TP Design Patterns
Conception orientée objet

Axel Fahy & Rudolf Höhn

January 20, 2016

1 Contexte de développement

Pour ce travail pratique, nous avons choisi le jeu du Sudoku. L'utilisateur peut jouer au Sudoku suivant les règles standardes pour une grille 9x9. Le jeu se fait en ligne de commande, à chaque tour, on demande à l'utilisateur s'il veut continuer ou arrêter le jeu, s'il veut utiliser un code de triche ou revenir dans un état précédent et dans quelle case veut-il placer le nouveau numéro.

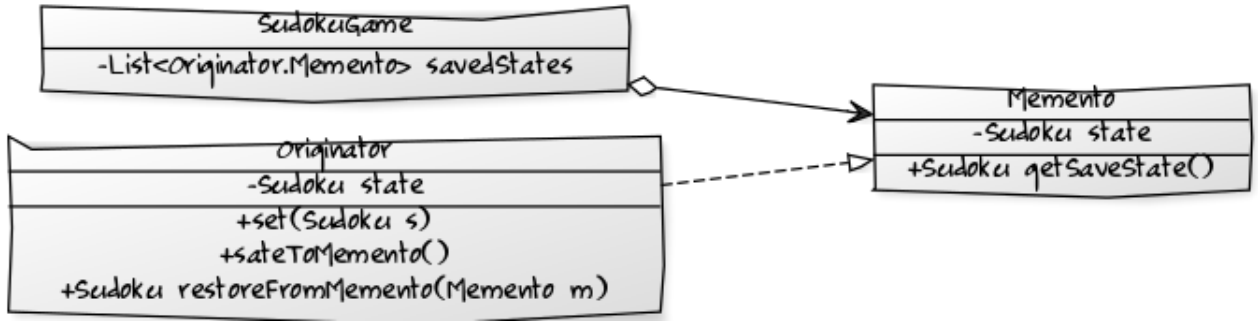
Pour la partie triche et mémoire du jeu, nous avons utilisé les Design Pattern Adapter et Memento.

2 Description des Design Patterns

2.1 Memento

Le Design Pattern Memento nous sert à mémoriser les précédents états du jeu, c'est-à-dire, des états de notre *Sudoku*. Dans la classe *SudokuGame*, nous avons une liste de *Memento* où chaque instance sera un état du jeu, et un objet *Originator* qui connaitra l'état actuel du jeu. Seul l'*Originator* peut aller lire ou modifier un état du jeu, soit un *Memento*.

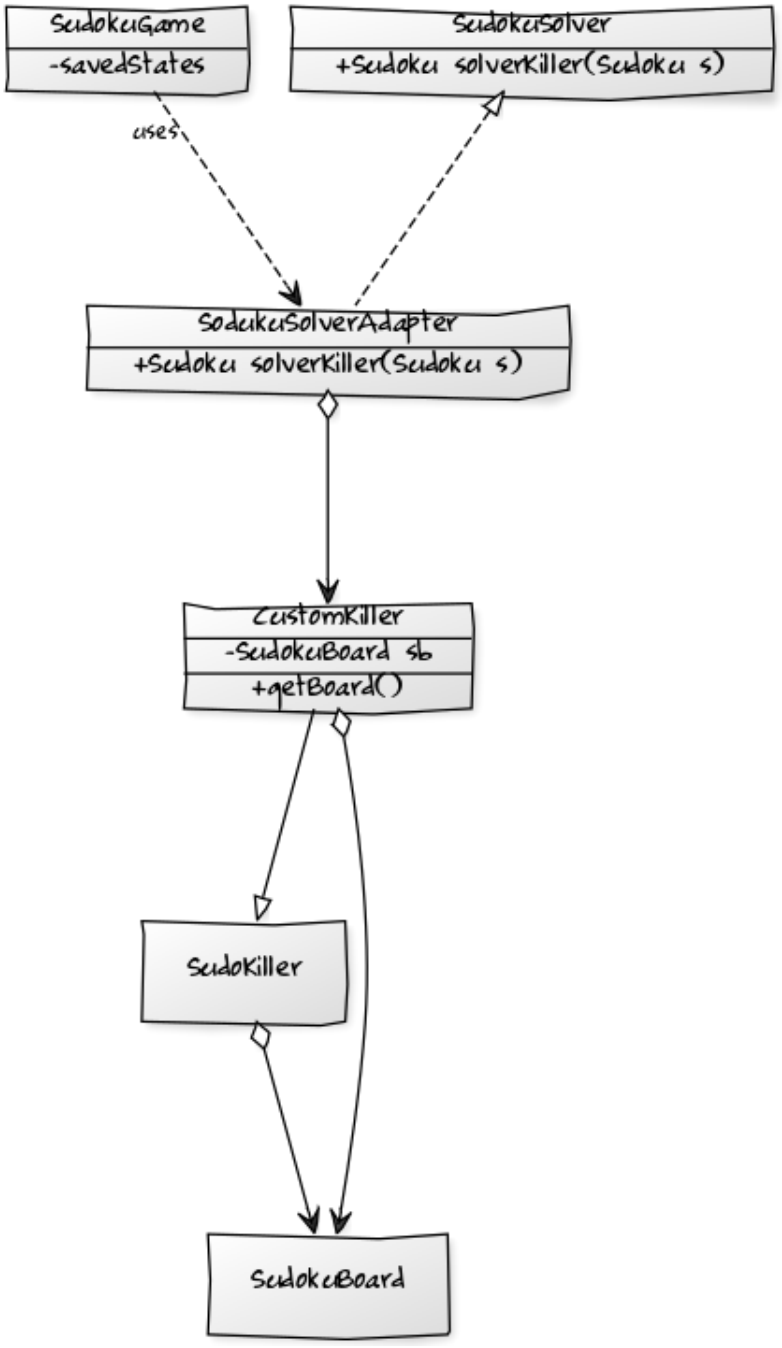
Si l'utilisateur décide de revenir à un état précédent, il pourra revenir d'autant de coups qu'il veut, et s'il est allé trop en arrière, il pourra revenir dans le temps (retour vers le futur, mais sans Dolorean).



Ce Design Pattern apporte une gestion des états très simple à mettre en œuvre. Le *Memento* est une classe opaque au jeu (ou *Caretaker* dans la littérature), et c'est pour garder le principe d'encapsulation qu'on passe par l'*Originator*.

2.2 Adapter

Le Design Adapter nous permet d'utiliser une fonction/algorithme provenant d'une autre classe sans devoir adapter notre code ou le code de l'algorithme pour se conformer à notre implémentation. Dans notre cas, nous avions à disposition un résolveur de Sudoku, *SudoKiller* qui, à partir d'un *SudokuBoard*, résoud le Sudoku. Nous avons donc pour cela implémenté une classe *SudokuSolverAdapter* qui reçoit un objet *Sudoku*, extrait la grille, crée un objet *SudokuBoard* et appelle une instance de *CustomKiller* pour trouver la bonne réponse. La classe *SudoKiller* étant abstraite, nous avons du définir une classe qui l'étend.



Ce Design Pattern nous permettrait de, par exemple, utiliser d'autres algorithmes de résolution de Sudoku sans devoir changer notre code de jeu, il faudrait juste rajouter une fonction *solverOther* où 'Other' serait l'autre algorithme. On comprend donc assez bien son utilité, elle permet de construire un lien entre deux programmes différents pour augmenter les capacités du premier.

3 Discussion

Les Design Pattern nous ont apporté une bonne modularité du code, le jeu en lui-même n'est pas lié au traitement de sauvegardes et d'algorithmie. Ils sont vus comme des modules par le jeu.

Grâce à l'Adapter, nous pourrions utiliser d'autres algorithmes de résolution et par exemple, faire des tests de performances entre les différents algorithmes. L'Adapter nous permet d'intégrer aisément du code externe sans avoir à modifier ni notre application, ni le code externe, il fait le lien entre les deux de manière transparente.

A la fin de la partie, nous pourrions afficher tous les états qui ont amené à la résolution du Sudoku en implémentant un auter Design Pattern qui est l'Iterator. Ce dernier nous permettrait d'accéder à tous les états de manière séquentielle sans casser l'encapsulation.

L'application pourrait être étendue en une version graphique, une version web ou une version mobile de manière à accéder sa partie de n'importe où et retrouver son état.