Des circuits sur lesquels on peut compter

PO&GL. Classes abstraites, héritage, redéfinition de méthodes.

Présentation

Dans ce TP, nous allons créer des circuits effectuant des opérations arithmétiques. Ces circuits sont constitués de nœuds reliés entre eux, chaque nœud effectuant une opération élémentaire. Le squelette de la classe principale est téléchargeable à l'adresse suivante :

`http://www.lri.fr/~blsk/POGL/TDTP/Circuit.java`

Ce squelette contient quelques définitions de base et une méthode `main`, qui elle-même contient une série d'exemples avec lesquels vous pourrez tester votre code. Pour l'instant, ces exemples sont placés en commentaires pour éviter les erreurs de compilation. Vous pourrez décommenter chaque exemple dès que les opérations dont il a besoin auront été définies.

1 Mise en jambes

Un objet de la classe circuit possède trois attributs privés

- `int entree` : un paramètre entier donné au circuit,
- `ArrayList<Noeud> noeuds` : l'ensemble des nœuds du circuit, et
- `Noeud sortie` : le dernier nœud du circuit, qui donne le résultat final,

ainsi que les méthodes publiques suivantes

- `int litEntree()` : renvoie la valeur de l'entrée,
- `int calcule(int e)` : renvoie la valeur calculée par le nœud `sortie` avec une entrée valant `e`,
- des méthodes `Noeud cree*` pour chaque classe de nœud pouvant être ajoutée au circuit,

et un constructeur `Circuit()`.

Tous les nœuds que nous allons manipuler hériteront de la classe abstraite suivante :

```
abstract class Noeud {
  abstract public int valeur();
}
```

Un appel à la méthode `valeur()` doit renvoyer le résultat calculé par ce nœud du circuit, et ce résultat sera un entier. Voici par exemple la définition d'un nœud donnant une valeur constante :

```
class Constante extends Noeud {
    private int cst;
    public Constante(int c) { this.cst = c; }
    public int valeur() { return this.cst; }
}
```

Les nœuds pourront de plus posséder un certain nombre d'attributs représentant les opérandes des opérations effectuées par le nœud. Ces attributs privés auront par exemple pour noms `source1` ou `source2` et seront toujours de la classe `Noeud`.

Créer trois classes pour des nœuds effectuant les opérations suivantes :

- Une classe `Entree`, dont les objets possèdent un attribut `circuit` de la classe `Circuit`. La méthode `valeur()` doit renvoyer la valeur de l'entrée du circuit. Cette classe doit posséder un unique constructeur de signature `Entree(Circuit c)`.
- Une classe `Addition`, dont les objets possèdent deux attributs `source1` et `source2`. La méthode `valeur()` doit renvoyer la somme des valeurs calculées par les deux sources. Cette classe doit posséder un unique constructeur de signature `Addition(Noeud n1, Noeud n2)`.
- Une classe `Multiplication`, dont les objets possèdent deux attributs `source1` et `source2`. La méthode `valeur()` doit renvoyer le produit des valeurs calculées par les deux sources. Cette classe doit posséder un unique constructeur de signature `Multiplication(Noeud n1, Noeud n2)`.

Dans la classe `Circuit`, créer également les méthodes `creeEntree()`, `creeAddition(Noeud n1, Noeud n2)` et `creeMultiplication(Noeud n1, Noeud n2)` correspondantes. Vous pouvez vous inspirer de la méthode `creeConstante(int c)` déjà définie.

2 Une hiérarchie de nœuds

Pour hiérarchiser les nœuds et éviter les redondances comme celles déjà présentes dans les constructeurs des classes `Addition` et `Multiplication`, nous allons introduire une classe abstraite intermédiaire `NoeudBinaire` qui hérite de `Noeud` et y ajoute les ingrédients suivants (cette liste est un minimum, vous pouvez y ajouter d'autres attributs ou méthodes si cela vous semble utile, maintenant ou plus tard dans le TP) :

- Des attributs privés `source1` et `source2` de classe `Noeud`.
- Des méthodes publiques `valeurSource1()` et `valeurSource2()` qui renvoieent les valeurs calculées respectivement par les nœuds `source1` et `source2`.

Définir une telle classe abstraite, et redéfinir les classes `Addition` et `Multiplication` pour qu'elles héritent de `NoeudBinaire`. Ajouter une classe `Soustraction` pour représenter un nœud binaire qui calcule la différence entre sa première source et sa deuxième source.
Avant de passer à la partie suivante, ne pas oublier de vérifier que les exemples fonctionnent toujours.

3 Affichage

Ajouter à la classe `Circuit` une méthode `public void affiche()` qui affiche l'expression arithmétique calculée, en utilisant des parenthèses autour des opérations binaires pour éviter les ambiguïtés, et en représentant l'entrée par `x`. Cette méthode pourra se reposer sur la méthode `public String toString()`, à redéfinir pour les nœuds.

Un des objectifs de la question consiste à utiliser convenablement l'héritage, les méthodes abstraites et les redéfinitions pour écrire un minimum de code.

4 Des outils de diagnostic

On veut pouvoir analyser la taille d'un circuit et l'utilisation de ses nœuds. Pour cela, ajouter à la classe `Circuit` les deux méthodes suivantes :

- `public int nbNoeudsMult()` compte le nombre de nœuds de classe `Multiplication` dans le circuit.
- `public int nbOpEffectuees()` compte le nombre de fois où les méthodes `valeur()` des nœuds du circuit de classe autre que `Constante` ou `Entree` ont été appelées (si la méthode `calcule` du circuit a été appelée plusieurs fois, les comptes doivent être cumulés).

Ces deux méthodes pourront faire appel à d'autres méthodes à définir dans la classe `Noeud` ou ses sous-classes, et pourront également utiliser de nouveaux attributs à introduire vous-même.

Utiliser ces outils de diagnostic pour analyser le comportement du circuit produit par l'appel `expRapide(20)` (la méthode `expRapide` est présente dans le squelette de code). Que constate-t-on ? La partie suivante vise à éliminer ce problème.

5 Memoïsation

Créer une classe `MultiplicationMemoisee` qui hérite de `Multiplication` et qui redéfinit sa méthode `valeur()` de sorte à ce que :

- le premier appel à la méthode `valeur()` d'un objet effectue le calcul comme un nœud `Multiplication` normal, mais enregistre en plus le résultat dans un nouvel attribut privé `mem`,
- tous les appels suivants se contentent de renvoyer la valeur enregistrée (qu'on qualifie aussi de *mémoïsée*) sans refaire le calcul. Dans ce dernier cas, la méthode `nbOpEffectuees()` ne doit pas compter une nouvelle opération.

En modifiant la méthode `expRapide` pour qu'elle utilise cette multiplication améliorée, le calcul de la vingtième puissance de `2` doit renvoyer `1048576` en n'effectuant que 7 multiplications.

Remarque : certaines des multiplications effectuées par l'exponentiation rapide n'ont pas besoin d'être mémoïsées. D'où un objectif bonus : n'utiliser un nœud `MultiplicationMemoisee` que lorsque cela est nécessaire.

Créer ensuite un circuit `c` calculant la vingtième puissance de son entrée, et effectuer successivement les appels `c.calcule(2)` et `c.calcule(3)`. Que constate-t-on ?

Ajouter aux circuits et au nœuds une méthode `void reInit()` qui réinitialise les nœuds de multiplication mémoïsée.