctamente los puntos de extensión que queramos para permitir futuras ampliaciones. Además también resulta apropiado determinar cuáles son los puntos de extensión de los módulos que vamos a utilizar para tener constancia de las posibilidades y limitaciones de cada uno de ellos.

2.2. Ejecución de jPET

En este apartado vamos a hablar de forma global cómo se lleva a cabo la ejecución de jPET, puesto que ya hablaremos en detalle en el manual de uso. Volvemos a recordar que esto hace mención sólo a las funcionalidades incluidas en la versión 1.0.

Para la ejecución de jPET sólo es necesario seleccionar el método, o los métodos, a testear y pulsar en el icono jPET (cuadro verde de la imagen). Una ventana emergente nos mostrará un menú de opciones para configurar las características de la evaluación. Esta ventana emergente es la llamada ventana "preferencias", que explicaremos a continuación.[10]

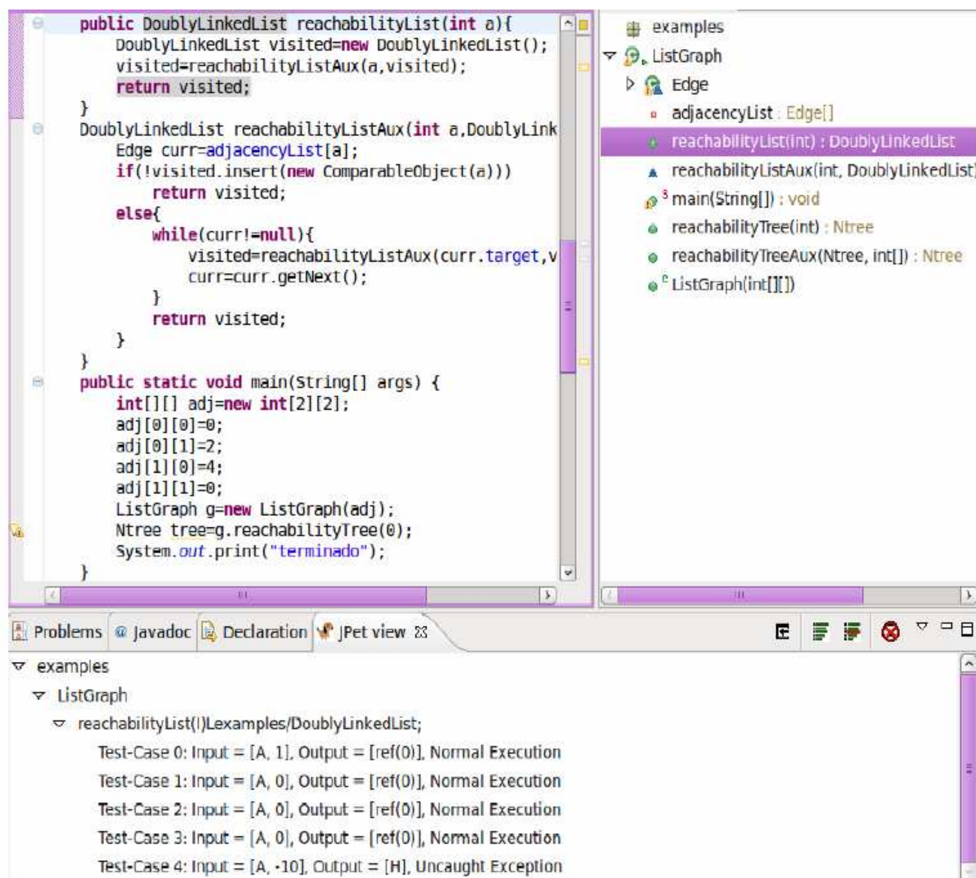


Fig2.1. Vista jPET y selección de métodos en la vista Outline

2.2.1. Enlazando con PET

Como ya se ha mencionado previamente, jPET es un plugin diseñado específicamente sobre PET para la generación de casos de prueba sobre código de bytes, para lo cual es necesario recoger todas y cada una de las opciones elegidas por el usuario y procesarlas para el funcionamiento deseado.

PET funciona en modo consola, traduciendo las opciones del usuario a una línea de texto en la que se recogen todas las preferencias para configurar la llamada. En muchas ocasiones nos encontramos con que para realizar el proceso de testing de una aplicación es necesario analizar una enorme cantidad de clases y métodos, por eso es importante que durante este proceso el programador sea capaz de continuar trabajando sin interrupciones.

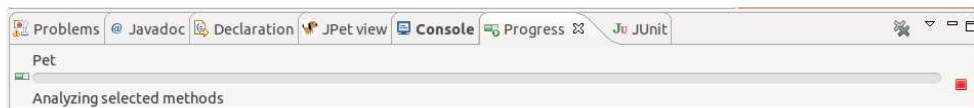


Fig.2.2. Enlazando jPET

2.2.2. Fichero XML

Al realizar una ejecución de PET, los resultados que recibimos en el modo con sola mantienen un formato propio y muy poco manejable. Debido a esto, el diseño de jPET se apoya en el uso de una herramienta capaz de generar un fichero XML con toda la información necesaria para representar la información. Este fichero se pensó para facilitar la comunicación con la interfaz gráfica diseñada en la etapa anterior de desarrollo, sin embargo también ha resultado de suma utilidad a la hora de generar los junits, que es la tarea que nos ha involucrado. No sólo hemos utilizado los campos que existían previamente, sino que se han ido agregando otros nuevos conforme iba surgiendo la necesidad, ampliando de esta manera de forma sencilla la comunicación y capacidad de información que se intercambia entre PET y la interfaz.

Como se puede comprobar, este fichero XML es una pieza clave en el desarrollo de jPET, puesto que se usa en el visor de casos de prueba, en los métodos de visualización de trazas y en la generación automática de junits. Ha sido necesario modificar en cierta medida el código de PET para introducir los cambios a la hora de añadir nuevos cambios en la información representada en el fichero XML y de esta forma aumentar su expresividad.

jPET crea, dentro de la carpeta indicada por el usuario el las preferencias, todas las carpetas que serán necesarias para el almacenamiento y organización de los ficheros XML, ordenados y diferenciados por clases y paquetes.

De manera abreviada, podemos decir que cada fichero XML contiene la información de todos los casos de prueba generados para un método.

Una vez que se ha generado el fichero XML es necesario extraer toda la información para poder tener acceso a ella y utilizarla en jPET. Para esto, se ha utilizado el API proporcionado por Java para este propósito, que viene incluida en el paquete *org.w3c.dom*, con las clases para realizar el parser y almacenar toda la información que recibimos contenida en el fichero XML.

2.2.3. Plugin para Eclipse

jPET como plugin para Eclipse incluye dos funcionalidades destinadas a facilitar la comprensión por parte del usuario de toda la información recibida por PET, así como visualizar de una forma mucho más intuitiva y natural tanto los casos de prueba como sus distintos flujos de ejecución.[11]

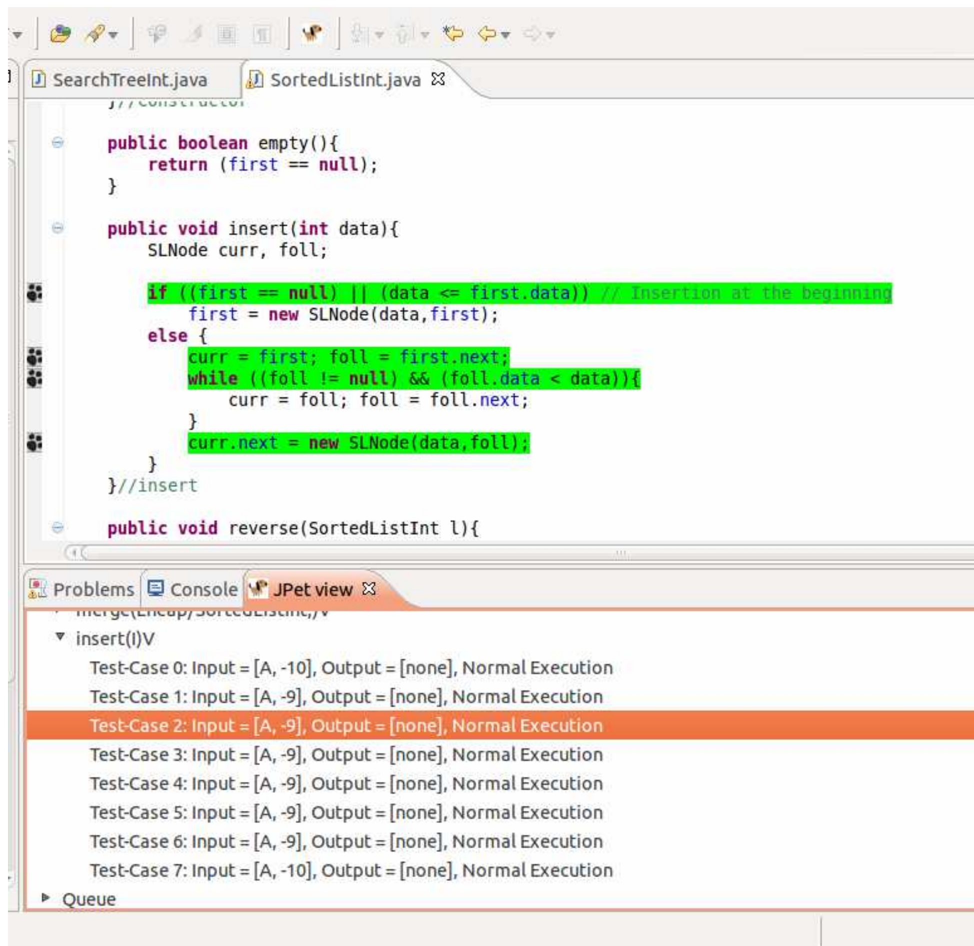


Figura 2.2.3.1. Traza de la ejecución de un caso de prueba

Como se puede apreciar en la figura, el plugin jPET permite al usuario ver claramente la rama de código que se está ejecutando en el caso de prueba seleccionado. Este camino en el código aparece remarcado en verde, facilitando de esta manera su identificación.

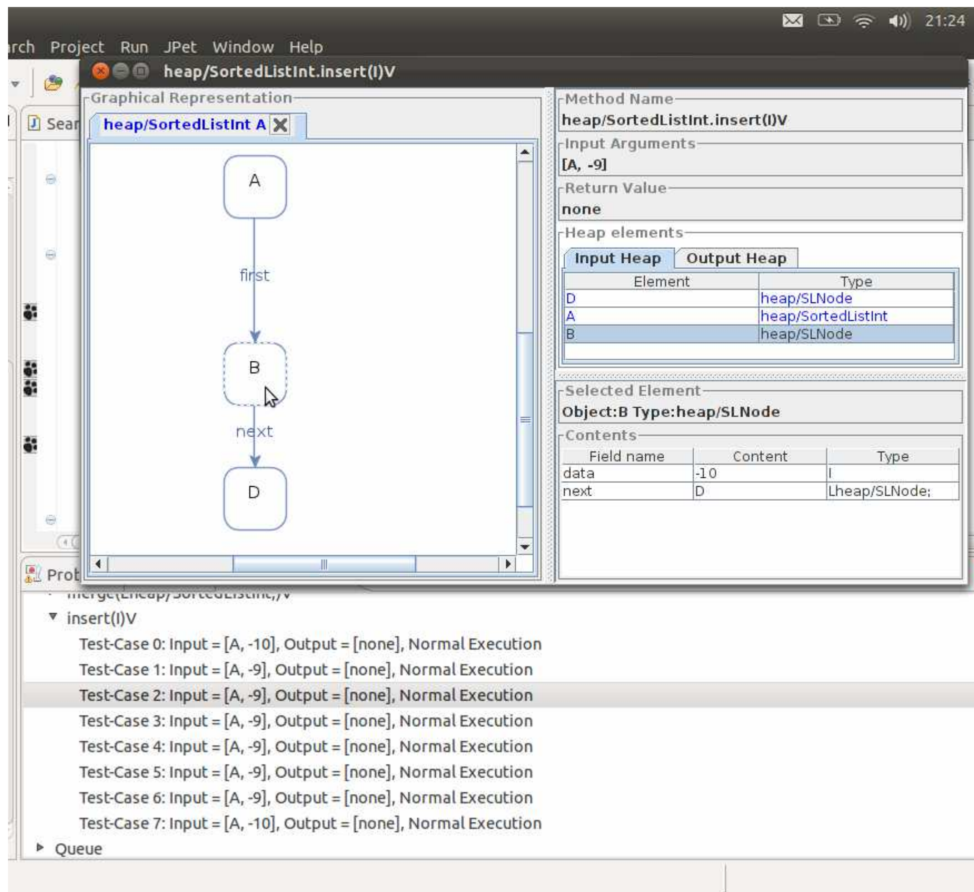


Figura 2.2.3.2. Visor de un caso de prueba

También es posible obtener información gráfica detallada de un caso de prueba. El plugin facilita información de los elementos contenidos tanto en el heap in como en el heap out. El usuario tiene total libertad de seleccionar el objeto que desee inspeccionar y desglosar su contenido para visualizarlo en forma de una lista enlazada.

2.3. Preferencias

PET utiliza una gran cantidad de parámetros configurables que hacen posible la generación de casos de prueba adecuados a las necesidades del usuario. Además, jPET tiene incorporado todas estas opciones además de añadir algunas nuevas para de esta forma ofrecer al usuario una experiencia más completa.

Las principales opciones incluidas en PET son:

- **Coverage Criterion**: Permite la selección del criterio de recubrimiento. Actualmente hay dos criterios diferentes implementados: el Block-K y el Depth-k. Ambos necesitan de un parámetro adicional que indica el número de veces que cada bloque será explorado (Block-k) o la profundidad del árbol(Depth-k).

- **Numeric test-cases or path-constraints**: Permite decidir si se desean obtener casos de prueba especificados como conjuntos de restricciones o como valores numéricos concretos. Si se selecciona la opción de valores numéricos será necesario establecer un rango para definir los posibles valores.

- **References Aliasing**: Señala si se permite la posibilidad de que existan alias, esto es, referencias diferentes que apunten al mismo objeto. Por lo general, al activar esta opción se generan una gran cantidad de casos de prueba adicionales, que en ocasiones puede resultar interesante.

- **Labeling Strategy**: COn este parámetro se permite transformar los *path-contraints* en valores concretos que satisfagan las restricciones haciendo uso de la función labeling existente el *CLP-FD*, *Constraint Logic Programming over Finite Domains,* una librería existente en Prolog que permite la descripción de condiciones que debe cumplir una solución.

- **Verbosity:** Indica el nivel de detalle de la información proporcionada.

- **Tracing**: Con esta opción se puede obtener la traza de ejecución de cada uno de los casos de prueba generados.

- **Generate Junit test**: Cuando se activa, se generan una serie de junits en base a los casos de prueba obtenidos por PET. Se puede seleccionar que los casos de prueba sean numéricos (opción numeric) o paramétricos (opción parametric).

Además, jPET también incluye las siguientes preferencias:

- **Junit Path**: Ofrece la posibilidad de especificar el directorio en el que queremos almacenar los ficheros Junits generados.

- **Show JUnit Test**: Al activar esta opción, cuando se finaliza el análisis de abrirá de manera automática en el editor el fichero con las pruebas de Junit.

- **XML Filespath**: Permite elegir la ruta en la que se van a crear los ficheros XML generados por la herramienta.

Hay que destacar que a pesar de que no sean visibles de forma directa, las siguientes preferencias también se han añadido en jPET para integrar ampliaciones o para facilitar el manejo al usuario:

- **Classpath**: Sirve para especificar un directorio diferente al que hay por defecto, donde se encuentra la clase a analizar. DEsde el punto de vista del usuario esta opción de ofrece la posibilidad de trabajar desde cualquier directorio.

- **Precond**: Envía las precondiciones a PET para aplicarlas a la hora de realizar el análisis.

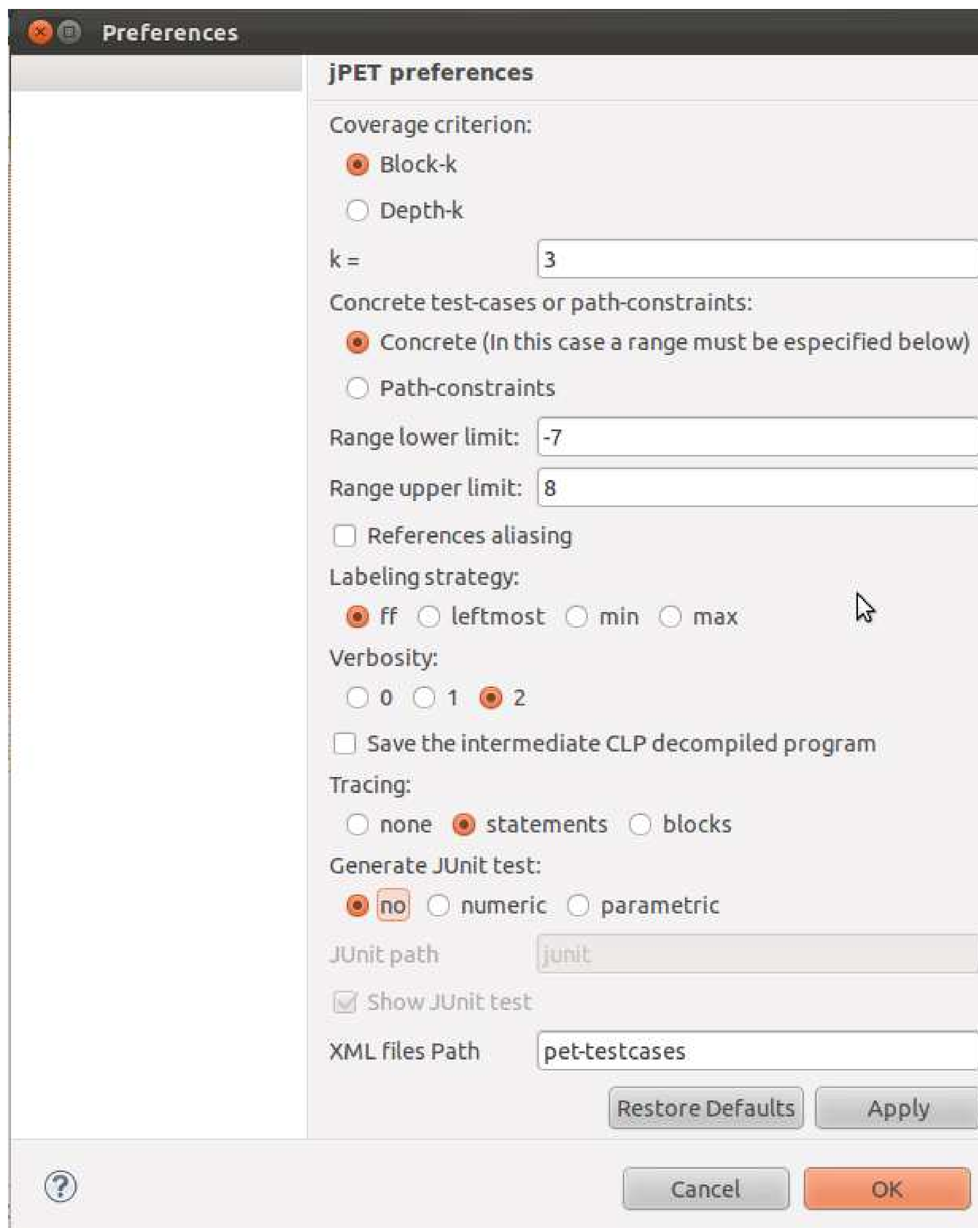


Figura 2.3. Ventana de preferencias

2.4. Ejemplo ilustrativo

Para mostrar la funcionalidad, vamos a utilizar un algoritmo sencillo, *reachabilityList* de la clase *ListGraph*, que implementa grafos dirigidos usando listas de adyacencia.

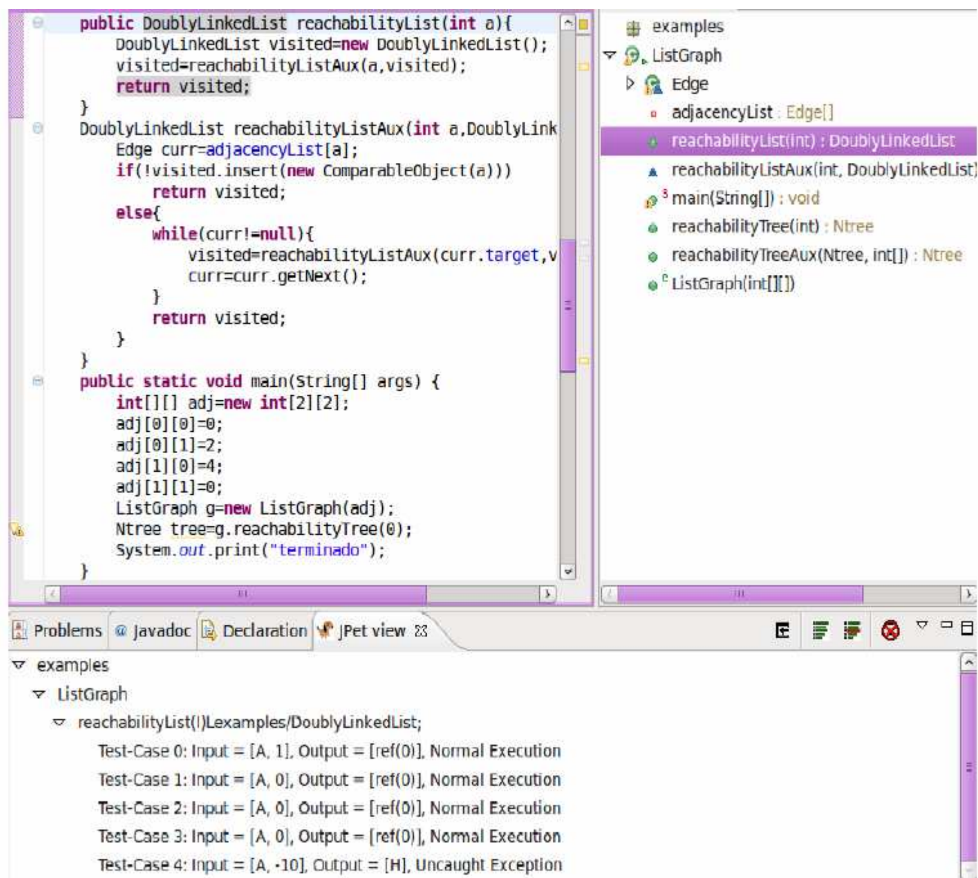
Para la ejecución de este método, se hace uso de la clase DoublyLinkedList para crear una lista doblemente enlazada; y un método auxiliar insert que indica dónde debe añadirse un elemento.

```
public DoublyLinkedList reachabilityList(int a){
      DoublyLinkedList visited= new DoublyLinkedList();
      visited=reachabilityListAux(a,visited);
      return visited;
}

DoublyLinkedList reachabilityListAux(int a,
DoublyLinkedList visited) {
      Edge curr=adjacencyList[a];
      if(!visited.insert(new ComparableObject(a)))
            return visited;
      else{
            while(curr!=null){
                  visited=reachabilityListAux(
                        curr.target,visited);
                  curr=curr.getNext();
            }
            return visited;
      }
}
```

Figura 2.6: Algoritmo de ejemplo: ListGraph.reachabilityList

Este método lo hemos ejecutado con un criterio de recubrimiento loop-k =2 y un rango de variables numéricas de -10 a 10. En la siguiente figura puede observarse que se generan 5 casos de prueba. Para ver tanto la traza como el visor de cada caso de prueba, basta con seleccionar uno y pinchar en “show test-case” para el visor y “show trace” para la traza.


public DoublyLinkedList reachabilityList(int a){
DoublyLinkedList visited=new DoublyLinkedList();
visited=reachabilityListAux(a,visited);
return visited;
}
DoublyLinkedList reachabilityListAux(int a,DoublyLink
Edge curr=adjacencyList[a];
if(!visited.insert(new ComparableObject(a)))
return visited;
else{
while(curr!=null){
visited=reachabilityListAux(curr.target,v
curr=curr.getNext();
}
return visited;
}
}
public static void main(String[] args) {
int[][] adj=new int[2][2];
adj[0][0]=0;
adj[0][1]=2;
adj[1][0]=4;
adj[1][1]=0;
ListGraph g=new ListGraph(adj);
Ntree tree=g.reachabilityTree(0);
System.out.print("terminado");
}
examples
ListGraph
Edge
adjacencyList : Edge[]
reachabilityList(int) : DoublyLinkedList
reachabilityListAux(int, DoublyLinkedList)
main(String[]) : void
reachabilityTree(int) : Ntree
reachabilityTreeAux(Ntree, int[]) : Ntree
ListGraph(int[][])
Problems
Javadoc
Declaration
JPet view
examples
ListGraph
reachabilityList(I)Lexamples/DoublyLinkedList;
Test-Case 0: Input = [A, 1], Output = [ref(0)], Normal Execution
Test-Case 1: Input = [A, 0], Output = [ref(0)], Normal Execution
Test-Case 2: Input = [A, 0], Output = [ref(0)], Normal Execution
Test-Case 3: Input = [A, 0], Output = [ref(0)], Normal Execution
Test-Case 4: Input = [A, -10], Output = [H], Uncaught Exception

3. Creación de junit

En este capítulo hablaremos de la generación de junit concretos, cómo lo hemos implementado y los problemas que hemos encontrado.

3.1. Descripción y funcionamiento

Esta funcionalidad se encarga de la generación de casos de prueba *junit* a partir de los casos de prueba proporcionados por jPET.

Como ya mencionamos anteriormente, un unit-test en programación, es realizar una prueba para verificar el funcionamiento de un módulo de código y con ello asegurar el correcto funcionamiento de cada módulo por separado. Un unit-test consta de tres partes: la construcción de los datos de entrada, la llamada al módulo y la comprobación del resultado.

Los unit-tests generados son plantillas genéricas que agilizan al usuario la tarea de testing. Las plantillas son creadas a partir de las estructuras de datos obtenidas de leer el fichero xml.

El fichero xml ha sido construido por el motor de los casos de prueba y contiene la información necesaria.

Las plantillas pueden ser de dos tipos: numéricas y paramétricas. En el primer caso, se trabaja con valores numéricos concretos dentro de un rango definido, mientras que en el segundo caso se generan dos ficheros, uno de ellos con valores variables y otro con valores concretos que satisfacen los casos del primer fichero.

Un ejemplo de lo que pretendemos obtener podría ser el siguiente:

Elegimos “abs” como método a testear que devuelve el valor absoluto de un número.

```
public static int abs(int x){
      if (x >= 0) return x;
      else return -x;
}
```

La clase junit que escribiría un desarrollador para probar el método sería algo como esto:

```
Public class Abs_I_I_Test extends TestCase {

      public Abs_I_I_Test() {
            super();
      }

      protected void setUp() {}

      protected void tearDown() {}


      /**
       * Test Number 1
       * Result: ok
       */
       public void test_1(){

            //initialize inputs of method
            int input0 = -10;

            //generate output
            int output = Arithmetic.abs(input0);

            //initialize output
            //show object after run the method

            int outputExpected = 10;

            //compare results
            assertEquals("The result expected",
                        outputExpected, output);
      }

      /**
       * Test Number 2
       * Result: ok
       */
       public void test_2(){

            //initialize inputs of method
            int input0 = 0;
            //generate output
            int output = Arithmetic.abs(input0);

            //initialize output
            //show object after run the method

            int outputExpected = 0;

            //compare results
            assertEquals("The result expected",
                        outputExpected, output);
      }

}
```

3.2. Detalles de implementación

3.2.1. Formato del fichero junit

A la hora de escribir el fichero junit, seguiremos una estructura que es muy comúnmente utilizada para realizar tests unitarios.

- imports:

```
package junit.heap;

//import of TestCase
import junit.framework.TestCase;

//import of Reflection
import java.lang.reflect.*;

//imports of objects
import heap.SortedListInt;
import heap.SLNode;
import java.lang.NullPointerException;
```

- constructora: Nos sirve para preparar todas las variables y objetos que vamos a utilizar para la ejecución de los tests, así como para realizar las inicializaciones básicas.

```
public Clase_metodo_Test() {
      super();
}
```

- setUp: Es un método que nos va a servir para llevar a cabo las inicializaciones pertinentes y comunes de todos los test.

```
protected void setUp() {
      _objClass = new SortedListInt();
}
```

- tearDown: Es un método de limpieza que se invoca después de realizar la llamada de cada test. Su tarea será la de eliminar los objetos y liberar los recursos que ya no van a ser necesarios.

```
protected void tearDown() {
      _objClass = null;
}
```

- test_X: Son los test propiamente dichos, donde se comparará la salida esperada con la salida obtenida para realizar un assert. En el fichero junit generado tendremos un test por cada uno de los casos de prueba obtenidos por PET.

```
/**
 * Test Number 1
 * Result: ok
 * @throws Exception
 */
public void test_1(){

      //initialize object fields

      //initialize inputs of method

      //generate output

      //initialize output

      //show object after run the method

      //compare results

}
```

- Método myEquals: (ver apéndice)
- Método setPrivateField:

```
/**
* This method set value in a selected field of object
                                                   obj
* @param obj: class of object
* @param fieldName: name of field of obj
* @param value
*/

private void setPrivateField(Object obj, String
                        fieldName, Object value){
      Field field;
      try{
            field =
      obj.getClass().getDeclaredField(fieldName);
            field.setAccessible(true);
            field.set(obj,value);
      }

      catch (Exception e) {
            //TODO Auto-generated catch block
            e.printStackTrace();
      }
}
```

Por la naturaleza de este proyecto y al apoyarse en un trabajo anterior, hemos reutilizado muchas de las herramientas y metodologías que nos proporcionaba ya la aplicación.

Fue necesario recurrir a las siguientes bibliotecas:

- *BCEL*: Esta biblioteca en sí no sólo la empleamos en la aplicación, si no que hacemos uso de ella también en el código que generamos para acceder a elementos privados de las clases. Ya dentro de nuestra implementación la utilizamos para acceder a la información protegida del método o los métodos seleccionados.[12]

- *JUNIT*: Como es obvio, es necesario el uso de esta biblioteca, puesto que es el recurso a generar. Por motivos de diseño, hemos decidido usar la versión 4.

- *XML Standard API*: Esta biblioteca la empleamos para parsear el xml. Aunque esta funcionalidad ya estaba anteriormente en jPET, nos hemos visto obligados a modificarlo añadiendo más información en él.

3.2.3. Estructura general

Como hemos dicho en apartados anteriores, se trata de una ampliación del proyecto de jPET, por lo que nos hemos valido en lo mayor posible de la estructura que ya había. A continuación nombraremos los pasos que ejecuta la aplicación desde que ejecuta jPET hasta que ha sido generado el fichero junit:

1. Se analiza el método y mediante el motor de jPET se generan todos los casos de prueba posibles.

2. Esos casos de prueba se guardan en un xml con el fin de tener la información necesaria recopilada.

3. Se extrae la información del xml mediante un parseador y se pasa a una estructura de datos.

4. Se escribe el fichero junit en la carpeta “test-cases” a partir de la estructura de datos.

Ahora describiremos el orden en el que rellenamos el fichero junit:

Escribimos los “imports” con las clases que va a necesitar, como las relacionadas con junit y reflection.

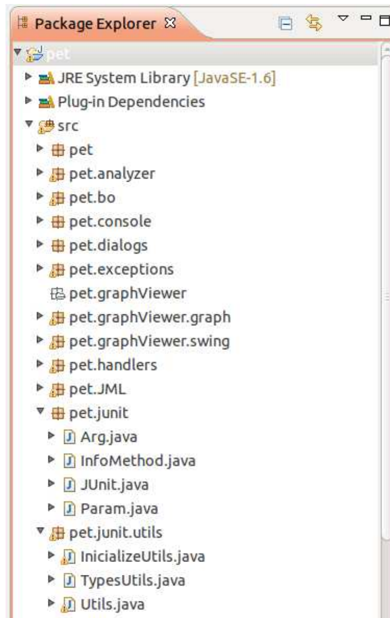
A continuación, añadimos la cabecera de la clase, así como la constructora y los métodos propios de junit: setUp() y tearDown().

Escribimos un método auxiliar llamado “setPrivateField(…) cuya función es modificar el valor de un atributo de una clase, ya sea público o privado. Esto es necesario porque no conocemos las clases que se van a tratar a priori y cuantas menos restricciones se pongan, más facilitamos el trabajo a los desarrolladores.

Añadimos un método test por cada caso de prueba que se haya generado en el motor. Si estamos en el caso de junit paramétrico, será necesario incluir al comienzo de cada test una comprobación de todas las precondiciones, y al final una comprobación de todas las postcondiciones.

Por último, insertamos un método “myEquals” cuyo funcionamiento es el mismo que el del método “equals”, con la particularidad de que es un método genérico que en lugar de basarse en las referencias de memoria, compara cada uno de los elementos de las clases a comparar. Este método fue introducido al archivo para no obligar a los desarrolladores a tener que implementar el método “equals” en cada una de las clases que creaban.

Una vez explicados los pasos que realiza jPET, vamos a mostrar en detalle la estructura. Esta funcionalidad se encuentra dentro de jPET en un paquete llamado “junit”:



Estructura del paquete junit:

- *Clase Arg*: Estructura de datos que asocia el nombre de una variable del xml con su identificador en el junit.

    - Atributos:
        - Identifier: identificador dado
        - Name: nombre de la variable en el xml

    - Métodos: accesores y modificadores de los atributos identifier y name.

- *Clase InfoMethod*: Esta clase almacena información relevante del método seleccionado:

    - Atributos:
        - *argsMethod*: argumentos de entrada del método
        - *isStatic*: informa si el método seleccionado es estático o no.
        - *methodName*: método seleccionado
        - *myClass*: clase del método seleccionado.
        - *myMethod*: nombre del método seleccionado
        - *nameTest*: nombre que va a llevar el archivo
        - *petMethod*
        - *test*

        - *typesMethod*: tipos de los argumentos de entrada del método

    - Métodos: accesores y modificadores de los atributos anteriormente nombrados.

- *Clase Junit*: Clase principal para le generación de junit. Es la encargada de crear el fichero y rellenarlo con la información adecuada.

    - Atributos:
        - *info*: Objeto de tipo InfoMethod
        - *flag*: Entero que indica el modo de creación de junit. Si flag = 0 entonces creará un junit de tipo numérico, pero si el flag vale 2 creará un junit de tipo paramétrico.
    - Métodos:
        - *Constructora*: inicializa info.
        - *checkConstraints*: Si flag = 2, añade al fichero la comprobación de las precondiciones.
        - *classInit*: Añade al fichero la cabecera de la clase a crear y los métodos setUp() y tearDown().
        - *classInitConst*: En el caso de que flag = 2, inicializa la clase concreta creada.
        - *generateFile*: método principal. Se encarga de crear el fichero junit, llamando al resto de métodos en un orden determinado.
        - *getParameters*:
        - *getSolutionsParameters*:
        - *importWrite*: Escribe en el fichero los imports necesarios.
        - *outputGenerate*: escribe la llamada al método
        - *setField*: añade al fichero el método auxiliar setPrivateField.
        - *testWrite*: Añade un caso de prueba en el fichero en forma de test.
        - *writeMethodEquals*: escribe el método genérico myEquals.

- *Clase Param*: Asocia el nombre de un parámetro con su valor concreto.

    - Atributos:
        - *name*: nombre del parámetro
        - *value*: valor del parámetro

    - Métodos: accesores y modificadores de *name* y *value.*

- *Paquete utils*: Contiene clases cuya función es clarificar la programación de la clase Junit.

    - *Clase InicializeUtils*: contiene métodos que recorren el xml y escriben en el junit cada objeto reflejado en él.
    - *Clase TypesUtils*: Utilidades sobre tipos de objetos.
    - *Clase Utils*: otros métodos auxiliares.

3.3. Incidencias en el desarrollo

Aunque muchas incidencias de las que nombraremos a continuación son comunes para los dos tipos de junits, las incluiremos aquí puesto que primero implementamos este caso y fue donde encontramos todos esos problemas.

El primer inconveniente que encontramos era que, al ser un proyecto ya desarrollado, debíamos intentar hacer uso del mayor número de recursos que poseía y, aunque eso nos ha facilitado el trabajo, a priori nos encontramos con dificultades hasta que conseguimos encontrar y aprender a usar dichos recursos.

Cuando empezamos a crear nuestros primeros ficheros encontramos el problema de que no podíamos escribir el fichero en otra ruta distinta a la ruta donde se encontraba el fichero xml. Cuando elegíamos rutas distintas, la primera vez nos creaba bien tanto el fichero xml como el fichero junit, pero una vez que escribía el junit, ya no creaba ningún fichero más. A día de hoy, aún no hemos sido capaces de solventar este problema.

El problema de crear casos de prueba de unidad automatizados, es que es necesario saber a priori cómo es el método a testear (estático o no estático), las entradas necesarias para ejecutarlo con sus tipos, así como el tipo que devuelve. También es necesario saber si los atributos son visibles o no y si hay métodos accesores y modificadores para dichos atributos. Para solucionar este problema, hemos hecho uso de la librería BCEL, pues esta librería se encarga de procesar y extraer información de archivos de bytecode (.class).

Uno de los problemas que solucionamos gracias a esta la librería BCEL es el caso en el que leíamos del xml un objeto cuyo valor era irrelevante. A priori no parecía complicado, puesto que se diferenciaba de una variable de tipo entero por el tipo, pero en el xml no siempre teníamos el tipo definido, por lo que sin la librería nos era imposible solucionarlo.

Cuando un desarrollador define un objeto, los atributos los suele hacer privados. Con el fin de no restringir mucho al programador y, por tanto, no obligarle a hacer los atributos públicos o a definir los métodos modificadores, decidimos también añadir un método al junit que utilizara la librería BCEL, más concretamente la clase *reflection*, cuya función es asignar un valor dado a un atributo de un objeto, ya sea privado o público.

Una vez terminado un junit, era necesario ejecutarlo para ver si se había construido correctamente. Al usar el método "equals" veíamos que los tests fallaban aun sabiendo que debían funcionar. Esto ocurría porque el método "equals" usado en objetos compara sólo las referencias a memoria, cuando lo que nos interesa es que compare cada uno de los componentes del objeto. Cuando un desarrollador crea clases, normalmente no implementa este método, y si hiciera él su propio junit test, en algunos casos sería más rápido hacer cada comprobación que implementarlo. Con el fin de facilitar el trabajo a desarrolladores, se optó por escribir un método "equals" genérico en el junit creado que hiciera esas comprobaciones.

3.4. Ejemplo de junit

Para mostrar un ejemplo bastante completo, hemos decidido utilizar como ejemplo el método merge, cuya funcionalidad es mezclar una lista enlazada ordenada con otra lista enlazada ordenada:

```
public void merge(SortedListInt l){
      SLNode p1, p2, prev;

      p1 = first; p2 = l.first;
      if(p1.data <= p2.data)
            p1 = p1.next;
      else { first = p2; p2 = p2.next;}

      prev = first;
      while((p1 != null) && (p2 != null)){
            if (p1.data <= p2.data){
                  prev.next = p1;
                  p1 = p1.next;
            }
            else {
                  prev.next = p2;
                  p2 = p2.next;
            }
            prev = prev.next;
      }

      if (p1 == null)
            prev.next = p2;
      else prev.next = p1;
}
```

Al ejecutar jPet con un rango de [-10, 10] y una profundidad en el árbol de ejecución k = 2, obtenemos un junit test con 9 casos distintos. Como mostrar todo el código nos ocuparía alrededor de 10 páginas, hemos preferido enseñar sólo dos casos:

```
/**
 * Test Number 1
 * Result: ok
 */
public void test_1(){
      //initialize object fields
      SLNode H = new SLNode();
      SLNode C = new SLNode();
      SLNode E = new SLNode();
      setPrivateField(E,"data",-10);
      setPrivateField(E,"next",H);
      setPrivateField(_objClass,"first",C);
      setPrivateField(C,"data",-9);
      setPrivateField(H,"data",-10);
      setPrivateField(H,"next",null);

      //initialize inputs of method
      SortedListInt input0 = new SortedListInt();
      setPrivateField(input0,"first",E);

      //generate output
      _objClass.merge(input0);

      //initialize output
      //show object after run the method
      SLNode newH = new SLNode();
      SLNode newC = new SLNode();
      SortedListInt new_input0 = new SortedListInt();
      SortedListInt new_objClass = new
                                  SortedListInt();
      SLNode newE = new SLNode();
      setPrivateField(newE,"data",-10);
      setPrivateField(newE,"next",H);
      setPrivateField(new_objClass,"first",E);
      setPrivateField(new_input0,"first",E);
      setPrivateField(newC,"data",-9);
      setPrivateField(newH,"data",-10);
      setPrivateField(newH,"next",C);

      //compare results
      assertTrue("The object expected",
                 myEquals(_objClass,new_objClass));
```

En este caso, la lista enlazada principal es de la forma [-9] y la lista enlazada que le vamos a mezclar es [-10, -10]. En un principio se crean las dos listas y se añaden tantos elementos como sea necesario para tener las dos listas mencionadas anteriormente. Después de llamar al método deberíamos obtener la lista [-10, -10, -9] resultante de unir las anteriores. Si miramos el código vemos que efectivamente, se consigue el resultado esperado.

```
/**
 * Test Number 7
 * Result: java.lang.NullPointerException
 */
public void test_7(){
      //initialize object fields
      SLNode C = new SLNode();
      setPrivateField(_objClass,"first",C);
      setPrivateField(C,"data",-10);

      //initialize inputs of method
      SortedListInt input0 = new SortedListInt();
      setPrivateField(input0,"first",null);

      //generate output
      try{
            _objClass.merge(input0);
      }
      catch(Exception ex){
            assertEquals("excepcion",
                  "java.lang.NullPointerException",
                  ex.getClass().getName());
            java.lang.NullPointerException
                  exceptionExpected = new
                  java.lang.NullPointerException();

            assertEquals("excepcion",
                  exceptionExpected,ex);
            return;
      }
      fail("Did not find expected exception");
}
```

En este caso una de las listas no existe, por lo que al ir a acceder a esa lista, el test nos da un error.

4. Junit de tipo paramétrico

En este capítulo vamos a hablar de la implementación de la generación de unit-test paramétricos. Como recordatorio, en los unit-test paramétricos agrupamos los distintos valores de entradas que dan lugar al mismo caso de prueba.

La implementación de estos tests es prácticamente la misma que en los unit-tests concretos, por lo que nos vamos a ahorrar explicar de nuevo esa parte y vamos a hablar de las incidencias ocurridas durante el desarrollo.

4.1. Incidencias en el desarrollo

Este tipo lo desarrollamos una vez terminado el junit de tipo numérico, por lo que casi todas las incidencias estaban solucionadas. El único problema que encontramos fue la falta de información en el xml para poder generar estos nuevos ficheros de prueba, tema que trataremos en el siguiente apartado, pues necesitamos extender el propio motor de la aplicación.

4.2. Extensiones

En una primera modificación del motor de jPet, añadimos la opción de un nuevo comando para generar los casos de prueba paramétricos.

Para escribir los ficheros junit, era necesario incluir más información al .xml, como las precondiciones, las postcondiciones, las variables de entrada y los valores concretos que satisfacen cada uno de los casos de prueba del método a testear. Esta información se encuentra en las restricciones CLP que proporciona PET, por lo que tuvimos que volver a modificar el motor para adjuntar dicha información, quedando el fichero xml con la siguiente estructura:

- **Firma**: La firma del método seleccionado para la generación de casos de prueba.

- **Argumentos de entrada**: La ristra de argumentos de entrada para el caso de prueba. Puede contener tanto punteros a objetos del heap como valores concretos.

- **Heap de entrada**: Muestra información de cada uno de los objetos contenidos en el heap de entrada, su tipo, información de sus elementos, indicador de si es array y de sus campos si se trata de un objeto, etc.

- **Heap de salida**: Igual que para el heap de entrada, pero en este caso para los objetos de salida.

- **Retorno**: Contiene el valor de retorno o la referencia al objeto a devolver en el caso de prueba.

- **Flag de excepción**: Flag de excepción que es activado siempre que el caso de prueba genere una excepción no capturada.

- **Traza**: Traza del caso de prueba con la información del nombre de las instrucciones que se ejecutan y el contador de programa para poder identificar cada instrucción de manera unívoca.

- **Constraints de entrada:** Restricciones de los objetos contenidos en el heap de entrada.

- **Constraints de salida:** Igual que para los constraints de entrada pero en este caso de los objetos contenidos en el heap de salida.

- **Parámetros:** Conjunto de pares (nombre, valor) que recibe cada test como entrada.

Al aumentar el xml, también fue necesario modificar su parseador y la estructura en la que se almacena después de tratar el xml.

4.3. Ejemplo de junit paramétricos

Como ejemplo a mostrar, hemos decidido usar el mismo método que en el caso de junit de tipo numérico, pues así pueden apreciarse mejor las diferencias entre uno y otro. Para el primer fichero también vamos a optar por enseñar los tests generados 1 y 7:

```
public void test_Constraints (){
      try{
            _testClass.test_1(-9,-10,-10);
            _testClass.test_2(-9,-10,-9);
            _testClass.test_3(-9,-10);
            _testClass.test_4(-10,-10);
            _testClass.test_5(-10,-10,-9);
            _testClass.test_6(-10,-10,-10);
            _testClass.test_7(-10);
            _testClass.test_8();
            _testClass.test_9();
      }catch(Exception e){
            e.printStackTrace();
      }
}
```

Fichero con los valores concretos

Aquí se muestran las llamadas concretas que haríamos para los casos paramétricos. Hemos señalado los casos 1 y 7 que son los que mostraremos a continuación.

```
/**
 * Test Number 1
 * Result: ok
 * @throws Exception
 */
public void test_1(int G,int I,int L) throws
                                      Exception{
      if(!(L <= G + -1 && I <= G + -1))
            throw new Exception("preconditions
                                            failed");
      else {
            //initialize object fields
            SLNode J = new SLNode();
            SLNode C = new SLNode();
            SLNode E = new SLNode();
            setPrivateField(E,"data",I);
            setPrivateField(E,"next",J);
            setPrivateField(_objClass,"first",C);
            setPrivateField(C,"data",G);
            setPrivateField(J,"data",L);
            setPrivateField(J,"next",null);

            //initialize inputs of method
            SortedListInt input0 = new
                                  SortedListInt();
            setPrivateField(input0,"first",E);

            //generate output
            _objClass.merge(input0);

            //initialize output
            //show object after run the method
            SLNode newJ = new SLNode();
            SLNode newC = new SLNode();
            SortedListInt new_input0 = new
                                  SortedListInt();
            SortedListInt new_objClass = new
                                  SortedListInt();
            SLNode newE = new SLNode();
            setPrivateField(newE,"data",I);
            setPrivateField(newE,"next",J);
            setPrivateField(new_objClass,"first",E);
            setPrivateField(new_input0,"first",E);
            setPrivateField(newC,"data",G);
            setPrivateField(newJ,"data",L);
            setPrivateField(newJ,"next",C);

            //compare postconditions

            //compare results
            assertTrue("The object expected",
                  myEquals(_objClass,new_objClass));
            assertTrue("The input expected",
                  myEquals(input0,new_input0));
      }
}
```

Como podemos observar, primero comprobamos que se cumplen unas ciertas precondiciones, es decir, que los valores de las entradas satisfacen la ejecución de ese caso de prueba. Una vez hecha dicha comprobación, creamos las dos listas enlazadas, [G] y [I, L], siendo I mayor o igual que L. Después de ejecutar el método, obtenemos la lista [I, L, G], siendo G mayor o igual que L.

```
/**
 * Test Number 7
 * Result: java.lang.NullPointerException
 * @throws Exception
 */
public void test_7(int F) throws Exception{
      if(false)
            throw new Exception("preconditions
                                                failed");
      else {
            //initialize object fields
            SLNode C = new SLNode();
            setPrivateField(_objClass,"first",C);
            setPrivateField(C,"data",F);

            //initialize inputs of method
            SortedListInt input0 = new
                                    SortedListInt();
            setPrivateField(input0,"first",null);

            //generate output
            try{
                  _objClass.merge(input0);
            }
            catch(Exception ex){
                  assertEquals("excepcion",
                  "java.lang.NullPointerException",
                  ex.getClass().getName());

                  java.lang.NullPointerException
                  exceptionExpected = new
                  java.lang.NullPointerException();

                  assertEquals("excepcion",
                  exceptionExpected,ex);

                  return;
            }
            fail("Did not find expected exception");
      }
}
```

En este caso una de las listas no existe, por lo que al probarlo, nos dará un error indicando ese problema.

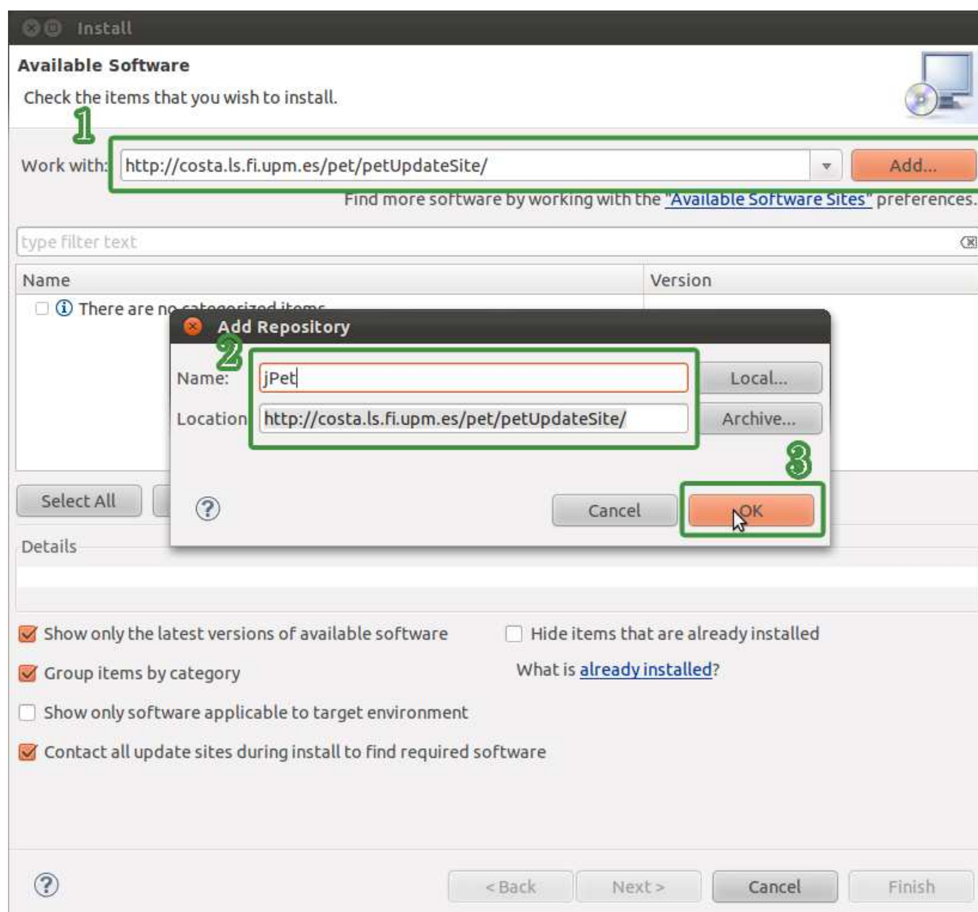
5. Manual de usuario

5.1. Requisitos

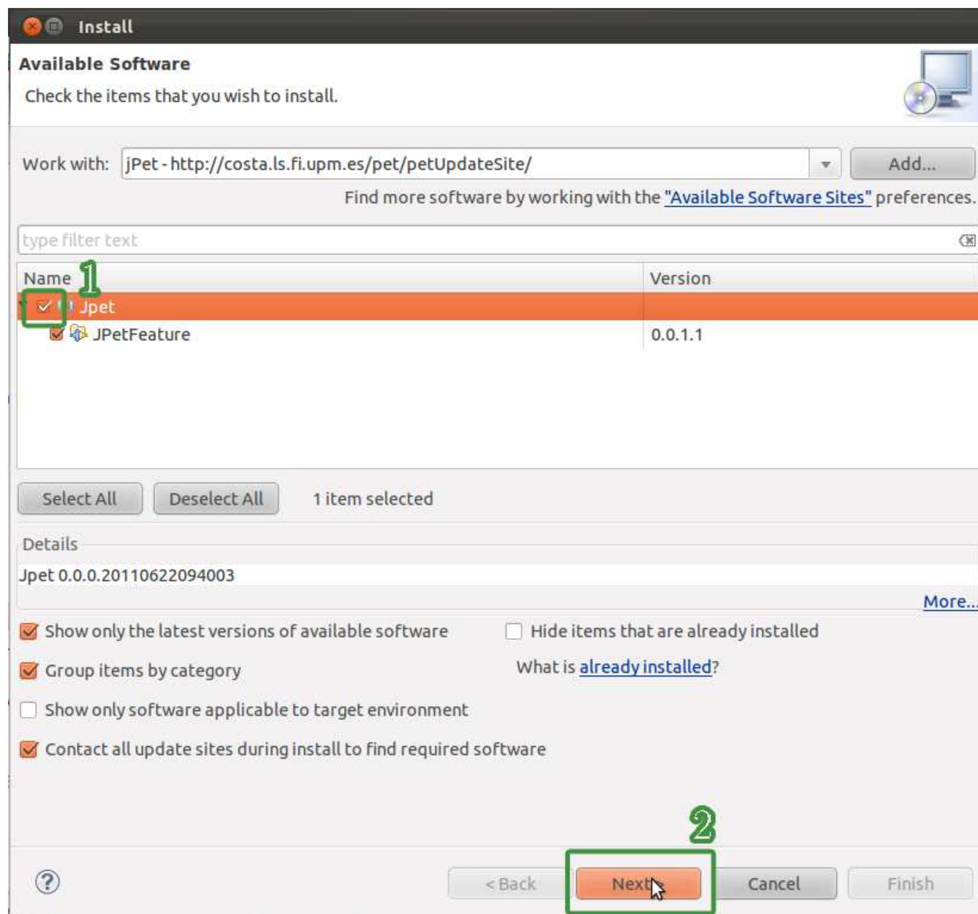
Los requisitos mínimos del sistema son:

- Procesador a 350MHz
- Memoria RAM de 256MB
- Sistema operativo: Linux Ubuntu versión 10.4 o posterior
- Software necesario:
    - Eclipse
    - JDK
    - SWI-Prolog o SICStus Prolog

5.2. Instalación de jPET



Para instalar jPet necesitamos abrir el asistente de instalación de nuevo software de Eclipse, pulsando en el menú 'Help' y después en 'Install new software'. En el campo 'Work with', pinchamos en el botón 'Add' e incluimos la dirección indicada en la figura en el menú de instalación (http://costa.ls.fi.upm.es/pet/petUpdateSite/) en el campo 'Location', rellenando también el campo de 'Name' como en la figura. Click en Ok.

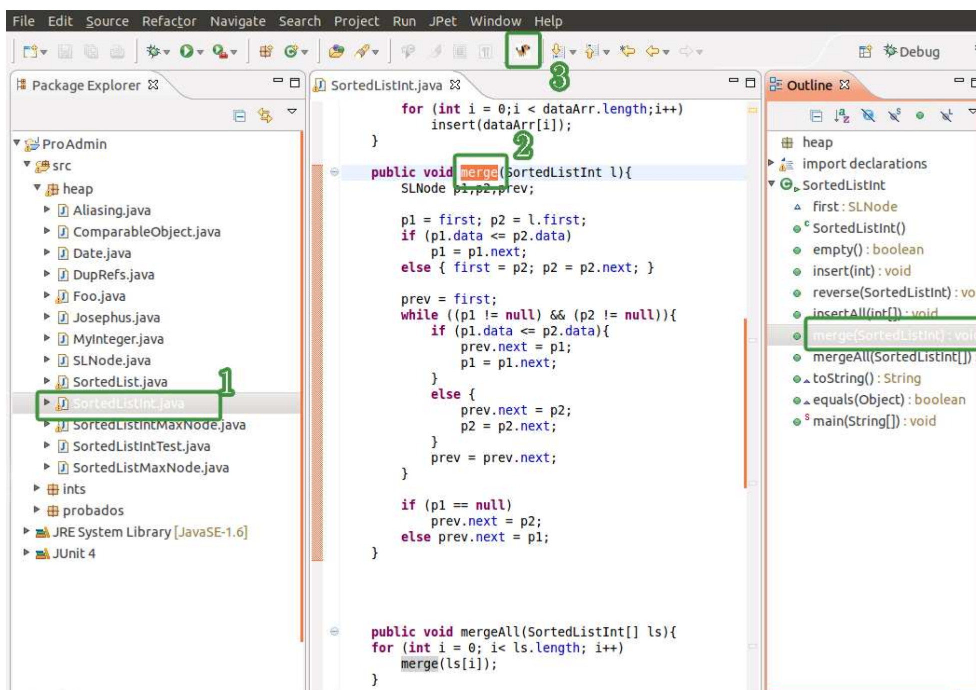


Una vez conectados, marcamos jPet y jPetFeature y pinchamos en el botón 'Next'. Y así completamos la instalación de jPet en Eclipse.

5.3. Manual de uso

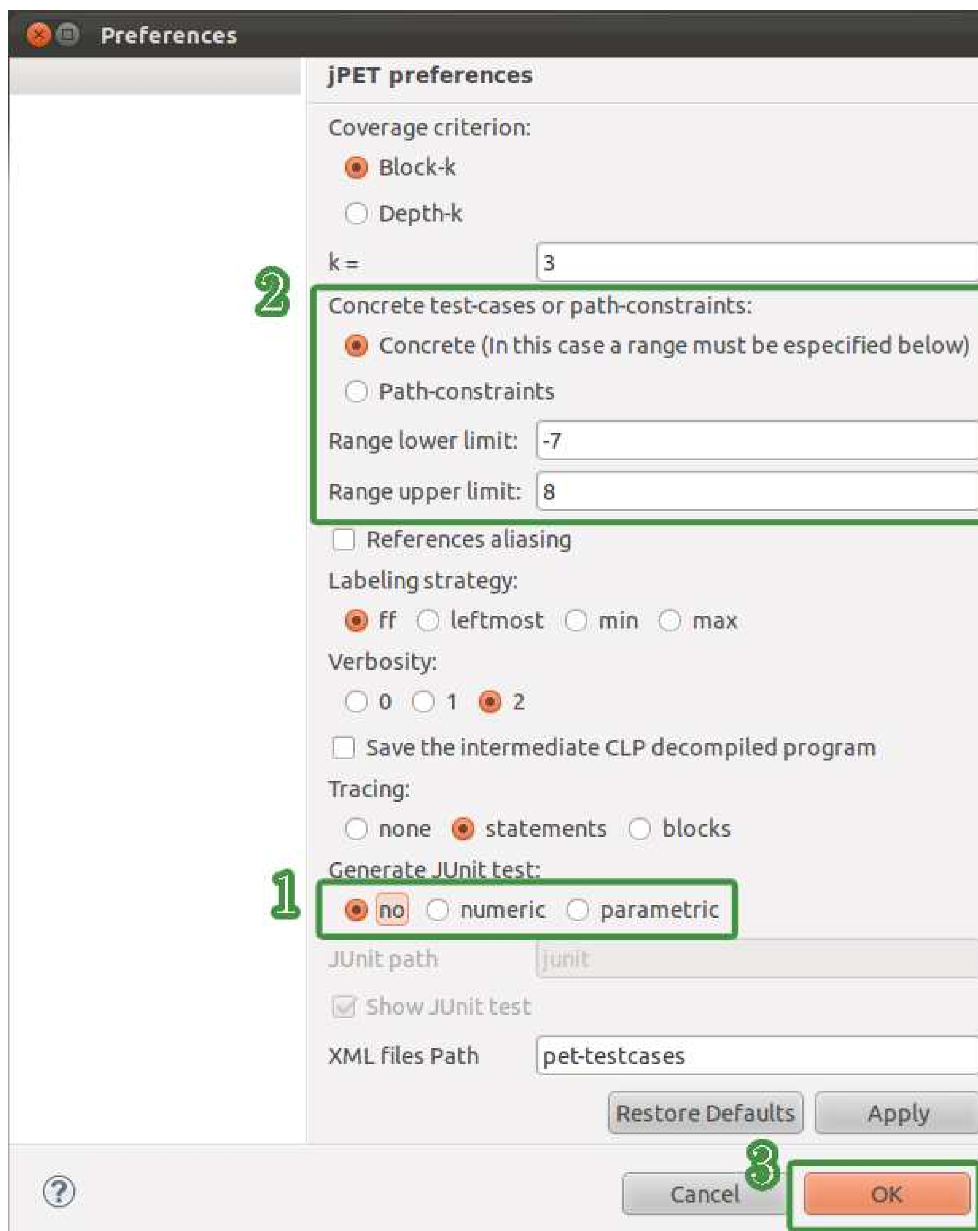
5.3.1. Ejecución

Una vez tengamos instalado jPet, podemos empezar a trabajar con él. La figura sig. ilustra el entorno de trabajo que veríamos y a la izquierda, las distintas clases de un proyecto de prueba. Para empezar usar jPet, seleccionamos una clase del proyecto. Dentro de ella, buscamos el método de la clase que queramos probar. Véase que a la derecha se listan todos los métodos de la clase, podemos seleccionar mas de uno para que jPet los trate. Pulsamos el botón de jPet, el nuevo icono con el perro en la parte superior de la interfaz de eclipse como se muestra en la siguiente figura.



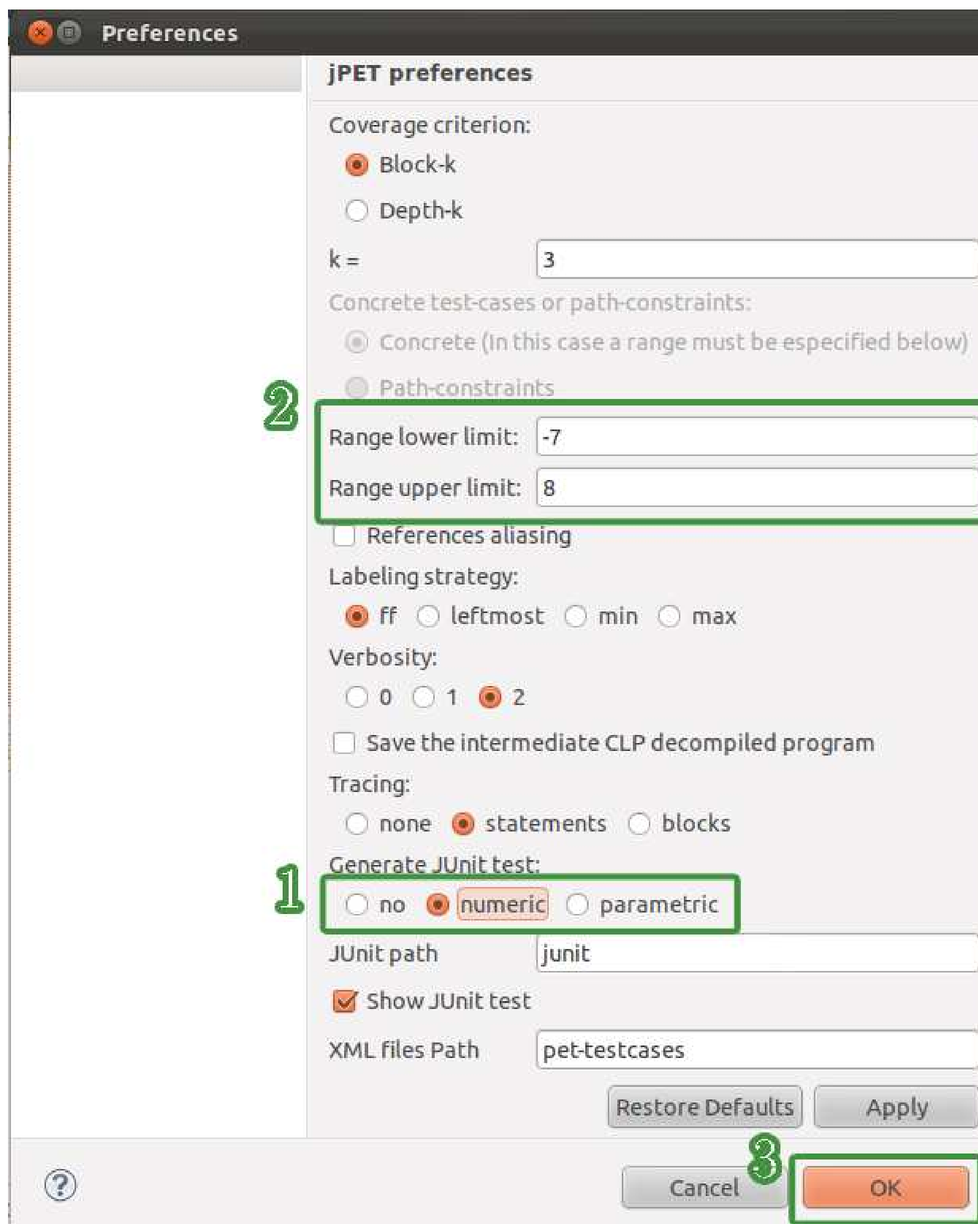
5.3.1.1. Sin JUnit

Después de hacer click en el botón, se abrirá la ventana de preferencias de jPet, en el que se muestran todas las opciones en la generación de junits. Criterio de recubrimiento, casos de prueba concretos, etc. Para generar casos de prueba sin junit, marcamos, junto al título 'Generate jUnit test:' la opción 'no' y pinchar en el botón 'Ok'. De esta manera, jPet no producirá ningún caso de prueba en formato junit. Nótese que seguirá cumpliendo con las especificaciones normales de jPet.



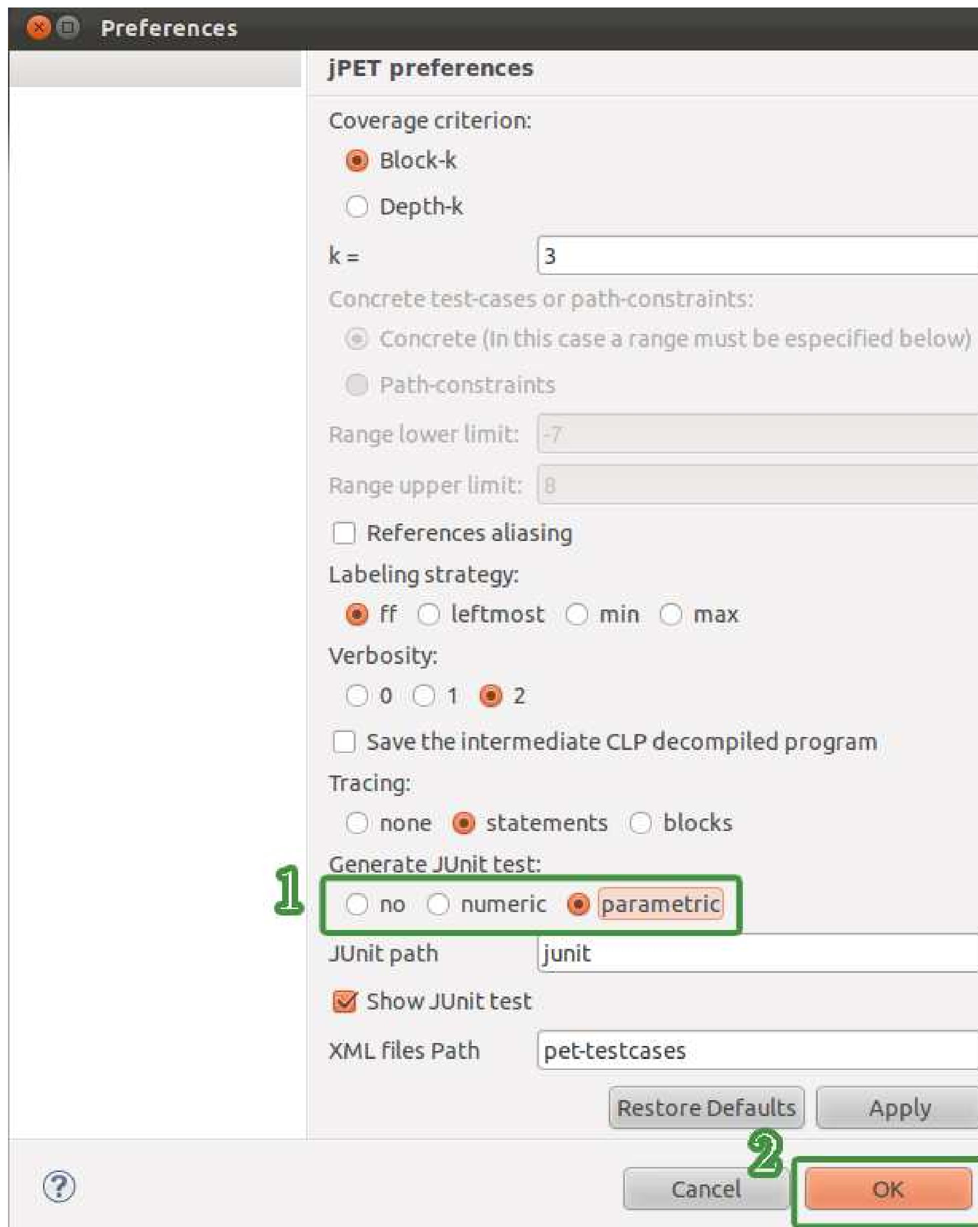
5.3.1.2. Con JUnit numérico

Para generar junits numéricos basta con marcar en la ventana de preferencias nombrada antes la opción ‘numeric’ del título ‘Generate jUnit tests’. También podemos definir el rango de valores con los que van a ser generados dichos junits. En el campo ‘Range lower limit:’ introduciremos el valor mínimo con el que queremos que se representen estos valores numéricos y en el campo ‘Range upper limit’ el valor máximo. Y para terminar hacemos click en el botón ‘Ok’.



5.3.1.3. Con JUnit paramétrico

Para generar junits paramétricos marcamos en la ventana de preferencias nombrada antes la opción ‘parametric’ del título ‘Generate jUnit tests’ y pinchamos en el botón ‘Ok’.



6. Conclusiones y mejoras de la aplicación

A continuación vamos a describir las características de la aplicación, así como las mejoras que se podrían realizar, cómo podrían llevarse a cabo y cuánto esfuerzo costaría.

6.1. Conclusiones

JPet es un generador automático de casos de prueba de programas Java. En versiones anteriores, esta herramienta generaba ficheros .xml con la información recopilada en los .clp de todos los casos de prueba posibles de un método seleccionado. Estos xml se representaban en una interfaz gráfica donde se mostraba la estructura de los casos de prueba.

Otra de las funcionalidades de la versión anterior era la muestra de la traza de cada uno de los tests de manera visual, coloreando cada línea de código por dónde pasaba.

En este proyecto se presenta un jPET ampliado, una herramienta de generación automática de casos de prueba integrada en el entorno de desarrollo Eclipse con el propósito de ayudar a los desarrolladores a probar sus programas Java sin tener que escribir un solo fichero de casos de prueba de unidad. Las contribuciones de este proyecto son:

- Permite crear casos de prueba de unidad a partir del recubrimiento realizado anteriormente. Existen dos opciones a elegir de casos de prueba de unidad:
  - Numéricos: en este caso creará un único fichero en el que las variables serán todo valores numéricos.
  - Paramétricos: En este caso se crean dos ficheros, en uno se muestran todos los casos de prueba con variables,

mientras que en el otro archivo se muestran llamadas a esos casos de prueba con valores concretos.

Esta funcionalidad cumple con los objetivos buscados. La creación de casos de prueba de unidad consigue que jPET sea una herramienta muy útil para los desarrolladores de software gracias al tiempo que ahorran al obtener los tests de forma automática.

6.2. Qué se puede mejorar

Las posibles mejoras que hemos contemplado para la aplicación serían: contemplar varias técnicas de testing, fichero xml, permitir distintos tipos primitivos, permitir la ejecución en Windows.

En este proyecto hemos incorporado la funcionalidad del testing, creando ficheros JUnit. Sin embargo, existen múltiples técnicas a la hora de escribir estos ficheros y nosotros sólo contemplamos uno. Añadir dichas técnicas sería provechoso para el desarrollador al hacer esta herramienta más cómoda, dejándole la posibilidad de elegir la manera que más se ajuste a su forma de trabajar.

El fichero xml se creó en un primer momento según los requisitos de la aplicación y con el tiempo ha sido necesario ir añadiendo información pero con el fin de ser lo más escueto posible, ha perdido parte de expresividad. Sería interesante reestructurar el xml de cara a facilitar futuros desarrollos.

En la actualidad, la aplicación sólo funciona con tipos primitivos enteros, por lo que su uso queda muy restringido. Incluyendo otros tipos primitivos, ampliaríamos las capacidades del plug-in y sería bastante más útil.

Gran cantidad de desarrolladores trabajan en el sistema operativo Windows, si esta herramienta corriera también bajo Windows, estos desarrolladores podrían hacer uso de este plug-in.

6.3. Posibles ampliaciones

Como futuras ampliaciones se pueden considerar varias opciones: poder editar los objetos en el visor, crear ficheros junit a partir del visor, permitir al desarrollador añadir precondiciones y postcondiciones mediante JML.

Actualmente, tanto el fichero xml como el junit, son plantillas creadas a partir del motor de JPet. Para facilitar al desarrollador a probar sus métodos con los datos deseados y no tener que modificar estos dos archivos manualmente, se podría ofrecer la opción de editar las estructuras de datos en el visor.

7. Bibliografía

[1]. Jiantao Pan, 1999
Software Testing
http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/

[2]. TestPlant
Black-box vs. White-box testing
http://www.testplant.com/wp-content/uploads/downloads/2011/06/BB_vs_WB_Testing-1.pdf

[3]. King, J. C. 1976. Symbolic execution and program testing. Commun. ACM 19, 7, pages 385-394.

[4]. M. Gomez-Zamalloa, E. Albert and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. 26th International Conference on Logic Programming (ICLP'10). TPLP Special Issue, Vol. 10 (4-6), pages 659-674, Cambridge U. Press, July 2010.

[5]. Tillmann, N. and de Halleux, J. 2008. Pex-white box test generation for .NET. In TAP. pages 134-153.

[6]. E. Albert, M. Gómez-Zamalloa, and G. Puebla. PET: A Partial Evaluation-based Test Case Generation Tool for Java Bytecode. In *Proc. of. PEPM'10*. ACM Press, 2010.

[7]. Junit & Junit Factory
http://www.junit.org/node/16

[8]. Automatic Junit Test Creator
https://developers.google.com/java-dev-tools/codepro/doc/features/junit/test_case_generation

[9]. Christian Engel and Reiner Hähnle, 2007. Generating Unit Tests from Formal Proofs. Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland

[10]. E. Albert, Israel Cabañas, Antonio Flores-Montoya, M. Gomez- Zamalloa, and Sergio Gutierrez. jPET: an Automatic Test-Case Generator for Java. 18th Working Conference on Reverse Engineering (WCRE'11). IEEE Computer Society, pages 441-442, October 2011.

[11]. Elvira Albert, Israel Cabañas, Antonio Flores- Montoya, Miguel Gómez-Zamalloa, and Sergio Gutiérrez. Software Testing using jPET. http://costa.ls.fi.upm.es/pet/papers/AlbertCFGG11.pdf, 2011.

[12]. Markus Dahm, Jason van Zyl, Enver Haase, Dave Brosius, and Torsten Curdt. BCEL. http://jakarta.apache.org/bcel/manual.html , 2006

8. Apéndice

A. Especificación de fichero XML

```
[XML] : :=    <pet>
                    [ TestCase ]  *
              </ pet>

[ TestCase ] : :=     <t e s t c a s e>
                            [ Firma ]
                            [ ArgsIn ]
                            [ HeapIn ]
                            [ HeapOut ]
                            [ Return ]
                            [ ExcFlag ]
                            [ Trace ]
                            [Constraints_in]
                            [Constraints_out]
                            [params]
                      </ t e s t c a s e>

[ Firma ] : := <method>
                    ' Firma de l metodo '
              </method>

[ ArgsIn ] : := <a r g s i n>
                    ( [ Ref ] j [ Data ] )  *
              </ a r g s i n>

[ HeapIn ] : := <heap_in>
                    [ Elem]  *
              </ heap in>

[ HeapOut ] : :=      <heap out>
                            [ Elem]  *
                      </ heap out>
```

```
[ Return ] : := <return>
                    ' Valor de r e t o rno '
                </return>

[ ExcFlag ] : := <exceptionflag>
                    ' Flag de excepcion '
                </exceptionflag>

[ Trace ] : := <trace>
                    [ Call ] _
                </trace>

[ Ref ] : :=    <ref>
                    'Nombre del parametro'
                </ref>

[ Data ] : :=   <data>
                    ' Valor del parametro'
                </data>

[ Elem] : :=    <elem>
                    [Num]
                    ( [ Obj ] j [ Array ] )
                </elem>

[Num] : :=      <num>
                    'Nombre del elemento'
                </num>

[ Obj ] : :=    <object>
                    <class name>
                        'Nombre de la clase'
                    </class name>
                    [ Fields]
                </object>

[Fields] : :=   <fields>
                    [Field] _
                <fields>
```

```
[ Field] : :=   <field>
                        <field name>
                                'Nombre del campo '
                        </field name>
                        ( [ Ref ] j [ Data ] ) _
                </field>

[ Array ] : := <array>
                        <type>
                                'Tipo del array'
                        </type>
                        <num elems>
                                'Numero de elementos '
                        </num elems>
                        [Args]
                </array>

[ Args ] : :=   <args>
                        [ Ref ]
                        [ Arg ] _
                </args>

[ Arg ] : :=    <arg>
                        'Nombre o valor de la variable'
                </arg>

[ Call ] : :=   <call>
                        [ Depth ]
                        [ ClassName ]
                        [MethodName ]
                        [ Instruction ] _
                </call>

[ Depth ] : :=          <depth>
                                ' Profundidad de la llamada '
                        </depth>

[ ClassName ] : :=      <class name>
                                ' Clase en la que se  esta ejecutando '
                        </class name>
```

[MethodName] : := <method name>
' Firma del metodo en el que se esta ejecutando '
</method name>

[Instruction] : := <instruction>
[PC]
[Bytecode]
</ instruction>

[PC] : := <pc>
' Contador de programa para la instruccion de
codigo de bytes'
</pc>

[Bytecode] : := <bytecode name>
'Nombre de la instruccion de codigo de bytes '
<bytecode name>

[Constraints-in] ::= <constraints_in>
[const]*
</constraints_in>

[Constraints-out] ::= <constraints_out>
[const]*
</constraints_out>

[const] ::= or [const] [const] | leq [ArExp] [ArExp] |
eq [ArExp] [ArExp] | neq [var] [var]

[ArExp] ::= Data | add[Data] [Data] | sub[Data] [Data] |
mul[Data] [Data] | div [Data] [Data] |
mod [Data] [Data]

B. Implementación del método “myEquals”

```
public final boolean myEquals(Object obj1, Object obj2) {

      boolean blEqual = false;
      Class obj1Class = obj1.getClass();
      Class obj2Class = obj2.getClass();
      if(obj1Class.getName().equals(obj2Class.getName())){
            if(obj1Class.equals(Integer.class))
                  return obj1 == obj2;
            else if (obj1Class.getName().equals("[I")){
                  if(((int[])obj1).length ==
                                          ((int[])obj2).length){
                        int sizeObj = ((int[])obj1).length;
                        for(int size = 0; size < sizeObj; size++)
                              blEqual = (((int[])obj1)[size]) ==
                                          (((int[])obj2)[size]);
            }
      }
      else if (obj1Class.getName().contains("[")){//si es un array
            if(((Object[])obj1).length ==
                                    ((Object[])obj2).length){
                  int sizeObj = ((Object[])obj1).length;
                              for(int size = 0; size < sizeObj;
size++)
                                    blEqual =
myEquals((((Object[])obj1)[size]),(((Object[])obj2)[size]));
                        }
                        else return false;
                  }
                  if (obj1 == null && obj2 == null || (obj1 ==
obj2))
                        return true;
                  else if( (obj1 == null && obj2 != null) || (obj1
!= null && obj2== null))
                        return false;
```

```
else{
      Field[] fields = obj1Class.getDeclaredFields();
      Object obj1FieldValue = null;
      Object obj2FieldValue = null;
      int count = 0;
      int i = 0;
      count = fields.length;
      while (i < count) {
            try {

      fields[i].setAccessible(true);
      obj1FieldValue = fields[i].get(obj1);
      obj2FieldValue = fields[i].get(obj2);
      if (obj1FieldValue != null && obj2FieldValue !=
null) {
      //comprobar el tipo si es primitivo o no
            if(obj1FieldValue instanceof Integer){
                  blEqual =
obj1FieldValue.equals(obj2FieldValue);
      }
            else
                  blEqual =
myEquals(obj1FieldValue,obj2FieldValue);
      }
      else if((obj1FieldValue == null && obj2FieldValue !=
null) || (obj1FieldValue != null && obj2FieldValue==
null))
            return false;
            } catch (IllegalAccessException e) {
                  e.printStackTrace();
            }
                  if (!blEqual) {
                        return blEqual;
                  }
                        i++;
                  }
                  return blEqual;
            }
      }
      else
            //different class
            return false;
}
```