POLYTECH LILLE

Université de Lille

Conception & modélisation objet

Compte-rendu de projet

Projet CMO

SE2A4

Kevin Doolaeghe

Sommaire

1. Introduction

L'objectif de ce projet est de pouvoir modéliser des circuits logiques à entrées et sorties en Java par l'intermédiaire de la conception et de la modélisation objet.

2. Classes de portes

Dans un premier temps, on se propose de réaliser la hiérarchie de classes ci-dessous qui permet de représenter les différents types de composants d'un circuit.

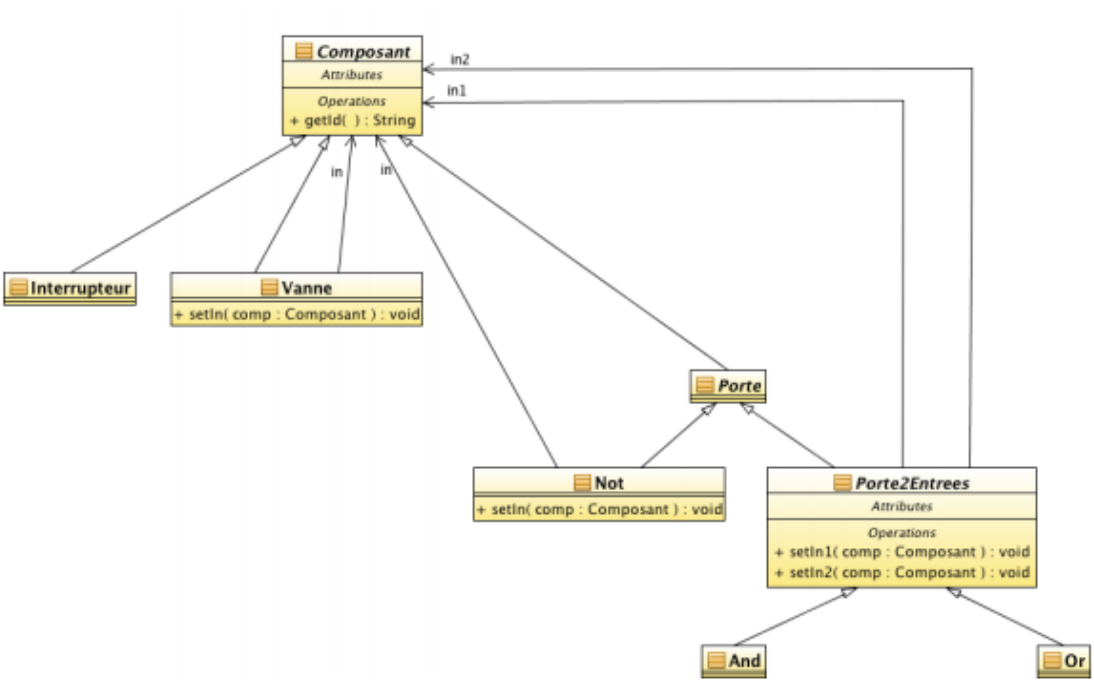

Figure n°1 : Hiérarchie des classes de composants.

La classe `Composant` est abstraite et est à la base de la hiérarchie. Elle permet d'identifier le composant à partir de sa méthode `getId()`. Cette méthode renvoie une représentation sous forme de `String` de l'objet en récupérant le nom de la classe de l'objet et son identifiant unique. Ces informations sont obtenues à l'aide des méthodes de la super-classe `java.lang.Object`.

```
public String getId() {
   return getClass().getSimpleName() + "@" + this.hashCode();
}
```

Les classes `Interrupteur` et `Vanne` sont des composants correspondant respectivement aux entrées et aux sorties des circuits. Une `Vanne` possède une entrée `in` tandis qu'un `Interrupteur` n'en a pas. Cette entrée est de type `Composant` car elle peut être n'importe quel composant.

Les portes logiques sont des composants pouvant avoir une ou deux entrées. Ainsi, on ajoute les classes `Porte` et `Porte2Entrees` désignant respectivement les portes à une

entrée (*not*) et à deux entrées (*and* et *or*). La classe `Porte2Entrees` est une `Porte` qui elle-même est un `Composant`. Ainsi, la classe `Porte` est abstraite et définit toutes les portes logiques. La classe `Porte2Entrees` est également abstraite et possède deux variables d'instance `in1` et `in2` de la classe Composant et leurs accesseurs correspondant aux deux entrées de la porte.

Dans un second temps, on réalise en programmation TDD (*Test-Driven Development* ou développement piloté par les tests) l'implémentation d'une méthode `public String description()` dans les classes de composants. Cette méthode doit permettre de fournir une chaîne de caractères formée de :

- ➔ l'identifiant du composant
- ➔ pour les composants disposant d'entrée(s), l'identifiant des composants connectés ou le message "*non connecte*" si l'entrée n'est pas connectée

Pour cela, on commence par créer la classe `DescriptionTest` avec ses méthodes de tests unitaires puis on ajoute une méthode abstraite `description()` à la classe `Composant` qui sera ensuite redéfinie dans ses sous-classes. Par exemple, un Interrupteur n'a pas d'entrée, sa méthode `description()` doit donc juste retourner son identifiant :

```
@Override
public String description() { return getId(); }
```

D'un autre côté, les classes `And` et `Or` n'ont pas à redéfinir cette méthode. En effet, c'est la classe `Porte2Entrees` à partir de laquelle elles héritent qui va redéfinir la méthode de la façon suivante :

```
@Override
public String description() {
   String in1 = getIn1() == null ? "non connecte" :
getIn1().getId();
   String in2 = getIn2() == null ? "non connecte" :
getIn2().getId();
   return getId() + " in1: " + in1 + " in2: " + in2;
}
```

Le test unitaire réalisé est effectué sur une porte *and* qui est connectée à une porte *not* sur son entrée n°2 et non connectée sur son entrée n°1 :

```
Not not = new Not();
And and = new And();
and.setIn2(not);
```

La vérification du résultat est effectuée avec une assertion. Après lancement, le test se déroule correctement et renvoie la chaîne attendue. Tant que le test unitaire sera correct, la méthode fonctionnera comme selon le cahier des charges.

3. Simulation

On souhaite désormais simuler le circuit suivant à partir des classes de composants créées précédemment.

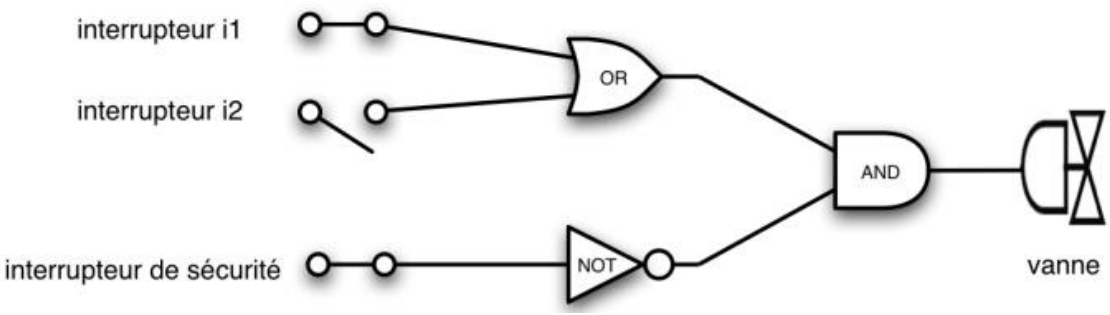

Figure n°2 : Circuit logique simulé.

Pour simuler le circuit, il faut pouvoir connaître l'état logique de chaque composant. On introduit donc la méthode abstraite `boolean getEtat()` au sein de la classe `Composant`. Cette méthode est à redéfinir dans les classes `Interrupteur`, `Vanne`, `Not`, `Or` et `And` pour récupérer l'état actuel de chaque type de composant. En effet, en fonction du nombre d'entrées, chaque composant peut répondre au message `getEtat()` à sa manière grâce au polymorphisme. Par exemple, les portes `Not` et `And` répondent respectivement à ce message des façons suivantes :

➔ Classe `Not` :

```
@Override
public boolean getEtat() throws NonConnecteException {
    try {
        return !in.getEtat();
    } catch (NullPointerException e) {
        throw new NonConnecteException(description(), e);
    }
}
```

➔ Classe `And` (l'opération est analogue pour la classe `Or` mis à part le ET logique "`&&`" qui est remplacé par un OU logique "`||`") :

```
@Override
public boolean getEtat() throws NonConnecteException {
    try {
        return getIn1().getEtat() && getIn2().getEtat();
    } catch (NullPointerException e) {
        throw new NonConnecteException(description(), e);
    }
}
```

Si le composant n'est pas correctement connecté, la méthode provoque et propage alors une exception `NonConnecteException`. Cette classe hérite de la classe `Exception` et possède un constructeur pour propager un message et une cause à l'exception survenue. La déclaration de cette classe est la suivante :

```
public class NonConnecteException extends Exception {
    public NonConnecteException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Un `Interrupteur` peut ne possède pas d'entrée mais peut être positionné à *true* ou *false* par ses méthodes respectives `on()` et `off()`.

Maintenant que les composants peuvent retourner l'état logique de leur sortie en fonction de leur(s) entrée(s), on crée la classe `TestCircuit` contenant une méthode `main()` dans laquelle on instancie tous les composants nécessaires à la réalisation du montage précédent. Le câblage du circuit est réalisé à l'aide des constructeurs de chaque classe permettant de directement spécifier les entrées de chaque composant si celui-ci en possède. Enfin, on stocke tous les composants dans la variable locale `List<Composant> composants`.

```
public static void main(String[] args) {
    Interrupteur i1 = new Interrupteur();
    Interrupteur i2 = new Interrupteur();
    Interrupteur is = new Interrupteur();
    Or or = new Or(i1, i2);
    Not not = new Not(is);
    And and = new And(or, not);
    Vanne vanne = new Vanne(and);

    i1.on();
    i2.off();
    is.on();

    List<Composant> composants = List.of(i1, i2, is, or, not, and,
vanne);
}
```

On programme ensuite une méthode statique `traceEtats()` qui reçoit en paramètre un tableau de composants pour afficher leur description et leur état. La méthode `getEtat()` des composants peut renvoyer une exception `NonConnecteException` qui doit être capturée à l'aide d'un bloc *try/catch*. La déclaration de cette fonction est donnée ci-dessous :

```
public static void traceEtats(List<Composant> composants) {
    composants.forEach(composant -> {
        System.out.print("\t- " + composant.description() + " ->
");
        try {
            System.out.println(composant.getEtat());
        } catch (NonConnecteException e) {
            System.out.println("non connecte");
        }
    });
}
```

On constate que le bloc *try/catch* est situé à l'intérieur du bouclage sur chaque élément du tableau afin de pouvoir continuer sur l'élément suivant en cas d'exception. Avant de lancer le programme, il reste à appeler la méthode `traceEtats()` dans la méthode `main()` avec pour argument le tableau de composants. Le programme affiche le résultat suivant sur la console au lancement de la méthode `main()` :

```
- Interrupteur@1313922862 -> true
- Interrupteur@1791741888 -> false
- Interrupteur@1595428806 -> true
- Or@1072408673 in1: Interrupteur@1313922862 in2: Interrupteur@1791741888 -> true
- Not@1531448569 in: Interrupteur@1595428806 -> false
- And@1867083167 in1: Or@1072408673 in2: Not@1531448569 -> false
- Vanne@1915910607 in: And@1867083167 -> false
```

Figure n°3 : Résultat obtenu sur la console.

4. Modèle de circuit

On souhaite désormais programmer une classe `Circuit` qui est composée d'une collection de composants manipulés avec l'interface `java.util.List<E>`. Le diagramme UML représentant cette hiérarchie de classe est le suivant :

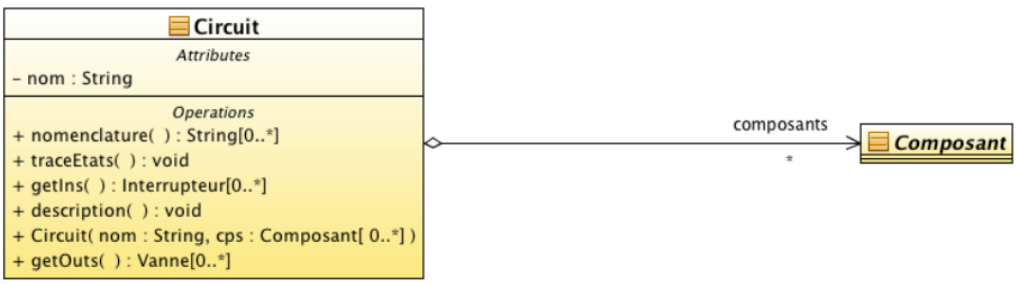

Figure n°4 : Représentation de la classe Circuit au sein de la hiérarchie.

On souhaite que la liste des composants soit triée par leur identifiant. Pour cela, la classe `Composant` implémente l'interface `Comparable<Composant>` et redéfinit la méthode `compareTo(Object o)` pour retourner l'identifiant unique de l'objet :

```
@Override
public int compareTo(Object o) { return this.hashCode(); }
```

Ensuite, on définit le constructeur `public Circuit(String nom, Composant[] cps)` permettant de construire un circuit dont le nom est `nom` et dont la liste de composant est `cps`. La classe `Circuit` dispose des variables d'instance privées `nom` et `composants` auxquelles son constructeur va attribuer ses paramètres. Voici l'implémentation de ce code :

```
public Circuit(String nom, Composant[] cps) {
   this.nom = nom;
   this.composants = new ArrayList<>();
   this.composants.addAll(Arrays.asList(cps));
   Collections.sort(this.composants);
}
```

Par l'intermédiaire de ce constructeur, un circuit peut donc bien être créé et posséder un nom et une liste de composants triés.

Ensuite, la classe Circuit doit disposer des méthodes :

- ➔ `public List<String> nomenclature()` : renvoie la liste des identifiants triée de ses composants.
- ➔ `public void description()` : affiche le nom du circuit et la description de chaque composant.
- ➔ `public void traceEtats()` : affiche le nom du circuit et trace l'état de chaque composant.
- ➔ `public List<Interrupteur> getInputs()` : renvoie la liste des interrupteurs.
- ➔ `public List<Vanne> getOutputs()` : renvoie la liste des vannes.

Pour filtrer les données de la liste de composants, on utilise une classe supplémentaire `CircuitFilter`. Cette classe reçoit un `Circuit` à son instanciation et permet de filtrer les composants pour ne récupérer que les entrées, les sorties ou encore la liste des identifiants. Pour cela, on utilise l'API *Stream* de Java 8 et les fonctions fléchées sur les collections :

```
public List<String> filtrerId() {
   return circuit
      .getComposants()
      .stream()
      .map(c -> c.getId())
      .collect(Collectors.toList());
}

public List<Interrupteur> filtrerEntrees() {
   Predicate<Composant> filtre = c -> c.isEntree();
   return circuit
```

```
        .getComposants()
        .stream()
        .filter(filtre)
        .map(c -> (Interrupteur) c)
        .collect(Collectors.toList());
}

public List<Vanne> filtrerSorties() {
    Predicate<Composant> filtre = c -> c.isSortie();
    return circuit
        .getComposants()
        .stream()
        .filter(filtre)
        .map(c -> (Vanne) c)
        .collect(Collectors.toList());
}
```

Les méthodes `isEntree()` et `isSortie()` ont été ajoutées à la classe abstraite `Composant` et sont redéfinies dans chaque type de composant pour retourner si celui-ci est respectivement une entrée et une sortie. Chaque méthode renvoie une valeur booléenne *true* ou *false*. Les interrupteurs retournent donc *true* à la méthode `isEntree()` tandis que les vannes retournent *true* à `isSortie()`.

Ainsi, les méthodes `nomenclature()`, `getInputs()` et `getOutputs()` de la classe `Circuit` deviennent simplement :

```
public List<String> nomenclature() {
    return new CircuitFilter(this).filtrerId();
}

public List<Interrupteur> getInputs() {
    return new CircuitFilter(this).filtrerEntrees();
}

public List<Vanne> getOutputs() {
    return new CircuitFilter(this).filtrerSorties();
}
```

Les méthodes `traceEtats()` et `description()` doivent toutes les deux afficher le nom du circuit, on crée donc une nouvelle méthode `afficherNom()` qui affiche le nom du circuit. Excepté l'affichage du nom du circuit, la méthode `traceEtats()` est la même que celle créée précédemment dans la classe `TestCircuit`. Enfin, la méthode `description()` affiche la description de chaque composant. Pour cela, on boucle simplement sur la liste de composant et on appelle la méthode `description()` de chaque composant pour afficher le résultat sur la sortie standard.

```
public void afficherNom() {
    System.out.println(this.nom + ":");
}

public void description() {
```

```
    afficherNom();
    composants.forEach(c -> System.out.println("\t-" +
c.description()));
}

public void traceEtats() {
    afficherNom();
    for (Composant composant : composants) {
        System.out.print("\t- " + composant.description() + " ->
");
        try {
            System.out.println(composant.getEtat());
        } catch (NonConnecteException e) {
            System.out.println("non connecte");
        }
    };
}
```

Enfin, pour tester le bon fonctionnement de la classe `Circuit`, on ajoute à la classe `TestCircuit` la méthode `static void test(Circuit circ)` qui appelle progressivement sur `circ` les méthodes :

- ➔ `nomenclature()` (affiche la liste obtenue)
- ➔ `description()`
- ➔ `getInputs()` et `getOutputs()` (affiche l'identifiant de chaque entrée/sortie)
- ➔ `traceEtats()` (après avoir forcé certains interrupteurs)

Le résultat obtenu est le suivant :

```
monCircuit:
    -Interrupteur@883049899
    -Interrupteur@1791741888
    -Interrupteur@1595428806
    -Or@1072408673 in1: Interrupteur@883049899 in2: Interrupteur@1791741888
    -Not@1531448569 in: Interrupteur@1595428806
    -And@1867083167 in1: Or@1072408673 in2: Not@1531448569
    -Vanne@1915910607 in: And@1867083167
ID list:
    - Interrupteur@883049899
    - Interrupteur@1791741888
    - Interrupteur@1595428806
    - Or@1072408673
    - Not@1531448569
    - And@1867083167
    - Vanne@1915910607
Inputs:
    - Interrupteur@883049899
    - Interrupteur@1791741888
    - Interrupteur@1595428806
Outputs:
    - Vanne@1915910607
monCircuit:
    - Interrupteur@883049899 -> true
    - Interrupteur@1791741888 -> false
    - Interrupteur@1595428806 -> true
    - Or@1072408673 in1: Interrupteur@883049899 in2: Interrupteur@1791741888 -> true
    - Not@1531448569 in: Interrupteur@1595428806 -> false
    - And@1867083167 in1: Or@1072408673 in2: Not@1531448569 -> false
    - Vanne@1915910607 in: And@1867083167 -> false
```

Figure n°5 : Résultat obtenu après appel des différentes méthodes de la classe Circuit.

5. Modélisation des signaux logiques

Pour finir, on choisit de simuler des signaux logiques au sein d'un `Circuit`. Pour cela, on modélise les deux types de signaux `SignalHaut` et `SignalBas` correspondant respectivement aux valeurs logiques 1 et 0. Ces objets se propagent au travers du circuit par rétro-évaluation.

Dans un premier temps, on modélise la classe abstraite `SignalLogique` dont les classes `SignalHaut` et `SignalBas` vont hériter. Cette classe décrit les méthodes suivantes :

- ➔ `value()` : renvoie la valeur booléenne du signal.
- ➔ `not()` : renvoie une instance de `SignalLogique` inverse.
- ➔ `and(SignalLogique s)` : renvoie une instance de `SignalLogique` adéquate au résultat du ET logique réalisé entre les deux signaux.
- ➔ `or(SignalLogique s)` : renvoie une instance de `SignalLogique` adéquate au résultat du OU logique réalisé entre les deux signaux.
- ➔ `toString()` : renvoie une chaîne de caractère représentant l'état du signal.

Les méthodes `and()` et `or()` peuvent être définies dans la classe `SignalLogique` car elles sont les mêmes pour les classes `SignalHaut` et `SignalBas`. Les autres méthodes sont abstraites car elles dépendent du type de signal. On obtient donc la classe abstraite `SignalLogique` suivante :

```
public abstract class SignalLogique {
    public abstract boolean value();

    public abstract SignalLogique not();

    public SignalLogique and(SignalLogique s) {
        if (value() && s.value())
            return new SignalHaut();
        else
            return new SignalBas();
    }

    public SignalLogique or(SignalLogique s) {
        if (value() || s.value())
            return new SignalHaut();
        else
            return new SignalBas();
    }

    public abstract String toString();
}
```

Les classes `SignalHaut` et `SignalBas` héritent de la classe `SignalLogique` et implémentent les différentes méthodes précédentes. On dispose finalement des classes suivantes :

```java
public class SignalHaut extends SignalLogique {
    @Override
    public boolean value() {
        return true;
    }

    @Override
    public SignalLogique not() {
        return new SignalBas();
    }

    @Override
    public String toString() {
        return "True";
    }
}
```

```java
public class SignalBas extends SignalLogique {
    @Override
    public boolean value() {
        return false;
    }

    @Override
    public SignalLogique not() {
        return new SignalHaut();
    }

    @Override
    public String toString() {
        return "False";
    }
}
```

Pour évaluer le transit des signaux au travers des composants, on utilise l'interface `Evaluable` dont la définition est la suivante :

```java
public interface Evaluable {
    public SignalLogique evaluate() throws NonConnecteException;
}
```

```java
public abstract class Composant implements Comparable, Evaluable {
    //...
}
```

Cette interface est implémentée à la classe `Composant` et la méthode `evaluate()` qu'elle déclare sera redéfinie dans les sous-classes de `Composant`. Cette méthode peut propager une `NonConnecteException` dans le cas où un composant n'est pas correctement câblé.

Ainsi, la redéfinition de la méthode `evaluate()` dépend du composant. En effet, la porte *not* doit renvoyer une instance de `SignalHaut` si son entrée est une instance de `SignalBas` et inversement. Pour cela, on utilise la méthode `not()` du `SignalLogique` obtenue en entrée. De la même façon, les portes *and* et *or* doivent réaliser les opérations `and()` et `or()` respectivement. Enfin, la classe `Vanne` retourne simplement le signal obtenu par l'appel `in.evaluate()`.

➔ Classe `Not` :

```
@Override
public SignalLogique evaluate() throws NonConnecteException {
    try {
        return in.evaluate().not();
    } catch (NullPointerException e) {
        throw new NonConnecteException("non connecte", e);
    }
}
```

➔ Classe `And` (l'opération est analogue pour la classe `Or` mis à part le ET logique "`and()`" qui est remplacé par un OU logique "`or()`") :

```
@Override
public SignalLogique evaluate() throws NonConnecteException {
    try {
        return getIn1().evaluate().and(getIn2().evaluate());
    } catch (NullPointerException e) {
        throw new NonConnecteException("non connecte", e);
    }
}
```

La gestion des signaux par la classe `Interrupteur` était jusqu'à présent effectuée avec des booléens du type primitif *boolean*. On change donc le type de l'ancienne variable d'instance `etat` par la classe `SignalLogique`. Les méthodes `on()` et `off()` instancient respectivement un `SignalHaut` et un `SignalBas` sur `etat`. La méthode `getEtat()` doit à présent retourner `etat.value()` tandis que la méthode `evaluate()` se contente de retourner `etat`.

```
public class Interrupteur extends Composant {
    private SignalLogique etat;

    public Interrupteur() { off(); }

    public void on() { etat = new SignalHaut(); }

    public void off() { etat = new SignalBas(); }

    //...

    @Override
    public boolean getEtat() { return etat.value(); }

    @Override
    public SignalLogique evaluate() {
        return etat;
    }
}
```

Pour évaluer le circuit avec les signaux logiques, on ajoute la méthode statique `traitementSignaux()` dans la classe `TestCircuit`.

```
public static void traitementSignaux(Circuit c) {
    SignalCreator selector = new SignalCreator();
    c.getInputs().forEach(i -> {
        if (selector.saisieSignal(i.getId()).value()) i.on();
        else i.off();
    });

    System.out.print("Le circuit " + c.getNom() + " est ");
    try {
        System.out.println(c.evaluate().value() ? "allumé." :
"éteint.");
    } catch (NonConnecteException e) {
        System.out.println(e.getMessage());
    }
}
```

Cette méthode parcourt la liste des interrupteurs du circuit. Pour chaque interrupteur, la saisie de l'état du signal correspondant est demandée à l'aide de la classe `SignalCreator`. Après avoir parcouru la liste, le circuit est évalué et l'état du circuit est imprimé.

La classe `SignalCreator` permet de réaliser la saisie de l'état d'un interrupteur. Pour cela, elle utilise une instance de la classe `CommandLineInterface` qui permet de réaliser la saisie d'une chaîne de caractères :

```
public class CommandLineInterface {
    private Scanner scanner;

    public CommandLineInterface() {
        scanner = new Scanner(System.in);
    }

    public String scanCommand() {
        return scanner.nextLine();
    }
}
```

La classe `SignalCreator` possède une méthode `saisieSignal()` qui tente de lire un état saisie au clavier et recommence tant que l'exception `SignalBuilderException` est renvoyée. La classe `SignalBuilder` tente de construire un `SignalLogique` à partir d'une chaîne de caractères et propage une `SignalBuilderException` en cas d'échec.

```
public class SignalCreator {
    private CommandLineInterface cli;

    public SignalCreator() {
        cli = new CommandLineInterface();
    }
```

```
    public SignalLogique saisieSignal(String composant) {
        SignalBuilder builder = new SignalBuilder();
        try {
            System.out.print("Enter state for switch " + composant
+ ": ");
            return builder.build(cli.scanCommand());
        } catch (SignalBuilderException e) {
            System.out.println(e.getMessage());
            return saisieSignal(composant);
        }
    }
}
```

```
public class SignalBuilder {
    public SignalLogique build(String data) throws
SignalBuilderException {
        try {
            int state = Integer.parseInt(data);
            switch (state) {
                case 1:
                    return new SignalHaut();
                case 0:
                    return new SignalBas();
                default:
                    throw new SignalBuilderException("Data input
must be either 0 or 1");
            }
        } catch (NumberFormatException e) {
            throw new SignalBuilderException("Data input must be
integer", e);
        }
    }
}
```

Finalement, après le lancement du programme, l'utilisateur est invité à saisir l'état de chaque interrupteur du circuit. S'il se trompe, il lui est demandé de recommencer avec une indication. Le programme continue bien de fonctionner malgré les erreurs et affiche le résultat final.

```
----------------------------------------------------------------
Enter state for switch Interrupteur@1313922862: toto
Data input must be integer
Enter state for switch Interrupteur@1313922862: 1.5
Data input must be integer
Enter state for switch Interrupteur@1313922862: 3
Data input must be either 0 or 1
Enter state for switch Interrupteur@1313922862: 1
Enter state for switch Interrupteur@1791741888: 0
Enter state for switch Interrupteur@1595428806: 0
Le circuit monCircuit est allumé.
----------------------------------------------------------------
```

Figure n°6 : Résultat obtenu après appel de la méthode test() de la classe TestCircuit.

6. Conclusion

Le projet réalisé est fonctionnel et répond aux attentes. De plus, même si l'on déconnecte un composant du circuit, le programme continue de fonctionner correctement comme l'illustre l'exemple ci-dessous qui reprend le circuit précédent pour lequel la porte *not* a été déconnectée :

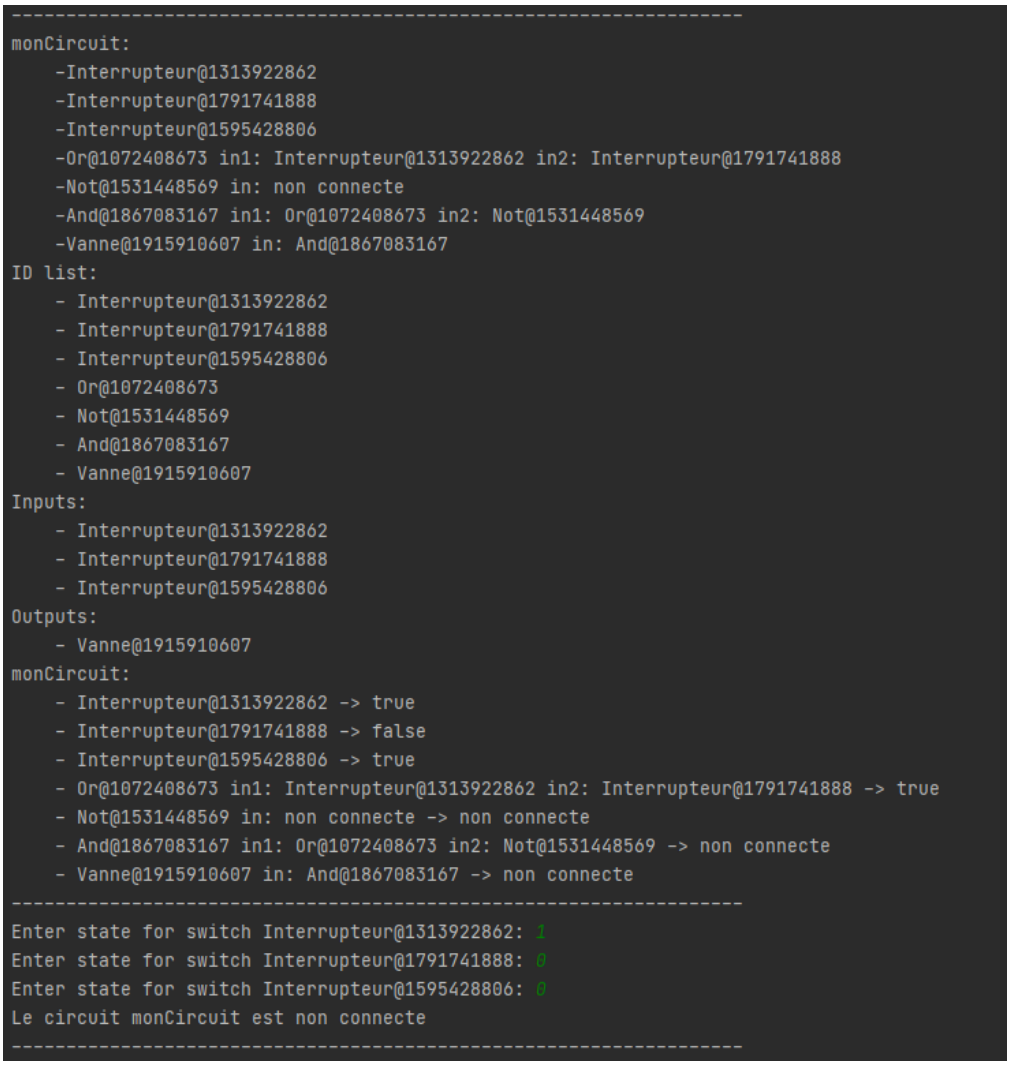

Figure n°7 : Résultat obtenu avec la porte Not non connectée.

Il est important d'essayer de concevoir un programme orienté objet en prenant en compte la potentielle évolution du produit. Il faut également tenter de séparer au mieux les différentes tâches pouvant être effectuées.