Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire **2**
Étudiant(s)	Hadrien Renault
	Myron Di Wei Ye
Code(s) permanent(s)	RENH23089909
	YEXM29049702
Cours	LOG121
Session	Automne
Groupe	2
Professeur	Sègla Kpodjedo
Chargés de laboratoire	Olivier Michaud
Date de remise	12 Novembre 2019

1 Introduction

Ce laboratoire/logiciel, même si il semble simple à première vue, implémente de nombreux concepts fondamentaux en programmation. En effet, l'utilisation d'un cadriciel pour permettre de réutiliser du code est un concept clé de la programmation objet. Le logiciel nécessite également l'utilisation de 3 patrons clés, le patron itérateur, stratégie et méthode template. Puis, pour vérifier le fonctionnement du code, des tests unitaires sont obligatoires, tests qui facilitent grandement la tâche pour la vérification du fonctionnement du code.

La présente solution n'est pas parfaite, mais elle répond aux objectifs globaux de l'extensibilité d'un cadriciel. On y retrouve une bonne implémentation du cadriciel et des divers patrons. En effet, Le patron template méthode permet de faire l'exécution d'un cadriciel de jeu de dés. Le patron stratégie permet de spécifier le type et les règles de jeu. Le patron itérateur permet d'accéder et parcourir des collections de dés et de joueurs pour permettre des interactions avec les autres classes.

Premièrement, le rapport expliquera le choix des différentes classes et leurs dépendances aux autres classes. Par la suite, nous allons vous faire part de notre diagramme de classe ainsi que vous expliquer les faiblesses de conception qui pourraient en découler. Puis, nous allons vous montrer notre diagramme de séquences qui démontrent l'utilisation du patron stratégie. Notre deuxième diagramme de séquence portera sur le patron méthode template. Finalement, nous discuterons des décisions de conception et d'implémentation que nous avons faites et nous conclurons.

2. Conception

2.1 Choix et responsabilités des classes

Discutez du choix des classes et de la répartition des tâches. Vous pouvez utiliser un tableau qui résume vos classes importantes.

- *Responsabilités : utilisez des phrases descriptives, ne listez pas de méthodes*
- *Dépendances : les classes de votre application nécessaires au fonctionnement de la classe en question.*

Classe	*Responsabilités*	*Dépendances*
– Joueur	– Classe qui sauvegarde les informations d'un joueur.	– «Interface» Comparable
– De	– Classe qui sauvegarde les informations d'un de.	– «Interface» Comparable
– Fabrique	– Classe qui gère la création des joueurs et des des.	–
– «Interface» IStrategie	– Interface qui contient les méthodes en lien avec le comptage d'un jeu.	–
– CollectionDes	– Classe qui contiennent une liste de De et permet de parcourir ce liste.	– «interface» Collection – «interface» Iterator
– CollectionJoueurs	– Classe qui contiennent une liste de Joueur et permet de parcourir ce liste.	– «interface» Collection – «interface» Iterator
– JeuDe	– Class qui contient la logique et les fonctions du jeu. Incluant les méthodes de calculs du pointages. De plus, elle contient la liste des instances des joueurs et dés.	– CollectionDes – CollectionJoueurs – Joueur – Fabrique – «interface» IStrategie
– BuncoStrategy	– Classe qui implémente les règles spécifiques de Bunco pour calculer les points et les vainqueurs..	– «interface» IStrategie
– Demo	– Classe main qui permet de jouer au jeu	– JeuDe – BuncoStrategy
– DeTest	– Classe qui fera les test pour les valeurs négatives, un dé nul et la comparaison entre 2 dés pour voir	– De

	si l'un est supérieur, inférieur ou égal à l'autre.	
– JeuTest	– Classe qui fera les tests du changements de joueurs et du maintien du même joueur par rapport au pointage obtenu avec les dés et de l'exécution d'une partie.	– Bunco – JeuDe – De – Joueur
– JoueurTest	– Class qui fera le test du compareTo avec 2 joueurs dans le cas où il est inférieur, supérieur ou égale. De plus, comparer un joueur avec un joueur inexistant, ajouter les points négatifs à un joueur et le set du pointage.	– Joueur

2.2 Diagramme des classes

Voir le document L2_DIAGRAMME-DE-CLASSE_HRenault_MYe DANS LE FICHIER ZIP.

2.3 Faiblesses de la conception

Décrivez les faiblesses de votre conception et des solutions possibles pour remédier à ces points faibles. (maximum 1 page)

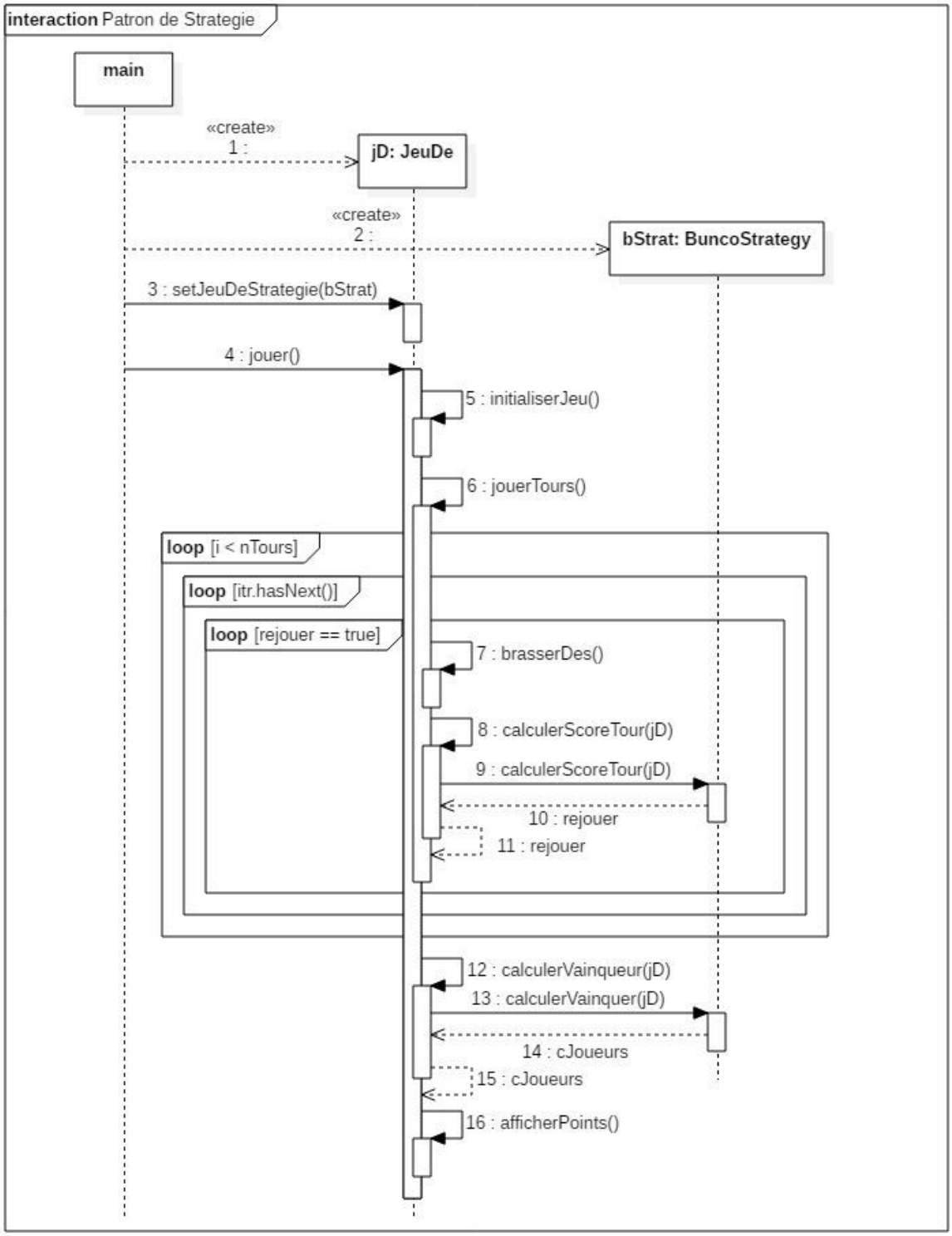
Le but du laboratoire était de créer un code extensible à plusieurs jeux de dés, puis d'implémenter une instance spécifique pour un jeu de Bunco en utilisant ce squelette. Notre première faiblesse serait d'avoir trop miser sur cette implémentation du Bunco. Notre squelette (JeuDe, Fabrique, etc.) n'est pas une implémentation directe du Bunco, mais a été beaucoup inspiré par les règles de ce dernier, et nécessiterait d'être re-travailler pour pouvoir implémenter la majorité de jeux de dés. En effet, notamment pour la méthode jouerTours(), on a implémenté une version assez spécifique d'un jeu de dé. Plutôt que de forcer les joueurs à jouer selon une logique spécifique dans un ordre spécifique ou les points sont affichés à chaque 'tour', il aurait été préférable de faire une méthode plus générale qu'on aurait implémenté plus spécifiquement dans une sous-classe spécifique à Bunco. Ainsi, notre cadriciel aurait été plus flexible et aurait nécessité moins de travail dans un cas ou on aurait à implémenter plusieurs autres jeux de dés.

Notre deuxième faiblesse de conception se situe au niveau de l'extensibilité des classes Joueur et De. En ce moment, celles-ci ne peuvent pas être étendues facilement afin de personnaliser les attributs propres aux Joueurs et aux Dés. Notre conception actuelle crée des joueurs ayant comme seul attribut un nom. Dans l'éventualité qu'un prochain jeu demanderait d'avoir une description ou tout autres type d'informations, notre classe Joueur ne sera pas facilement implémenter ces nouveaux attributs. La solution serait de créer des classes Joueur propres à chaque jeu par exemple JoueurBunco. Ces nouvelles classes hériteront de la superclasse Joueur et pourrons redéfinir leurs attributs dans leur constructeur. Grâce à cette solution, notre cadriciel serait plus extensible pour personnaliser le logiciel pour une variante particulière du jeu de dés.

2.4 Diagramme de séquence (uml)

2.4.1. Exemple qui illustre la dynamique du patron stratégie

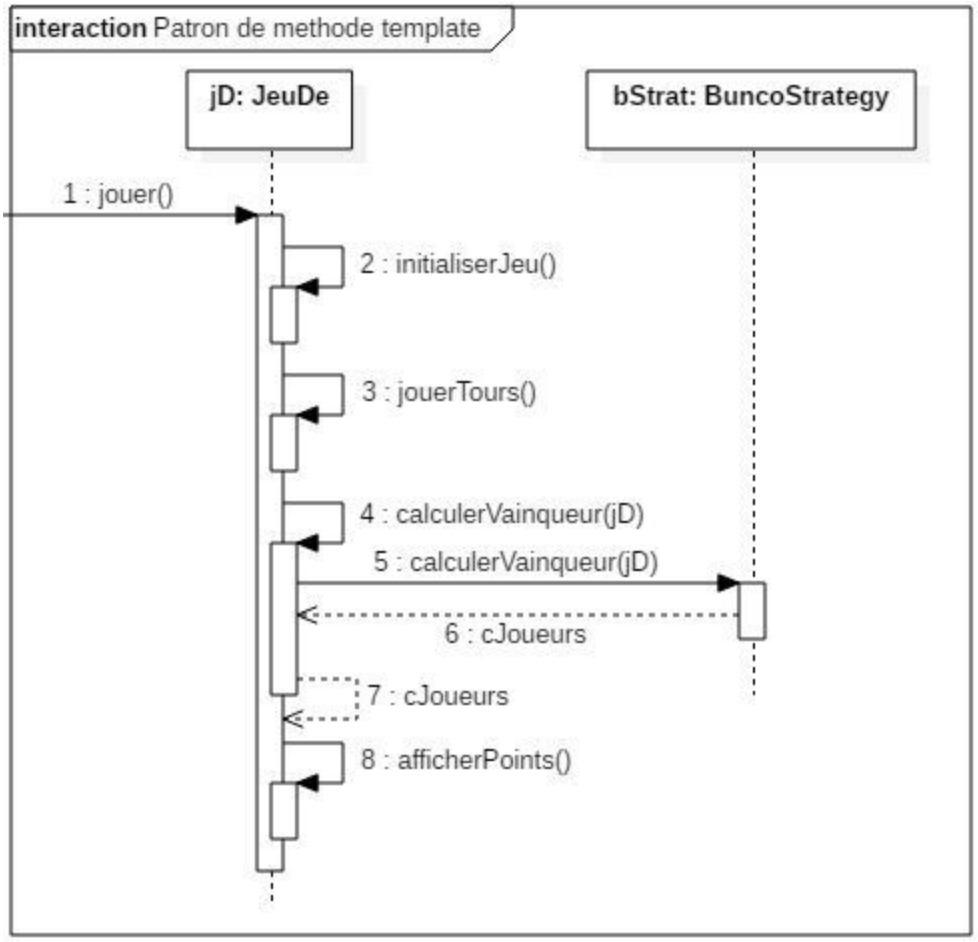
(Prochaine page)



L'application commence dans la classe de «Demo» qui crée un objet de «JeuDe» et de «BuncoStrategy». Ensuite, la méthode de setJeuDeStrategie() est appelée et on passe en

paramètre bStrat qui dira à la classe de JeuDe quel stratégie elle doit utiliser pour faire les calculs du jeu. On fait appel a la methode de jouer() pour commencer le jeu. La méthode initialiserJeu() va créer la liste des joueurs et des dés selon les paramètres que l'utilisateur va fournir. Selon ces informations, on peut commencer le jeu avec la méthode jouerTours() qui jouera toutes les tours du jeu avec l'aide des boucles. Le jeu fait appel à brasserDes() pour le lancement des dés et à calculerScoreTour() qui utilisera la stratégie qu'on lui avait fourni au début. Elle retourne un booléen nommé rejouer qui dira si le joueur peut rejouer un tour. Après les boucles, le jeu appel la méthode de calculerVainqueur() pour trier la liste des joueurs selon leurs points. Finalement, afficherPoints() affichera le classement des joueurs en ordre décroissant selon les points.

2.4.2. Exemple qui illustre la dynamique du patron template method



La méthode de jouer() de la classe de JeuDe suit le principe du patron de la méthode template. Donc, elle fait appel à initialiserJeu() pour créer les instances de joueurs et des dés selon les

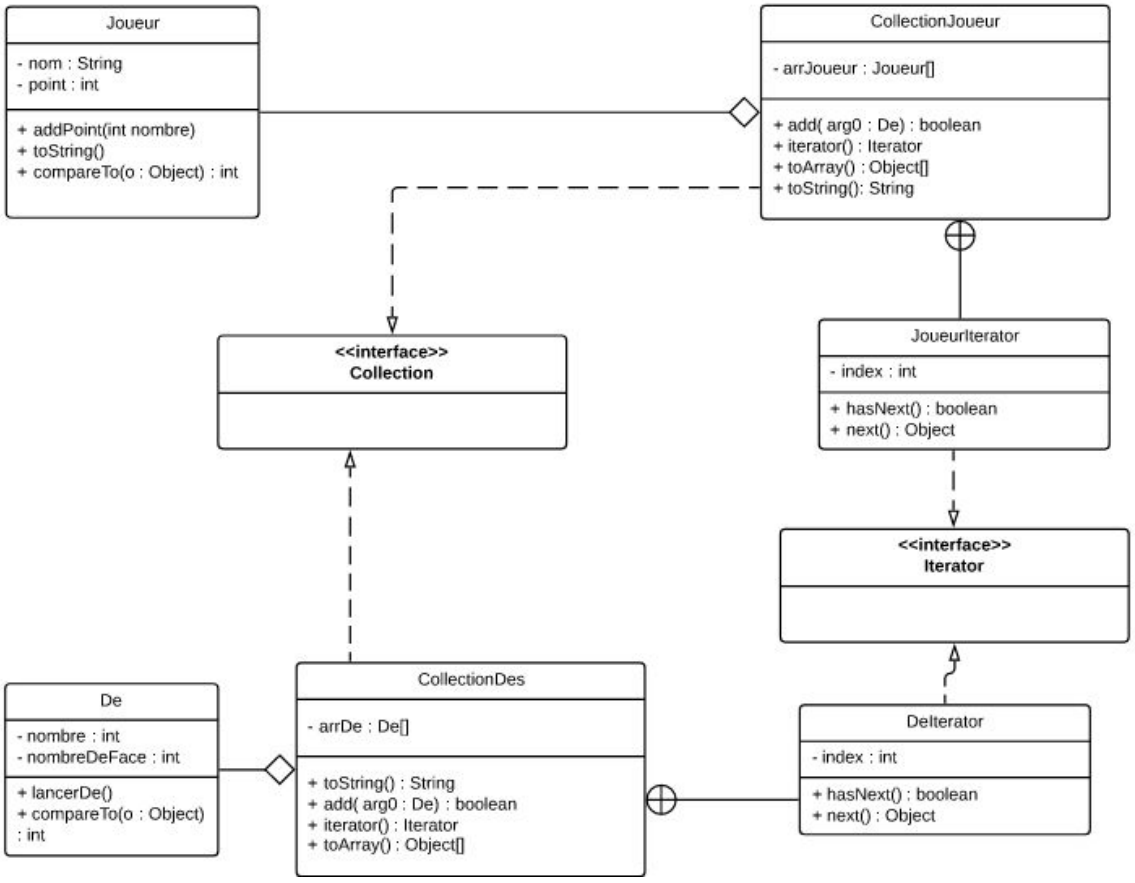
paramètres obtenues de l'utilisateurs. Ensuite, la méthode jouerTours() simule le fonctionnement de chaque tours du jeu et calcul les points en même temps. Après, jeuDe fait appel à la méthode de calculerVainqueur() pour trier la liste des joueurs en ordre décroissant selon leurs points. Finalement, la méthode afficherPoints() affiche le classement des joueurs.

3 Décisions de conception/d'implémentation

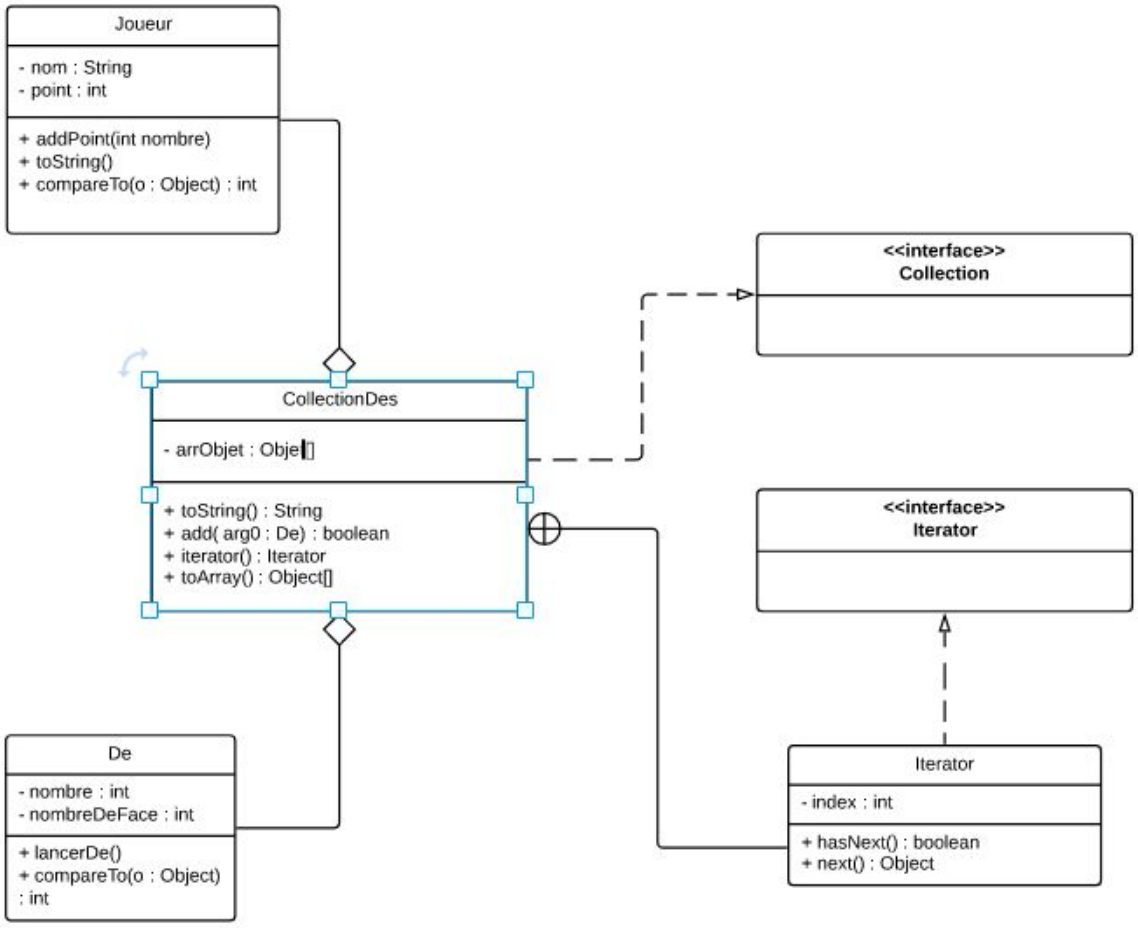
3.1 Décision 1 : Patron itérateur

Décrivez, en subdivisant bien votre texte selon chaque aspect :

- **Contexte**: Le cadriciel exigeait que la classe de Dé et de Joueur soit sauvegarder dans une collection avec l'utilisation du patron itérateur.
- **Solution 1**: Pour la première version de notre conception, nous avons déterminé d'utiliser deux classe de Collection dont une est pour la classe de Dé et l'autre pour la classe de Joueur. L'avantages de cette solution nous permet d'implémenter des méthodes spécifiques pour chaque Collection. Cependant, les inconvénients sont une redondance des méthodes similaires entre les deux collections, nécessite plus de temps a implémenté et elle ne sont pas réutilisables dans un autre contexte.



- **Solution 2** : Une deuxième solution a été conçu pour résoudre les inconvénients de la première solution. Cette solution suggère une seule classe de Collection qui implémente aussi les méthodes du patron d'itérateur. De plus, cette classe pourrait être réutilisés pour d'autres contextes. Dans le cas où nous voulons des méthodes spécifiques, cela permet aussi que la classe de Collection de se faire hériter pour répondre à ce besoin.



- **Choix de la solution et justification :**

  Nous avons choisi la solution #1 parce qu'en séparant les collection en deux instances différentes (CollectionDe et CollectionJoueur), on simplifie la logique du code et comme nous ne sommes pas encore très familier avec le travail d'équipe, git, etc. Nous avons préféré miser sur un code clair et logique qui nous permettrait de moins se mélanger lorsqu'on travaillerait sur différentes classes et méthodes de notre côté. Pour un travail futur, nous aurions plutôt privilégié la deuxième solution et misé sur des commentaires et des bons messages de commit pour rester sur la même page.

3.2 Décision 2 : patron de methode template

Décrivez, en subdivisant bien votre texte selon chaque aspect :

- **Contexte**: Afin de faciliter le fonctionnement du cadriciel, l'implémentation du patron de méthode template est nécessaire.
- **Solution 1**: Notre première itération de conception pour le patron de méthode template était de l'utiliser pour la classe de Fabrique. Les opérations de cette classe est de

demander l'utilisateur des paramètres pour créer des instances de dés et de joueurs. Cependant, cette implémentation est très limité comparé à ce que le patron est capable d'offrir.

- **Solution 2** : Nous avons proposé une autre solution d'où la classe de JeuDe utilisera ce patron. Le fonctionnement des jeux de dés sont plus ou moins les mêmes, jouer un tour, lancer les dés, calculer le score et d'afficher le gagnant du jeu. Donc, le patron est utilisé pour faciliter l'implémentation d'un jeu de De et nous avons ajouté aussi la configuration du jeu pour faciliter la tâche d'implémenter le cadriciel.

- **Choix de la solution et justification :**

  On a choisi la solution 2 parce qu'elle offre une solution beaucoup plus extensible au problème du laboratoire. En effet, lorsque la classe JeuDe utilise le patron, on a beaucoup plus d'options puisque tout le squelette du jeu de dés se situe dans la classe. Ainsi, on a pu utilisé le patron pour créer une méthode template qui définit l'ordre dans lequel les différentes méthodes doivent être appelés. On a ainsi créé un code plus logique et plus clair, tout en respectant le but du cadriciel et en implémentant que les règles générales d'un jeu de dé dans la méthode template.

4 CONCLUSION

Dans la création du cadriciel du jeu de dés, nous avons dû faire usage du patron template méthode pour définir le déroulement général d'un jeu de dés. Nous avons utilisé le patron stratégie pour implémenter les règles de jeu Bunco. Nous avons aussi fait usage du patron itérateur pour les dés et les joueurs. Nous avons utilisé JUnit pour les tests des différentes classes comme le joueur, le dé et le jeu.

Nous avons pu effectuer de manière efficace les tests pour les classes JeuDe, BuncoStrategy, Joueur, De, etc. Nous avons pu implémenter tous les patrons qui ont été demandés pour ce travail. Un des points faibles qu'on a rencontré est la décision de certaines étapes du déroulement du jeu de dés. Le tri des joueurs selon le meilleur pointage a aussi été une étape qui a été difficile, et c'est pourquoi on a opté pour une méthode assez simple plutôt qu'un algorithme complexe. Après, nous avons facilement pu nous adapter pour la création des tests pour valider nos méthodes dans les classes. Notre solution ne satisfait pas à 100% à l'extensibilité d'un cadriciel, mais je dirai que nous sommes arrivé très proche et que les

changements nécessaires au cadriciel ne serait pas énorme pour la plupart des autres jeux de dés.

On aurait pu être introduit plus en détail sur les étapes des bonnes pratiques de contrôle de version et sur le fonctionnement de git pour le travail d'équipe. On aurait également pu organiser mieux notre temps pour permettre de tester au maximum l'extensibilité du cadriciel.