Polytech Lille IMA4 SC 2018-2019

Conception Logicielle à Objets

TP3

Simulation

L'état logique d'un composant est fourni par sa méthode `boolean getEtat()`. Un interrupteur peut être positionné à `true` ou `false` par ses méthodes respectives `on()` et `off()`. L'état des autres composants peut-être calculé en fonction de l'état des composants connectés sur ses entrées. Cependant, si le composant n'est pas complètement connecté, ce calcul ne peut être effectué et provoque une exception `NonConnecteException`.

Les classes fournies précédemment sont complétées comme suit (`~bcarre/public_html/ima4-FSC/circuits/simulation`) :

```
public class NonConnecteException extends Exception {}

public abstract class Composant {
 //...
 public abstract boolean getEtat() throws NonConnecteException;
}

public class Interrupteur extends Composant {
 //...
 protected boolean etat;
 public void on() {
   etat = true;
 }
 public void off() {
   etat = false;
 }
 public boolean getEtat() throws NonConnecteException {
   return etat;
 }
}
public class Vanne extends Composant {
 //...
 public boolean getEtat() throws NonConnecteException {
  if (in == null) {
   throw new NonConnecteException();
  } else {
   return in.getEtat();
  }
}
```

```
public abstract class Porte extends Composant {}

public class Not extends Porte {
 //...
 public boolean getEtat() throws NonConnecteException {
  if (in == null) {
   throw new NonConnecteException();
  } else {
   return !in.getEtat();
  }
}
```

1 Evaluation des portes

— Récupérer le code fourni
`~bcarre/public_html/ima4-FSC/circuits/simulation`
et l'examiner
— Complétez vos classes avec ce code
— Programmer de façon similaire la méthode `getEtat()` des `And` et `Or`.

Test

— Dans la classe `TestCircuits` programmer une méthode `traceEtats` paramétrée par un tableau de composants qui affiche leur description et leur état et l'appliquer dans la section `//Affichage` du `main` sur le tableau `composants`.
— Compiler et noter l'erreur "`unreported exception NonConnecteException; must be caught or declared to be thrown`" si vous n'avez pas tenu compte de cette exception

1. 1ère expérience : laisser se propager l'exception ("`declare to be thrown`") jusqu'au système : arrêt de programme avec trace de l'exception)
   (a) déclarer `throws NonConnecteException` sur `traceEtats`. Recompiler et noter la même erreur au niveau du main (en effet, le main appelle `traceEtats` qui peut provoquer cette exception)
   (b) déclarer `throws NonConnecteException` sur le `main`
   (c) exécuter et noter la sortie de programme sur l'exception dès qu'il y a un composant non connecté (sans vous dire "`Au revoir!`").
2. 2ème expérience : capturer l'exception dans le `main`
   (a) supprimer la déclaration `throws NonConnecteException` sur le `main`
   (b) entourer l'appel à `traceEtats` dans la section `//Affichage` par un `try/catch` pour afficher "`Au moins un composant non connecte`".
   (c) exécuter (et noter que le programme est plus poli).
3. 3ème expérience : capturer l'exception dès `traceEtats`
   (a) supprimer la déclaration `throws NonConnecteException` sur `traceEtats`
   (b) entourer l'appel à `getEtat()` par un `try/catch` pour afficher "`Composant non connecte`"
   (c) exécuter et noter que le programme ne s'arrête plus au premier composant non connecté mais poursuit l'exécution sur tous les composants (jusqu'à vous dire "`Au revoir!`").

2 OO++ (toujours plus orienté objet...)

2.1 Trace des états

Reporter la trace d'état d'un composant (description + état) dans la classe Composant en programmant une méthode `public String traceEtat()` qui fournit une chaîne formée de la concaténation de sa description et de son état ou "`Composant non connecte `". Noter l'économie de code et la généricité implicite de cette méthode grâce au polymorphisme de redéfinition : une seule méthode `traceEtat()` commune à tous les composants liée dynamiquement à leur propre (`this`) méthode `description()` et `getEtat()`.

Test

Modifier la méthode `traceEtats` de la classe `TestCircuits` en conséquence et tester.

2.2 Porte2Entrees

Remarquer la similitude du code de `getEtat()` des portes `And` et `Or` : le contrôle des connexions en entrée est commun, seule l'évaluation logique leur est propre. Factoriser `getEtat()` dans `Porte2Entrees` en introduisant une méthode `boolean eval()` qui "reporte" l'évaluation logique dans chaque sous-classe.

Test

Bien noter la compilation séparée et le chargement dynamique de code : il suffit de recompiler les classes `Porte2Entrees`, `And` et `Or` seulement. Les autres classes ne sont pas impactées par ce changement, notamment `TestCircuits` qui peut être exécutée sans recompilation (le protocole des portes qu'elle utilise n'a pas changé).

3 Sondes

Sonder des composants consiste à positionner sur certaines de leurs entrées des sondes interactives qui demandent à l'utilisateur quelle valeur logique forcer afin d'observer leur sortie ou de simuler interactivement un circuit.

3.1 Sondes interactives

Programmer une classe `Sonde` respectant les spécifications suivantes :

— `Sonde` est sous-classe de `Composant` de telle façon qu'une sonde puisse être connectée en entrée d'un composant au même titre que tout autre composant par les méthodes $\texttt{setIn}_{[i]}$

— une sonde mémorise le composant et le nom de l'entrée sur laquelle elle est connectée par paramétrage de son constructeur. Par exemple (remplacement de l'interrupteur i1 en entrée "`in1`" du `Or` du circuit exemple par une sonde interactive) :
```
or.setIn1(new Sonde(or,"in1"));
```

— la méthode `getEtat()` d'une sonde demande à l'utilisateur quelle est la valeur logique à forcer à des fins de test, par exemple :
```
in1 de Or@55e83f9, true ou false?
```

Test

— Dans la section `//Connexions` de `TestCircuits`, commenter certaines connexions et connecter des sondes à la place.

— Tester en traçant l'état.

3.2 Sondes interactives paresseuses

Les sondes précédentes s'appliquent bien pour tester un composant isolément. Par contre vous avez remarqué que le mode d'évaluation de tout un circuit (par "rétro-évaluation") fait que les composants en amont sont "sondés" interactivement plusieurs fois (ce qui est inconfortable pour l'utilisateur et risque surtout d'être incohérent si les réponses varient !).

— Programmer une classe `LazySonde` [1] sous-classe de `Sonde` qui permet de résoudre ce problème :
  — mémoriser la valeur fournie par l'utilisateur la première fois et la renvoyer les fois suivantes
  — permettre un `reset()` pour un autre "sondage".
— Retester. Grâce au polymorphisme de redéfinition il doit suffire de remplacer la classe `Sonde` par `LazySonde` dans les tests de connection de la question précédente.

1. sondes paresseuses car elles ne demandent une valeur qu'une seule fois.