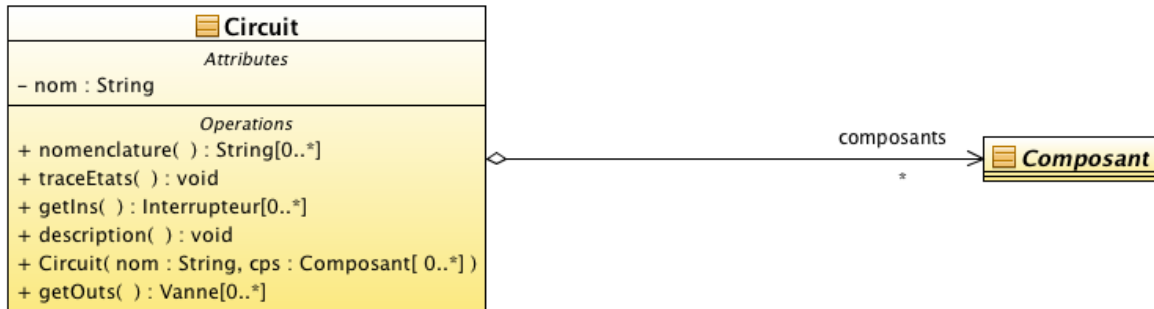


## TP4

## 1 Objets circuits

Jusqu'à maintenant vous avez remarqué que les circuits n'ont pas d'existence propre et sont manipulés comme des vulgaires tableaux de composants dans la classe de test. Nous allons ici leur donner le statut d'objets à travers une classe `Circuit` selon le schéma UML suivant :



- Un `Circuit` est composé de `Composant` au travers du rôle `composants` : utiliser une classe de `java.util.List<E>` pour représenter cela.
- Le constructeur `public Circuit(String nom, Composant[] cps)` permet de construire un circuit en fournissant 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> getIns()` renvoie la liste des interrupteurs<sup>1</sup> :
  - créer une liste `list_interrupteurs` qui sera retournée après traitement

1. on considère pour simplifier des circuits dont les entrées sont des interrupteurs et que ceux-ci ne sont dédiés qu'à cela (plus proches de la notion de commutateur). Une généralisation consisterait à introduire une surclasse pour les entrées (interrupteurs, capteurs, commutateurs, ...), idem en sortie (appareils, actionneurs, ... dont les vannes ne sont qu'un exemple).

- la remplir avec les composants de type `Interrupteur` grâce au test `instanceof`
- De manière similaire la méthode `getOuts()` renvoie la liste des vannes.

## 1.1 Classe Circuit

Programmer la classe `Circuit` respectant ces spécifications et tester progressivement ses méthodes comme suit.

## 1.2 Test

Dans la classe `TestCircuits` :

- En fin de la section `//Connexions` : instancier le circuit exemple 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()`
  - `getIns()` et `getOuts()` (afficher leurs ids)
  - forcer certains interrupteurs (récupérés par `getIns()`) et afficher l'état des vannes ou toute la trace par `circ.traceEtats()`.

# 2 Sondage de circuits

Sonder un circuit complet consiste à substituer tous ses interrupteurs d'entrées par des sondes interactives (`LazySonde`). Pour cela on ajoute le protocole suivant à la classe `Circuit` :

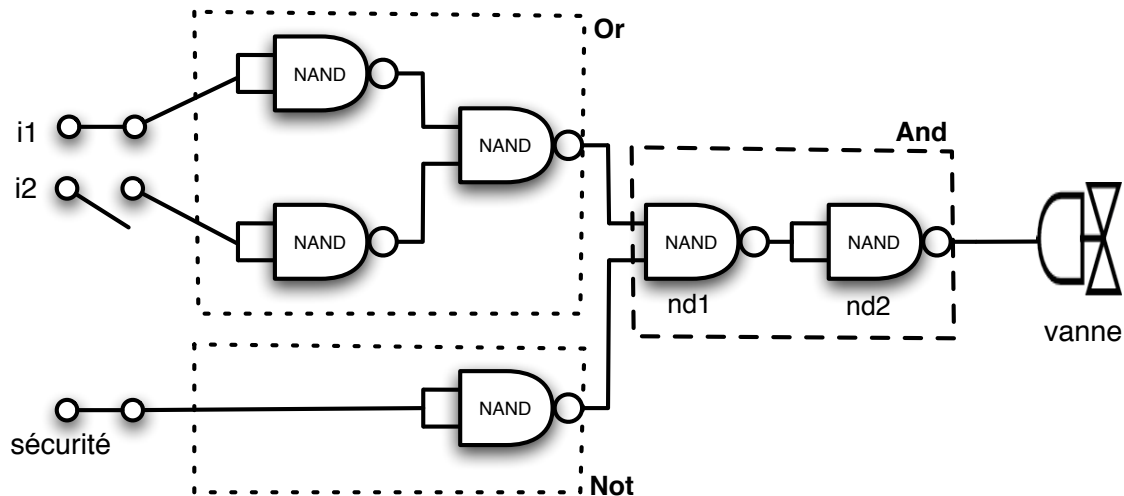
- `public void probe()` qui fait ces substitutions
- `public void resetSondes()` qui permet de réinitialiser les sondes (par appel à leur `reset()`) pour un nouveau sondage
- `public void unProbe()` qui défait les substitutions : rétablit les connexions sur les interrupteurs et détruit les sondes.

Pour vous aider à programmer ces méthodes, appliquer la démarche suivante.

## 2.1 Table d'association Sonde ↔ Interrupteur

Le circuit doit mémoriser les substitutions `Sonde ↔ Interrupteur` lors du `probe()` pour 2 raisons :

- lors du `probe()` trouver la sonde correspondant à un interrupteur si celui-ci a déjà été substitué, pour éviter les duplications et les incohérences. En effet le même interrupteur (comme tout autre composant d'ailleurs) peut être connecté sur plusieurs entrées comme dans le circuit suivant (cf. `i1`, `i2` et `nd1`).



— lors du `unProbe()` retrouver l'interrupteur correspondant à une sonde pour rétablir la connexion.  
 Les Map (HashMap ou TreeMap) permettent de résoudre le problème dans un sens : Sonde → Interrupteur  
 ou Interrupteur → Sonde, mais pas les 2.

## A faire :

Programmer une classe `SondesTable`<sup>2</sup> définie comme suit :

- `SondesTable` encapsule 2 Maps (variables d'instance) : `sondesToInterrupteurs` (Sonde → Interrupteur) et `interrupteursToSondes` (Interrupteur → Sonde)
- La méthode:  
`Interrupteur getInterrupteur(LazySonde sonde)`  
 renvoie l'interrupteur associé à la sonde passée en paramètre
- La méthode:  
`LazySonde getSonde(Interrupteur interrupteur, Composant cible, String entree)`  
 — renvoie la sonde associée à l'interrupteur passé en paramètre si elle existe déjà  
 — sinon la crée (avec les paramètres `cible` et `entree`), ajoute l'association dans `sondesToInterrupteurs` et `interrupteursToSondes` et la renvoie.
- La méthode void `resetSondes()` fait un `reset()` sur toutes les sondes
- La méthode void `clear()` défait les associations Sonde - Interrupteur par `clear()` sur `sondesToInterrupteurs` et `interrupteursToSondes`.

## 2.2 Utilisation

- Munir les circuits d'une `SondesTable` en déclarant dans la classe `Circuit` :  
`protected SondesTable tableSondes = new SondesTable();`
- La méthode `probe()` d'un circuit doit parcourir ses portes logiques en appliquant leur propre méthode void `probe(SondesTable tableSondes)`. Cette méthode teste si la porte logique est connectée sur des interrupteurs, auquel cas elle met à jour la table `tableSondes`. Appliquer le même principe pour l'opération de `unProbe()`.
- Tester `probe()`, `resetSondes()` et `unProbe()` dans la méthode `test(Circuit circ)` de `TestCircuits`.

---

<sup>2</sup>. pas trop bruyante...