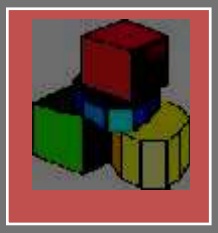
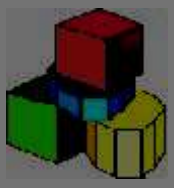


# Boole

## Le lego de la logique

Simulateur de circuit logique





# 1 Objectif

Le but de ce travail est de réaliser une modélisation objet en utilisant le langage Java fournissant l'environnement **Boole** décrit dans la suite de ce document. Vous prendrez soin de concevoir un programme structuré mettant en évidence les concepts majeurs. Vous aurez à l'esprit lors de son élaboration que votre programme doit pouvoir : Evoluer dans le temps, admettre de nouvelles spécifications venant naturellement compléter votre réalisation, accepter des optimisations permettant une plus grande performance de votre environnement, etc.

Ce travail permet aussi de revoir les principes de base de construction et d'évaluation qui sont sous-jacents à toute machine d'exécution.

# 2 Présentation générale de l'environnement Boole

On souhaite réaliser un système de simulation de circuits logiques constitués d'opérateurs qui sont des composants élémentaires ou composites permettant des assemblages ayant des fonctions particulières dans un contexte sans barrière temporelle de telle sorte que le calcul sera une composition fonctionnelle pure.

Ces composants que l'on qualifie souvent de "portes" peuvent soit transformer des entrées en sorties (Transformateur) comme les portes "Et", "Ou", "Non", … ; soit engendrer des niveaux logiques en sortie (Générateur) "Vcc"[1], "Gnd"[2], "Itr"[3],… ; soit recevoir des niveaux logiques en entrée (Récepteur) comme une "Led"[4], …

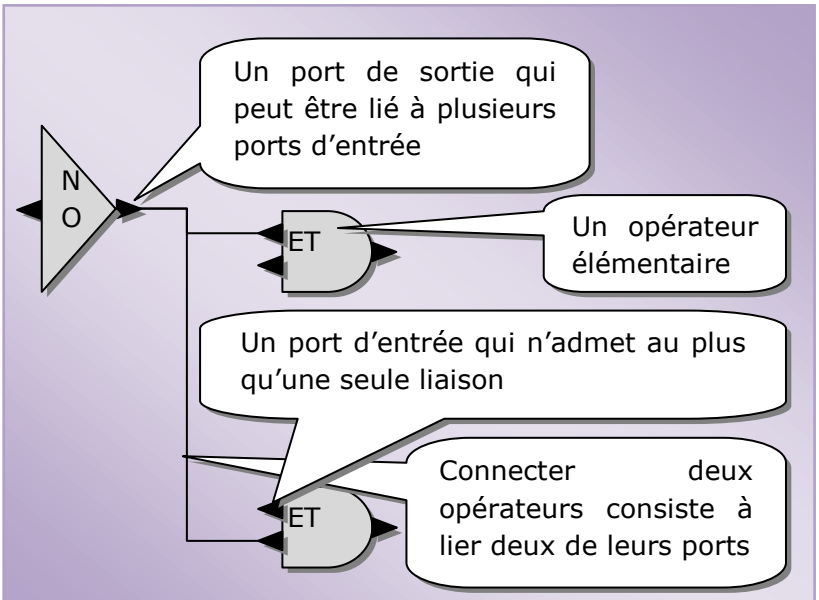



**Figure 1 : éléments constitutifs d'un circuit**

Un composant est donc un opérateur qui possède zéro, un ou plusieurs ports d'entrée et zéro, un ou plusieurs ports de sortie, mais en tout état de cause il possède au moins une entrée ou une sortie. Par exemple le composant "Et" possède 2 ports d'entrée et 1 port de sortie. Les nombres de ports d'entrée et de sortie d'un type de composants sont immuables, ces ports d'entrée (respectivement de sortie) sont précisément identifiés et seront repérés par un numéro d'ordre commençant à 1 pour chaque type de ports (entrée ou sortie).

[1] Générateur d'un niveau haut(1)
[2] Générateur de niveau bas (0)
[3] Générateur de niveau haut ou bas dépendant d'un choix de l'utilisateur final
[4] Voyant lumineux pouvant être allumé ou éteint

Un circuit est un assemblage de composants reliés entre eux par des connexions réalisant une opération plus ou moins complexe. Une connexion (un fil) est orientée et relie une sortie d'un composant à l'entrée d'un autre. Une sortie d'un composant peut être reliée à plusieurs entrées distinctes, mais une entrée ne peut recevoir qu'une connexion. Si des niveaux logiques existent aux ports d'entrée d'un opérateur, alors son activation fournira des niveaux logiques aux ports de sortie de celui-ci. **Seuls les états des ports sont observables dans notre système** ; Au contraire, les opérateurs seront des boites noires dont les activités respectives consistent à positionner leurs ports de sortie en fonction des états de leurs ports d'entrée. Cependant les générateurs comme les récepteurs doivent être observables via une valeur qui caractérise leur état, ceci afin de fournir un mécanisme de visualisation. Le choix de cette valeur est libre et peut être différente de celle utilisée pour la représentation textuelle de l'objet.

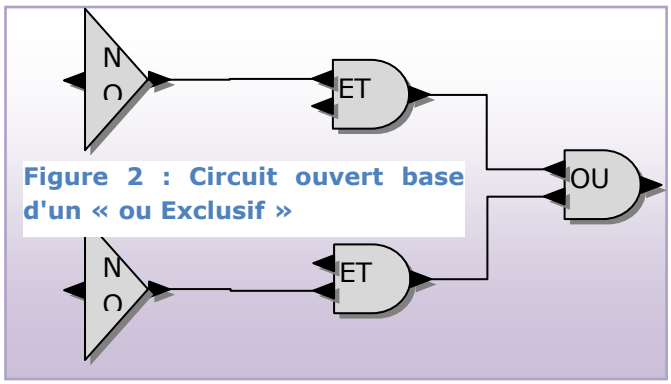


**Figure 2 : Circuit ouvert base d'un « ou Exclusif »**

Nous nous intéresserons aussi à des circuits particuliers, ceux pour lesquels aucun port d'entrée ou de sortie n'est libre, c.-à-d. qu'il existe une connexion (fil) partant de chaque port de sortie et une connexion arrivant sur chaque port d'entée. Il s'ensuit, comme nous l'avons déjà dit, que tout composant ne possède pas nécessairement des ports d'entrée et de sortie. Cependant, il est doté de ce qu'il faut pour être utile dans un assemblage. Nous disposerons, pour pouvoir interagir avec la simulation de façon simple, de composants électroniques qui "affichent" une information (Récepteur) et d'autres qui engendrent des informations (Générateur). Ceci permettra à un utilisateur de pouvoir fournir des entrées à un sous-circuit et de récupérer les résultats produits par ce sous-circuit. Nous considérerons qu'un circuit n'ayant aucun port libre est un **circuit fermé**.

A contrario, si un circuit possède au moins un port libre, il est dit **ouvert**. On souhaite pouvoir définir de nouveaux composants sur la base de circuits ouverts. Cette possibilité permet d'étendre aisément l'ensemble des composants mis à disposition pour la construction de circuits. Un composant composite est un composant pour lequel la fonction de calcul est réalisée à partir d'un circuit ouvert dont les ports libres sont connectés aux ports du composite qu'il constitue. On propose dans ***Figure 2 : Circuit ouvert base d'un « ou*** Exclusif **»**, un circuit ouvert qui peut être la base d'un "ou exclusif" et dans ***Figure 3*** le composite de ce « ou exclusif ». Cette définition confère au composite à la fois les propriétés de composant et de circuit. De même, on constate qu'un port d'un composite est un port d'entrée lorsqu'on considère celui-ci comme un composant atomique connectable au sein d'un circuit et un port de sortie lorsqu'on considère le composite comme un circuit ouvert dont les ports libres doivent être connectés aux «connexions» vers l'extérieur.

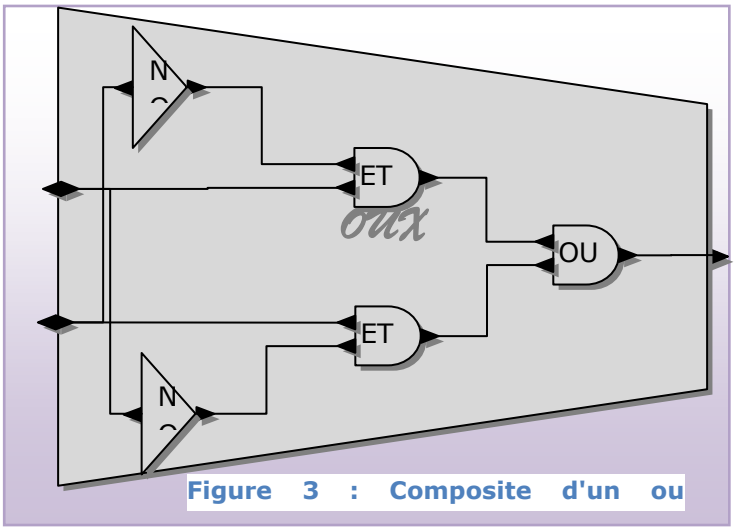


**Figure 3 : Composite d'un ou**

## 2.1 Représentation des circuits

Pour représenter des circuits, on propose un format textuel qui décrit les composants et les connexions qui les relient. Chaque composant est identifié par un numéro unique et le nom du type auquel il appartient. Si le composant est un générateur ou un récepteur une information complémentaire sur l'état du composant peut être mentionnée. A chaque composant est associée la liste des connexions sortantes en indiquant pour chacune d'elles la liste des connexions entrantes atteintes. Une connexion entrante est caractérisée par le numéro du composant d'arrivée et le numéro de la connexion entrante dans ce composant. Nous proposons dans la ***Figure 4*** une syntaxe pour cette notation :

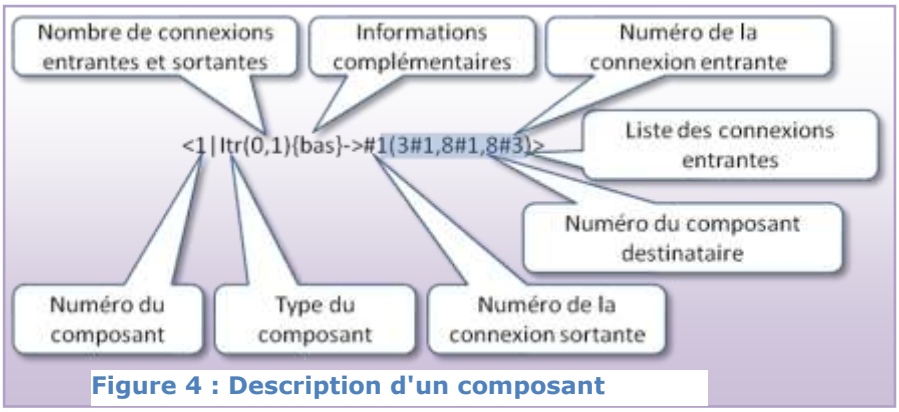



**Figure 4 : Description d'un composant**

Si le composant est composite, on peut souhaiter développer la structure interne qui le caractérise, nous proposons une syntaxe pour ce cas là dans la ***Figure 5***. Une connexion d'interface en entrée, représentée par son numéro d'ordre (#i), est reliée à une connexion entrante décrite par le numéro du composant interne et le numéro du port d'entrée à laquelle elle aboutit. Une connexion sortante d'un composant interne (#j) est reliée une connexion d'interface décrite pas son numéro d'ordre (#k).

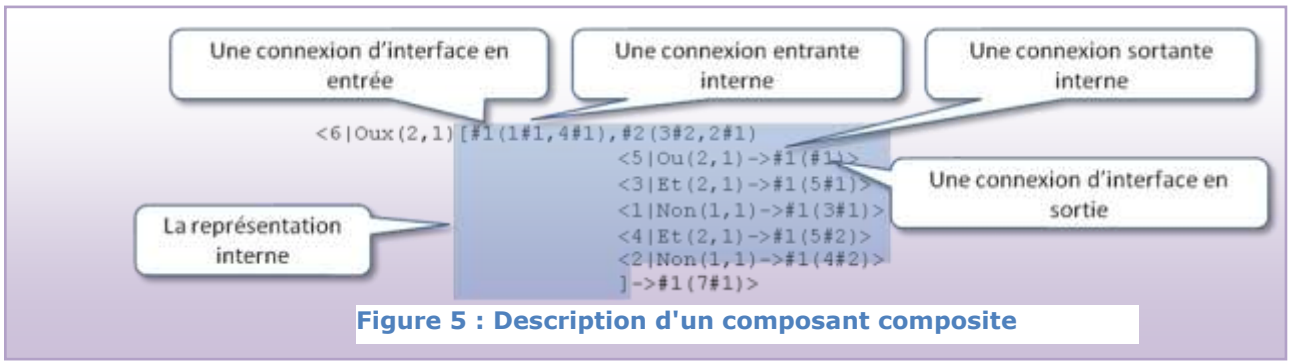



**Figure 5 : Description d'un composant composite**

```
CircuitTest[
    <7|Led(1,0){Eteint}>
    <6|Oux(2,1)[#1(1#1,4#1),#2(3#2,2#1)
        <5|Ou(2,1)->#1(#1)>
        <3|Et(2,1)->#1(5#1)>
        <1|Non(1,1)->#1(3#1)>
        <4|Et(2,1)->#1(5#2)>
        <2|Non(1,1)->#1(4#2)>
        ]->#1(7#1)>
    <3|Ou(2,1)->#1(6#1)>
    <8|Aff4b(4,0){0}>
    <1|Itr(0,1){bas}->#1(3#1,8#1,8#3)>
    <4|Non(1,1)->#1(6#2)>
    <2|Itr(0,1){bas}->#1(4#1,8#2,8#4)>
    <5|Vcc(0,1)->#1(3#2)>
]
```

**Figure 6 : Visualisation complète d'un circuit**

# 3 Analyse du problème

## 3.1 Identifier les concepts et les constituants majeurs

La modélisation objet se résume à élaborer des concepts en définissant des classes, des attributs, des méthodes ou des objets. L'approche objet favorise l'identification des concepts propres à l'application ou domaine de l'application que l'on souhaite réaliser, elle incite à produire des solutions proches du monde réel que l'on veut modéliser. A partir du texte fourni, surligner les mots qui vous paraissent importants et affectez-les soit à une classe (type), un objet, un attribut ou une méthode. Fournissez la classification des termes retenus. Construisez une hiérarchie de classes à partir de cette première analyse du domaine.

La structure hiérarchique des classes doit refléter les concepts du domaine, sa "géométrie" est nécessairement un élément d'appréciation de sa bonne élaboration. Vous serez attentif à définir les "classes interface" java nécessaires pour éviter les problèmes d'héritage multiple inhérents à ce langage. On peut estimer qu'une quinzaine de concepts majeurs permettent de déterminer le cadre du domaine de notre application. Ceci peut nous conduire à élaborer un peu plus d'une vingtaine de classes ou interfaces. Cette première décomposition pourra être étendue par la suite si l'intérêt s'en fait ressentir.

## 3.2 Spécifier les classes principales

Vous compléterez ce travail en définissant plus précisément la spécification de chaque classe élaborée : constructeur et méthodes de son interface.

Chaque classe sera pertinemment commentée. Vous préciserez, de façon circonstanciée, les choix qui ont présidé aux solutions que vous avez retenues. Vous utiliserez à cet effet tous les moyens de spécifications qui sont à votre disposition.

## 3.3 Structurer l'ensemble des concepts

Proposez une décomposition en package qui permette une bonne répartition des différentes classes à réaliser. Ceci facilitera la lisibilité du projet et rendra plus aisé son développement et sa maintenance. Par exemple un composant booléen peut être réalisé soit en concevant une classe d'un composant élémentaire, soit en réalisant une classe d'un composant composite élaboré à partir d'un sous-circuit. On peut raisonnablement vouloir que les 2 versions puissent exister et être utilisées simultanément. Cela peut aussi avoir un intérêt plus technique sur la manipulation introspective qui apparaît dans toute application moderne.

Vous pourrez aussi dans votre projet scinder l'espace des sources en plusieurs parties permettant, là encore, de rendre plus lisible l'ensemble du code développé.

# 4 Réalisations à élaborer

Ayant structuré l'application, on doit commencer par réaliser les éléments conceptuels majeurs inhérents au domaine de l'application à concevoir. Ensuite seront développés les éléments secondaires comme les composants logiques usuels, les divers outils ayant un intérêt dans l'élaboration de composant ou de circuit, enfin un environnement interactif dédié à l'utilisateur final.

## 4.1 Le noyau (kernel) du système boole

Réalisez les classes qui constituent le noyau de l'application boole. Celles-ci sont donc indépendantes des aspects d'interaction dans le pur respect du modèle MVC. A ce point du développement on disposera d'un framework (cadre) minimal qui pourra être complété par la suite pour construire une application finalisée destinée à un utilisateur. Le principe d'un framework se met naturellement en œuvre dans le modèle objet. Il a pour but de fixer les éléments conceptuels minimaux qui permettront, sur sa base, de construire des variantes d'une application, voire des applications différentes mais recouvrant une thématique commune. L'héritage est un moyen qui se prête bien à l'élaboration d'une telle architecture, offrant la possibilité d'étendre, d'adapter les concepts initiaux vers ceux nécessaires dans l'application à réaliser. Le schéma architectural du framework peut préciser les possibilités d'extensions offertes et les parties qui resteront figées.

## 4.2 Le niveau applicatif (application) du système boole

Cette partie vient compléter le framework réalisé précédemment en fournissant divers composants usuels offrant ainsi la possibilité à un usager d'élaborer des circuits. 2 grandes catégories de composants sont prévues : les composants élémentaires dont les comportements sont décrits dans le langage hôte et les composants composites dont les comportements pourront être décrits dans le langage des circuits.

### 4.2.1 Les composants élémentaires

Ecrivez les classes des composants élémentaires de type transformateur **Et**, **Ou, Non** et **Oux** (ou exclusif). Vous pourrez mettre en place les abstractions qui vous semblent utiles de rajouter pour assurer les mises en commun les plus pertinentes possibles.

Ecrivez les classes des composants élémentaires de type générateur, **Vcc** (engendre un niveau Haut), **Gnd** (engendre un niveau Bas), **Itr** (interrupteur permettant à l'utilisateur d'engendrer un niveau Haut ou Bas).

Ecrivez le composant élémentaire de type récepteur **Led** permettant d'afficher à l'utilisateur l'état du niveau (Haut ou Bas) reçu en entrée.

### 4.2.2 Les tests des composants élémentaires

Définissez la classe représentant un circuit fermé TestOuX permettant de tester votre composant OUX. Ce circuit comportera un interrupteur à chaque entrée du composant et une LED à la sortie. En outre, on dotera la classe d'une méthode tester() affichant l'état de la LED

pour chacune des positions possibles des interrupteurs suivi de la représentation textuelle du circuit. Dans ce cas particulier, seules 4 configurations des entrées sont à considérer, ce qui peut se faire simplement en les programmant toutes séquentiellement.

### 4.2.3 Généralisation du test à tout composant

On souhaite généraliser ce test en évitant la redondance de code : ne pas faire un programme pour chaque composant à tester. Sur le même modèle que la classe de test précédemment écrite, définir une classe CircuitTestUnitaire permettant de tester un composant quelconque de l'ensemble des composants utilisables. La partie construction du circuit se généralise aisément, il n'en est pas tout à fait de même pour la partie « tester ». Celle-ci revient à construire l'extension de la fonction représentée par ce composant.

Pour ce faire, il faut pouvoir engendrer la combinatoire de l'ensemble des générateurs du circuit de test. Dans le cas particulier où ceux-ci sont restreints aux interrupteurs il y a $2^n$ combinaisons possibles avec n interrupteurs. Il ne faut cependant pas envisager ce seul cas, on généralisera aussi au circuit fermé constitué de générateurs quelconques. Pour cela on va considérer que chaque Generateur est un Iterable<Void>, ce qui revient à lui faire énumérer tous les états possibles qu'il peut prendre. Cet itérateur ne produit donc aucune valeur caractéristique mais évolue au travers de tous les états du générateur considéré. Dans le cas d'un Interrupteur celui sera dans l'état Haut puis après dans l'état Bas (ou vice-versa).

Elaborer l'extension d'une fonction revient à constuire pour chaque combinaison des entrées la combinaison de sorties. Chaque élément de cette extension sera donc une paire dont le premier élément est la combinaison des entrées et le second élément la combinaison des sorties. L'extension est une collection de telles paires. Chaque paire sera construire par évaluation du circuit pour la configuration des entrées fournie.

Vérifier que votre solution fonctionne pour les composants réalisés précédement mais aussi pour un circuit fermé comme le TestOuX que vous avez aussi réalisé.

## 4.3 Les composants composites

Un composant composite est un composant dont la fonction de calcul est réalisée par un circuit, ce qui revient à dire qu'il pourrait être écrit en langage des circuits à la différence du composant élémentaire dont la description du comportement est faite à l'aide du langage hôte (dans notre cas Java). Réaliser la classe d'un composant composite ayant le comportement d'un additionneur 3 bits (Add3b) ayant 3 ports d'entrée et 2 ports de sortie et dont la sémantique est donnée dans le tableau ci-contre.

| E1 | E2 | E3 | S1 | S2 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Utiliser votre CircuitTestUnitaire pour tester ce composant, ce qui devrait reproduire, à la présentation près, la tabulation ci-contre qui pour les entrées E1, E2 et E3 données fixe les sorties S1 et S2 où S2 est l'unité et S1 la retenue sortante de l'opération d'addition (ou vice-versa à votre convenance).

### 4.3.1 Construction d'un Additionneur 4 bits

Pour réaliser l'addition de 2 nombres et vérifier que cet opérateur fonctionne, on va tout d'abord réaliser 2 composants. Le premier définit un composant émetteur correspondant à un générateur 4 bits (**Gen4bC2**) en complément à 2. L'état du générateur sera défini par un entier compris entre -8 et +7, comme décrit partiellement dans le tableau ci-contre. Le second est un composant récepteur correspondant à un afficheur 4 bits (**Aff4bC2**) assurant la fonction inverse, qui pour 4 entrées définissant un nombre en Cà2 sur 4 bits, associe l'état compris entre -8 et +7.

| Etat | S1 | S2 | S3 | S4 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| … | … | … | … | … |
| -1 | 1 | 1 | 1 | 1 |
| -2 | 1 | 1 | 1 | 0 |
| -3 | 1 | 1 | 0 | 1 |
| … | … | … | … | … |

Réalisez, sur la base de l'opérateur Add3b, un composant composite (**Add4x4**) fournissant l'additionneur de 2 nombres relatifs représentés sur 4 bits en complément à 2. Le composant fournira en sortie la retenue et le débordement en plus de la valeur sur 4 bits, en suivant le principe[5] de l'addition binaire en complément à 2. Tester le composant grâce à votre CircuitTestUnitaire.

## 4.4 Mise en place d'une spécification (description) de circuit

Pour décrire plus synthétiquement (aisément) un circuit, on se propose d'utiliser un langage de description basé sur celui de la représentation qui a été proposé au début de ce document. La description d'un circuit fera éventuellement apparaitre les descriptions de composites utilisés suivi de la description du circuit proprement dit comme le montre l'exemple ci-contre. La description d'un circuit permet d'énoncer l'assemblage à réaliser en mentionnant l'ensemble des composants et les liaisons qui les relient. Il s'ensuit que certaines informations utiles dans la représentation n'ont pas d'intérêt dans la description. De plus, de cette manière on évite d'avoir la description d'un composite au sein de la description d'un circuit.

```
composite Implique(2,1)
          <1|Non->#1(2#1)>
          <2|Ou>
[#E1(1#1);#E2(2#2);#S1(2#1)]


circuit TestImplique
          <1|Itr->#1(2#1)>
          <2|Implique->#1(4#1)>
          <3|Itr->#1(2#2)>
          <4|Led>
```

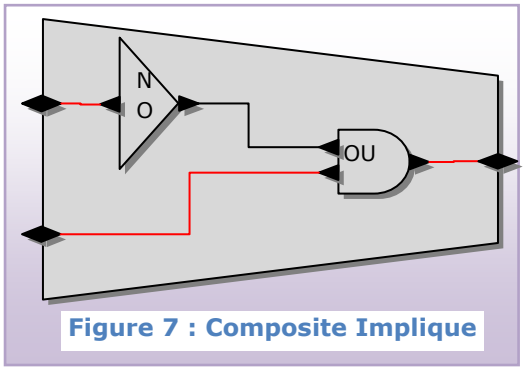


**Figure 7 : Composite Implique**

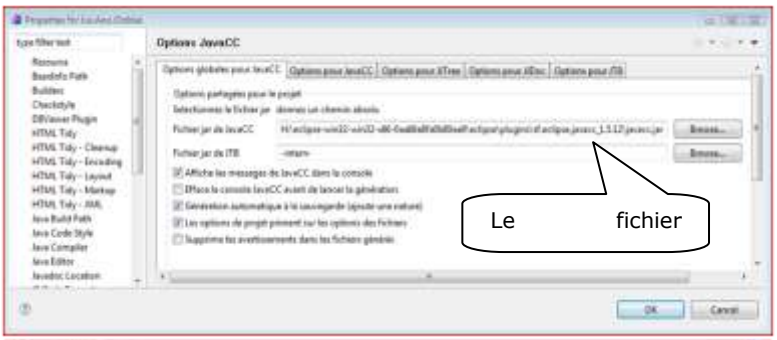
La description d'un composite est constitué de celle d'un circuit à laquelle on doit ajouter les caractéristiques du composite : Nombre d'entrées et de sorties ainsi que les connexions du circuit à celles-ci. Dans la proposition faite, les connexions sont fournies dans une seule liste, on utilise une syntaxe discriminante pour savoir s'il s'agit d'un port d'entrée (E) ou de sortie (S) du composite.

Cette syntaxe est une proposition vous pouvez l'adapter si vous estimez pouvoir faire mieux en ne manquant pas de le justifier avec les arguments qui vous auront conduits à le faire. Dans la suite du sujet des propositions sont suggérées.

[5] Voir 8.3Addition en complément à 2

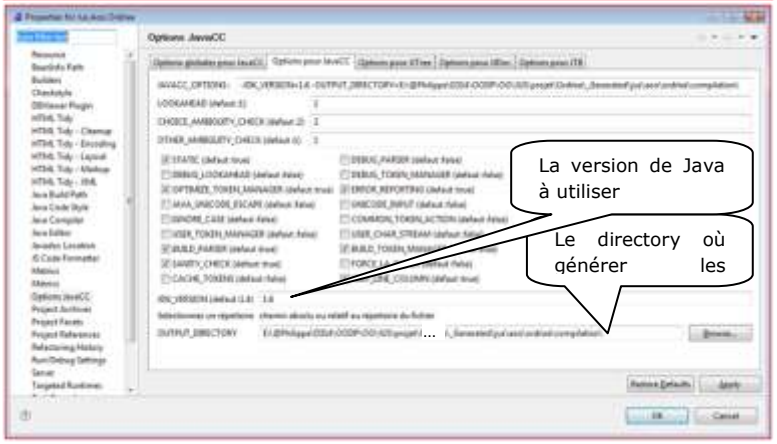
### 4.4.1 L'acquisition d'un circuit

Ceci se fera via une lecture dans un fichier comme ci-dessus. Pour cela nous allons utiliser un générateur d'analyseur syntaxique qui s'intègre aisément à Java.




### 4.4.2 Configuration de JavaCC

Cette description se trouve dans un fichier qui peut se nommer « Reader.jj ». Si vous avez installé le plugin JavaCC (http://sourceforge.net/projects/eclipse-javacc) dans votre environnement Eclipse, la mise à jour et la génération des classes nécessaires se fera automatiquement du moment où vous aurez fixé les options dans les propriétés de votre projet.




Celles-ci permettent de déterminer le fichier jar à utiliser pour l'analyse de la grammaire, ainsi que le lieu où vous souhaitez engendrer les classes de l'analyseur.

On vous conseille d'utiliser un autre directory source dans votre projet pour isoler correctement les différentes parties de votre application.

### 4.4.3 Prémisse de la grammaire

Le Reader permet d'acquérir la description d'un circuit sur l'entrée sélectionnée. On vous fournit les prémisses du programme d'analyse syntaxique sous forme de la grammaire du langage décrit en JavaCC. La classe Reader comportera principalement 1 méthode permettant l'acquisition d'un circuit : read(). D'autres méthodes pourront être définies, si vous en avez le besoin, dans la classe mère `ReaderUtilities`.

```
options{STATIC=false;}
PARSER_BEGIN(Reader)
package jus.aoo.boole.reader;

import jus.aoo.boole.circuit.*;

public class Reader extends ReaderUtilities {
    public _Circuit read() throws Exception {return DEF_CIRCUIT();}
}

PARSER_END(Reader)
SKIP :{" " |"\r" | "\t" | "\n"}
TOKEN :{
  <NUM: ( <DIGIT> )+ >
| <ID:  <LETTER> (<DIGIT> | <LETTER>)+>
```

```
| <#LETTER: ["a"-"z","A"-"Z"] >
| <#DIGIT: ["0" - "9"] >
}
TOKEN :{
   Définir les lexèmes du langage
}
<Type> DEF_CIRCUIT() : {variables locales }{
   (DEF_COMPOSANT())*
   ("circuit" name=<ID> <variable> =CIRCUIT() { return …;})?
   { return null;}
}
```

## 4.5 Mise en place d'une spécification de test d'opérateur

La combinatoire inhérente aux circuits réalisables pouvant prendre une large proportion, nous ne souhaitons pas nécessairement les explorer toutes, il s'ensuit qu'il faut pouvoir spécifier les cas particuliers que l'on veut contrôler. Nous allons mettre en place une description qui fixe les configurations des entrées que l'on veut vérifier en y associant les configurations des sorties attendues. Pour cela définir ce qui vous paraît nécessaire pour produire un rapport de test indiquant, pour chaque cas testé, si le résultat calculé est conforme ou non à celui attendu. La description d'une spécification du test d'un opérateur pourrait avoir la forme suivante, on y indiquerait le nom du test, l'opérateur devant être testé et l'extension partielle de la fonction que couvre ce test. La dtd associée à la représentation xml d'un tel fichier serait :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Test (Operateur, Couple*)>
<!ATTLIST Test name CDATA #REQUIRED>
<!ELEMENT Operateur EMPTY>
<!ATTLIST Operateur name CDATA #IMPLIED >
<!ELEMENT Couple (Entrees, Sorties)>
<!ELEMENT Entrees (#PCDATA)>
<!ELEMENT Sorties (#PCDATA)>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Test SYSTEM "TestUnitaire.dtd">
<Test name="test Ou">
   <Operateur name ="jus.aoo.boole.operateur.elementaire.Ou"/>
   <Couple>
      <Entrees> 0 0 </Entrees>
      <Sorties> 0 </Sorties>
   </Couple>
   <Couple>
         <Entrees> 0 1 </Entrees>
         <Sorties>1</Sorties>
   </Couple>
   <Couple>
         <Entrees> 1 1 </Entrees>
         <Sorties> 1 </Sorties>
   </Couple>
</Test>
```

# 5 Mémorisation des portes (prototypes ou modèles)

Pour faciliter la construction d'un circuit, on souhaite ne pas redéfinir dans son fichier de description systématiquement tous les composites qui pourraient y apparaître et qui caractérisent les portes non élémentaires utilisées. Pour cela on autorise l'utilisateur à pouvoir définir des fichiers de prototypes ou modèles permettant d'élaborer une base de composites où l'on pourrait puiser les différents composites nécessaires aux circuits à réaliser comme on le fait avec les composants élémentaires. Deux stratégies principales sont envisageables, correspondant toutes deux à 2 manières de mémoriser une information : soit par sa valeur, soit par le calcul de cette valeur. Dans le premier cas on parlera de prototype : Objet reproductible, dans le second cas on construit un modèle permettant d'engendrer l'objet : La classe s'impose d'office dans l'approche objet.

Dans les 2 cas la mémorisation est externalisée via un système de fichiers ou un mécanisme de sauvegarde plus sophistiqué. Dans les 2 cas, il est préférable d'utiliser le principe des ressources accessibles via le « classpath » de l'application, ce qui la rend indépendante de sa localisation.

## 5.1 Principe du prototype

On construit un objet composite qui sera mémorisé dans une "base" de composites. Une façon simple de mettre en œuvre cette base consiste à avoir une table associative reliant le nom au prototype. Il faut dans ce cas que le prototype soit « duplicable » pour produire autant d'exemplaires que nécessaire. Une seconde façon consiste à associer à chaque prototype (qui est nommé) un fichier de sauvegarde de l'objet prototype. La sérialisation, fournie en standard dans Java, est une solution aisée et rapide à réaliser. Par restauration de cette sauvegarde on obtient un nouvel exemplaire du composite.

```
prototype Implique(2,1)
          <1|Non->#1(2#1)>
          <2|Ou>
[#E1(1#1);#E2(2#2);#S1(2#1)]
```

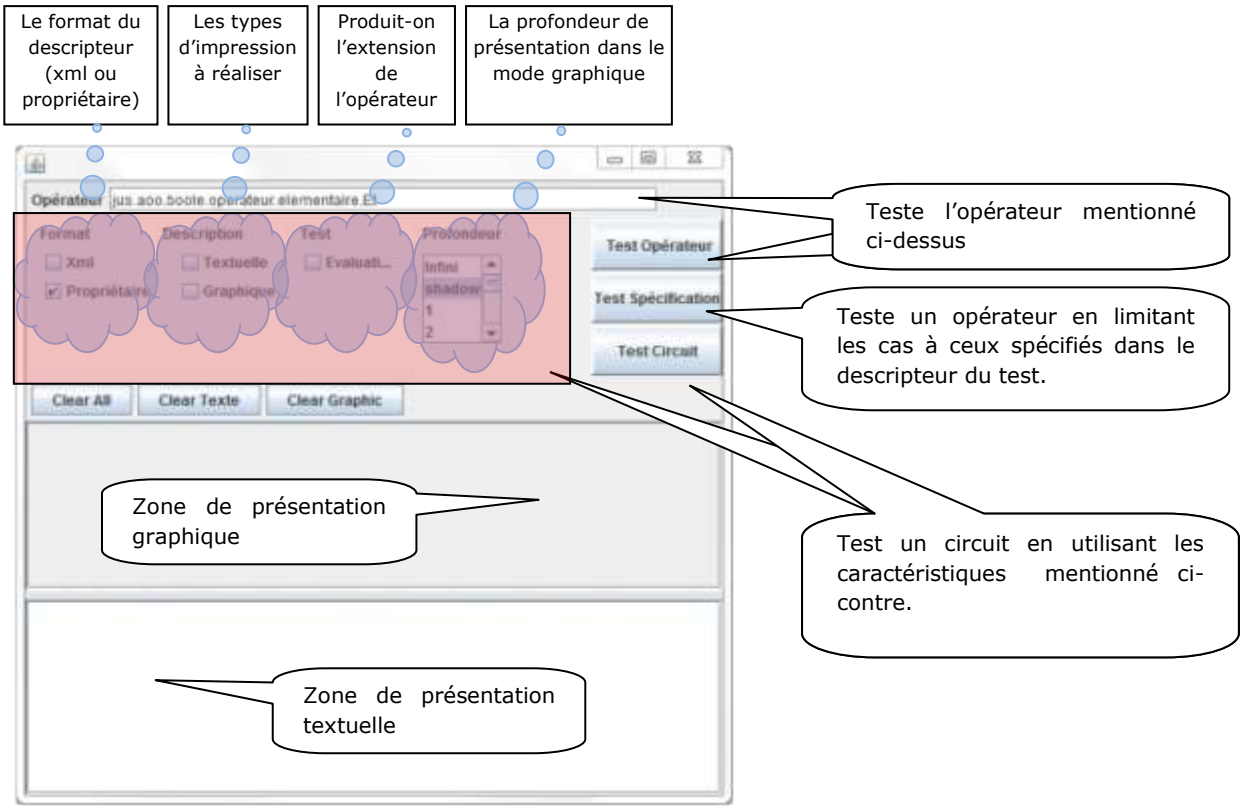
## 5.2 Principe du modèle

On construit la classe spécifique correspondant au modèle, par instanciation de celle-ci on obtient un exemplaire du composite. Il faut donc construire cette classe portant le nom du modèle. Plusieurs façons sont à disposition, la plus élémentaire consiste à construire le fichier .java et de réaliser sa compilation en utilisant les outils de compilation classiques de Java soit via une commande, soit via la classe Compiler qui fournit cette abstraction. Une autre consiste en l'utilisation de librairies permettant la manipulation des classes Java comme des objets. Il en existe plusieurs, une des mieux adaptée à notre problème serait Javassist.

```
modele Implique(2,1)
          <1|Non->#1(2#1)>
          <2|Ou>
[#E1(1#1);#E2(2#2);#S1(2#1)]
```

# 6 L'interface du logiciel

On pourra proposer une interface graphique offrant les présentations souhaitées et les commandes correspondant aux différentes possibilités offertes par le logiciel comme le montre l'exemple. Dans un premier temps on peut se contenter de mettre en place une zone textuelle pour faire apparaitre les représentations des circuits et les résultats des différents tests et un bandeau de commandes offrant 5 commandes : le test d'un opérateur (soit un composant soit un circuit fermé), le test d'une spécification, le test d'un circuit, le clear du support et enfin un quit.




Pour réaliser une version graphique, vous pouvez utiliser la libraire JgraphX qui offre un widget dédié à la manipulation de structures de graphe. Cet aspect du projet reste cependant secondaire, une interface utilisateur n'ayant un réel intérêt que lorsque le cœur de l'application existe. Il ne faut pas oublier aussi que l'application en elle-même doit rester entièrement indépendante de l'interface qui simplement en facilite l'usage.

# 7 Représentation XML d'un circuit.

Au lieu d'avoir un langage ad-hoc comme nous l'avons fait précédemment, on se propose de disposer d'une représentation xml de la structure d'un circuit, dont on vous propose ici une formulation possible avec la dtd associée :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Test SYSTEM "Circuit.dtd">
<Test>
  <Composite name="Xor" entrees="2" sorties="1">
     <Composant idf="5" type="Ou"/>
     <Composant idf="4" type="Et">
        <Connexion sortie="1">
           <Destination composant="5" entree="2"/>
        </Connexion>
     </Composant>
     <Composant idf="3" type="Et">
        <Connexion sortie="1">
           <Destination composant="5" entree="1"/>
        </Connexion>
     </Composant>
     <Composant idf="2" type="Non">
        <Connexion sortie="1">
           <Destination composant="4" entree="2"/>
        </Connexion>
     </Composant>
     <Composant idf="1" type="Non">
        <Connexion sortie="1">
           <Destination composant="3" entree="1"/>
        </Connexion>
     </Composant>
     <Interface>
        <Entree port="1">
           <Destination composant="1" entree="1"/>
           <Destination composant="4" entree="1"/>
        </Entree>
        <Entree port="2">
           <Destination composant="2" entree="1"/>
           <Destination composant="3" entree="2"/>
        </Entree>
        <Sortie interface="1" composant="5" port="1"/>
     </Interface>
  </Composite>

  <Circuit name="TestXor">
     <Composant idf="4" type="Led"/>
     <Composant idf="3" type="Xor">
        <Connexion sortie="1">
           <Destination composant="4" entree="1"/>
        </Connexion>
     </Composant>
     <Composant idf="2" type="Itr">
        <Connexion sortie="1">
           <Destination composant="3" entree="2"/>
        </Connexion>
     </Composant>
     <Composant idf="1" type="Itr">
        <Connexion sortie="1">
           <Destination composant="3 " entree="1"/>
        </Connexion>
     </Composant>
  </Circuit>
</Test>
```

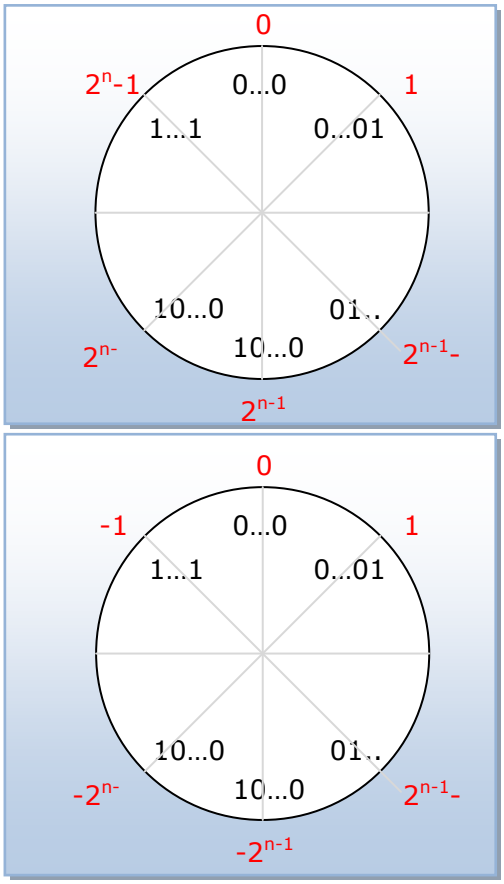
```xml
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT Test ((Composite | Modele)*, Circuit)>

<!ELEMENT Circuit (Composant+)>
<!ATTLIST Circuit name NMTOKEN #REQUIRED>
<!ELEMENT Composant (Connexion*)>
<!ATTLIST Composant
      idf NMTOKEN #REQUIRED
      type NMTOKEN #REQUIRED
>
<!ELEMENT Connexion (Destination+)>
<!ATTLIST Connexion sortie CDATA #REQUIRED>
<!ELEMENT Destination EMPTY>
<!ATTLIST Destination
   composant NMTOKEN #REQUIRED
   entree CDATA #REQUIRED
>
<!ELEMENT Composite (Composant+,Interface)>
<!ATTLIST Composite
   name NMTOKEN #REQUIRED
   entrees CDATA #REQUIRED
   sorties CDATA #REQUIRED
>
<!ELEMENT Modele (Composant+,Interface)>
<!ATTLIST Modele
   name NMTOKEN #REQUIRED
   entrees CDATA #REQUIRED
   sorties CDATA #REQUIRED
>
<!ELEMENT Interface (Entree*,Sortie*)>
<!ELEMENT Entree (Destination+)>
<!ATTLIST Entree
   port CDATA #REQUIRED
>
<!ELEMENT Sortie EMPTY>
<!ATTLIST Sortie
   interface CDATA #REQUIRED
   composant CDATA #REQUIRED
   port CDATA #REQUIRED
>
```

# 8 Annexes

## 8.1 Codage des nombres entiers

Avec n bits on dispose de $2^n$ représentations différentes ce qui permet donc de représenter $2^n$ nombres entiers.




Les nombres naturels ($\mathbb{N}$) sont codés en binaire standard par une suite de bits $\{b_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$ qui représentent le nombre entier dont la valeur est : $\sum_{k=0}^{n-1} b_k 2^k$ *où* $b_k \in \{0,1\}$. Il s'agit d'une arithmétique circulaire (modulo) telle que les opérations (addition par exemple) sont modulo $2^n$.

Pour représenter des nombres relatifs ($\mathbb{Z}$) on dispose du même nombre de représentations qui devront être réparties de façon équitables entre les positifs et les négatifs. Plusieurs associations sont possibles celle qui retient le plus l'intérêt en informatique est celle dite du complément à 2 (Cà2). Elle à l'avantage de coder $2^n$ nombres relatifs différents en ayant $2^{n-1}$ nombres négatifs et $2^{n-1}$-1 nombres strictement positifs, d'avoir des propriétés de calcul qui rendent les algorithmes performants et compatibles avec le codage standard.

D'autres codages ont été utilisés ou le sont encore dans certaines circonstances. Le codage VAS qui attribue à $b_{n-1}$ la signification de signe (0 : le nombre est positif, 1 le nombre est négatif) et $\{b_{n-2}, \ldots, b_1, b_0\}$ la valeur absolue. On peut constater qu'en Cà2 le bit $b_{n-1}$ représente aussi le signe.

## 8.2 Addition en standard

Soit A ($\{a_{n-1}, a_{n-2}, \ldots, a_1, a_0\}$) et B ($\{b_{n-1}, b_{n-2}, \ldots, b_1, b_0\}$) 2 nombres en codage standard. Leur somme est un nombre C ($\{c_{n-1}, c_{n-2}, \ldots, c_1, c_0\}$) tel que sa valeur S(C) = (S(A)+S(B)) % $2^n$ où $c_i$ est $a_i+b_i+r_i$ avec $r_i$ la retenue de la somme précédente. On pose $r_0 = 0$. Si $r_n$ est égal à 0 alors C est le codage standard de la somme des valeurs de A et B sinon on dit qu'il y a débordement et la valeur de C est celle de (A+B)%$2^n$.

## 8.3 Addition en complément à 2

Le principe de l'addition en complément à 2 est de réaliser l'addition en standard puis de voir comment s'interprète le résultat. On note par S(A), où A est de la forme $\{a_{n-1}, \ldots, a_1, a_0\}$ avec $a_i$ valant 0 ou 1, l'interprétation standard de A = $\sum a_i 2^i$. On note V(A) la valeur correspondant à l'interprétation en complément à 2 de A, on rappelle que $C_2(A,n) = 2^n - S(A)$

où n représente le nombre de bits de représentation, donc $V(A) = S(A)-a_{n-1}2^n$. On rappelle que le complément à 1 $C_1(A,n)= 2^n-1-S(A)$

Définition de ce que l'on doit calculer

1. $V(A)+V(B) = S(A)+S(B) -2^n(a_{n-1}+b_{n-1})$
2. $V(C) = S(C)-c_{n-1}2^n$

Définition de ce qui est calculé (relativement à la définition de l'addition en standard)

1. $2r_n+c_{n-1} = a_{n-1}+b_{n-1}+r_{n-1}$
2. $S(C) = S(A)+S(B)-r_n2^n$

On développe l'équation n°1

$V(A)+V(B) = S(C)+r_n2^n-2^n(a_{n-1}+b_{n-1})$ 4

$S(C)+2^n(r_n-a_{n-1}-b_{n-1})$

$V(C)+c_{n-1}2^n+2^n(r_n-a_{n-1}-b_{n-1})$ 2

$V(C)+2^n(c_{n-1}+r_n-a_{n-1}-b_{n-1})$

$V(C)+2^n(r_{n-1}-r_n)$ 3

Une autre façon de trouver la condition de validité de l'opération consiste à résoudre l'équation suivante :

$S(A)+S(B) -2^n(a_{n-1}+b_{n-1}) = S(C)-c_{n-1}2^n$

On remplace, dans la définition de ce que l'on doit calculer, S(C) par la valeur effectivement calculée, on obtient l'équation suivante :

$S(A)+S(B) -2^n(a_{n-1}+b_{n-1}) = S(C)+r_n2^n-c_{n-1}2^n$

$S(C)-2^n(-r_n+c_{n-1})$

Pour que l'opération soit valide il faut que $a_{n-1}+b_{n-1} = -r_n+c_{n-1}$.

La somme en interprétation standard de 2 nombres en complément à 2 donne la valeur en complément à 2 de la somme, au débordement près.

Montrez que ceci est nécessairement vrai quand les signes des opérandes sont différents.

## 8.4 Exemple de construction d'une classe avec Javassist.

Ci-après une description minimaliste de l'usage que l'on peut faire de la librairie Javassist qui offre de nombreuses fonctionnalités de réification de Java et de manipulation à un niveau plus fin que ce que propose le package java.reflect de la librairie standard.

```
// Construction d'une classe ayant pour nom <name> héritant de la classe de nom <superClassName>
ClassPool pool = ClassPool.getDefault();
try{
   // La classe existe-t-elle déjà dans le pool des classes, si oui rien à faire
   pool.get(name);
}catch(NotFoundException e){
   // On construit la classe de nom name
   CtClass cc = pool.makeClass(name);
   // On établit la relation d'héritage
   cc.setSuperclass(pool.get(superClassName));
   // On élabore un constructeur pour cette classe avec pour définition la chaîne <cstrDef>
   CtConstructor cstr = CtNewConstructor.make(cstrDef,cc);
   cc.addConstructor(cstr);
   // On élabore une méthode pour cette classe avec pour définition la chaîne <mthDef>
   CtMethod m = CtNewMethod.make(mthDef, cc);
   cc.addMethod(m);
   // On peut sauvegarder la classe dans le directory de nom <directoryName>
   cc.writeFile(directoryName);
   // On peut charger la classe correspondante
   cc.toClass();
}
```

cstrDef et mthDef correspondent aux textes que l'on écrit en Java pour ces 2 caractéristiques de la classes que sont le constructeur et la méthode. Des exceptions peuvent être levées par ce code, il faut les prendre en compte.