Labo 05 Matrices - Rapport

Sacha Bron - Valentin Minder // HEIG-VD // POO // 29.10.2014

Introduction

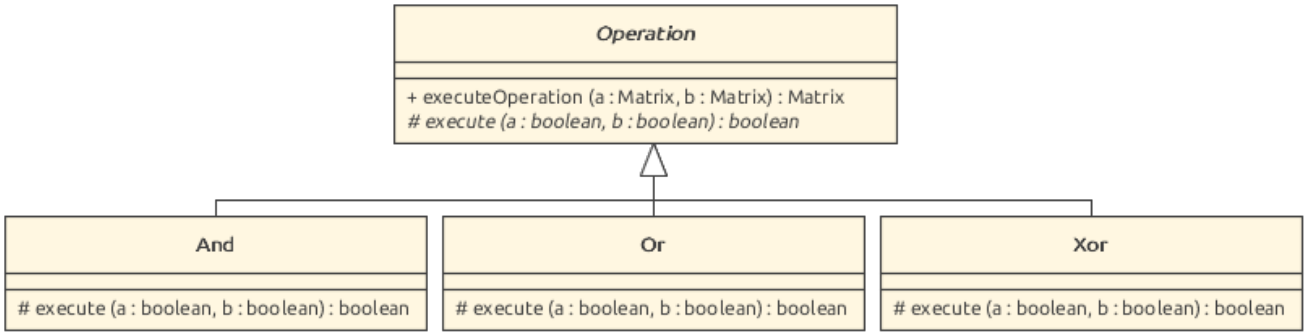
Le but de ce laboratoire est d'implémenter des opérations de base sur des matrices carrées de booléens, de manière dynamique sans effectuer de tests sur le type des opérations utilisées. Les opérations de base sont `And`, `Or` et `Xor`.

Méthodologie

Pour réaliser cela, nous avons créé un objet `Matrix` qui stocke de manière interne et privée le tableau de tableau de `boolean`. Au moment de la création (constructeur), la taille fixe est passée en paramètre, puis le tableau de tableau est automatiquement rempli aléatoirement. Cette classe contient des accesseurs (getters) et des modifieurs (setters), permettant d'accéder aux éléments et de les modifier, par leur position `(x, y)` dans la matrice. Par ailleurs, la méthode `toString()` est redéfinie pour afficher correctement la matrice comme demandé.

Note sur la liaison dynamique des liens

Afin de bénéficier du principe de liaison dynamique des liens, nous avons créé une classe abstraite `Operation` et trois sous-classes concrètes pour les opérations en elles-mêmes, selon le schéma ci-dessous.



La classe abstraite contient une méthode abstraite `execute(a, b)` qui sera implémentée différemment par les sous-classes. Elle contient de plus une méthode concrète qui créé la matrice pour stocker le résultat et gère le parcours des matrices opérandes. Dans cette boucle, elle appelle `execute(a, b)` sur deux éléments.

Ainsi, le code `execute(a, b)` fonctionne toujours depuis la classe abstraite, et va appeler automatiquement la bonne méthode dans la bonne sous-classe à l'éxécution, en fonction du type réel de `op` dans le main (`op.executeOperation(m1, m2)`). En effet, on peut librement réaffecter `op = new Or();` sans affecter le comportement.

Pour aller plus loin

A plusieurs reprises, il est nécessaire de vérifier que les tailles sont compatibles (p.ex les opérations ne sont applicables que sur des matrices de même taille, l'accès et la modification des éléments doit se faire dans les limites de la taille, etc.). Lors d'arguments illégaux, nous avons lancé une `IllegalArgumentException`.

Il serait sans doute plus propre de ne pas instancier à chaque fois les opérations, mais d'avoir un deisgn de `Singleton` avec un contructeur privé et une méthode `getInstance()`. Toutefois, cela semble hors du cadre de ce labo.

Conclusion

Nous constatons qu'au moment de l'execution, la bonne méthode est appelée en fonction du type réel de l'objet `Operation`, bien que celui-ci soit de type abstrait. Ainsi, grâce au polymorphisme, aucun test sur le type des objets et aucune structure de contrôle n'est nécessaire.

Annexes

Le code imprimé est annexé. Il se compose des 4 opérations, de l'objet `Matrix` et d'un launcher (`main`) pour instancier les objets. Vous trouverez de plus ci-après un exemple d'éxécution.

Exemple d'exécution

```
.  .   .                  ,      , .       .  .    ,        |
|  | _ | _. _ ._ _  _    -+-_   -+-|_  _    |\/| _ -+-._.*\./|
|/\|(/,|(_.(_)[ | )(/,   |(_)    | [ )(/,  |  |(_] | |  |/'\*

Matrix one
0 0 0 1
1 1 0 1
1 0 0 0
1 0 1 1

Matrix two
1 0 1 0
0 1 0 0
0 0 1 1
1 1 0 0

oneOrTwo = one or two
1 0 1 1
1 1 0 1
1 0 1 1
1 1 1 1

oneAndTwo = one and two
0 0 0 0
0 1 0 0
0 0 0 0
1 0 0 0

oneXOrTwo = one xor two
1 0 1 1
1 0 0 1
1 0 1 1
0 1 1 1
```