Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire **2**
Étudiant(s)	Bui Quach, Catarina Castro, Ugo Costantini, Abderraouf Haddouni
Code(s) permanent(s)	COSU16129907
Cours	LOG121
Session	E2020
Groupe	02
Professeur	**GALARNEAU Benoit**
Chargés de laboratoire	**ALCHALBI Bilal**
Date de remise	2020-06-13

1 INTRODUCTION

Le sujet de ce laboratoire #2 est la réalisation d'un cadriciel pour un jeu de dés.Un cadriciel est un ensemble de classes que l'on peut étendre afin d'implémenter une ou plusieurs variantes de notre jeu de dés. L'objectif principal ici est de développer ce cadriciel afin de produire des classes réutilisables et d'avoir une bonne conception facilement extensible.

Ce laboratoire a plusieurs objectifs : Le premier objectif est de bien comprendre le rôle du patron Stratégie, Fabrique et Méthode Template dans le cadre d'un logiciel. Le deuxième objectif est d'apprendre à concevoir une application facilement extensible, afin qu'elle puisse être facielement evoluable et maintenable. Enfin, un dernier objectif est de comprendre l'utilité des tests unitaires, ainsi que leur utilisation et leur implémentation dans un logiciel.

Ce rapport portera notamment sur la conception de l'application, tout d'abord en présentant les choix et responsabilités des classes. Le diagramme de classes UML de l'application sera ensuite présenté. Puis, les faiblesses de conception du logiciel seront énumérées et détaillées. 2 diagrammes de séquence, détaillant l'implémentation du patron Stratégie ainsi que du patron Fabrique seront montrés. Enfin, 2 décisions importantes de l'application seront détaillées, l'implémentation du calcul du score à chaque tour et du vainqueur ainsi que la gestion des joueurs. Nous conclurons ensuite en rappelant les objectifs du laboratoire, en synthétisant sur la conception choisie et sur les pistes possibles d'amélioration.

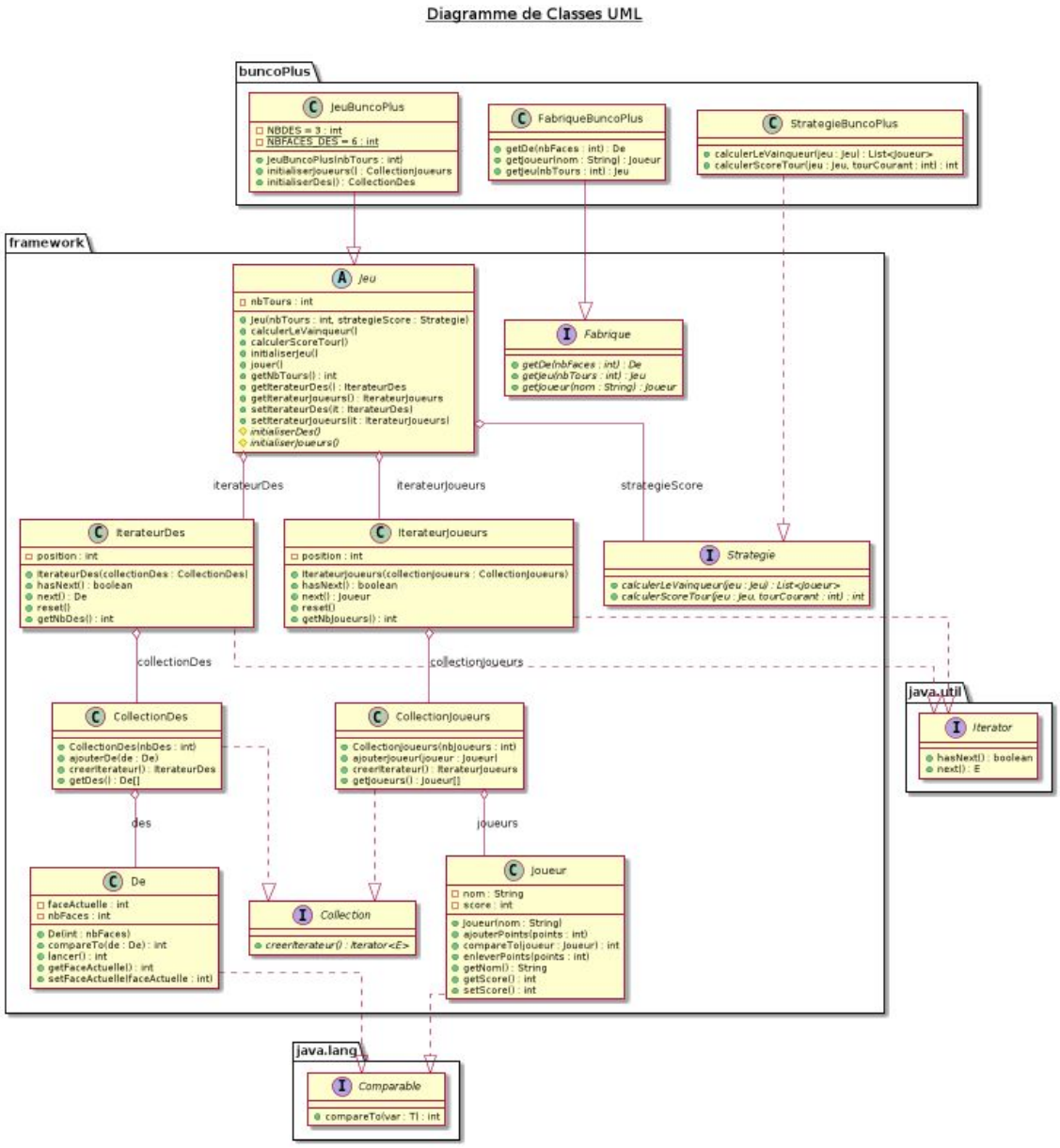
2. CONCEPTION

Classe	*Responsabilités*	*Dépendances*
Joueur	– Classe qui sert à instancier les joueurs. Un joueur possède un nom ainsi qu'un score.	– Comparable
IterateurJoueurs	– Classe qui sert à itérer sur chaque joueur de la partie. L'itérateur possède une collection de joueurs.	– Iterator – CollectionJoueurs
CollectionJoueurs	– Classe qui contient les joueurs du jeu. Elle possèdes des méthodes pour ajouter un joueur ou créer un itérateur.	– Collection – Joueur – IterateurJoueurs
De	– Classe pour instancier les dés. Elle a un nombre de faces et une face actuelle. La méthode lancer permet de mettre un nombre aléatoire au face actuelle selon le nombre de faces que le De possède.	– Comparable
CollectionDes	– Classe qui implémente Collection pour contenir les dés. Elle possède une méthode pour ajouter un dé. Elle crée aussi l'itérateur pour sa collection.	– Collection – De – IterateurDes
IterateurDes	– Classe pour itérer sur les dés. Elle possède une collection de dés. Dans cette classe, on a des méthodes pour parcourir la collection. Elle a aussi une méthode pour réinitialiser la position au début.	– CollectionDes – Iterator

Collection	- Interface qui contient la signature de la méthode qui permet de créer les itérateurs.	Aucune
Comparable	- Interface qui contient la signature de la méthode qui permet de comparer 2 objets.	Aucune
Jeu	- Cette classe abstraite oblige les classes hésitantes à initialiser un jeu avec un certain nombre de joueurs et un certain nombre de dés. Aussi, Elle permet d'initialiser un jeu et de jouer durant un certain nombre de tours. Enfin, cette classe possède une stratégie de base permettant le calcul du nombre de points gagnés par tour et une autre permettant de déterminer le vainqueur de la partie.	- IterateurJoueurs - IterateurDes - Strategie
StrategieDeBase	- Cette classe, faisant partie du framework, implemente une stratégie concrète d'un jeu classique, un jeu de base. Il s'agit d'une stratégie pré-intégrée au framework.	- Jeu - Strategie - IterateurJoueurs - IterateurDes
JeuBuncoPlus	- Cette classe implémente concrètement la classe abstraite Jeu. Elle contient la logique permettant d'intialiser les dés et joueurs pour le jeu buncoPlus, et appeler la straþegie concrète.	- Jeu - StrategieBuncoPlus - FabriqueBuncoPlus - IterateurDes - IterateurJoueurs
StrategieBuncoPlus	- Cette classe implémente la stratégie concrète pour joueur au jeu buncoPlus (déroulement, calcul du score à chaque tour), ainsi que le calcul concret du vainqueur.	- Strategie - JeuBuncoPlus - IterateurJoueurs - IterateurDes
FabriqueBuncoPlus	- Cette classe implémente la fabrique abstraite permettant de créer un jeu buncoPlus, un joueur, ou encore un dé.	- De - Joueur - JeuBuncoPlus

2.2 Diagramme des classes

Le diagramme de classes est également disponible dans le fichier diagramme_classes.png



2.3 Faiblesses de la conception

Une des faiblesses est qu'il y a des classes qui ne sont pas immuable comme notre classe Jeu. Une classe immuable nous permet d'avoir une classe qui est plus facile à implémenter et utiliser qu'une classe qui n'est pas immuable puisqu'elle ne change pas une fois construite. Le problème avec notre classe Jeu, c'est qu'elle possède des modificateurs pour nos membres iterateurDes et iterateurJoueurs (setIterateurDes et setIterateurJoueurs). Elle n'est donc pas immuable.

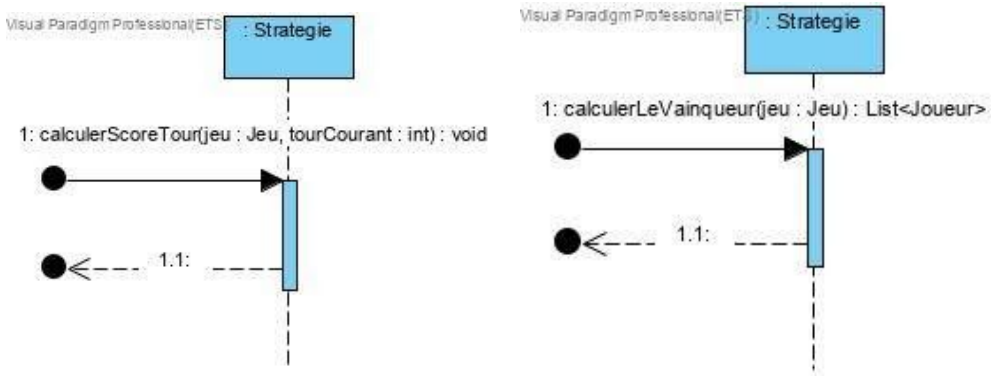
Une solution possible est d'ajouter l'iterateurDes et l'iterateurJoueur dans le constructeur de la classe Joueur. Ensuite, on enlève nos méthodes setIterateurDes et setIterateurJoueurs. Finalement, la classe Joueur n'aura pas de modificateur pour nos membres et elle va donc être une classe immuable. C'est possible, car notre jeu demande tous les informations de nos dés et joueurs avant d'initialiser la classe Jeu.

Une autre faiblesse concerne la conception du framework : En effet, on peut y remarquer de la duplication de code au niveau des collections : Les collections héritent en effet d'une interface Collection, qui leur permet de créer un Itérateur. En revanche, les méthodes pour ajouter un Joueur ou ajouter un Dé, bien qu'identiques, ont 2 noms différents et 2 implémentations différentes, car le paramètre d'entrée est différent. On se retrouve donc avec 2 classes qui ont les mêmes attributs et méthodes, mais seulement avec un nom différent et des types de paramètres/retour différents.

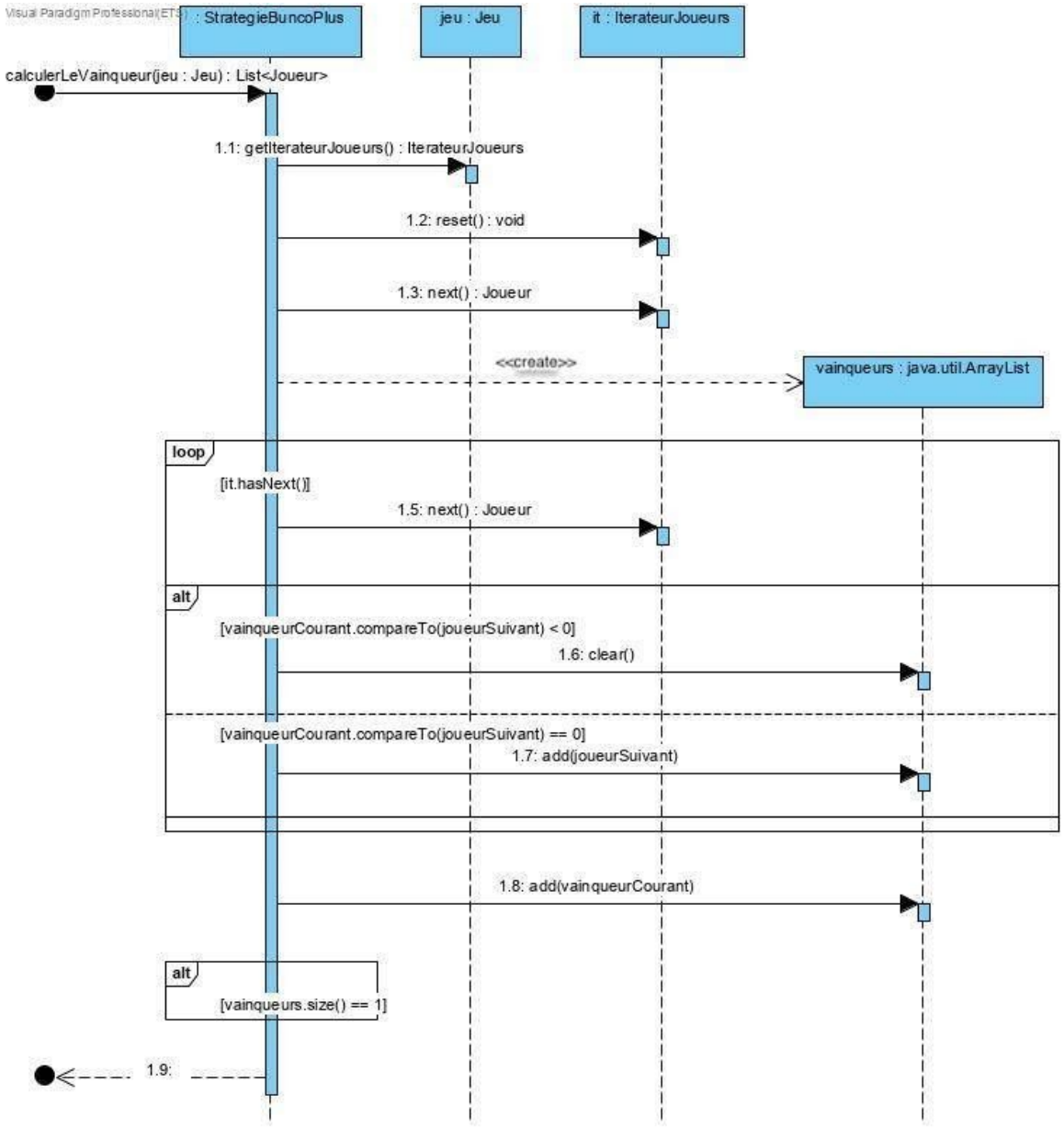
Une solution ici serait d'utiliser ce qu'on appelle la Généricité (Collection<T>), qui permettrait à notre interface Collection de dire aux classes concrètes que le type de retour peut-être n'importe quoi, tant qu'il est spécifié lors de la déclaration de la classe concrète (implements Collection<Joueur>). Ce système permettrait d'avoir des noms de méthodes similaires entre les collections, et de rajouter des méthodes dams l'interface. On peut même pousser la logique plus loin, en imaginant que l'attribut qui contient les valeurs de la collection serait lui-même générique (T[] valeurs).

2.4 Diagramme de séquence (uml)

2.4.1. Exemple qui illustre la dynamique du patron stratégie

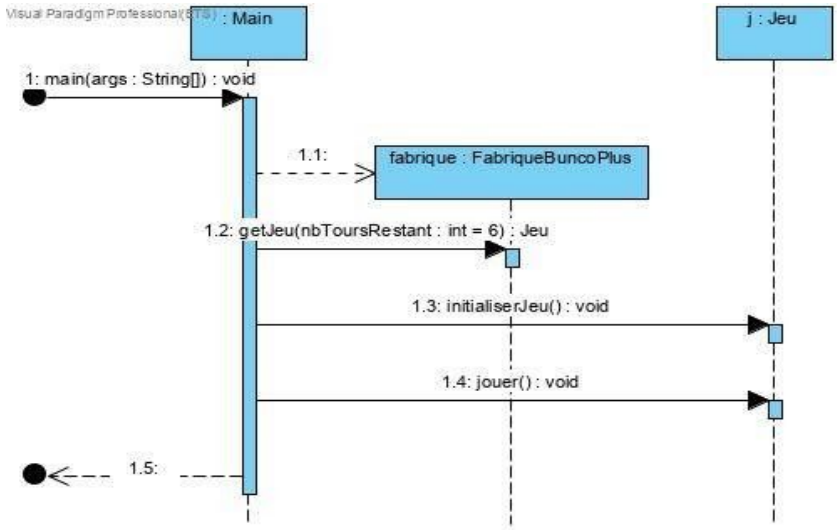


L'interface Strategie est composée de deux méthode, soit calculerScoreTour() et calculerLeVainqueur().

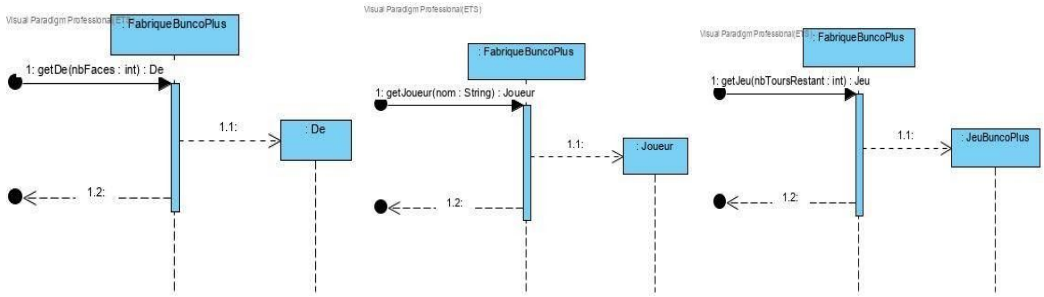


Prenons ici l'appel de la methode calculerLeVainqueur() de StrategieBuncoPlus. On commence par instancier un itérateur de joueur de la partie en question, pour ensuite le reset(), soit mettre la position à 0. Aussi on instancie un nouveau joueur, au premier de joueur de l'itérateur de joueurs et on crée un Arraylist de joueurs. Lorsqu'on rentre dans la boucle, on instancie un nouveau joueur avec le deuxième joueur dans l'itérateur de joueur. On va par la suite compareTo() le premier joueur au deuxième, pour déterminer lequel a le plus de points pour l'affecter au vainqueurCourant. Sinon, si leur nombre de points sont égaux, nous allons l'ajouter à la liste de vainqueurs. Ceci va être répété tant qu'il y aura des joueur dans l'itérateur.

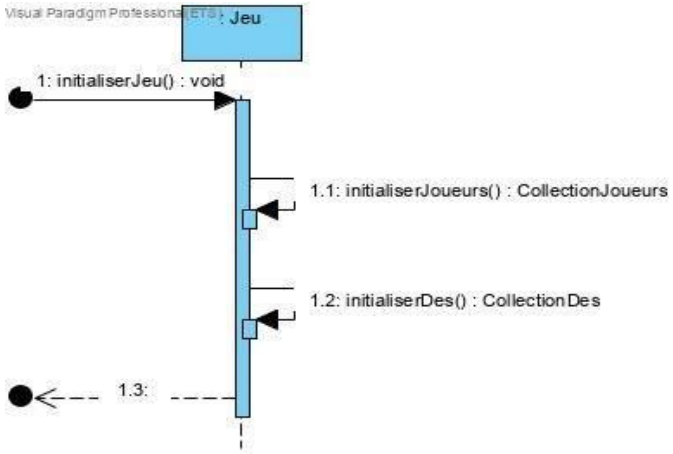
2.4.2. Exemple qui illustre la dynamique du patron Fabrique



Ici nous pouvons voir le déroulement du main qui sert à instancier une fabrique pour initialiser le jeu.



Nous remarquons ici que la classe FabriqueBuncoPLus est composé de trois méthode, soit getJoueur(), getDe() et getJeu().



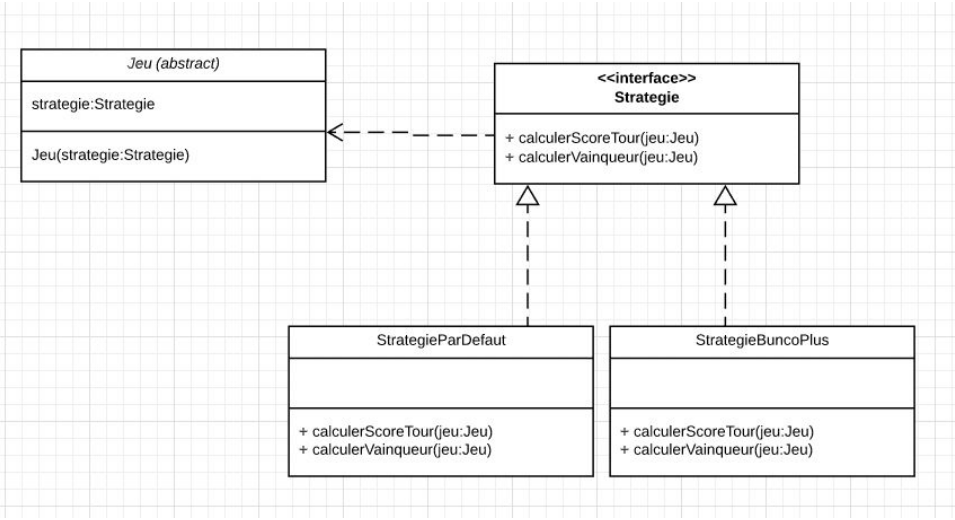
La méthode initialiserJeu() de la classe Jeu, initialise la collection de joueurs et la collection de dés.

3 Décisions de conception/d'implémentation

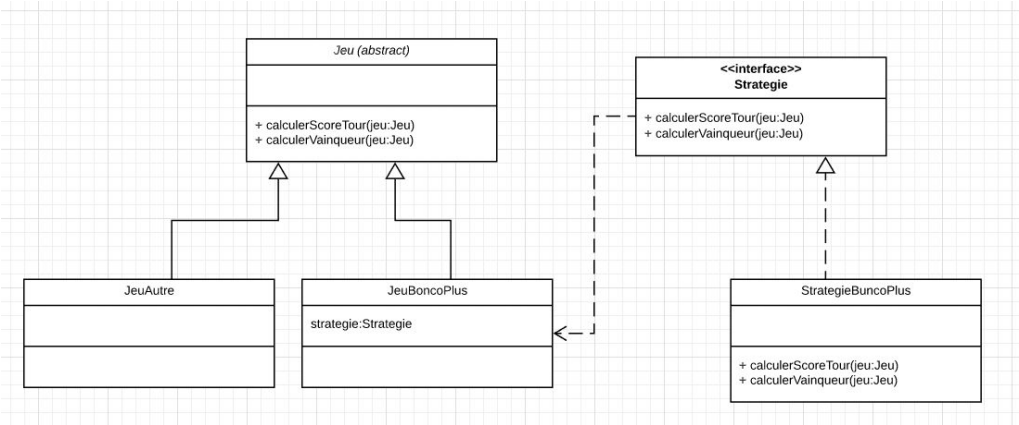
3.1 Décision 1 : L'implantation du calcul de points et du vainqueur

- **Contexte**: En décortiquant l'énoncé, deux choix s'offraient à nous quant à l'implémentation des méthodes *calculerLeVainquer()* et *calculerScoreTour()*. Étant déclaré dans la classe abstraite Jeu, on pouvait soit définir une stratégie pour l'implémentation de ses méthodes ou les définir directement dans la classe *Jeu*.

- **Solution 1**: Créer une nouvelle stratégie qui héritera de l'interface *Strategie* et qui devra implémenter les méthodes *calculerLeVainquer()* et *calculerScoreTour()*. De cette façon, la classe jeu aura comme attribut privé une stratégie qui sera définie à l'instanciation. Ainsi, la classe jeu délaisse la responsabilité de l'implémentation et offre aux classes filles la possibilité de changer de stratégie si celle de base ne leur convient pas. Par ailleurs, cette solution oblige les classes héritantes à être définies avec une stratégie de base.



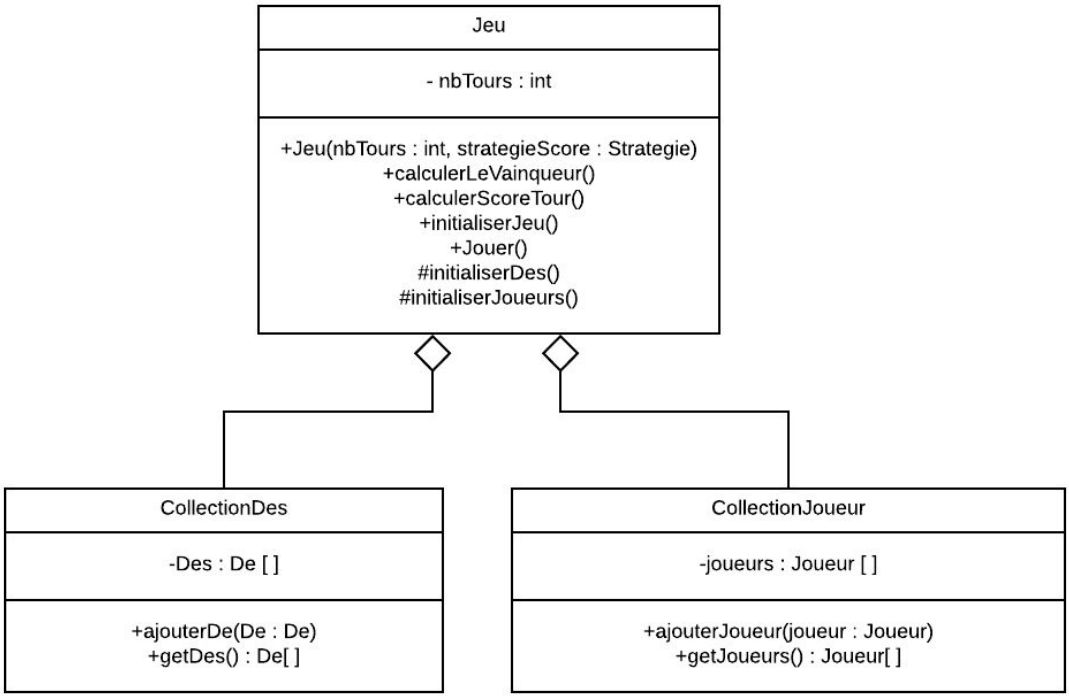
- **Solution 2** : Les méthodes *calculerLeVainquer()* et *calculerScoreTour()* seront définies et implémentées dans la classe *Jeu*. Par ailleurs, chaque classe héritante sera responsable de redéfinir cette méthode en appelant une nouvelle stratégie. Ainsi, aucune stratégie ne sera créée à l'instanciation de la classe Jeu. Néanmoins, chaque classe qui étend de Jeu et qui veut établir une stratégie de calcul de point devra avoir un nouvel attribut. De plus, chaque nouvelle classe devra implémenter l'interface Strategie. On ne bénéficiera donc plus de tout l'avantage de l'héritage puisque cet attribut sera redondant pour plusieurs classes.



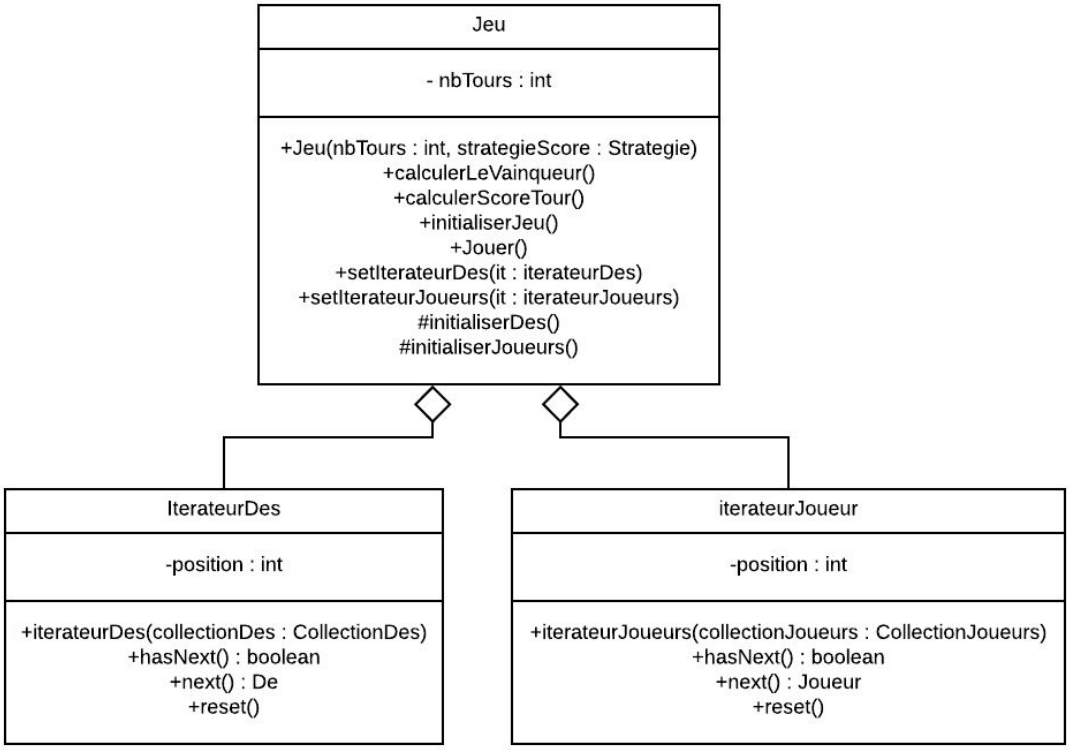
- **Choix de la solution et justification :** Nous avons choisi la solution 1 puisque la notion d'héritage est mieux exploitée. En effet, à notre sens, tous les jeux ont une stratégie de calcul de points et du vainqueur. Toutes les classes qui étendent *Jeu* auront un attribut de type *Strategie*, ils pourront redéfinir cet attribut s'ils veulent utiliser une autre méthode de calcul. Aussi, de cette façon, seules les classes qui implémentent l'interface *Strategie* seront responsables de l'implémentation des méthodes *calculerLeVainquer()* et *calculerScoreTour()*. Enfin, cette solution permet une meilleure extensibilité de la classe Jeu et du squelette proposé.

3.2 Décision 2 : Gérer les joueurs

- **Contexte**: Au moment où nous devions choisir comment gérer les joueurs, deux options s'offraient à nous. On pouvait dans le premier cas, gérer une collection de joueur directement dans la classe Jeu où bien avoir un IterateurJoueurs dans la classe Jeu.
- **Solution 1**: Notre première solution était que la classe Jeu contient une collection de joueurs de la classe CollectionJoueurs. Dans ce cas, c'est la classe jeu qui devra gérer les joueurs. En effet, ce sera la responsabilité à la collection de joueur d' itérer sur chaque joueur. Cette solution semble efficace, mais ne respecte pas le principe de l'encapsulation et ne respecte pas les contraintes de conception demandées.



- **Solution 2** : Notre deuxième solution était que la classe Jeu contient un itérateur joueur, qui lui contient une collection de joueurs. Ainsi c'est l'itérateur qui va gérer la collection de joueurs et non le jeu. Dans ce cas, on respecte le principe de l'encapsulation, étant donné qu'on ne connait pas la collection, mais on va tout de même itéré sur elle. En effet, la classe ItérateurJoueurs devra être instanciée avec une collection de joueurs sur lesquelles on pourra itéré. Cette façon de faire, en plus de respecter l'encapsulation, suit mieux le squelette qui nous a été proposé.



- **Choix de la solution et justification :** Nous avons décidé de choisir la solution 2, étant donné qu'en utilisant cette méthode, l'itérateur permet d'abstraire comment accéder à la collection de joueurs, ainsi on ne touche pas directement à la collection de joueurs. Aussi on peut modifier comment la liste marche, si c'est un arraylist, un tableau etc.

4 CONCLUSION

En conclusion, les objectifs ont été complètement atteints. En effet, grâce à ce laboratoire, nous avons appris le rôle, l'utilité et tous les bénéfices qu'apportent les patrons Stratégie, Fabrique, Itérateur et Méthode Template. Ainsi, un squelette extensible d'un jeu de dés a été réalisé. Ensuite, Tout au long de la conception, nous avons gardé en tête le mot évoluable. Ainsi, les classes du paquet template rendent l'héritage facile tout en réduisant au maximum le couplage. Aussi, grâce au test unitaire, nous avons essayé le jeu avec différents dés et différents joueurs. Toutes les classes ont été testé. Notons par ailleurs qu'un Mockito a été réalisé pour la classe abstraite *Jeu*. Tout cela a renforcé nos connaissances pour la partie test.

Avec ces objectifs en tête, nous avons réalisé un projet avec plusieurs points forts et nous avons aussi remarqué quelques points faibles. Un des points forts du projet est comment nous avons gérer Bunco+. Avec la méthode "Stratégie", la classe jeu n'est pas responsable des méthodes calculerScoreTour() et calculerLeVainqueur(). De cette manière, ce n'est pas nécessaire de changer la classe jeu si l'on veut changer la façon de calculer les scores. Un autre point fort est l'encapsulation de nos collections de dés et de joueurs. Puisque que c'est notre itérateur qui contient la collection, nous ne touchons pas directement à nos collections.

Pour les points faibles, il y a des classes qui ne sont pas immuables comme notre classe Jeu. Nous avons des méthodes qui changent nos membres de la classe comme setIterateurDes() et setIterateurJoueur(). Il y a aussi du code qui se répète. Par exemple, nous avons les classes CollectionDes et CollectionJoueurs qui sont très similaires.

La porte de ce travail est grande. De nombreuses améliorations peuvent être faites étant donné qu'il s'agit là d'un squelette simple. L'ajout d'une interface graphique pourrait être intéressant. On pourrait voir l'interaction des différents joueurs et le lancer des dés. Il serait aussi intéressant d'appliquer d'autre patron de conception telle qu'observateur. Le jeu pourra ainsi être bonifié du temps. De cette manière, une partie pourrait durer un certain nombre de tours ou un certain temps ou les deux.