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

ESI 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)

LES PATRONS GANG OF FOUR

TP PDC : Livrable2

Réalisé par

- Zerrouk Madjda
- Abdiche Fatima Zahra

Proposé par

- Mme. N.Bousbia

DECEMBER 19, 2018

PROMOTION 2015

2CSSIL1

Table des matières:

I. Introduction :

Dans le premier livrable, on a reconstruit le système donné en utilisant les patrons GRASP, ces derniers sont essentiellement des principes théoriques et règles abstraites qui renforcent l'approche orientée objet tous comme les principes SOLID ou LoD mais ils ne représentent pas des solutions adaptées pour une conception complète et finale.

Pour cette raison les patrons de conception ou Design Patterns ont été créés. Ils représentent un ensemble de solutions prouvées pour régler des problèmes liés à la conception et qui occurrent fréquemment.

Il existe plusieurs ressources qui définissent et structurent ces patrons selon le type de problème, l'intention de l'utilisation et ses conséquences. Parmi ces ressources on cite les Patrons de GoF qui sont groupés sous trois catégories :

- Des patrons de création
- Des patrons structurels
- Des patrons comportementaux

Donc afin d'ajuster le système existant et éventuellement fixer les parties instables ou mal-conçues, on va exploiter quelques patrons de conception de GoF en précisant le problème trouvé, le patron utilisé et la raison derrière cette utilisation.

Objectifs

L'utilisation de ces patrons nous permettra d'avoir :

- Une application **Maintenable.**
- Avec des modules **Réutilisables.**
- Et un système ouvert à **l'extensibilité.**

Tout en respectant les principes fondamentaux de l'Orienté Objet.

II. Identification et résolution des problèmes :

Dans cette partie, nous allons recenser les problèmes de notre conception (proposée dans la première partie), qui peuvent être résolu en utilisant les patrons de GoF ou bien pour améliorer ce qui a été déjà réalisé.

i. L'instanciation des cases :

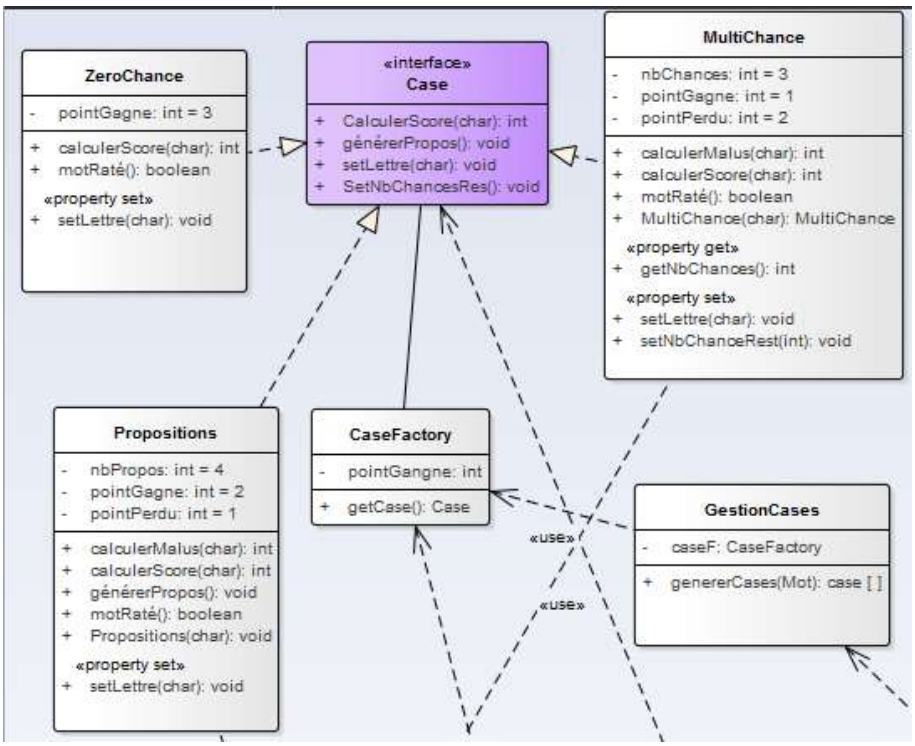
Puisque les mots sont constitués à chaque fois de différents types de cases, on aura pour chaque session un grand nombre d'objet case dont on n'aura pas besoin de leurs identités.

Pour réduire le nombre d'instance, il suffit d'instancier chaque type une seule fois et se servir après pour fournir des instances virtuelles de ce type lors de la création d'un mot.

Et cela est assuré par le patron **Poids-mouche (Flyweight).** Pour appliquer ce patron, il faut distinguer entre les états intrinsèques (pour le stocker avec l'objet) et extrinsèques (qui vont être calculés au moment de l'exécution).

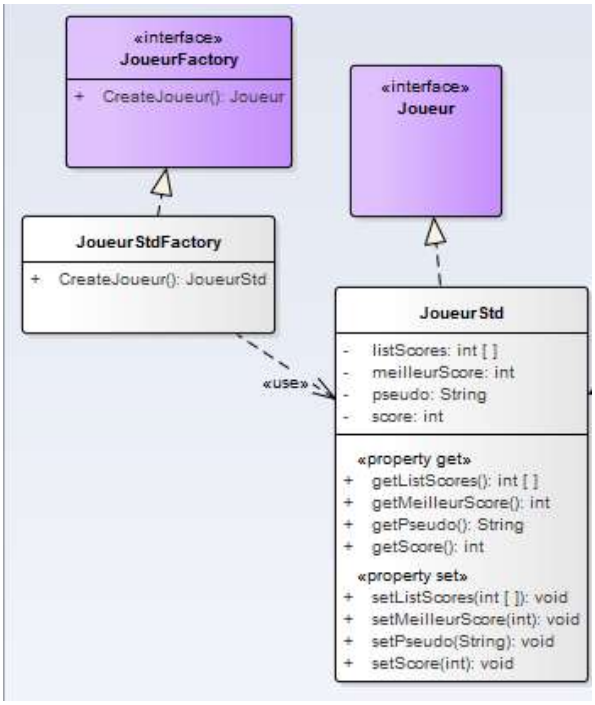
- Case sera le flyweight ses dérivées seront les concret flyweight .
- On ajoute une classe CaseFactory qui représente notre flyweightFactory.
- Le tableau ci-dessous résume les états intrinsèques et extrinsèques des flyweights.

Classe	états intrinsèques	états extrinsèques
MultiChance	nbChance pointGangne pointPerdu	lettre, score, NbchanceRes
ZeroChance	pointGangne	lettre, score
Proposition	nbProposition pointGangne pointPerdu	lettre, score, propo



ii. Création de joueur :

En pensant à l'évolution de système, on aura probablement des différents types de joueurs, donc vaut mieux avoir une interface pour la création de joueur mais on en délégant l'instanciation aux sous classes, en d'autres termes on les laisse le choix des classes à instancier. Ce qui fait appel au patron **Fabrication (Factory Method)**



- *JoueurFactory* qui sera notre Creator : une interface qui contient la méthode de fabrication : *CreateJoueur*
- JoueurStd Factory qui sera le Creator : elle implémente Joueur Factory et donc la méthode *CreateJoueur .*
- *Joueur* qui est le Product
- *JoueurStd* qui est le Concret Product

iii. La création d'un mot :

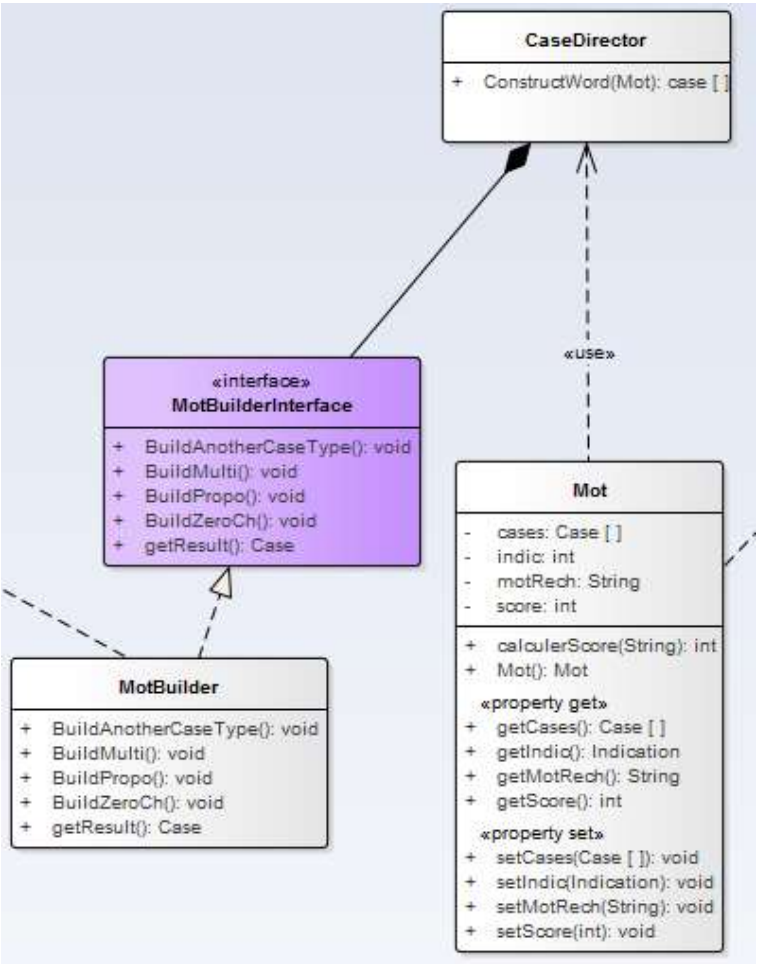
Durant une session du jeu « Le pendu », 10 mots sont créés aléatoirement basant sur une composition de différents types de cases (Multichance, proposition, Zerochance ... etc.)

Cette composition est générée dynamiquement lors l'exécution du jeu et la construction d'un objet assit complexe comme les mots doit être séparée de sa représentation afin de pouvoir créer différentes formes avec un processus unique.

Pour cette raison on opte pour l'utilisation d'un Builder ou les différentes parties construites sont les cases (i.e. Type de case + contenu + traitement approprié).

Donc on aura une Interface **MotBuilderInterface** *(Builder)* qui contient des méthodes abstraites pour l'instanciation des cases selon l'ordre et le type choisis aléatoirement par *Director.*

Dans l'autre côté, les classes *ConcreteBuilders* comme **MotBuilder** … etc ; vont redéfinir et implémenter les méthodes précédentes (BuildPropo, BuildMulti,



BuildZeroCh ...) selon le besoin du Director. Par suite, on peut avoir des produits (i.e. représentations des mots) différents et construits dynamiquement selon la demande.

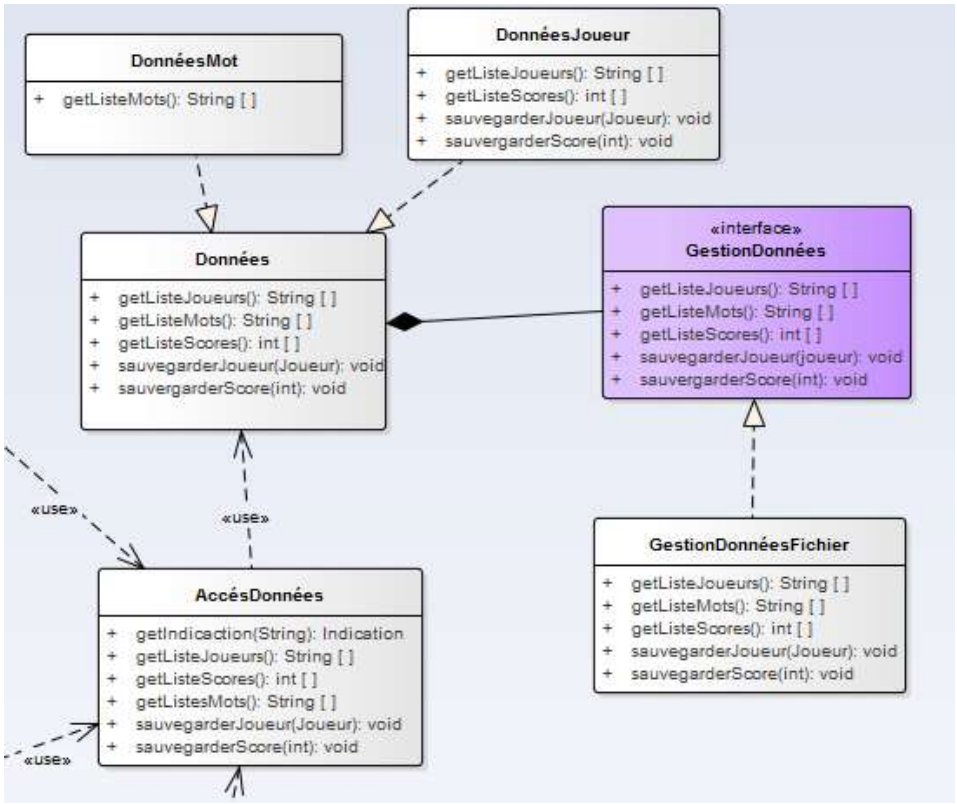
iv. L'accès aux données sauvegardées :

Selon le système existant, la récupération ainsi que la sauvegarde des données (des joueurs ou des mots ...) sont manipulées par des fichiers textes associés à notre application.

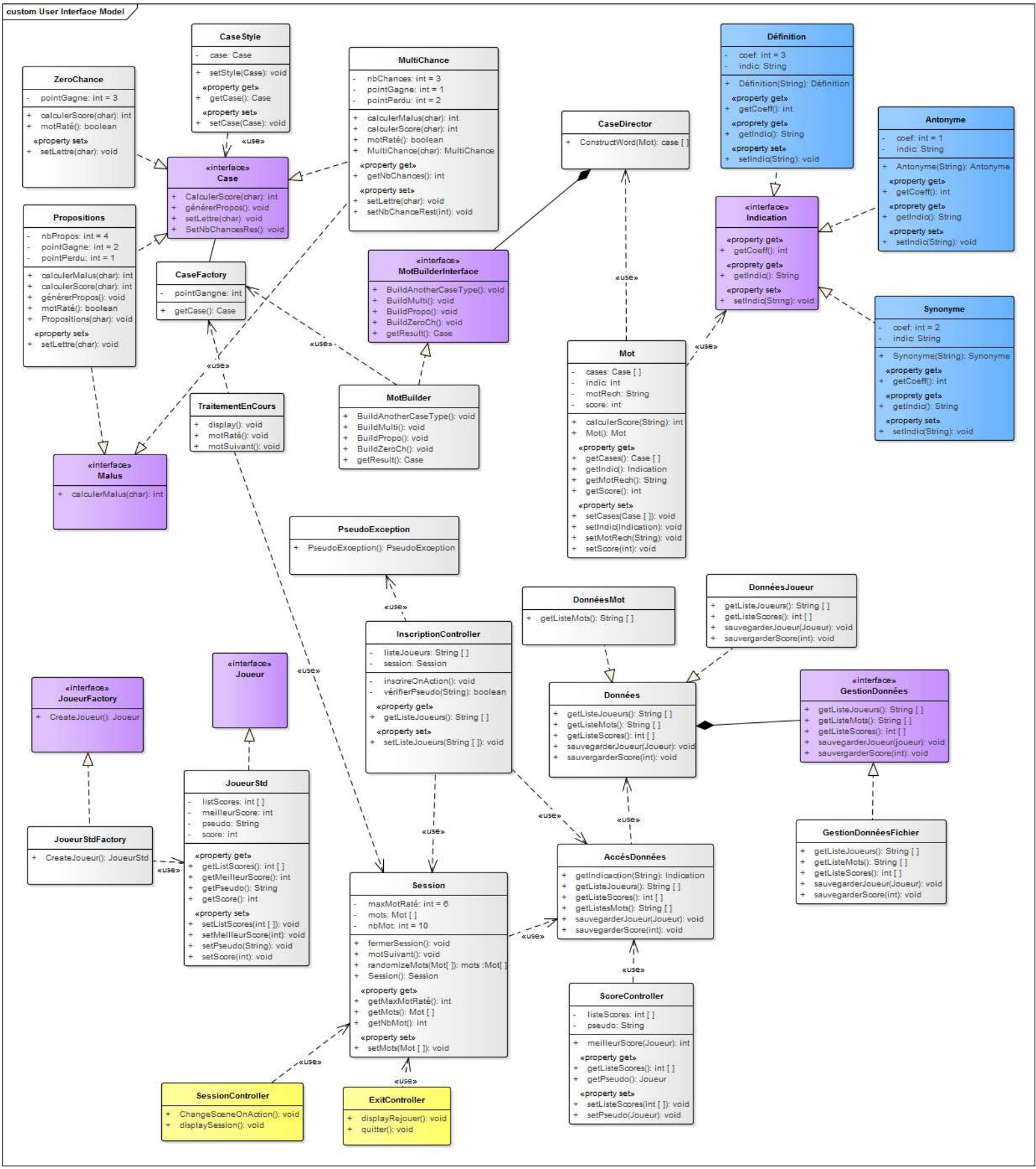
Si on veut améliorer la gestion des données en utilisant une BDD hébergée sur un serveur externe ou par des fichiers cryptés qui nécessitent un accès spécial ; ou bien si on ajoutera un nouveau type de données pour la sauvegarde (par exemple des informations concernant une session non-terminée), alors on doit ajouter autant de classes -*qui implémentent / héritent la classe* **gestionDonnées**- que le nombre des fichiers multiplié par leurs types. Ce dernier contredit le principe de la conception Orientée Objet qui indique « la favorisation de la composition sur l'héritage ».

Afin de rendre notre système plus flexible et maintenable, on exploite **le patron de structure Bridge (monteur)** qui sert à séparer l'implémentation de l'abstraction. Donc on aura une classe abstraite **Données** hérité par des classes filles spécialisées selon le type de fichier *(pour le système actuel on possède des données concernant les joueurs et autres concernant les mots utilisés seulement).*

Puis on ajoute une interface **GestionDonnées** qui contient les différents traitements reliés aux informations sauvegardés et que autres classes (i.e. *GestionDonnéesFichier*, *GestionDonnéesSQL* … etc.) implémentent selon le type de données et la nature des traitements.



III. Nouveau diagramme de classe :



IV. Conclusion :

L'analyse de la conception de la première partie nous a permis d'identifier des problèmes qui ont une relation avec la réutilisation, l'évolution de système ainsi que l'optimisation de l'espace mémoire et aussi la génération dynamique des objets. La résolution de ses problèmes nous a conduit à appliquer quelques patrons de Gang Of Four afin d'accomplir les failles des patrons GRASP.

Donc, dans le but d'avoir un logiciel fiable et de qualité, il faut faire une conception efficace et cela nécessite de prendre en considération l'utilisation des patrons de conception qui permettent d'accélérer le processus de développement par l'apport de paradigmes de développement à l'efficacité éprouvé pour une problématique donnée.

Dernier point et non des moindres, ces patrons de conception permettent une meilleure communication entre développeur en établissant des termes et noms d'interactions logicielles connus et bien compris de tous.