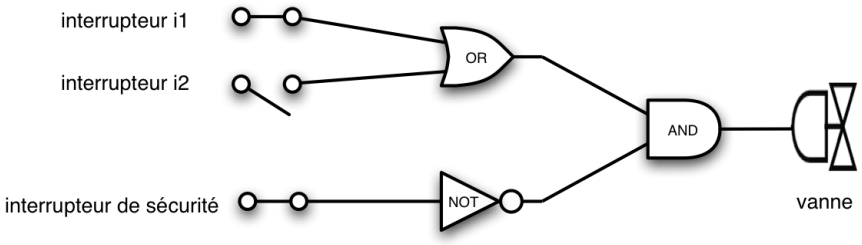
Bases de la conception orientée objet
Portes logiques et simulation de circuit

Steven Costiou
Stéphane Ducasse
Inria

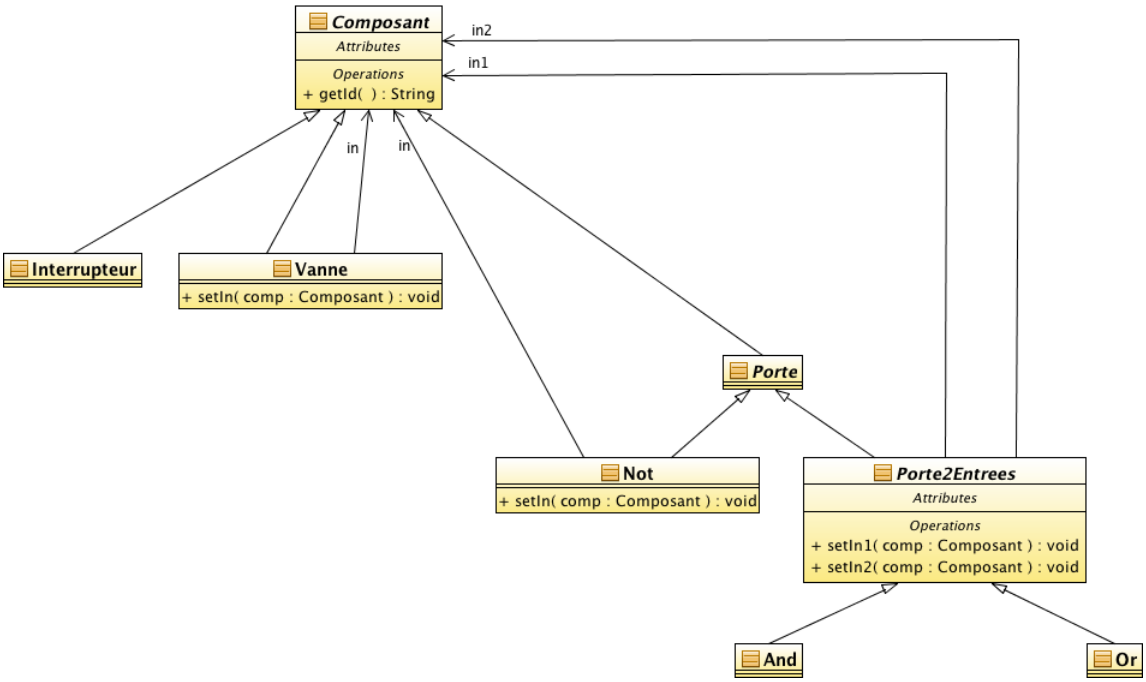
11 juin 2021

On considère des circuits constitués de portes logiques *and*, *or* ou *not* connectées en entrée sur des interrupteurs et en sortie sur des appareils. Par exemple :



Composants

La hiérarchie de classes suivante permet de représenter les types de composants :



— `Composant` est la classe racine de la hiérarchie. Elle est abstraite et fournit une méthode `getId()` qui permet d'identifier le composant.
— un `Interrupteur` n'a pas d'entrée
— une `Vanne` est connectée en entrée sur un composant (`in`)
— `Porte` est la classe abstraite racine des portes logiques
— un `Not` a une connection en entrée sur un composant (`in`)
— `Porte2Entrees` est abstraite et factorise les portes logiques à 2 entrées (And, Or, ...) connectées en entrée sur 2 composants (`in1` et `in2`)

1 Classes de portes

1. Développez le modèle de composants spécifié ci-dessus
2. Programmer en **TDD** une méthode `public String description()` dans les classes de composants qui fournit une chaîne de caractères formée de :
   — leur identifiant (donné par `getId()`)
   — pour les composants disposant d'entrée(s), les identifiants (`getId()`) des composants correspondants ou la chaîne de caractères **`"non connecte"`** si l'entrée n'est pas connectée.

   Par exemple pour un and (`And@48d6c16c`) non connecté en entrée 1, connecté en entrée 2 sur un not (`Not@5abb7465`) : `And@48d6c16c in1: non connecte in2: Not@5abb7465`

2 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 développées précédemment sont complétées comme suit :

```
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();
        }
        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();
        }
        return !in.getEtat();
  }
}
```

2.1 Évaluation des portes

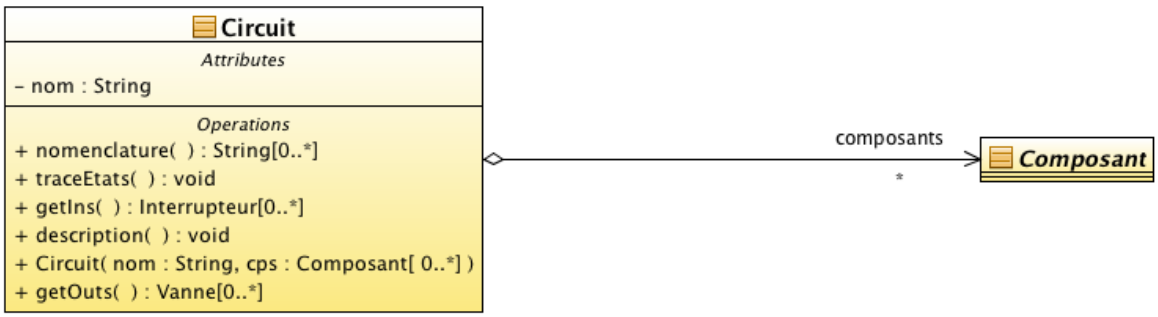
1. Complétez vos classes avec le code ci-dessus et testez ce code,
2. testez et programmez de façon similaire la méthode `getEtat()` des `And` et `Or`.

2.2 Test d'un circuit

1. Créez une classe `TestCircuits` contenant une méthode `main`. Instanciez et connectez des composants pour construire le circuit du début du TP. Vous stockerez les composants dans un tableau après leur instantiation.
2. Dans la classe `TestCircuits` programmez 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.`
3. Si l'erreur **''`unreported exception NonConnecteException; must be caught or declared to be thrown`''** apparaît, c'est que vous n'avez pas tenu compte de cette exception. Capturez cette erreur dans un `try/catch` pour gérer cette erreur. le programme ne doit pas s'arrêter sur le premier composant non connecté mais poursuivre son exécution et passer sur tous les composants.

3 Modèle de circuit

Jusqu'à maintenant, les circuits n'ont pas d'existence propre et sont manipulés comme des tableaux de composants dans une classe de test. Nous allons modéliser les circuits sous forme d'objets définis par la classe Circuit :



3.1 La classe Circuit

Programmer la classe Circuit suivant les spécifications ci-dessous :

- Un Circuit est composé de Composant au travers du rôle composants. Les composants seront stockés dans une collection, que vous manipulerez via l'interface java.util.List<E>.
- Le constructeur public Circuit(String nom, Composant[] cps) permet de construire un circuit en founissant son nom (String) et un tableau cps de composants connectés (formé comme précédemment.) :
  - ajouter les composants de cps dans sa liste composants de la façon suivante (voir la javadoc) : composants.addAll(Arrays.asList(cps))
  - trier les composants par ids en utilisant Collections.sort(composants). Pour cela il est nécessaire que la classe Composant implémente l'interface Comparable<Composant> pour fournir un ordre (méthode compareTo(Composant)) sur leur id.
- La méthode public List<String> nomenclature() renvoie la liste des ids de ses composants (triée parce que les composants le sont).
- La méthode public void description() affiche le nom du circuit et la description de ses composants.
- La méthode public void traceEtats() affiche le nom du circuit et trace l'état de ses composants.
- La méthode public List<Interrupteur> getInputs() renvoie la liste des interrupteurs :
  - créer une liste interrupteurs qui sera retournée par getInputs()
  - trouvez un moyen, en parcourant la liste des composants du circuit, de stocker uniquement les interrupteurs dans cette liste :
    - sans utiliser de test if sur le type pour chaque composant,
    - sans utiliser la méthode instanceof.
  - de manière similaire la méthode getOutputs() renvoie la liste des vannes, stockées dans une liste vannes lors d'un parcours de la liste des composants.

3.2 Instanciation et test du modèle de circuit

Dans la classe TestCircuits :

- Instancier le circuit en lui fournissant un nom et le tableau de composants connectés précédemment créé.
- Appliquer sur ce circuit la méthode suivante : static void test(Circuit circ) qui appelle sur circ progressivement ses méthodes :
  - nomenclature() (afficher la liste résultat)
  - description()
  - getInputs() et getOutputs() (afficher leurs ids)
  - forcer certains interrupteurs (récupérés par getInputs()) et afficher l'état des vannes ou toute la trace par circ.traceEtats().

4 Modélisation des signaux logiques

Nous allons maintenant simuler des signaux logiques en entrée de notre circuit pour démarrer la vanne. Pour cela, nous allons modéliser deux types de signaux : SignalHaut et SignalBas. Les instances de la classe SignalHaut correspondent à la valeur logique 1 (booléen true). Les instances de la classe SignalBas correspondent à la valeur logique 0 (booléen false).

Les objets représentant les signaux se propagent au travers du circuit par retro-évaluation. Pour décider de son statut, l'objet en fin de circuit (la vanne) évalue son état en évaluant ses composants en entrée. Chacun de ces composants évaluent alors leurs propres composants en entrée, et ainsi de suite jusqu'à atteindre les interrupteurs. Lors de leur évaluation, les interrupteurs renvoient alors des objets *signaux* qui se propagent en chemin inverse par retour de composant jusqu'à la vanne.

Lors de leur passage par un composant, les signaux peuvent être modifiés en fonction de l'opérateur logique appliqué par le composant. Par exemple, lorsqu'un composant Not récupère une instance de SignalHaut, le composant renvoit une instance de SignalBas. Désormais, nous allons manipuler des objets qui représentent les signaux logiques. Nous allons donc leur déléguer l'application des opérations logiques : ce sont les objets qui savent comment ils doivent être transformés par une opération logique.

4.1 Les classes SignalHaut et SignalBas

Les classes de signaux logiques sont indépendantes du projet de modélisation de circuit. Elles doivent être utilisables comme une librairie dans n'importe quel autre projet. Pour parler des classes de signaux de manière générique, nous faisons référence dans le texte à une classe SignalLogique.

Les objets signaux savent répondre au protocole suivant :

- **value()** : renvoie la valeur logique du signal (true ou false).
- **not()** : renvoie le signal "inverse".
  Exemple : new SignalHaut().not() → renvoie une instance de *signal bas*.
- **and(SignalLogique s)** : opérateur logique "*and*".
  Exemple : new SignalHaut().and(new SignalBas()) → renvoie une instance de *signal bas*.
- **or(SignalLogique s)** : opérateur logique "*or*".
  Exemple : new SignalHaut().or(new SignalBas()) → renvoie une instance de *signal haut*.
- **toString()** : renvoie une chaine de caractère représentant l'état du signal.

Comme le modèle de signaux logiques est indépendant du reste, vous fournirez une classe séparée avec une méthode main dans laquelle vous testerez les deux types de signaux SignalHaut et SignalBas. Vous instancierez des signaux et vous vous assurerez que les méthodes implémentées renvoient un résultat correct. Pour cela vous imprimerez le résultat de chaque opération sur la sortie standard.

4.2 Transit des signaux par les différents composants

Les composants doivent maintenant pouvoir évaluer un signal d'entrée et renvoyer un signal en sortie. Pour cela, on donne l'interface Evaluable :

```
public interface Evaluable {
   public SignalLogique evaluate();
}
```

Les objets evaluables implémentent tous une méthode evaluate() : les instances de Circuit et de Composant. Cette méthode suit les spécification suivantes :

La méthode evaluate() des circuits considère la liste des composants en sortie, leur envoie le message evaluate() puis traite le résultat.

La méthode evaluate() des interrupteurs renvoie un objet signal. Modifiez la variable d'instance etat de la classe Interrupteur pour contenir un object signal au lieu d'un booléen. La méthode evaluate() d'un interrupteur retourne simplement l'état de l'interrupteur.

Pour tout autre composant, la méthode evaluate() :

- récupère le résultat de l'évaluation de ses composants en entrée (un ou plusieurs signaux),
- applique une opération logique sur ce résultat,
- renvoie le signal produit par l'opération.

Les opérations logiques sont appliquées en appelant les méthodes présentes dans le protocole des objets signaux. Par exemple, prenons un composant Not et sa méthode evaluate(). L'opération exécutée par cette methode sera return in.evaluate().not().

Note importante : le remplacement du type de la variable d'instance etat de Interrupteur d'un booléen vers un signal logique impacte le fonctionnement du code existant reposant sur cette variable. Vous veillerez à ce que votre architecture repose sur cette nouvelle conception et adapterez judicieusement le code impacté dans la hiérarchie de classes de Composant.

4.3 Évaluation du circuit avec des signaux

Nous allons maintenant jouer avec notre circuit en donnant des signaux en entrée à nos interrupteurs, puis en essayant de démarrer le circuit. Vous devez :

1. Parcourir votre liste d'interrupteurs,
2. pour chaque interrupteur, demander à l'utilisateur d'entrer une valeur correspondant à un signal logique (1 ou 0), et instancier dans la variable d'instance de l'interrupteur l'objet signal correspondant,
3. évaluer votre circuit et imprimer le résultat.

Vous implémenterez cela dans une méthode statique de votre classe d'exemple où se situe la méthode main, puis l'appellerez à partir de main.

Pour connaître l'état final de votre circuit, appelez la méthode value() sur le signal obtenu lors de l'évaluation du circuit. Un résultat égal à true correspond à une circuit *allumé*, *éteint* sinon.