Práctica: Pruebas basadas en cobertura de grafos (I)

Departamento de Teoría de la Señal y Comunicaciones y Sistemas Telemáticos y Computación, URJC

29 de noviembre de 2017

Desarrolla estos ejercicios en diferentes repositorios de git en el que para cada tarea distinta utilices una rama distinta: escritura de un test nuevo, escritura de nueva funcionalidad o corrección de un fallo en el código que estás probando, etc. Ve mezclando en la rama `master` los commits de las ramas que vayas creando.

En cada *commit* escribe comentarios descriptivos que expliquen/justifiquen el código escrito o modificado en cada *commit*, y cualquier otro detalle que consideres relevante para entender la razón por la que has pensado que es oportuno crear el *commit*.

Ejercicio 1 (individual)

Repasa el ejemplo visto en clase con la clase `Quadratic` para entender lo que tienes que hacer en este ejercicio.

Escribe pruebas de integración para el código de la clase `Stutter.java`[1]. Analiza para ello el flujo de datos a través de las interfaces de los métodos, definiendo los pares-du alrededor de las llamadas que hacen entre sí los métodos de `Stutter`.

1. Dibuja el grafo de control de flujo de cada uno de los métodos de la clase `Stutter`: `main()`, `stut()`, `checkDupes()`, `isDelimit()`. Desde los nodos en los que se realice una llamada a otro método, traza un arco al método llamado. Puede venirte bien identificar estos arcos de algún modo para realizar los siguientes apartados (pinta el arco con línea discontínua). Desde los nodos de un método `m` en los que se haga `return`, traza arcos a los nodos de otros métodos desde los que se hicieron llamadas a `m` (un arco distinto para retornar a cada uno de los lugares desde los que se llama a `m`).

2. En una tabla de dos columnas identifica los lugares en el código en los que se realizan llamadas a métodos: para cada llamada identifica la línea de código y el nombre del método que realiza la llamada, y el método al que se llama.

3. Etiqueta los nodos y arcos con los últimos *def* y primeros *use* de las variables globales (atributos de clase *static* `Stutter`) y los parámetros que se definen y usan en las interfaces de los métodos llamados:

   a) *def* justo antes de un lugar de llamada y *use* justo después del método llamado

   b) *def* justo antes de un método que ejecuta `return` y *use* justo después del lugar de llamada al que se retorna

4. Recoge en una tabla de tres columnas los *pares-du* definidos en el anterior apartado:

   a) 1ª columna: número de línea de la llamada a método alrededor del que está definido el par-du

   b) 2ª columna: método, variable y número de línea del último-def

   c) 3ª columna: método, variable y número de línea del primer-uso

[1] Dado que los tests JUnit que acabarás diseñando en esta práctica ejecutan el método *main* de la clase *Stutter*, podemos considerar estos tests como pruebas de sistema.

Para identificar los números de línea utiliza los números definidos en el fichero `Stutter.num` que se proporciona. Alternativamente, puedes utilizar los números de línea de `Stutter.java` mostrados por Eclipse (o tu editor de confianza).

Recuerda incluir tanto los pares-du al hacer una llamada, como los pares-du al retornar una llamada.

5. Diseña pruebas con JUnit que satisfagan el requisito *All-uses coverage (AUC)* pero sólo para los pares-du identificados en los anteriores apartados (los definidos alrededor de los lugares en los que un método llama a otro).

   Utiliza el fichero `StutterTest.java` para realizar las pruebas.

   Añade las pruebas que consideres necesarias a este fichero, identificando en el comentario de cada test qué pares-du de la tabla del anterior apartado satisfacen.

   Puede ayudarte a realizar este apartado utilizar la herramienta de comprobación de cobertura de Eclipse. Podrás así ir comprobando qué sentencias quedan cubiertas por cada uno de los test indivuduales que diseñes, facilitándote la identificación de pares-du que quedan cubiertos por cada uno de los test.

Ejercicio 2 (individual)

Las variables que definen el comportamiento del termostato que regula la temperatura de un hogar son las siguientes:

```
partOfDay : {Wake, Sleep}
temp : {Low, High}
```

Se considerará que inicialmente los valores de estas variables al arrancar el termostato son siempre estos: `partOfDay == Wake`, `temp == Low`.

Los métodos que permiten modificar estas dos variables son los siguientes:

```
// Efecto: avanza a la siguiente parte del día: Wake => Sleep => Wake,...
public void advance();

// Efecto: Si Temp == Low, sube Temp a High
public void up();

// Efecto: Si Temp == High, baja Temp a Low
public void down();
```

1. Dibuja el grafo de una máquina de estados finitos (MEF) para el termostato. Etiqueta cada estado con los valores de las variables que lo definen, y los arcos con las acciones que provocan la transición entre estados.

2. Diseña los caminos de pruebas que sean necesarios para satisfacer el criterio de cobertura de arcos (*edge coverage*) relativo al grafo de la MEF del termostato.

3. Usando JUnit escribe el código de las pruebas en el fichero `ThermostatTest.java`.

4. Escribe después el código de la clase `Thermostat` en el fichero `Thermostat.java`.

Normas de entrega

Se dispondrá una tarea de entrega en el Aula Virtual en la que deberás especificar la url del repositorio GitHub de cada Ejercicio.