Práctica 5
Depuración
Análisis estático de código

Javier Nogueras Iso y Miguel Ángel Latre
Escuela de Ingeniería y Arquitectura
Depto. de Informática e Ingeniería de Sistemas

1. Introducción

Los objetivos de esta práctica son los siguientes:

1. Utilizar el depurador de IntelliJ[1].
2. Utilizar el *inspector* de ItelliJ para el análisis estático de código[2].

2. Utilización del depurador de IntelliJ

2.1. El depurador de IntelliJ

La depuración, junto con la ejecución normal y la medición de cobertura, son los tres modos en los que IntelliJ puede ejecutar código. En modo de depuración, IntelliJ permite interrumpir la ejecución del programa en puntos previamente definidos, ejecutar el código paso a paso y observar y modificar los valores de las variables.

Para utilizar el depurador de IntelliJ, hay que definir al menos un *punto de interrupción* en el código. Cuando un programa se lanza en modo depuración (a través del icono con un insecto en la barra de herramientas, o a través del menú Run y la opción Debug), se ejecuta de forma

[1] Aunque en el guion de la práctica se mencione únicamente IntelliJ, todo lo expuesto es aplicable también al depurador de Android Studio. Sin embargo, el proyecto con el que se va a trabjar es un proyecto IntelliJ.
[2] El inspector también está disponible en Android Studio, donde añade también reglas específicas para Android.

normal hasta alcanzar alguno de los puntos de interrupción. En ese momento, la ejecución del programa se interrumpe e IntelliJ abre su panel de depuración, con información sobre los hilos en ejecución y sus pilas de invocaciones; un área denominada Variables en la que se pueden observar y modificar los valores de los parámetros y variables locales del método que se está ejecutando y en la que pueden añadirse otras variables en ámbito y expresiones para su evaluación tras la ejecución de cada instrucción. En el editor de código fuente, la próxima instrucción a ejecutar estará resaltada.

Una vez interrumpida la ejecución, puede analizarse el estado del programa observando la pila de ejecución y los valores de las variables. Las acciones relativas a cómo seguir ejecutando el código son las siguientes:

Step into F7 Se ejecuta la instrucción señalada por la línea actual. En caso de ser una llamada a un método, la ejecución se interrumpirá en la primera instrucción de ese método.

Step over F8 Se ejecuta la instrucción señalada por la línea actual. La ejecución se interrumpirá en la línea siguiente, independientemente de que en la línea actual se invoquen métodos o no.

Step out Mayús + F8 Continúa la ejecución del programa hasta que finalice la ejecución del método al que pertenece la instrucción señalada por la línea actual. La depuración continuará en la siguiente línea a la instrucción en la que se invocó al método actual.

Resume Program F9 Continúa la ejecución del programa hasta que alcance un punto de interrupción o hasta que finaliza la ejecución del programa.

Stop Ctrl + F2 Finaliza la ejecución del programa.

Es posible establecer que determinados paquetes Java se excluyan del proceso de depuración. Para ello, hay que ir al menú File > Settings y seleccionar Build, Execution, Deployment > Debugger > Stepping.

2.2. Variables

En el panel Variables, es posible observar variables que se encuentran en ámbito cuando el programa está interrumpido y sus valores correspondientes. En el caso de que sean objetos, se muestra como valor el resultado de invocar a su método `toString()`. En el caso de variables de tipos primitivos o cadenas de caracteres, su valor **puede ser modificado** sin más que hacer clic con el botón derecho del ratón y elegir la opción Set Value... o presionar la tecla F2.

Para ver atributos estáticos de las clases, o visualizar los tipos declarados de los atributos y de las variables, hay que seleccionar del menú contextual la opción Customize Data Views....

2.3. Puntos de interrupción

Para acceder al diálogo de puntos de interrupción hay que hacer clic en el botón View Breakpoints en la barra de herramientas vertical que hay a la izquierda del panel de depuración o presionar la combinación de teclas Ctrl + Mayús + F8. El diálogo muestra una lista con todos los puntos de interrupción del proyecto y permite añadir nuevos, borrar los existentes, activar o desactivarlos y configurar sus propiedades. De esta forma, puede establecerse el comportamiento de un punto de interrupción activo: suspender la ejecución de todos los hilos del programa (opción predeterminada) o solo del que ha alcanzado el punto de interrupción, o escribir un mensaje fijo en la consola o con una expresión evaluada en el momento en el que se alcanza el punto de interrupción. También se puede desactivar una vez que haya sido alcanzado (Remove once hit) o hacer que se active solo cuando se haya pasado por él un determinado número de veces (Pass count), cuando se satisfaga una determinada condición (Condition) o cuando se haya alcanzado otro punto de interrupción concreto (Disabled until selected breackpoint is hit).

Para desactivar todos los puntos de interrupción, hay que hacer clic en el botón Mute Breakpoints de la barra de herramientas vertical del panel de depuración.

Existen puntos de interrupción más especializados y que se describen a continuación:

Punto de interrupción de métodos («*Java Method Breakpoints*»). Es posible establecer puntos de interrupción que se activan cuando comienza la ejecución de un determinado método, cuando concluye o en ambos casos. Para ello, puede hacerse clic en la barra de la izquierda del editor de código, a la altura de la línea correspondiente a la cabecera del método en cuestión o añadirse desde el diálogo de puntos de interrupción. Este tipo de puntos de interrupción pueden ralentizar significativamente la ejecución en modo depuración.

Punto de observación («*Java Field Breakpoints*»). Se denominan así los puntos de interrupción asociados a un atributo de una clase. El depurador se detendrá cada vez que se consulte o modifique el valor del atributo. Para establecerlo, se puede hacer clic en la barra de la izquierda del editor de código, a la altura de la línea correspondiente a la declaración del atributo, o añadirse desde el diálogo de puntos de interrupción.

Punto de interrupción de excepciones («*Java Exception Breakpoints*»). Es posible establecer puntos de interrupción que se activan cuando se lanza una determinada excepción. Pueden añadirse y configurarse desde el diálogo de puntos de interrupción. Es posible configurar si el punto de interrupción debe activarse al lanzarse excepciones que se capturan, al lanzarse excepciones que no se capturan o en ambos casos.

2.4. La pila de llamadas

En la parte izquierda del panel de depuración, podemos encontrar la pila de llamadas de cada uno de los hilos de ejecución de que conste el programa (pestaña Threads) o solo la

pila del hilo que ha alcanzado el último punto de interrupción (pestaña Frames). Al pulsar en cualquiera de los elementos de la pila, podemos ver las variables en ámbito en esa invocación, así como sus valores. El depurador de IntelliJ permite seleccionar cualquier nivel de la pila, y hacer que la máquina virtual de Java vuelva a ejecutar desde ese nivel. Para ello, hay que elegir la opción Drop Frame del menú contextual. No es exactamente lo mismo que volver a ejecutar el programa desde ese punto, puesto que todas las modificaciones que se hayan hecho a variables y objetos que sigan existiendo se mantendrán.

2.5. Actividades a realizar en la práctica

En esta práctica trabajaréis con los módulos «`GildedRose0`», «`GildedRose1`» y «`GildedRose2`» del proyecto «`unizar-vv-practica5`» disponible en GitHub[3], correspondientes a la *kata* Gilded Rose[4]. Los proyectos «`GildedRose1`» y «`GildedRose2`» tienen un defecto cada uno y deberéis utilizar el depurador de IntelliJ para encontrarlo y corregirlo.

Antes, para familiarizaros con el código, completaremos el ejercicio de diseño de pruebas realizado en clase a través del proyecto «`GildedRose0`».

2.5.1. Análisis de las pruebas existentes en el proyecto «`GildedRose0`»

1. Echad un vistazo a las clases `Item` e `Inventory` de la carpeta «`src/main/java`» y a la clase `InventoryTest` de la carpeta «`src/test/java`».
2. Ejecutad las pruebas correspondientes a la clase `InventoryTest` y comprobad que todas las pruebas pasan sin detectar errores.
3. Comparad las pruebas de la clase `InventoryTest` con las que diseñamos en la clase de pizarra. Anotad las correspondencias entre ellas a través de comentarios en los métodos de la clase `InventoryTest`.

   Utilizad como referencia la hoja de cálculo disponible en Moodle, donde también podéis anotar qué método de la clase `InventoryTest` implementa cada una de ellas. Como la clase `Item` no cuenta con un método `equals()`, algunos test de la clase `InventoryTest` comprueban el valor del atributo `sellIn`, mientras que otros, comprueban el valor del atributo `quality`. Esta es la causa de que en la hoja de cálculo, haya dos columnas (una por atributo) para establecer la correspondencia.

   ¿Hay métodos de prueba en la clase `InventoryTest` que no diseñáramos en la clase de pizarra?

   ¿Hay métodos que diseñáramos que no aparezcan en la clase `InventoryTest`? Si es así,

[3] `https://github.com/miguel-latre/unizar-vv-practica5`

[4] El proyecto «`unizar-vv-practica5`» está basado en la versión de Java preparada por Alex Aitken (`https://github.com/alexaitken/GildedRose_java`) de la kata original de Terry Hughes y Bobby Johnson para C# (`https://github.com/NotMyself/GildedRose`)

no los implementéis todavía.

4. Medid la cobertura de las pruebas de `InventoryTest` en la clase `Inventory` con IntelliJ y comprobad que no es del 100 %. Recordad modificar la configuración de la medición de la cobertura para utilizar la opción *«Tracing»* en lugar de la de *«Sampling»*.
5. Implementad los métodos de prueba que diseñamos en clase y que faltan de la case de pruebas proporcionada. Tras la implementación de cada método, volved a medir la cobertura y comprobar cómo se va incrementando, pero como sigue sin alcanzarse el 100 %.
6. Averiguad qué instrucciones y condiciones quedan sin cubrir y escribid nuevas pruebas en la clase `InventoryTest` hasta lograr una cobertura del 100 %.

2.5.2. Actividades de depuración

A continuación, vais a tener que utilizar el depurador de IntelliJ para encontrar (y, posteriormente, corregir) los defectos que hay en los proyectos «`GildedRose1`» y «`GildedRose2`».

Con el proyecto «`GildedRose1`» haced lo siguiente (aunque es más para que os habituéis al depurador de IntelliJ, que para identificar el problema en sí):

1. Copiad el conjunto de pruebas de la clase `InventoryTest` del proyecto «`GildedRose0`».
2. Ejecutad las pruebas para ver cuáles fallan. Utilizad el modo de ejecución normal, no el de medición de cobertura[5].
3. Estableced un punto de interrupción en la invocación al método `updateQuality()` de la prueba que falla.
4. Ejecutad las pruebas en modo depuración y, cuando la ejecución se interrumpa en el punto de interrupción que habéis establecido, continuadla dentro del método `updateQuality()`.
5. Solo por experimentar, cambiad el valor de un atributo de un artículo.
6. Estableced un formato detallado para los objetos de la clase `Item` que permita conocer el nombre del objeto, el número de días que faltan para vender y la calidad de cada artículo.

   Para ello, seleccionad un objeto de la clase `Item` del panel de variables y, en su menú contextual, seleccionad las opciones View as » Create... . Habréis entrado a un diálogo de configuración de formatos para objetos. En la sección encabezada con When rendering a node, marcad Use the following expression: y escribid una expresión similar a la que escribiríais para completar la instrucción **`return`** de un método `toString()` para la clase `Item`.

[5] Con la cantidad de pruebas con las que cuenta este código, los resultados de la medición de cobertura prácticamente identifican el defecto.

7. Estableced un punto de observación en el atributo `quality` de la clase `Item` y reiniciad la depuración.
8. Estableced un punto de interrupción cuando se ejecute el método `setQuality` de la clase `Item` y reiniciad la depuración.
9. Estableced un punto de interrupción cuando se cargue la clase `Item` y reiniciad la depuración.
10. Configurad los puntos de interrupción (activación, desactivación y *pass count* o condición) de tal forma que, cuando se inicia la depuración, la ejecución se detenga únicamente una vez en la prueba que falla y luego, en el método `updateQuality()` cuando se va a procesar el artículo que dará un fallo.

3. Análisis estático del código con IntelliJ

IntelliJ incluye un componente (denominado *Inspections*) para el análisis estático de código cuyo funcionamiento se basa en detectar patrones correspondientes a defectos que pueden manifestarse como fallos o errores en tiempo de ejecución, código que no se puede ejecutar nunca, código que puede ser optimizado, expresiones lógicas que puedan ser simplificadas, malos usos del lenguaje, código replicado, etc.

3.1. Actividades a realizar en la práctica

En esta parte de la práctica volveremos a trabajar con el módulo «`GildedRose0`» de la *kata* Gilded Rose.

1. Ejecutad la herramienta de análisis estático: menú Analyze › Inspect Code... . Elegid el módulo «`GildedRose0`» como ámbito y dejad seleccionada la opción Include test sources.
2. En el panel que se abrirá, observad el tipo de problemas que detecta y los problemas en sí mismos en el código. Tomad nota de ellos para la próxima clase de teoría e idlos corrigiendo. Apuntad también si os parecen razonables o no. Id refrescando periódicamente el inspector de código (botón Rerun inspections de la barra de herramientas vertical del panel de inspección) y los tests.
3. Una vez que estén corregidos, volved a ejecutar el inspector (a través del menú principal), pero cambiando la configuración: en el cuadro diálogo, en la sección Inspection profile, haced clic en los puntos suspensivos y marcad todas las reglas de Java.
4. De nuevo, observad y tomad nota de los problemas detectados. No os obsesionéis con corregirlos todos, el objetivo es el de discutir esas reglas en clase.

4. Entrega de resultados de la práctica

Como resultado de esta práctica se entregará un documento de texto (texto plano, Word, OpenDocument o PDF) en el que se indique lo siguiente:

1. Qué métodos de prueba en la clase `InventoryTest` no habíamos diseñado en la clase de pizarra (sección 2.5.1).
2. Qué métodos que diseñamos en la clase de pizarra no aparecen en la clase `InventoryTest` (sección 2.5.1).
3. Las pruebas que hayáis añadido para lograr una cobertura del 100 % (sección 2.5.1).
4. Descripción de los defectos de los módulos «`GildedRose1`» y «`GildedRose2`» (sección 2.5.2).
5. Descripción de las violaciones de las reglas del inspector que aparecen en el módulo «`GildedRose0`» y opinión que os merezcan las reglas violadas (sección 3.1).

El límite para la entrega es el día anterior a la siguiente sesión de prácticas. Dada la situación actual y para poder ser operativos en la sesión de prácticas, esta práctica se realizará en equipo, de acuerdo con la distribución de grupos para el trabajo de la asignatura. Solo es necesario que realice la entrega un miembro de cada grupo.

El análisis de los resultados de inspección se discutirá en la clase del lunes 14 de mayo.