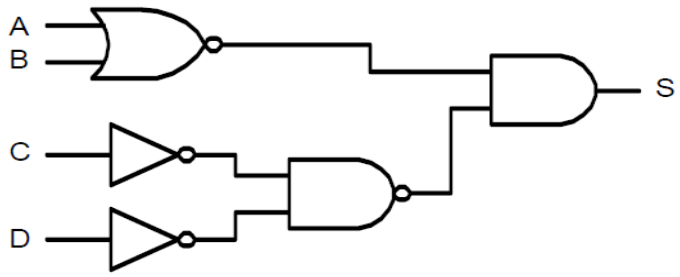
POBJ – Examen
Mai 2016

Tous documents autorisés

Durée : 2 heures

PROBLÈME : CIRCUITS (PAS TRÈS) LOGIQUES

On souhaite développer un programme permettant de construire et simuler des circuits logiques. Ces circuits sont constitués de hiérarchies de modules, qui eux-mêmes contiennent des générateurs de signaux binaires aléatoires et des portes logiques réalisant par exemple des AND, des XOR ou des OR entre des entrées binaires. Une particularité est que ces circuits sont définis progressivement au travers d'une interface graphique, si bien qu'il est possible de manipuler des circuits dont les portes ne sont que partiellement connectées.



Façon générale de noter : en pourcentage. De l'ordre de -10 % pour chaque petites fautes de syntaxe (oubli de ;, {}, ()... Pénaliser une seule fois pour chacune de ces erreurs (-10%) l'absence d'import, de packages, les visibilités absentes ou erronées (attributs publics ou protected)...

Ne pas pénaliser les solutions lourdes, l'étudiant se pénalise tout seul en perdant du temps.

Tous les développements se situeront dans un package *units*.

1 : Unités simples

Pour représenter une unité de base du circuit, on définit l'interface suivante :

```
package units;

import java.util.List;

public interface Unit {
	// renvoie le nom symbolique de l'unité (fonction + numéro)
	public String getName();
	// renvoie les unités connectées en entrée
	public List<Unit> getInputUnits();
	// renvoie l'unité connectée en sortie
```

```
	public Unit getOutputUnit();
	// renvoie le signal de sortie de l'unité
	public boolean getOutputSignal();
	// calcule le signal de sortie de l'unité sachant les signaux d'entrées
	// lève une exception si l'unité ne possède pas un nombre d'entrées valide
	public void run() throws InvalidConnectionException;
	// branche l'unité à une autre en sortie
	public void plugTo(Unit destination);
}
```

Q1.1 : Définissez une classe *InvalidConnectionException* qui sera utilisée dans la suite. Le constructeur de cette classe ne prend pas de paramètres.

```
package units;

public class InvalidConnectionException extends Exception {
	private static final long serialVersionUID = 1L;

	public InvalidConnectionException(){
		super("Error : unit not connected");
	}
}
```

le serialVersionUID n'est pas obligatoire
-50 % si pas extends Exception
-20 % si pas de message d'erreur (idéalement, passé dans super)
NB : je n'en ai qu'un qui l'a fait...
-20 % si pas de super dans le constructeur (constructeur par défaut (absent) possible, mais attention au message)

Afin de factoriser quelques comportements génériques des unités, on va les regrouper dans deux niveaux de classes abstraites.

Q1.2 : Définissez une classe abstraite *AbstractUnit* qui implémente l'interface *Unit :*

- La méthode *public void run() throws InvalidConnectionException* est laissée abstraite.
- Le constructeur prend le nom de l'unité en paramètre.
- La méthode `public void plugTo(Unit destination)` connecte le récepteur (this) à sa destination en sortie et connecte le récepteur en entrée de l'unité de destination.
- Une méthode *protected void setOutputSignal(boolean b)* permet d'affecter le signal de sortie de l'unité.

```
package units;

import java.util.ArrayList;
import java.util.List;

public abstract class AbstractUnit implements Unit {
	private List<Unit> inputs = new ArrayList<Unit>();
	private Unit output;
	private boolean outputSignal;
	private String name;

	public AbstractUnit(String name){
		this.name = name;
```

```
		}
	@Override
	public List<Unit> getInputUnits() {
		return inputs;
	}
	public Unit getOutputUnit() {
		return output;
	}
	@Override
	public String getName() {
		return name;
	}
	@Override
	public boolean getOutputSignal() {
		return outputSignal;
	}
	protected void setOutputSignal(boolean b) {
		outputSignal= b;
	}
	@Override
	public void plugTo(Unit destination){
		destination.getInputUnits().add(this);
		output = destination;
	}
	@Override
	public abstract void run() throws InvalidConnectionException;
}
```

Question un peu pénible à corriger

10 % pour la déclaration (public abstract, implements Unit), 20 % pour les attributs et les accesseurs, 10 % pour setOutputSignal et 20 % par autre méthode juste (constructeur, plugTo, run). Attention à l'init de l'arraylist.

Fautes vues : new List, pas mal d'erreurs sur plugTo, des {} sur méthode abstract (je mets 0/20%)

Q1.3 : Les unités simples se distinguent des unités quelconques par le fait qu'elles ont une fonction de transfert et un nombre d'entrées connu a priori. Définissez une classe abstraite *AbstractSingleUnit* qui étend *AbstractUnit*. Cette classe définit deux méthodes publiques et abstraites (qui seront redéfinies par des classes concrètes dans les questions suivantes) :

- `public boolean callTransferFunction()` qui calcule le signal de sortie de l'unité en fonction de ses entrées,
- `public int getRequiredInputNumber()` qui renvoie le nombre d'entrées nécessaires pour calculer la sortie.

Par ailleurs, elle définit deux méthodes publiques concrètes dont vous donnerez le code :

- `boolean isConnectionValid()`, qui renvoie *true* si le nombre d'entrées de l'unité est égal au nombre requis, *false* sinon,
- `void run() throws InvalidConnectionException`. Cette méthode lève l'exception `InvalidConnectionException` si la connection n'est pas valide. Sinon, elle appelle la fonction de transfert et affecte le signal de sortie de l'unité.

```
package units;
```

```
public abstract class AbstractSingleUnit extends AbstractUnit {

	public abstract boolean callTransferFunction();
	public abstract int getRequiredInputNumber();

	public AbstractSingleUnit(String name){
		super(name);
		}

	public void run() throws InvalidConnectionException {
		if (!isConnectionValid()) throw new InvalidConnectionException();
		setOutputSignal(callTransferFunction());
	}
	public boolean isConnectionValid() {
		return (getInputUnits().size()==getRequiredInputNumber());
	}
}
```

10 % pour la déclaration (public abstract, `extends AbstractUnit`, 10 % pour chaque méthode abstraite, 10 % pour le constructeur 20 % pour isConnectionValid() et 40 % pour run()

Fautes fréquentes : oubli du constructeur, ajout d'un entier « nbEntree » et comparaison à ce nombre dans isConnectionValid(), oubli d'abstract sur les méthodes abstraites

Q1.4 : A titre d'exemple, on va se doter de deux classes concrètes d'unités simples, les *AndUnit* et les *OrUnit*. Définissez les classes *AndUnit* et *OrUnit, qui étendent AbstractSingleUnit* et représentent respectivement le « Et » logique (noté « && » en java) et le « Ou » logique (noté « || » en java) entre deux entrées. Le constructeur prendra le nom de l'unité en paramètres. Ces classes doivent définir les méthodes qui sont restées abstraites jusque là.

```
package units;

public class AndUnit extends AbstractSingleUnit {

	public AndUnit(String name){ super(name);}

	@Override
	public int getRequiredInputNumber() {
		return 2;
	}

	public boolean callTransferFunction(){
		return
getInputUnits().get(0).getOutputSignal()&&getInputUnits().get(1).getOutputSignal
();
	}
}
package units;

public class OrUnit extends AbstractSingleUnit {

	public OrUnit(String name){ super(name);}
	@Override
	public boolean callTransferFunction(){
		return getInputUnits().get(0).getOutputSignal()||
getInputUnits().get(1).getOutputSignal();
```

```
	}
	@Override
	public int getRequiredInputNumber() {
		return 2;
	}
}
```

50 % pour chaque classe : 10 % déclaration, 10 % constructeur, 10 % getrequiredNumber(), 20 % callTransferFunction().

Attention à l'accès aux entrées, ils peuvent avoir ajouté un accesseur non demandé dans AbstractUnit

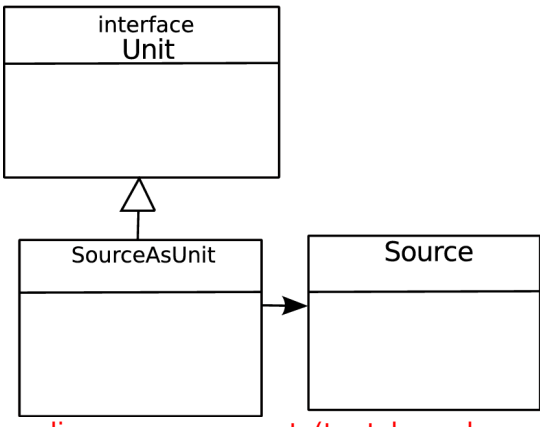
2 : Générateurs de signaux : les sources

Pour exciter les circuits logiques, on utilise des sources de signaux aléatoires. Une particularité des sources est qu'elles n'ont pas de signaux d'entrée. D'une façon générale, les sources sont définies par l'interface suivante :

```
package units;

public interface Source {
	// renvoie le nom symbolique de l'unité (fonction + numéro)
	public String getName();
	// calcule le prochain signal
	public void generateSignal();
	// renvoie le signal de sortie courant
	public boolean getOutputSignal();
}
```

On souhaite que les Sources puissent être manipulées comme des unités quelconques.

Q2.1 : A quel *design pattern* peut-on faire appel pour obtenir cette fonctionnalité ? Proposez l'instanciation décrite ci-dessous du pattern par un diagramme de classes dont vous ne ferez apparaître que les éléments publics.



50 % Adapter, 50 % pour un diagramme correct (tout branchement erroné = 0, pas mal de patterns « à l'envers » et des randomsource à la place de source (0 aussi))

Q2.2 : Définissez une classe *RandomSource* qui soit conforme à l'interface *Source* et qui émet des signaux booléens qui changent aléatoirement à chaque fois qu'on génère un nouveau signal. Le constructeur prend le nom de l'unité en paramètre. Vous pourrez faire appel à la classe *Random* qui dispose d'une méthode *public boolean nextBoolean()*.

```java
package units;

import java.util.Random;

public class RandomSource implements Source {
		private Random generator = new Random();
		private String name;
		private boolean signal;

	public RandomSource(String name){
		this.name = name;
	}
	@Override
	public boolean getOutputSignal() {
		return signal;
	}
	public void generateSignal() {
		signal = generator.nextBoolean();
	}
	@Override
	public String getName() {
		return name;
	}
}
```

Le generator peut être une variable locale. Question facile, être attentif aux petites bêtises.

Q2.3 *:* Proposez une classe *SourceAsUnit* pour réaliser le pattern conformément à la Q2.1.

```java
package units;

import java.util.Collections;
import java.util.List;

public class SourceAsUnit implements Unit {
	private Source source;
	private Unit output;

	public SourceAsUnit(Source s){
		source = s;
	}
	@Override
	public List<Unit> getInputUnits() {
		return Collections.emptyList();
	}
	@Override
	public Unit getOutputUnit() {
		return output;
	}

	@Override
	public boolean getOutputSignal() {
		return source.getOutputSignal();
	}
```

```
	@Override
	public void run()  throws InvalidConnectionException{
		source.generateSignal();
	}
	@Override
	public String getName() {
		return source.getName();
	}
	@Override
	public void plugTo(Unit destination) {
		destination.getInputUnits().add(this);
		output = destination;
	}
}
```

On peut aussi étendre AbstractUnit, ce qui évite de recopier plugTo().

La solution d'étendre AbstractSingleUnit est encore plus compacte : il suffit de coder getRequiredNumber() => return 0 et callTransferFunction().

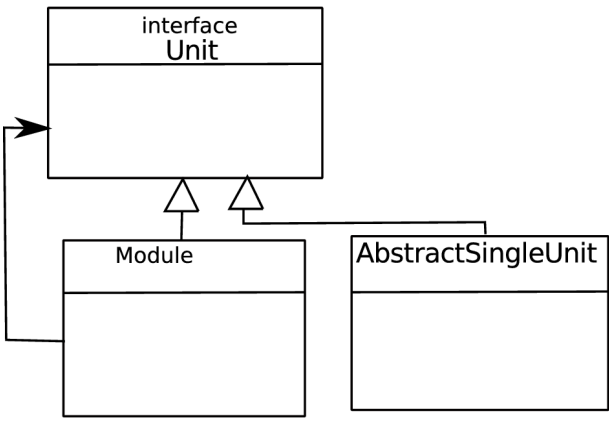
Question assez facile, avec beaucoup de solutions acceptables. Vérifier que toutes les méthodes requises y sont en fonction de la déclaration.

3 : Des modules constitués d'unités

On s'attaque à présent à la représentation des modules. Un module peut contenir des unités simples ou bien d'autres modules, ce qui permet de construire des hiérarchies.

Q3.1 : Quel *Design pattern* peut-on utiliser pour représenter des modules ? Faites un diagramme de classes présentant l'instanciation du pattern sur cet exemple ; vous préciserez la signature des opérations importantes dans le pattern.

Composite.



50 % pour identifier Composite, 50 % pour un dessin correct (des variantes possibles). On doit voir qu'un module implémente Unit et en contient une liste.

Le dessin leur a posé beaucoup de problèmes.

Q3.2 : Proposez une classe *Module* réalisée conformément au pattern choisi. Le constructeur prend en paramètres une liste d'unités supposées déjà branchées entre elles (et éventuellement à des unités externes au module) et une chaîne de caractères pour son nom.

Les entrées du module sont les unités non contenues dans le module, mais connectées à des unités du module.

Pour calculer le signal de sortie du module, on appelle séquentiellement le calcul de toutes les unités présentes dans le module (dans l'ordre où elles sont stockées) et on considère que le signal de sortie est produit par la dernière unité selon cet ordre.

```
package units;

import java.util.List;

public class Module extends AbstractUnit {
	private List<Unit> units;

	public Module(List<Unit> units, String name){
		super(name);
		this.units = units;
		setExternalInputs();
	}

	private void setExternalInputs() {
		for (Unit u : units){
			if (u.getInputUnits()!=null){
				for (Unit inp : u.getInputUnits()){
					if (!units.contains(inp)) inp.plugTo(this);
				}
			}
		}
	}

	public void run() throws InvalidConnectionException {
		for (Unit u : units){
				u.run();
			}
		setOutputSignal(units.get(units.size()-1).getOutputSignal());
		}
}
```

Code vraiment minimaliste.

Pas facile à corriger, on trouve des choses très variées.

Il doit y avoir au moins une bonne déclaration et un attribut `List<Unit>` (20%), un constructeur qui l'initialise (30%), et une méthode run qui fait ce qui est demandé (30%). Plus 20% pour setExternalInputs(). A adapter en fonction de vos copies.

4 : Générateur d'unités

Pour construire des unités, on fait appel à une classe *`UnitGenerator`* qui renvoie des unités du type souhaité. Cette classe attribue à chaque nouvelle unité un numéro unique croissant qui ne dépend pas du type de l'unité.

Q4.1 : Quel *Design pattern* reconnaissez-vous dans cette façon de créer des unités ?

Factory (0 ou 100%). La réponse exacte me semble être « factory method » mais on ne va pas chipoter... Par contre, je propose de rejeter Abstract Factory.

Q4.2 : Proposez la classe *UnitGenerator* et dotez-la des méthodes qui permettent de créer des sources aléatoires (vues comme des unités), des *AndUnit, des OrUnit,* et des modules. Pour ce dernier type d'unité, la méthode de création prendra en paramètres la liste d'unités contenue dans le module. Ce sont ces méthodes de création qui définissent le nom des unités, à savoir la chaîne de caractères « Random », « And », « Or » et « Module », suivi du numéro unique caractérisant l'unité.

```
package units;

import java.util.List;

public class UnitGenerator {
	public static int cpt = 0;

	public static Unit createOrUnit(){
		return new OrUnit("Or_" + cpt++);
	}
	public static Unit createRandomUnit(){
		return new SourceAsUnit(new RandomSource("Random_" + cpt++));
	}
	public static Unit createAndUnit(){
		return new AndUnit("And_" + cpt++);
	}
	public static Unit createModule(List<Unit> units){
		return new Module(units, "Module_" + cpt++);
	}
}
```

50 % pour la gestion du compteur (vu plusieurs fois en TD), 50 % pour les méthodes de création (on enlève des points s'il en manque, s'il y a des fautes, etc.)

5 : Parcours d'un chemin d'unités

On souhaite propager l'activité d'un circuit en partant d'une unité choisie au préalable et en parcourant les sorties successives à partir de cette unité (exclue) jusqu'à une unité ultime qui n'est connectée à rien. Pour parcourir le chemin, vous ferez appel au pattern *Iterator*.

Q5.1 : Proposez une classe *UnitIterator* qui implémente *Iterator<Unit>* et réalise le comportement défini ci-dessus. Pour rappel, l'interface *Iterator* impose de définir des méthodes **public boolean** hasNext() et **public** Object next(). Vous omettrez la méthode optionelle *remove()*. Le constructeur prend en paramètre l'unité d'où l'on souhaite démarrer.

```
package units;

import java.util.Iterator;

public class UnitIterator implements Iterator<Unit> {
		private Unit currentUnit;

	public UnitIterator(Unit u){
		currentUnit = u;
	}

	@Override
	public boolean hasNext() {
```

```
            return (currentUnit.getOutputUnit()!=null);
      }

      @Override
      public Unit next() {
            currentUnit = currentUnit.getOutputUnit();
            return currentUnit;
      }
}
```

20 % pour **implements** Iterator<Unit>, 20 % pour le constructeur, 20 % pour hasNext(), 40 % pour next() dont 20 % pour la mise à jour correcte de l'unité en cours.

6 : Gestionnaire de circuits

On construit à présent la classe *CircuitManager* qui permet d'effectuer des calculs le long du chemin défini à la question précédente en gérant le fait que des unités peuvent être incomplètement connectées.

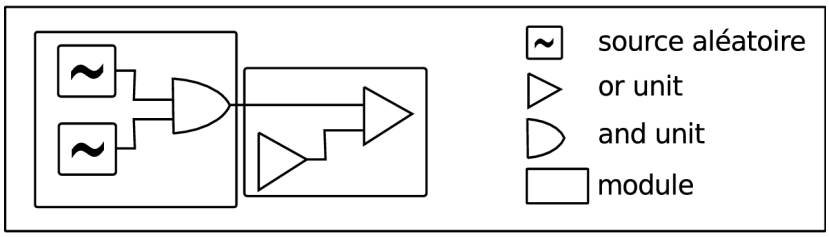
Q6.1 : Dans la classe *CircuitManager*, proposez une méthode **public boolean** checkPath(Unit u) qui utilise l'itérateur défini à la Q5.1 et propage le signal à partir de l'unité passée en paramètres. Elle appelle séquentiellement le signal de sortie de chaque unité rencontrée le long du chemin. Si elle rencontre une unité dont la connection n'est pas valide, elle s'arrête et renvoie *false*, sinon elle renvoie *true*.

```
package units;

public class CircuitManager {

      public boolean checkPath(Unit u){
            UnitIterator it = new UnitIterator(u);
            while(it.hasNext()){
                  Unit nextU = it.next();
                  try{
                        nextU.run();
                  }
                  catch(InvalidConnectionException e){
                        System.out.print("Chemin non valide");
                        return false;
                  }
            }
            return true;
      }
}
```

50 % pour l'utilisation correcte de l'itérateur, 50 % pour la gestion de l'exception.

7 : Testons l'ensemble

Q7.1 : Proposez une classe *Main* contenant la méthode principale pour tester tout le code que vous avez développé. Cette méthode utilise la classe *UnitGenerator* pour créer deux unités aléatoires, *une AndUnit, deux OrUnit*, et deux modules, connectés selon le schéma de la figure ci-dessous.



Une fois ces unités créées et connectées, la méthode vérifie le chemin partant de la première source aléatoire en utilisant la méthode de test *assert()*.

```
package units;

import java.util.ArrayList;
import java.util.List;

public class Main {

	public static void main(String args[]){
		List<Unit> units = new ArrayList<Unit>();

		Unit u1 = UnitGenerator.createRandomUnit();
		units.add(u1);
		Unit u2 =  UnitGenerator.createRandomUnit();
		units.add(u2);
		Unit u3 =  UnitGenerator.createAndUnit();
		units.add(u3);
		u1.plugTo(u3);
		u2.plugTo(u3);
		Unit m1 = UnitGenerator.createModule(units);

		List<Unit> units2 = new ArrayList<Unit>();
		Unit u4 =  UnitGenerator.createOrUnit();
		units.add(u4);
		Unit u5 =  UnitGenerator.createOrUnit();
		units2.add(u5);
		u4.plugTo(u5);
		Unit m2 = UnitGenerator.createModule(units2);

		u3.plugTo(u4);

		CircuitManager c = new CircuitManager();

		assert(c.checkPath(u1));
	}
}
```

70 % pour la création correcte de la structure, 30 % pour la vérification du chemin.