République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديمقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي و البحث العلمي

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

TP de Patrons De Conception

Livrable 02

Réalisé par :

- TADJER Amina
- FEKIR Nadjet

Groupe : SIL2

Projet TP3_3 : Football Game.

Proposé par : Mme BOUSBIA

Promotion : 2019/2020

I. Introduction :

Les Design Patterns permettent d'anticiper les problématiques et d'accélérer le processus de développement. En effet, leur application réduit les couplages au sein d'une application, apporte de la souplesse, favorise la maintenance et d'une manière générale aide à respecter les bonnes pratiques de développement.

Le design pattern du GoF surnommés la bande des quatres (Gang of Four) sont les patterns les plus connus et les plus cités, ils sont classés en trois catégories :

Les patrons de création décrivent comment régler les problèmes d'instanciation de classes, c'est à dire de création et de configuration d'objets.

Les patrons de structure décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas.

Les patrons de comportement décrivent la communication entre les classes et leurs responsabilités.

Après avoir analysé le code dans la première partie, il y'avait des problèmes qu'on ne pouvait pas régler seulement avec les patrons GRASP, d'où la nécessité d'appliquer les patrons GOF.

II. Solutions proposées :

1. Memento :

Après l'analyse du code, nous avons remarqué que les développeurs ont voulu implémenter la fonctionnalité de sauvegarde du jeu et l'état du tournoi, et ensuite restaurer la partie (voir dans Begin.java et SemiFinal.java et Finel.java et FenetrePrincipale.java avec l'objet ObjectOutputStream loadGame qui lit à partir d'un fichier loadGame.dat). Et de même pour la sauvegarde de la liste des équipes et des buts marqués (classe Teams.java dans le fichier TournoiOld.dat et classe Buts.java en utilisant le fichier saveGoal.dat).

Cette fonctionnalité d'enregistrement et de restauration est évidente, car le joueur doit avoir la possibilité d'arrêter le jeu et de le reprendre plus tard d'une part, et d'une autre part les résultats des matchs ainsi que le classement des équipes du début jusqu'à la fin du tournoi doivent être sauvegardés pour les afficher à la fin.

Problème :

En dépit des bugs que contiennent ces fonctionnalités lors de l'exécution, et sans tenir compte de l'enregistrement aléatoire, la base de la solution existante qui est sa conception est déjà mal faite. La solution existante comme est présentée viole un concept très important de la POO qui est l'encapsulation.

Solution proposée :

Le patron Memento est très adaptée pour cette situation, car il délègue la création des instantanés d'état au propriétaire réel de cet état, l'objet d'origine. Par conséquent, au lieu que d'autres objets tentent de copier l'état du tournoi et du jeu de «l'extérieur», la classe elle-même peut créer l'instantané car elle a un accès complet à son propre état.

Donc nous allons concevoir ces fonctionnalités d'enregistrement et de restauration en utilisant ce patron, et nous allons fournir la possibilité de sauvegarder dans un fichier pour pouvoir restaurer même en fermant la fenêtre du jeu et arrêtant son exécution en utilisant la sérialisation (qui est déjà utilisée dans la solution existante).

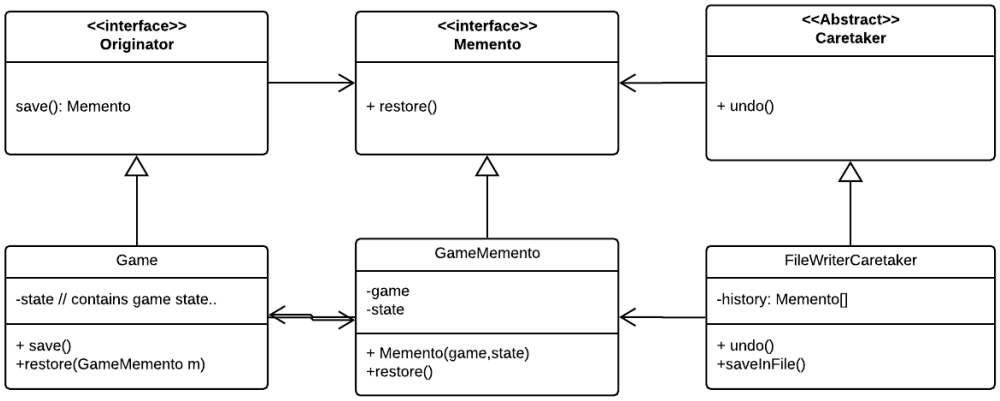
Nous allons appliquer notre solution sur une seule classe comme exemple qui est la classe jeu « Game », le jeu contient deux équipes qui s'affrontent, deux scores et l'état du jeu en général. Donc on va déléguer la sauvegarde de l'état du jeu courant à la classe Game elle-même, dans un objet spécial qui est GameMemento (classe imbriquée dans Game) qui est une capture de l'état du jeu à un moment donné, et nous allons créer une classe GameCaretaker qui saura quand et pourquoi sauvegarder et restaurer.

Nous allons utiliser la sérialisation en cas ou l'utilisateur veut quitter carrément le jeu et le reprendre (load Game).

Vu que la classe Game n'est pas la seule concernée par la sauvegarde et la restauration, nous allons créer 2 interfaces Memento et Originator, pour permettre aux autres classes de les implémenter, pour sauvegarder l'historique des équipes gagnantes par exemple.

Diagramme des classes :

Nous allons essayer d'adapter le diagramme d'un Memento simple à notre situation, en tenant compte de l'écriture dans des fichiers en utilisant la sérialisation.



Perspectives :

On peut combiner entre le patron Command qui sera responsable d'effectuer des opérations (sauvegarde, restauration.).

Le patron Iterator aussi peut être utilisé si on souhaite afficher le résultat du tournoi du premier tour jusqu'à la finale(match après match, tour après tour).

2. Builder :

L'équipe est composée d'un ensemble de joueurs, dont les attaquants, défenseurs et gardiens. Chaque équipe contient également un coach, et a un nom et une couleur. Et d'une liste pour les numéros des joueurs (voir la classe Team).

Pour instancier un objet de type Team et commencer le jeu, les développeurs ont donné la possibilité à la classe Test qui contient Main de créer les 11 joueurs (qui seront titulaires pendant le

match), et de les sauvegarder dans 2 listes (pour chaque équipe, la liste de ses joueurs titulaires est dans la classe Test, et de même pour le coach de la Team). Ensuite, instancier ces équipes en passant en paramètre le coach, la liste des joueurs, la couleur et le nom aussi.

Problème :

Le problème qui se pose est que la classe Test a été encombrée par la création des deux équipes, tout en connaissant la structure interne de l'équipe, le nombre d'attaquants et de défenseurs, la couleur (sachant que Test n'est pas censée connaitre la structure interne de chaque équipe).

Solution proposée :

Pour ne pas surcharger la classe Test, et pour ne pas donner la possibilité à une classe cliente de connaitre toutes les étapes de construction d'une équipe, nous allons utiliser le patron Builder. De plus le responsable de création de l'équipe sera la classe Game (Director dans notre cas).

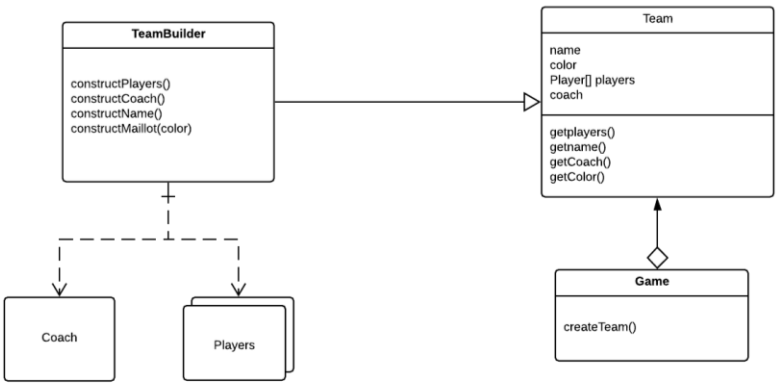
Nous allons créer une classe TeamBuilder qui fournit les différentes étapes de la construction de l'équipe. La classe Director qui est Game car c'est elle qui appelle la méthode créerTeam et définit l'ordre dans lequel appeler les étapes de construction, afin que vous puissiez créer et réutiliser des configurations spécifiques de produits. Le produit c'est justement l'équipe (la classe Team).

Et la classe Client qui utilise le Director et le Builder.

- Remarque : Si on prévoit différentes façons de création d'équipes selon la stratégie du coach par exemple (une équipe qui contient plus de défenseurs que d'attaquant qu'une autre) on doit créer un autre Builder, d'où la nécessité d'ajouter une interface Builder, qui sera implémenté par les Builders concrets.

Diagramme des classes :

Voici un diagramme simplifié de la solution :



2. Observateur :

Problème :

Après avoir effectué une analyse approfondie du code, nous avons remarqué qu'il y a un fort couplage entre les différentes classes du système, le fait de faire un changement au niveau d'une classe, implique d'en changer d'autres. Ces classes doivent donc être notifiées de chaque changement effectué, le système actuel ne respecte pas ce principe, les classes ne sont pas synchronisées entres elles, comme titre d'exemple les changements de position au niveau de la classe ballon ne sont pas pris en compte au niveau des classes Player.

Solution proposée :

Pendant le jeu les classes Player Gardien ainsi qu'arbitre, sont censées être à jour par rapport à la position du ballon. Ainsi tout changement de position au niveau de la classe ballon nécessite une mise à jour au niveau des 3 autres classes.

Le patron observateur est donc appliqué, il définit une relation 1 à n entre la classe ballon et les classes Player et Arbitre de façon que lorsque le ballon change d'emplacement, les autres classes seront notifiées et modifiées automatiquement.

Le ballon sera observé par les Player qui seront notifiés à chaque fois que le ballon change d'état, cela permet par exemple aux joueurs de savoir quand appeler la méthode 'passer'(un joueur ne pourra pas déplacer un ballon lorsque sa distance n'est pas 0), le ballon est donc le sujet et les joueurs et l'arbitre sont les observateurs.

Les participants :

Observer : c'est une interface qui va contenir une méthode update () qui sera appelé lors du changement de la position du ballon.

Concrete Observer : les classes Player, Arbitre qui vont maintenir une référence au ballon, elles implémentent l'interface Observer autrement dit la méthode update, pour mettre à jour la position du ballon qu'ils observent à chaque changement.

Subject : elle maintient une trace des Observateurs, responsable de l'ajout et retrait des Observateurs.

Concrete Subject :la classe ballon, c'est la classe observée, elle envoie une notification à chaque changement de sa position.

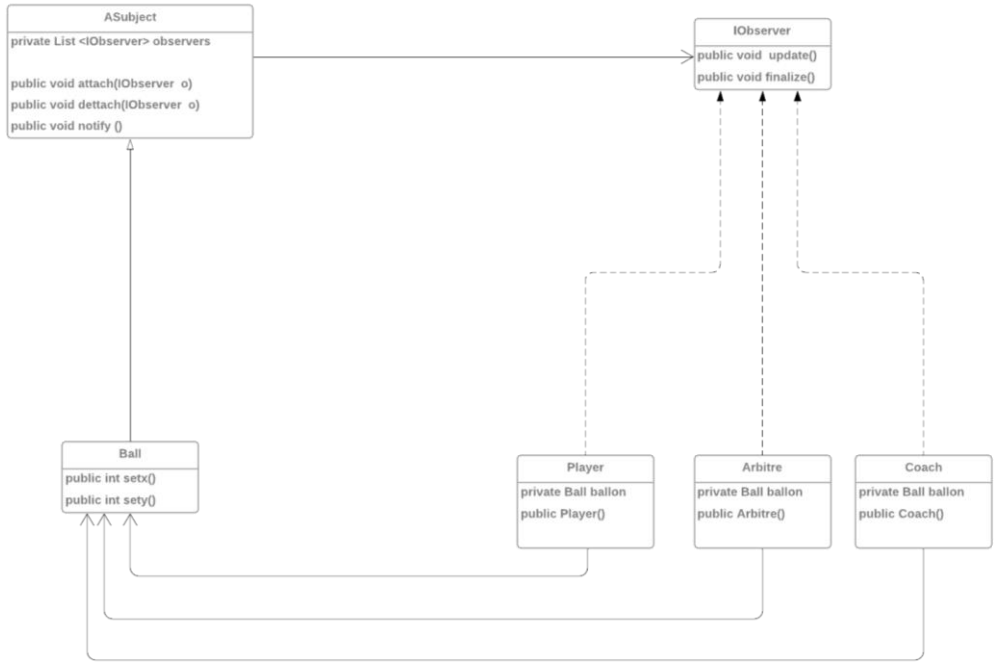
De même pour le cas suivant :

Les entraineurs des équipes peuvent hurler pendant le match pour demander à leurs joueurs d'avancer chacun à sa manière et cela selon son rôle dans l'équipe, Attaquant défenseur et gardien doivent être donc notifiés à chaque hurlement du coach.

Le coach sera donc observé par les Player (défenseur attaquant et gardien) et seront notifiés à chaque hurlement. Le coach jouera le rôle du sujet concret et les joueurs sont les observateurs.

Diagramme des classes :

Voici un diagramme simplifié de la solution :



1. Composite:

Problème :

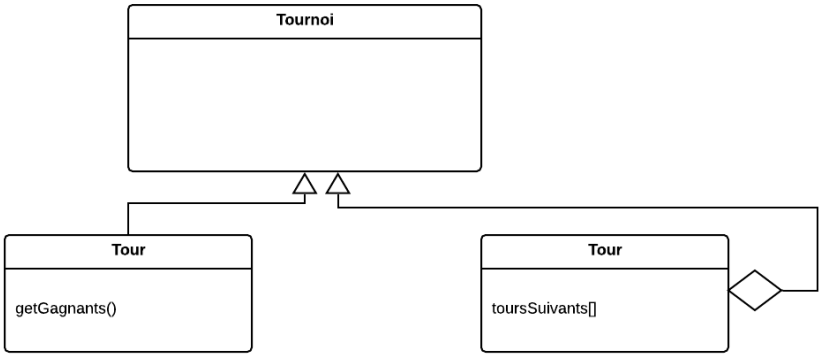
Dans le premier Livrable du TP, nous avons cité que nous allons travailler avec la classe Tournoi qui est composé de plusieurs tours (pour éliminer les classes Begin, SemiFinel,..) .

La solution comme elle est présentée nous pose un problème en cas où on voudra rajouter un tour (16 équipes par exemple comme la coupe du monde), donc nous serons amenés à modifier tout le code.

Solution proposée :

Vu que les tours sont présentés de façon hiérarchique, un tour qui contient 8 équipes se compose en deux groupes de 4 équipes. Dans chaque groupe de 4équipes, 2 équipes passent au tour suivant (cela ressemble à un arbre). Ce qui nous mène à appliquer la structure de composite pour créer la structure du tournoi. Dans ce cas la feuille (Leaf) est le match, le composite est le tour.

Diagramme des classes :



- **Bridge : (** Il n' y a pas un problème identifié, c'est juste en raison de réutilisabilité et de maintenabilité**)**
  Le patron Bridge aussi peut être utilisé comme patron de structure, car on remarque que dans l'interface graphique les joueurs sont représentés comme étant des triangles, et le ballon comme étant un cercle (les deux sous forme de shape). Si on veut améliorer l'interface et les visionner sous forme de vecteurs, ou 3D on trouvera un problème, car l'implémentation n'est pas séparée de l'abstraction.
  Donc on peut créer une classe Implémentation qui aura comme implémentations concrètes les classes ImplémentationAvecShapeSimple, ImplémentationVecteur et Implementation3D. Pour l'abstraction on aura les objets qu'on veut représenter tel que le ballon, les joueurs, le stade…

III. Conclusion :

Après avoir identifiés les problèmes et les avoir réglés avec les patrons GRASP et les patrons GoF, nous remarquons que le système est devenu beaucoup plus simple et plus réutilisable. Ce que nous venons de faire comme solutions sera amélioré dans le prochain livrable avec le patron MVC .