TP4 : À propos de calcul binaire :

on n'est jamais assez fort pour ce calcul...

notions : tout le programme...

Ce TP est hérité d'un examen sur machine Ensimag, réalisé en 2h.

Notez que vos enseignants POO Phelma déclinent toute responsabilité pour les malencontreuses contrepèteries éparpillées dans ce sujet. Toute récrimination à ce sujet sera adressée à Sylvain Bouveret, l'inénarrable 1er auteur de l'examen original.

attention : respectez bien les prototypes de méthode demandés pour ne pas avoir de soucis, à la fin du TP, avec le code fourni.

Introduction

Nous allons dans ce TP nous intéresser à nos amis les circuits logiques. Plus précisément, nous allons écrire un programme permettant de simuler le comportement de divers circuits logiques et de faire du calcul booléen (cf Figure 1) [1].

FIGURE 1 – Vue d'artiste de Georges Boole (1815–1864)

On considère un circuit digital *sans cycle* (circuit combinatoire). Il est entièrement défini par l'ensemble de ses entrées, de ses portes, des connexions entre ces éléments, et du sous-ensemble de ces éléments qui constituent ses sorties. Soit par exemple le circuit de la figure 2. Il possède 3 entrées et 6 portes. 4 de ces éléments constituent ses sorties, dans l'ordre : la porte « Et » désignée par « a », la porte « Et » désignée par « b », l'inverseur désigné par « f », et l'entrée numéro 2.

1. The Illustrated London News, 21 January 1865, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=39762976`

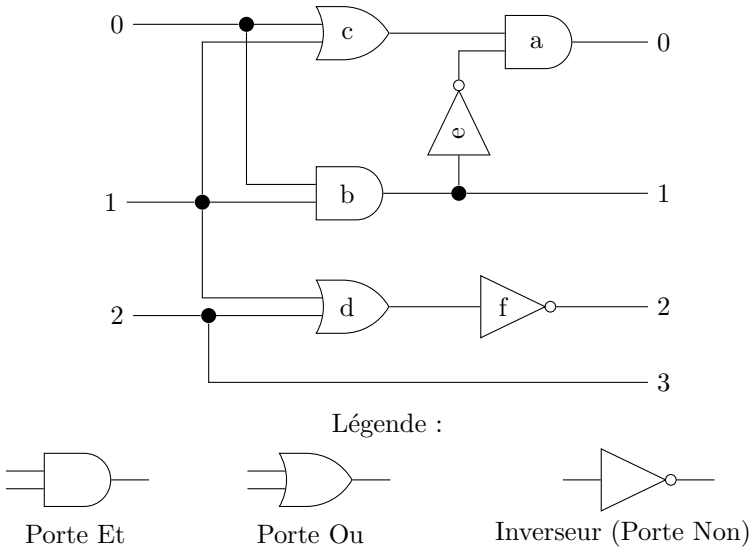

FIGURE 2 – Un circuit à 3 entrées et 4 sorties

1 Codage des portes logiques

1.1 Les classes de base

Avant de s'intéresser à la notion de circuit et au calcul d'un circuit, nous allons, dans cette partie, coder la base des éléments logiques constituant un circuit.

On appelle *élément* une entrée ou une porte. Les *portes* sont des inverseurs (porte « Non ») ou des portes binaires (dans l'exercice, on ne considère que des portes binaires « Et » et « Ou »). La figure 3 présente la hiérarchie de classes pour les éléments. Les classes `Element` et `PorteBinaire` sont abstraites, les 4 autres sont concrètes.

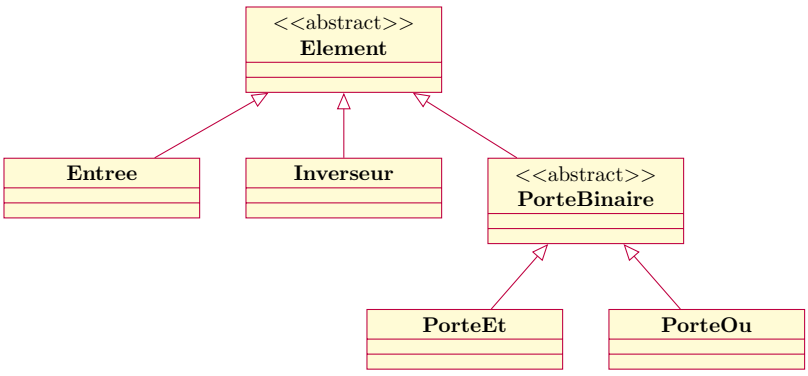

FIGURE 3 – Hiérarchie pour les éléments

1.1.1 Classe abstraite `Element`

La classe abstraite `Element` est constituée d'un attribut `nom` de type `String`. Codez cette classe (n'oubliez pas le constructeur !).

Pour le bon fonctionnement de la classe, le nom ne doit pas être `null`. Vous devez donc empêcher l'initialisation d'un élément avec un nom `null` (il serait de bon ton d'utiliser des exceptions...).

Ajoutez également à cette classe, **en respectant les usages Java** :

- une méthode qui renvoie une chaine de caractères `String` décrivant l'état de l'objet - c'est à dire retournant le nom de l'élément.
- une méthode qui teste l'égalité de `this` avec un autre objet passé en paramètre. On considèrera que deux éléments sont égaux si et seulement s'ils ont le même nom [2] ;
- une méthode qui retourne un code de hachage (entier) pour l'élément, qui permettra de regrouper des éléments dans une Collection de type "ensemble basé sur une table de hachage".
  Comme d'habitude en Java, cette méthode devra être cohérente avec la méthode qui teste l'égalité.
- une méthode définissant une relation d'ordre entre les éléments. Elle compare l'élément `this` avec un autre élément passé en paramètre et retourne un entier positif, nul ou négatif suivant que `this` est "plus grand", "égal" ou "'plus petit" que l'autre élément, compte tenu de la relation d'ordre considérée.
  L'existence de cette méthode permettra de regrouper des éléments dans une Collection de type "ensemble basé sur un arbre binaire de recherche".
  Comme d'habitude en Java, cette méthode devra être cohérente avec la méthode qui teste l'égalité.

1.1.2 Classe concrète `Inverseur`

La classe concrète `Inverseur`, en plus d'hériter de `Element`, possède un attribut `source` de type `Element`.

Le constructeur de cette classe prend en paramètre une chaîne de caractères (le nom de l'élément), et l'élément source. L'élément `source` ne doit pas être `null` (et donc vous devez empêcher la création d'un inverseur avec une source `null`). Codez cette classe. Attention : pour faciliter la suite de ce TP, respectez bien l'ordre des paramètres du constructeur.

1.1.3 Classe abstraite `PorteBinaire`

La classe abstraite `PorteBinaire`, en plus d'hériter de `Element`, possède deux attributs `source1` et `source2` de type `Element`. Codez cette classe. Comme précédemment, assurez-vous que l'on ne puisse pas initialiser une porte binaire avec l'une ou l'autre de ses sources valant `null`.

1.1.4 Classe concrètes `Entree` `PorteOu` et `PorteEt`

Les dernières classes concrètes de la hiérarchie, `Entree`, `PorteOu` et `PorteEt` (héritant respectivement de `Element`, `PorteBinaire` et `PorteBinaire`), n'ont pas d'attribut propre et, pour le moment, n'ont pas de méthode propre à part un unique constructeur.

2. Cela implique donc que, dans un circuit, tous les éléments devront avoir un nom différent.

Le constructeur de `Entree` prend en paramètre uniquement le nom de l'entrée. Quant aux constructeurs de `PorteOu` et `PorteEt`, ils prennent en paramètre une chaîne de caractères (le nom de l'élément) et les deux élémenst source.

Codez ces classes.

1.2 La liste des entrées

On souhaite connaître, pour une sortie d'un circuit, la liste des entrées dont elle dépend. Sur l'exemple de la figure 2, les sorties 0 et 1 dépendent des entrées numéro 0 et 1, la sortie 2 dépend des entrées numéro 1 et 2, la sortie 3 dépend de l'entrée 2.

Note : Nous supposerons ici (jusqu'à la section 5.1) que les circuits sur lesquels nous travaillons sont sans cycle et que tous les éléments portent un nom différent. Nous n'aurons pas à le vérifier pour l'instant.

La méthode `public Set<Entree> getEntrees()` de la classe `Element` renvoie en résultat une collection Java, dont les éléments sont des objets de la classe `Entree`. Cette collection doit contenir les entrées dont dépend l'élément logique sur lequel est appelé la méthode. L'ordre des éléments dans la collection retournée est sans importance.

On peut récupérer cet ensemble d'entrées sur n'importe quel type d'éléments, mais bien entendu, le code de cette méthode dépend du type d'éléments. Cette méthode est donc une méthode *abstraite* de `Element`, qui est redéfinie dans la hiérarchie des classes.

Ajoutez la méthode abstraite `getEntrees()` à la classe `Element`.

Redéfinissez ensuite cette méthode `public Set<Entree> getEntrees()` dans les sous-classes, sachant que :

— La liste des entrées dont dépend un objet de type `Entree` est une liste à un élément qui est l'entrée elle-même.
— Pour un inverseur, la liste des entrées est simplement constituée de la liste des entrées de la source.
— Pour une porte binaire, la liste des entrées est simplement constituée de l'union de la liste des entrées des deux sources.

Justifiez votre choix de la classe réalisant l'interface `Set<Element>`.

Que pouvez-vous dire de la complexité de la méthode `getEntrees()` ?

1.3 Un premier test

Écrivez un simple programme de test dans une classe `TestGetEntrees` qui crée une porte « Et » connectée à deux entrées, et affiche l'ensemble des entrées renvoyées par l'appel de la méthode `getEntrees()` sur cette porte « Et ».

Compilez, exécutez, et vérifiez que le résultat est cohérent.

2 La classe circuit

Maintenant que tous les éléments sont en place, nous allons pouvoir créer la classe `Circuit`.

En première approche, un circuit est constitué d'un ensemble d'éléments connectés. Toutefois, pour représenter un circuit dans la la classe, on peut se contenter de connaître l'ensemble de ses sorties. En effet, puisque les sorties connaissent leurs sources, et ainsi de suite jusqu'aux entrées du circuit, connaître les sorties permet d'accéder à tous les éléments du circuit.

Créez une classe `Circuit` aura un attribut `private Set<Element> sorties`, initialement vide.

Justifiez votre choix de la classe réalisant l'interface `Set<Element` que vous utilisez pour cet attribut.

Cette classe définira pour le moment les méthodes suivantes :
— `public void addSortie(Element e)`, qui ajoute une sortie au circuit ; Que pouvez-vous dire de la complexité de cette méthode ?
— `public Set<Entree> getEntrees()`, qui renvoie la liste des entrées du circuit.

Testez vos classes avec le programme de test `TestCircuit1` dont le fichier `.java` vous est gracieusement fourni. Ce fichier crée le circuit de la figure 2, et affiche ses entrées et ses sorties.

3 Evaluation et calcul

L'intérêt principal des circuits binaires est de pouvoir faire des calculs en propageant des valeurs d'entrée booléennes à travers les différentes portes logiques. C'est ce que nous allons faire ici.

Pour cela, nous allons nous appuyer sur l'utilisation d'un dictionnaire : une variable `Map<Entree , Boolean> environnement`, qui contient une liste d'affectations de valeurs booléennes aux entrées du circuit. C'est ce dictionnaire qui sert à représenter le vecteur des valeurs des entrées.

Étant donné un tel dictionnaire, n'importe quel élément du circuit doit être capable de s'évaluer, c'est-à-dire de retourner un booléen calculé de la manière suivante :
— pour un inverseur, on renvoie l'inverse de l'évaluation de sa source ;
— pour une porte « Et », on renvoie le résultat de l'application du « Et » logique sur l'évaluation de ses deux sources ;
— pour une porte « Ou », on renvoie le résultat de l'application du « Ou » logique sur l'évaluation de ses deux sources ;
— pour une entrée, si cette entrée est dans l'environnement, on renvoie directement cette valeur. Sinon, c'est une erreur (car nous ne pouvons pas calculer les valeurs de sortie du circuit dans ce cas-là). Il serait donc de bon ton de lancer une exception.

3.1 Modification de la hiérarchie de classe `Element`

Ajoutez dans la classe `Element` une méthode abstraite `evaluer` prenant en paramètre un environnement de type `Map<Entree, Boolean>` et renvoyant un booléen.

Donnez toutes les redéfinitions concrètes de cette méthode `evaluer` dans les sous-classes de `Element`, de manière à respecter la spécification ci-dessus.

3.2 Modification de la hiérarchie de classe `Circuit`

Ajoutez à la classe circuit une méthode qui évalue toutes les sorties du circuit. Cette méthode prend un paramètre un dictionnaire (environnement) précisant les valeurs voulues pour les entrées et retourne un autre dictionnaire associant les sorties à leurs valeurs.

3.3 Enrichissement de la classe de test `TestCircuit1`

Enrichissez la classe de test `TestCircuit1` de la section 2 pour qu'elle donne la table de vérité du circuit.

En d'autres termes, pour chaque combinaison de valeur des trois entrées (il y en a huit en tout), elle doit afficher sur la ligne de commande une ligne donnant la valeur des entrées, et la valeur des sorties.

Par exemple, les deux premières lignes affichées sont les suivantes :

```
false   false   false   ->      false   false   true    false
false   false   true    ->      false   false   false   true
...
```

4 Allez, un dernier calcul et on s'en va...

La classe `Additionneur`, dont le *bytecode* vous est fourni sous forme de fichier `.class` (et dont la Javadoc vous est fournie dans le répertoire `doc`), est une implantation d'un circuit réalisant l'addition entre deux entiers positifs.

Cette classe possède, entre autres, une méthode *statique* `additionner` prenant en paramètre deux entiers (de type `int`) et renvoyant la somme de ces entiers, calculée par un circuit binaire additonneur.

Si vos classes sont correctes, cette méthode doit effectivement renvoyer la somme algébrique correcte de ses deux paramètres.

Écrivez une classe `TestAdditionneur` qui utilise cette classe pour réaliser l'addition de deux entiers (que vous fixez dans le code), et vérifiez que le résultat est correct.

Bonus : modifiez votre programme pour que vous puissiez passer les deux entiers en paramètres de la ligne de commande, et qu'il vous affiche le résultat de l'addition de ces deux entiers.

5 Questions subsidiaires

5.1 Unicité des noms et absence de cycle

On souhaite modifier notre programme pour que deux contraintes soient toujours respectées sur les circuits :

1. un circuit ne doit jamais comporter deux portes logiques de même nom ;

2. un circuit ne doit jamais comporter de cycle : en parcourant de proche en proche les entrées d'un `Element` quelconque, on ne doit jamais retrouver cet élément.

Modifiez vos classes pour que, dès la construction du circuit, ces contraintes soient respectées.

Conseil : commencez par faire une copie de tous vos fichiers source, pour conserver votre code qui compile et s'exécute.

* * * * *

Tsss. Ça valait bien le coup de passer tout ce temps pour calculer 9+13...