

**Département de génie logiciel et des TI**

Rapport de laboratoire

|  |  |
| --- | --- |
| **No de laboratoire** | Laboratoire **3** |
| **Étudiant(s)** | Alexandre Malo  Marc-Antoine Hébert  Julien Lemonde  Jean-Michel Coupal |
| **Code(s) permanent(s)** | - |
| **Cours** | LOG121 |
| **Session** | Été 2015 |
| **Groupe** | 01 |
| **Professeur** | **Francis Cardinal** |
| **Chargés de laboratoire** | **Patrice Boucher** |
| **Date de remise** | 5 juillet 2015 |

1 Introduction

Le but de ce laboratoire est de développer un framework unique pouvant servir à créer plusieurs jeux de dés différents. Les objectifs du laboratoire sont d’implémenter une solution à l’aide de plusieurs patrons de conception tels que le patron GoF «Itérateur», le patron G0F «Méthode template» et le patron GoF «Stratégie». Par la suite, il faut concevoir des classes jumelles de test unitaire afin de faire la validation du code et sa conception. Toutes les solutions retenues dans le cadre de ce laboratoire répondre au critère préalablement établi. Les chapitres suivants de ce rapport comportent une section sur la conception telle que le choix et les responsabilités des classes, un diagramme de classe UML, une description de la conception, les faiblesses de la conception et un diagramme de séquence. Par la suite, la troisième section comporte les décisions d’implémentation avec les différentes options envisagées et la solution retenue.

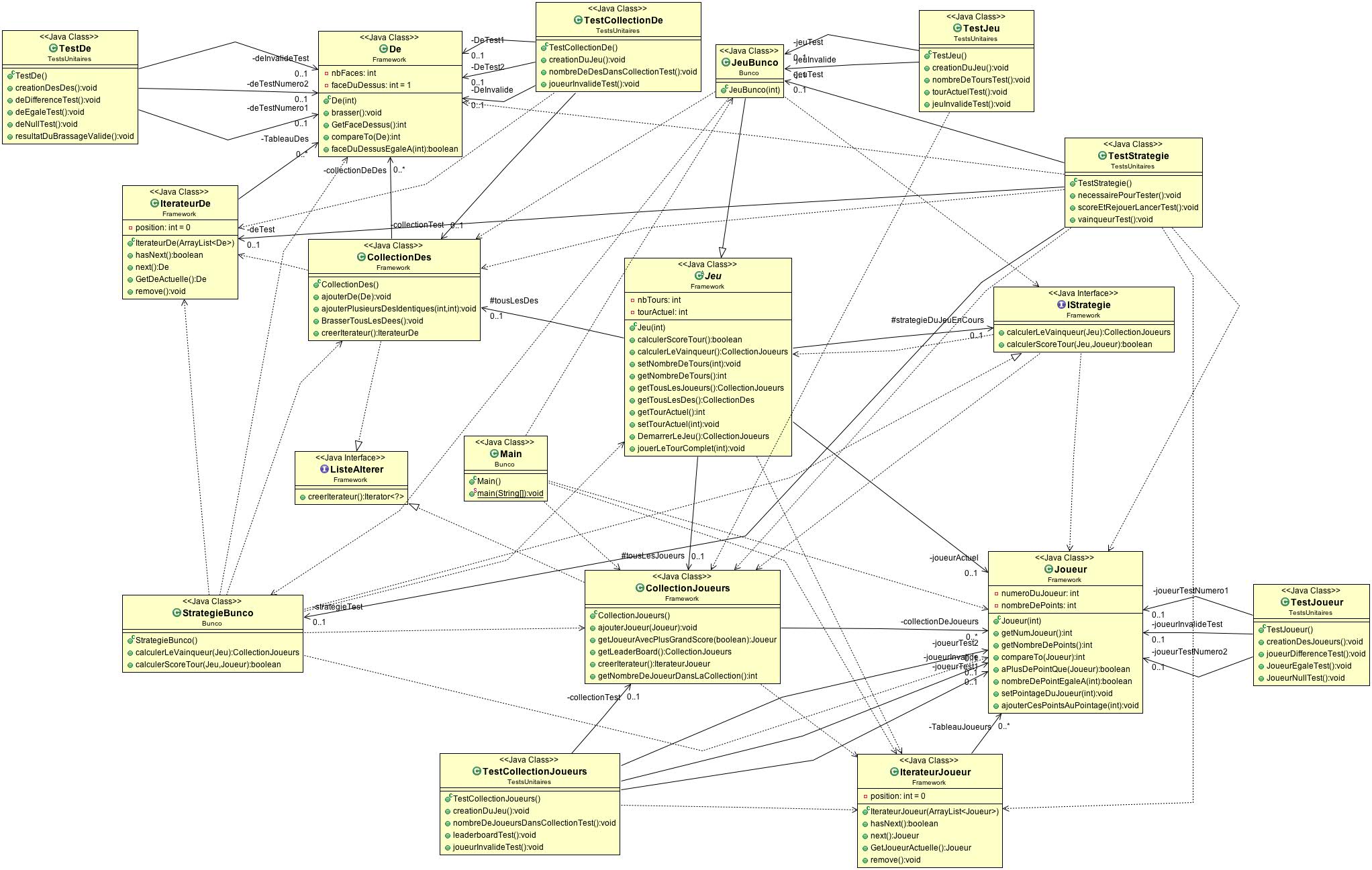
2. Conception

## 2.1 Choix et responsabilités des classes

Tableau présentant les classes présentes dans le laboratoire 3 incluant leurs responsabilités et leurs dépendancesrespectives.

|  |  |  |  |
| --- | --- | --- | --- |
| *Classe* | *Responsabilités* | | *Dépendances* |
| JeuBunco | * Cette classe a comme responsabilité de créer un nouveau jeu de Bunco avec le nombre de joueurs qu’elle a reçu en paramètre. | | * Jeu * StrategieBunco * CollectionDes * IStrategie |
| Main | * La classe *main* fait exécuter le jeu, affiche le score des joueurs tour après tour jusqu'à ce qu’un joueur gagne. Elle affiche aussi le classement des joueurs à la fin. | | * CollectionJoueurs * IterateurJoueur * Joueur * JeuBunco |
| StrategieBunco | * Cette classe assure comme fonction de calculer le pointage de chaque joueur à chaque tour. Elle retourne aussi le vainqueur du jeu Bunco. | | * IStrategie * IterateurDe * Jeu * Joueur * CollectionJoueurs |
| Framework | | | |
| CollectionDes | * La classe *CollectionDes* a comme responsabilité de créer une collection de dés, d’ajouter les dés dans la collection et de brasser tous les dés. | | * IterateurDe * De |
| CollectionJoueur | * Cette classe crée une collection de joueurs, elle ajoute les joueurs dans la collection, détermine quel joueur a le plus grand score et crée une collection de joueurs selon le classement de chacun. | | * Joueur * IterateurJoueur * ListeAlterer |
| De | * La classe *De* a la responsabilité de brasser le dé et d’obtenir le résultat. Elle compare aussi le résultat avec un autre dé et même elle compare le score du dé avec un nombre à vérifier qu’on lui a fourni en paramètre. | | * (Aucune) |
| IStrategie | * Interface qui implémente une méthode pour calculer le vainqueur et une pour calculer le score à chaque tour. | | * Joueur * CollectionJoueurs * Jeu |
| IterateurDe | * Classe qui créer un itérateur pour gérer les dés à jouer, peu importe le nombre, de dé. | | * De |
| IterateurJoueur | * Cette classe créer un itérateur pour gérer les joueurs du jeu, peu importe le nombre de joueurs. | | * Joueur |
| Jeu | * La classe *Jeu* a comme responsabilité de jouer un tour à la fois pour tous les joueurs à la fois. Elle contient aussi tous les accesseurs et mutateurs pour obtenir les informations qui attrait à la partie comme : calculer le score, définir le vainqueur, etc. | | * Joueur * CollectionJoueur * CollectionDes * IStrategie |
| Joueur | * Cette classe s’occupe de comparer deux joueurs ensemble, elle compare aussi un joueur pour déterminé celui-ci a le plus de points et elle ajoute les points du joueur au pointage. | | * (Aucune) |
| ListeAlterer | * Interface occupe comme fonction de définir le constructeur *creerIterateur*. | | * (Aucune) |
| Tests Unitaire | | | |
| TestDe | | * La classe *TestDe* a comme responsabilité de tester les méthodes des dés. Elle teste lorsque les résultats sont différents, si les résultats sont égaux et aussi elle teste pour s’assurer que le résultat du brassage est valide. | * De |
| TestJeu | | * Cette classe teste le jeu pour s’assurer que les tours sont bien respectés et que le programme retourne une erreur si on tente de démarrer une partie invalide. | * JeuBunco |
| TestJoueur | | * La classe *TestJoueur* teste les différentes possibilités par rapport aux joueurs. Elle teste lorsque leurs scores sont différents, lorsqu’ils sont égaux et lorsqu’on compare un jouer avec *null*. | * Joueur |
| TestStrategie | | * Cette classe s’occupe de tester la partie en tant que telle. On teste les fonctions de vérification des scores avec trois dés pareils ou pas pareil. La classe teste aussi la définition d’un joueur vainqueur pour s’assurer que le programme donne la victoire au bon joueur. | * StrategieBunco * JeuBunco * IterateurDe * CollectionJoueurs * IterateurJoueurs |
| TestCollectionDe | | * La classe *TestCollectionDe* teste le programme pour vérifier le nombre de dés qu’il y a dans la collection de dés. | * De * CollectionDe * IterateurDe |
| TestCollectionJoueurs | | * Cette classe a comme responsabilité de tester tout ce qui attrait à la collection de joueurs. Donc, elle vérifie le nombre de joueurs dans une collection et vérifie le classement des joueurs selon leur pointage. | * Joueur * CollectionJoueurs * IterateurJouer |

## 2.2 Diagramme des classes



## 2.3 Description de la conception

La conception de cette application est divisée en trois parties. La première partie est le *framework*, il contient toutes les classes génériques. Ces classes ont été conçues de manière à être polyvalentes et elles pourraient être réutilisées dans un autre jeu de dés autre que celui du Bunco. Exemple, le constructeur des dés prend en charge le nombre de face qu’on désir avoir. Donc, les dés qui sont créés ne sont pas automatiquement des dés à six faces.

La deuxième partie est le jeu en tant que tel. Elle contient trois classes (Main, JeuBunco, StrategieBunco). C’est dans ces classes que les règlements du jeu y sont définis. Elles utilisent le *framework* pour pouvoir jouer. De plus, l’une des fonctions majeures de cette section est que c’est dans la classe *StrategieBunco* qu’on effectue le calcul du score des joueurs en fonction du résultat des dés qu’ils ont brassé. Évidemment ces classes sont conçues de sorte à utiliser le framework et ces composantes pour jouer.

La troisième partie est les tests unitaires. Elle contient six classes, qui testent six sections du programme. Soit les dés, les joueurs, le jeu, la stratégie, la collection de dés et la collection de joueurs. Elle a été conçue de sorte à tester plusieurs fonctions de chacune des sections et de retourner un pointage selon la réussite ou l’échec des divers tests.

## 2.4 Faiblesses de la conception

Au niveau de la classe "Strategie" on se rend vite compte que le code ne respecte du mieux possible les règles apprises dans le cours. En effet la classe StrategieBunco connait le fonctionnement du jeu et de ses sous-composantes plus qu'elle ne le devrait. Elle s'occupe aussi de trop de tâche que d'autres classes auraient pu se charger. Par exemple, Dans la méthode "CalculerScoreTour", on lui passe un joueur et un jeu en paramètre et c'est ensuite cette méthode qui s'occupe de tout calculer les scores et de les ajouter au joueur passé en paramètre plus tôt. Le problème ici, est que la classe Strategie dépend maintenant d'une classe de plus, la classe Joueur. Une meilleure façon de faire ici serait que cette méthode "CalculerSocreTour" ne revoit qu'un "Int" et qu'ensuite, la classe Jeu ayant appelé cette méthode s'occupe d'ajouter ces points (Int reçu) au joueur Actuel. De cette façon, nous détacherions la classe Joueur de la classe "Strategie".

Ensuite, à plusieurs moment, dans le code on peut voir que les joueurs et les Des sont passé en accesseur directement par leur adresse mémoire pour être modifier directement. Cela facilite grandement la tâche du premier programmeur, mais il n'est pas conforme à la programmation orienté objet, que plusieurs classes possèdent la même adresse mémoire d'un même objet. Il devient très difficile à contrôler l'accès a cet objet par la suite. Lors de problèmes, plus tard, il sera difficile de trouver la cause d'une mauvaise modification sur un objet, car plusieurs classes pourraient être responsable. Une amélioration du code serait de se passer ces objet à l'aide de ".clone". Ainsi, ce ne serait pas l'adresse mémoire qui serait distribuer, mais bien un autre adresse contenant un clone du même objet. Ensuite, quand les modifications sont terminé, réacheminer ce clone vers le bon endroit pour remplacer son ancien objet passé date.

## 2.5 Diagramme de séquence (uml)

Chaque diagramme doit être appuyé d'un texte précis qui décrit verbalement le déroulement de l'exemple.

|  |
| --- |
| **Remarques :**   * *Les noms des méthodes, des classes et des objets doivent correspondre à ceux utilisés dans votre code.* * *Erreurs fréquentes :*    + *étiquette de l'objet incorrect. On doit avoir « nom:Type » pour un objet ayant un nom, «:Type» pour un objet anonyme et «Type» pour une méta-classe.*   + *l'objet n'apparaît pas au moment de son instanciation*   + *interaction anonyme*   + *présence de classe abstraite (impossible à moins qu'utilisée en tant que méta-classe)*   + *l'interaction ne reflète pas celle du code* |

### 2.5.1. Exemple qui illustre la dynamique du patron stratégie

### 2.5.2. Autre diagramme de séquence

3 Décisions d'implémentation

## 3.1 Décision 1 : Mettre les dés à une certaine face pour les tests unitaires

* **Contexte**: Afin de bien tester notre classe "Dé" durant les tests unitaires, il fallait nous assurer des valeurs sur la face du dessus des dés à tester.
* **Solution 1**: Ajouter un mutateur à la classe Dé pour que l'utilisateur d'un objet Dé puisse choisir la face du dessus de ce dé. Avec une méthode "deExemple.setFaceDuDessus(1)".
* **Solution 2** : Lors du test de notre classe. Ajouter un "while" qui brasse un dé tant qu'il n'est pas au résultat désirée. Comme cela, on peut choisir la face du dessus.
* **Choix de la solution et justification :** Pour ce qui est de l'efficacité, il est clair qu'il est plus performant de seulement affecter une face du dessus à l'aide d'un mutateur, que de lancer un dé jusqu'à ce qu'il aye la face du dessus désirée. Pourtant, nous avons choisi de choisir la deuxième méthode, car nous croyons pas qu'un dé devrait avoir la possibilité d'être placer sur la face de notre choix. Le but premier d'avoir un dé dans un jeu est d'être lancer et que se soit le "Hasard" qui choisi la face pour nous. Dans cette mentalité, nous trouvions que de laisser l'utilisateur l'opportunité de placer un dé sur la face de son choix n'était pas approprié.

## 3.2 Décision 2 : Rejouer le Tour ou Non selon le lancé

* **Contexte**: Il fallait savoir si le tour devait être rejouer ou non, dépendamment des dés brassés.
* **Solution 1**: Dans la classe jeu, dans la méthode "JouerLeTourComplet" nous aurions pu avoir le traitement du score reçu par la méthode "calculerScoreTour" qui mène à la méthode "calculerScoreTour" de la strategie courante. Si cette méthode avait retournée uniquement un "Int" pour le score de ce lancé, la méthode "JouerLeTourComplet" aurait pu décider par la suite, dépendamment du score obtenu, si le joueur rejouait son tour ou non. (Ex: if(this.calculerScoreTour()==21){rejouer = false})
* **Solution 2** : Dans la classe jeu, la méthode "JoueurLeTourComplet" ne recoit seulement qu'un "boolean" retourné par la méthode "calculerScoreTour" de la strategie actuelle. Si ce "boolean" nommé "rejouerLeTour" est à "True" la méthode "JouerLeTourComplet" sait qu'il faut faire rejouer ce joueur pour un autre lancé. Sinon, c'est au joueur suivant.
* **Choix de la solution et justification :** Nous avons choisi la Solution 2, car nous pensons, que le fait de rejouer ou non après l'obtention d'un certain score devrait être traité dans la stratégie du jeux, car il s'agit d'un règlement du jeu. Le jeu n'a donc pas besoin de savoir aucun règlement relié au fonctionnement du jeu Bunco+. Cependant, comme mentionné plus haut dans les faiblesses de conception, si nous voudrions changer ce que retourne la méthode "calculerScoreTour" de la classe "StrategieBunco", nous devrons choisir, ici, la solution 1.

4 Conclusion

1. Faites un rappel des objectifs du logiciel: ces objectifs ont-ils été atteints?
2. Synthétisez ce qui a été réalisé, les points forts et les points faibles de votre conception.
3. Commentez, rappelez, les améliorations possibles, la portée de votre travail pour d'autres applications et les développements futurs envisageables.

|  |
| --- |
| * *Ne discutez pas du contexte académique et de détails se rapportant au travail d'équipe (si vous éprouvez un problème dans votre équipe, contactez plutôt un chargé de laboratoire).* |

5.Références

* Mettez, au besoin, les références