Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	**Laboratoire 2**
Étudiant(s)	Charles Fleury, Thomas Lavergne
Code(s) permanent(s)	FLEC06129801, LAVT80050007
Cours	LOG121
Session	HIVER-20
Groupe	03
Professeur	**Cedric St-Onge**
Chargés de laboratoire	**Hind Errahmouni**
Date de remise	11 mars 2020

1 INTRODUCTION

Les cadriciels sont des classes logiciels destinées à fournir une base utilisable par des développeurs qui veulent créer des applications plus complexes. Le présent laboratoire a pour but de développer un cadriciel d'un jeu de dés, tout en y incluant les patrons Stratégie, Itérateur et Méthode Template. Nous pouvons donc déduire qu'une qualité essentielle d'un cadriciel est son extensibilité. Notre solution n'est pas parfaite et fournit uniquement une base très primitive d'un jeu, mais nous avons bien utilisé les trois patrons et croyons que notre cadriciel est facilement extensible, car il demande seulement de connaître et définir la stratégie de pointage et redéfinir au plus deux méthodes de l'usine qui créent le jeu pour avoir un jeu simple et fonctionnel.

Dans ce rapport, nous présenterons la conception de notre cadriciel, suivi de deux séquences d'appels importants entre les objets de notre cadriciel et de deux décisions clés de conception.

---

2. CONCEPTION

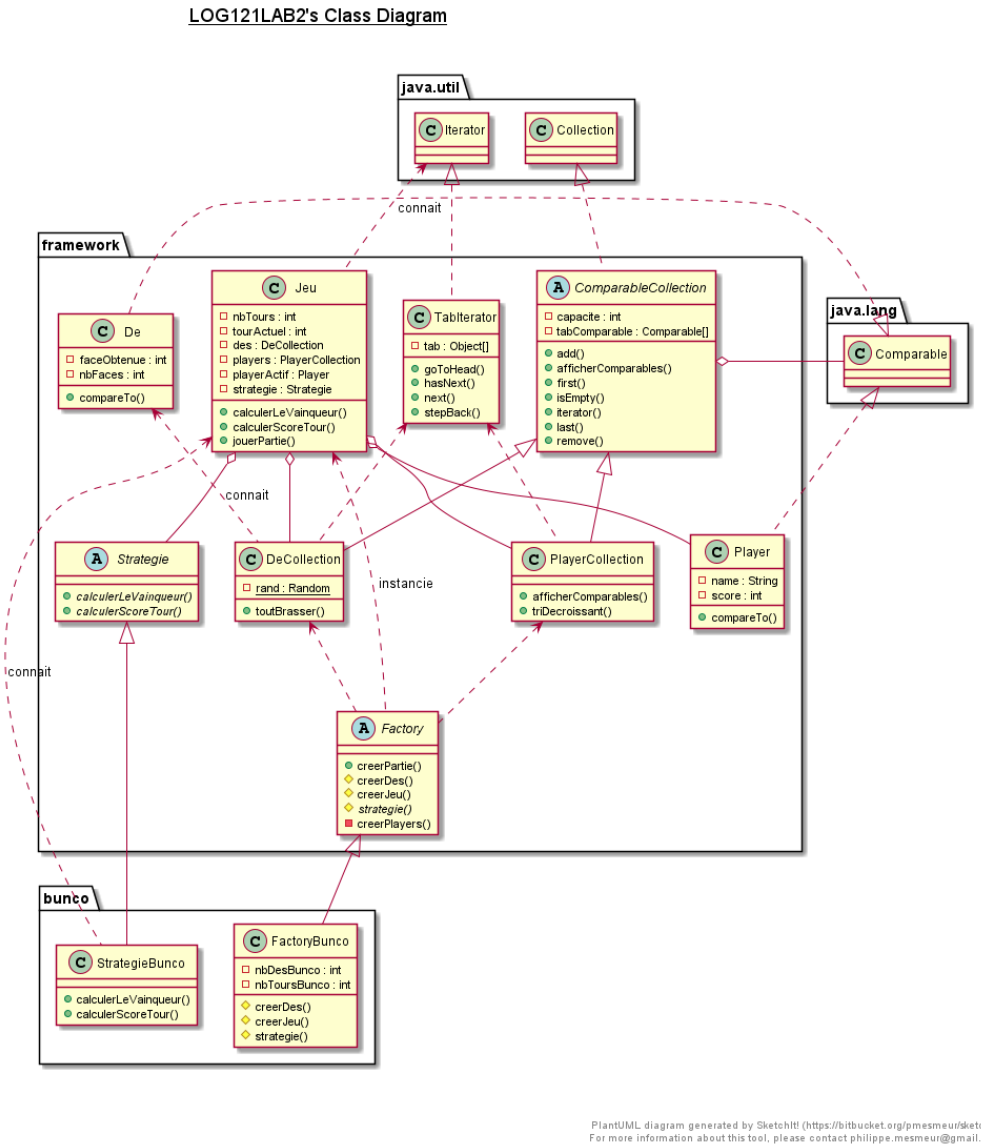
2.1 CHOIX ET RESPONSABILITÉS DES CLASSES

Classe	*Responsabilités*	*Dépendances*
– De	– Objet utilisé pour le jeu : le nombre de dés créé et le nombre de faces dépendent de la stratégie de jeu	–
– DeCollection	– Objet permettant de stocker les dés créés pour une partie de jeu	– De, ComparableCollection, TabIterator
– Player	– Objet utilisé pour le jeu : il représente le joueur avec son score, et le nombre créé sera initialisée à la création de la partie.	–
– PlayerCollection	– Objet permettant de stocker les joueurs pour une partie de jeu	– Player, ComparableCollection, TabIterator
– ComparableCollectio	– Classe abstraite utilisée pour analyser et comparer les objets des listes disponibles en classes filles	– TabIterator

n	(DeCollection et PlayerCollection) afin de pouvoir différencier les joueurs et dés et ainsi pouvoir déterminer le(s) gagnant(s)	
– TabIterator	– Classe Iterateur, permettant de parcourir les classes Listes mentionnées précédemment (PlayerCollection et DeCollection).	–
– Jeu	– Classe permettant de lancer le jeu – Contient les listes de dés et de joueurs nécessaires pour pouvoir faire tourner le jeu – Appelle les méthodes de la stratégie de la règle du jeu pour pouvoir réaliser des tours de jeu et déterminer le gagnant de la partie	– DeCollection, PlayerCollection, Stratégie, Player
– Stratégie	– Classe abstraite permettant de définir les règles du jeu : un objet fils de cette stratégie sera un type de jeu, avec des règles correspondantes. – Elle contient les méthodes abstraites permettant de réaliser un tour de jeu et de déterminer le gagnant d'une partie. – La classe fille stratégie sera celle qui sera créée hors du framework, afin que l'utilisateur puisse créer ses règles lui même	–
– Factory	– Classe abstraite permettant de créer l'objet jeu ainsi que tous les besoins de celui-ci, notamment les collections de dés et de joueurs ainsi que les	–

	objets nécessaires à ceux-ci. – La classe fille factory sera celle qui sera créée hors du framework, afin que l'utilisateur puisse régler les quantités des objets nécessaires à son jeu (par exemple le nombre de joueurs, le nombre de dés ou nombre de tours)	

2.2 DIAGRAMME DES CLASSES



2.3 Faiblesses de la conception

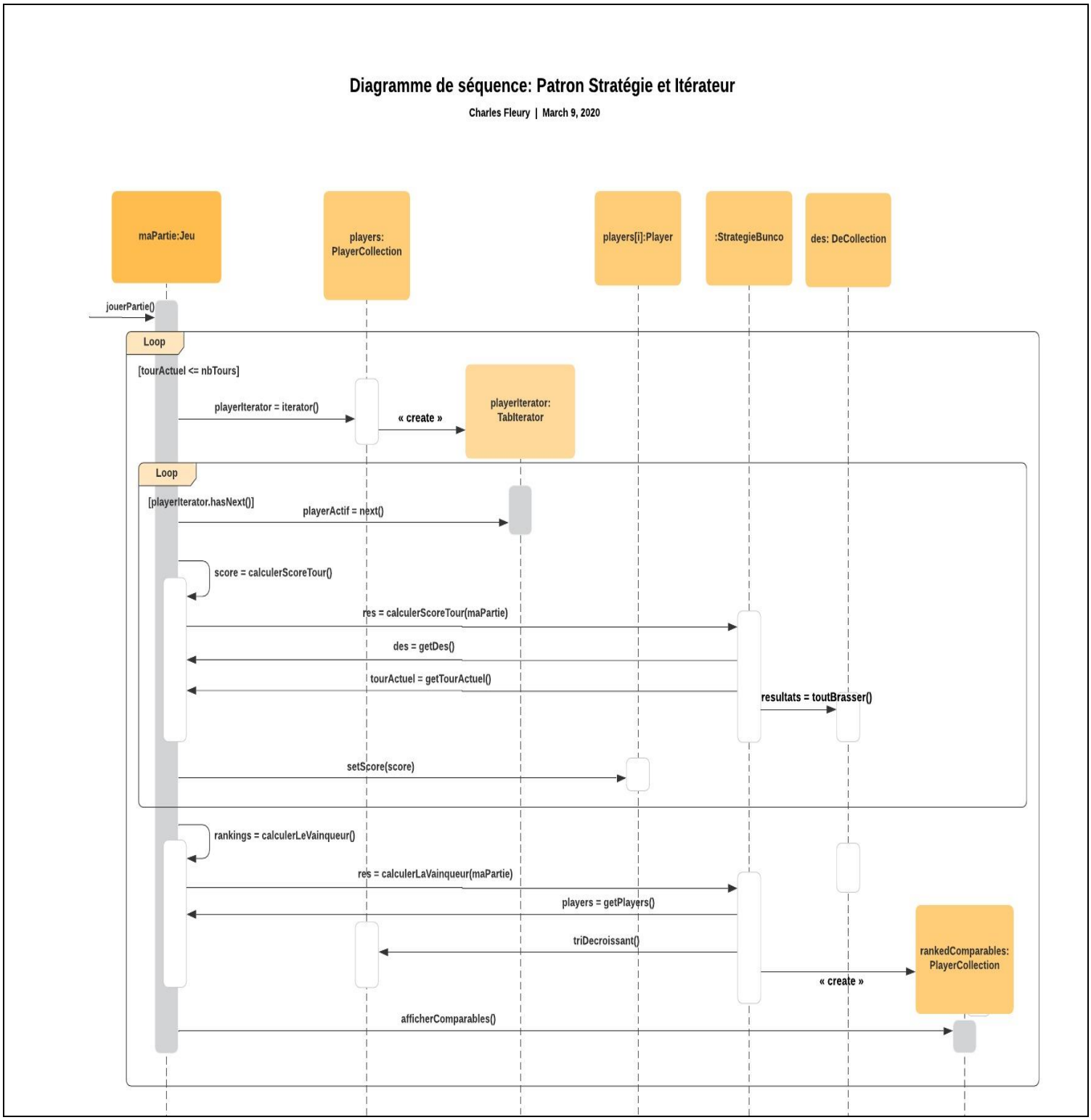
L'une des faiblesses de la conception réside dans l'utilisation de la classe Factory : en effet, celle-ci contient une méthode permettant de créer un jeu en passant en paramètre le nombre de joueurs, de dés et de tours de jeu, afin de pouvoir modifier certaines règles d'une stratégie. Or, certains jeux ont un ou plusieurs de ces nombres prédéfinis (par exemple le Bunco utilise trois dés et se joue en six tours), ce qui rend à la création du jeu utilisant la stratégie Bunco, certains paramètres inutiles (ici les dés et le nombre de tours). Nous aurions pu régler ce problème en ajoutant une autre méthode template avec moins de paramètres appelant des fonctions elles aussi sans paramètres afin de rendre le tout plus cohérent.

De plus, nous pourrions aussi dire que pour un utilisateur du cadriciel et l'ajout de nouvelles règles, il peut être compliqué d'implémenter certains jeux si la logique de passement de tour est plus complexe (par exemple, les joueurs jouent dans un ordre défini dans la stratégie) : le cadriciel permet juste de gérer le cas ou chaque joueur joue une fois à chaque tour de jeu.

Enfin, le jeu dépend tout de même de beaucoup de classes (notamment les collections) qui elles-mêmes dépendent d'autres classes, ce qui donne une dépendance plus forte encore, et cela rendrait la création de sous-classes de jeu plus complexe.

2.4 Diagramme de séquence (uml)

2.4.1. Exemple qui illustre la dynamique du patron stratégie

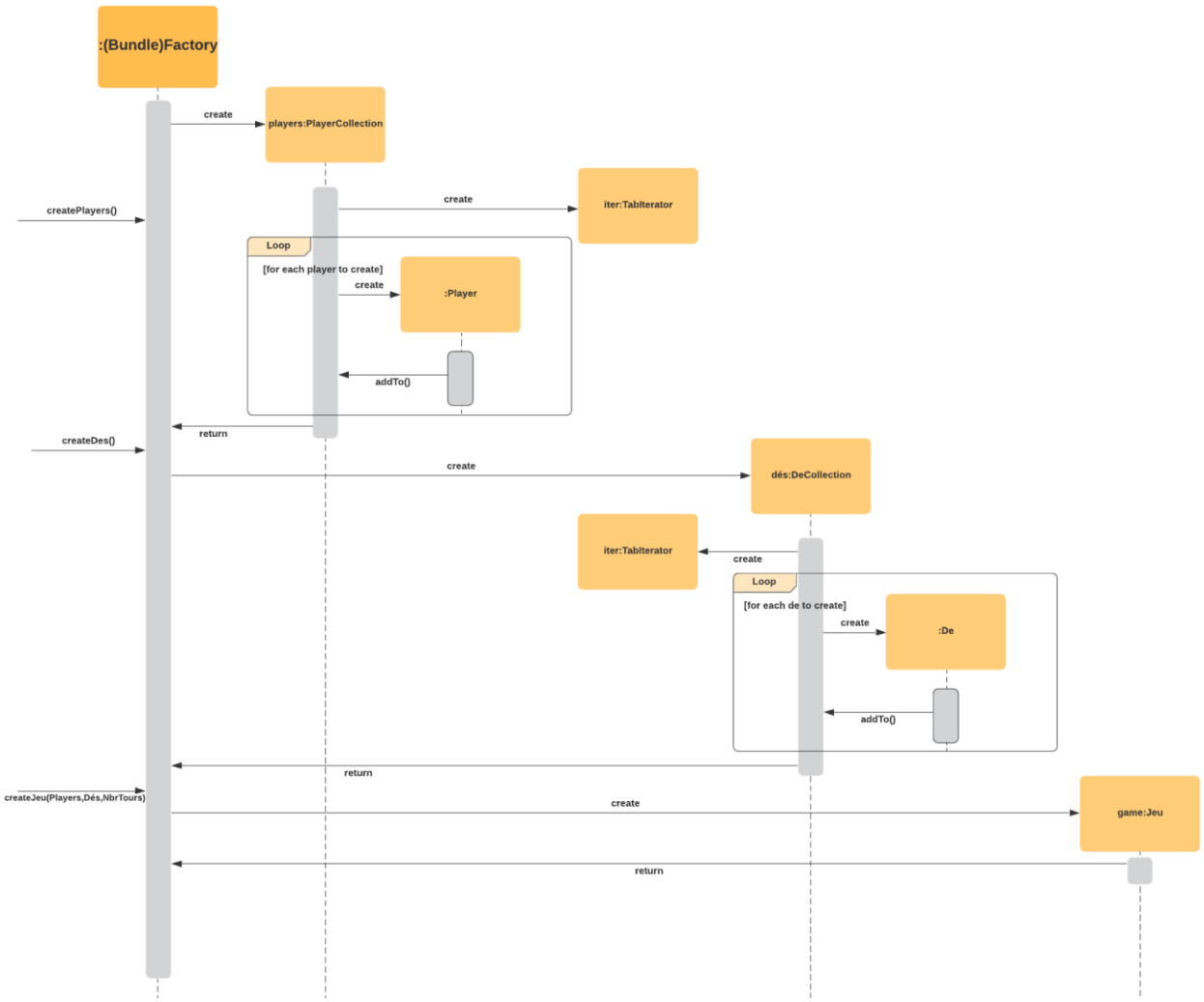


Description :

L'appel de la méthode jouerPartie() entraîne une série d'appels qui illustrent le patron Stratégie et le patron Itérateur. D'abord, le Jeu appelle la méthode iterator() de PlayerCollection, qui lui retourne son TabIterator. Ensuite, le jeu appelle sa méthode calculerScoreTour(), qui par le patron Stratégie, appelle la méthode du même nom de sa stratégie active, qui est dans notre cas

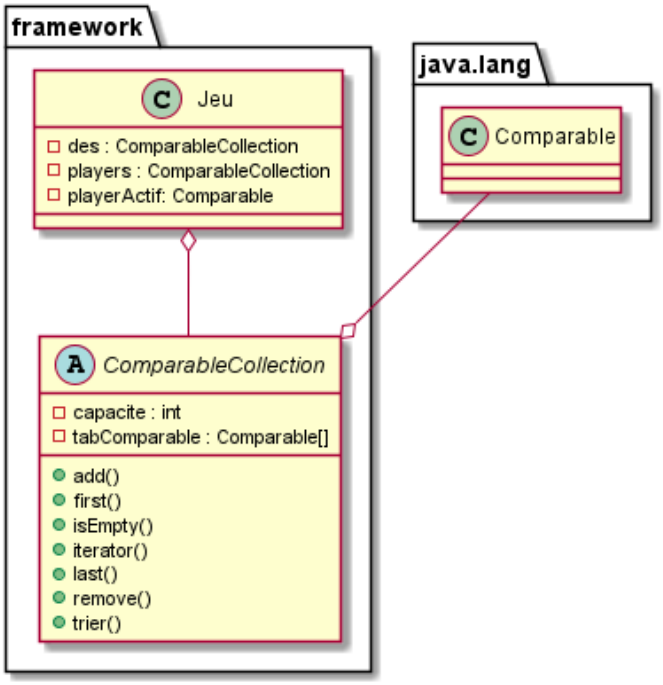
une StrategieBunco. Cette méthode récupère la collection de dés et le tour actuel du jeu et brasse tous les dés avec la méthode toutBrasser de DeCollection. Le score du tour est retourné au jeu, qui l'assigne au joueur actif. Une fois tous les joueurs passés, le jeu appelle sa méthode calculerLeVainqueur(), qui appelle la méthode du même nom de la StrategieBunco. Cette méthode appelle triDecroissant() de PlayerCollection avant de créer une copie triée de cette collection et la retourner au jeu. Le jeu finit par afficher le classement des joueurs avec afficherComparables().

2.4.2. Autre diagramme de séquence



3.1 Décision 1 : Décider de la complétude de la classe ComparableCollection

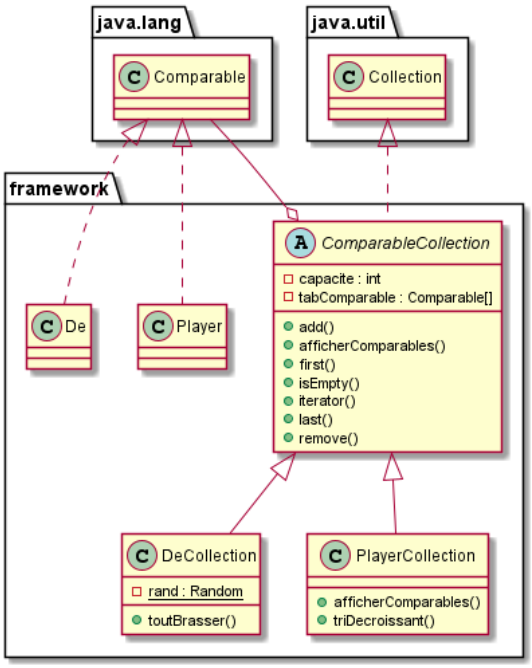
- **Contexte**: Un mandat de ce laboratoire est d'implémenter nos propres collections, sans se servir des collections Java existantes comme ArrayList ou Vector, qui implémentent l'interface Collection. Dans notre problème, nous avons besoin de stocker des dés et des joueurs (Player dans notre code) dans des collections. Plusieurs options s'offrent donc à nous, comme décider d'implémenter ou pas l'interface Collection, regrouper ou pas la collection de dés et de joueurs, faire une classe abstraite qui regroupe les dés et les joueurs ou encore les séparer complètement.
- **Solution 1**: Créer une classe concrète ComparableCollection qui n'implémente pas Collection et ne pas créer de DeCollection ni de PlayerCollection.



Cette solution comporte plusieurs avantages. D'abord, le fait de ne pas implémenter Collection rend le code beaucoup plus léger et lisible, puisque Collection comporte une multitude de méthode à implémenter dans ses sous-classes. Nous implémenterions seulement les méthodes d'une collection que nous jugeons nécessaire au fonctionnement d'un jeu quelconque dans ComparableCollection. Cette classe concrète

permet aussi d'empêcher le prolifération de classes en éliminant DeCollection et PlayerCollection : le Jeu dépend ainsi de moins de classes. Les défauts de cette conception sont son manque de complétude. Collection est une interface très complète qu'il serait important d'implémenter dans toute classe qui veut gérer une collection d'items. Aussi, cette conception ne permet pas d'ajouter des fonctions spécifiques aux dés ou aux joueurs dans leurs collections.

- **Solution 2** : Créer une classe abstraite ComparableCollection qui implémente Collection et qui est hérités par DeCollection et PlayerCollection



Cette conception permet une grande complétude des fonctions de base d'une collection puisque toutes les méthodes de Collection doivent être implémentées. Il permet aussi de mettre des fonctions spécifiques aux dés ou aux joueurs dans leurs sous-classes respectives. Par contre, la méthode de tri doit être redéfinie dans les sous-classes de ComparableCollection si on veut pouvoir retourner une instance triée de cette sous-classe, ce qui pourra entraîner de la duplication de code si un développeur voulait trier un objet comparable autre qu'un joueur. Aussi, la classe Jeu, qui n'est pas montrée dans le diagramme, devra connaître toutes les sous-classes de ComparableCollection, ce qui augmente les dépendances entre classes.

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

Nous avons choisi la solution 2. Le besoin de complétude dans un cadriciel est primordial car sa réutilisation à grande échelle est l'objectif principal, et c'est cette solution, en implémentaant toutes les méthodes de l'interface Collection, qui offre cette qualité. De plus, la classe abstraite ComparableCollection fournit plus de cohésion, car toutes les collections d'objets comparables vont se retrouver au même niveau et pourront rajouter des fonctions spécifiques à leur sous-classe, tandis que si on veut rajouter des fonctions spécifiques à une collection avec la classe ComparableCollection concrète, nous pourrions avoir à la fois des objets ComparableCollection et d'autres objets qui en héritent.

3.2 Décision 2 : Création d'une partie depuis la factory

- **Contexte :** Afin de créer une partie, plusieurs éléments sont nécessaires : des joueurs, des dés et des règles. Pour cela, nous devions créer une classe « usine » qui s'occupe de la création des joueurs, des dés et du jeu. Pour le Bunco par exemple, il faut un nombre indéfini de joueurs supérieur à 2, 3 dés et 6 tours de jeu (on considère les tours comme la règle, cette dernière étant implémentée dans la stratégie liée au jeu). Pour créer un jeu, comment faire pour pouvoir générer un « environnement » propice aux règles ? (Le Bunco se jouant à 3 dés, s'il n'y en a que deux les règles ne sont pas respectées)
- **Solution 1 :** Entrer les paramètres nécessaires au bon déroulement du jeu à la main dans le code.

  En effet, lors de la création de la partie, nous allons appeler une méthode de la classe usine permettant de créer la partie ainsi que la liste de joueurs et de dés pour enfin créer la partie à partir de ces deux derniers et du nombre de tours nécessaires pour la partie. Il faut donc définir ces nombres, les dés et les tours nécessaires pouvant changer selon les règles du jeu actuel et les joueurs pouvant changer entre chaque partie jouée.

  Cette solution était donc de rentrer directement les valeurs (ici 3 pour les dés et 6 pour les tours) dans la création de la partie, étant donné que la stratégie s'occupe du reste.

  Jeu partie = factory.créerPartie(joueurs = x, dés = 3, nombreTours = 6)

  Cependant, il y a un risque d'erreur dans la création de la partie, qui contredit les règles (un / des dé(s) ou tour(s) de trop / moins).

- **Solution 2 :** Créer une factory dédiée à une règle.

  Afin de pouvoir avoir les bonnes règles nécessaires au déroulement du jeu, il faut que les « obligations » soient disponibles pour créer correctement chaque nouvelle partie. Pour cela, nous avons décidé de passer la classe Factory en classe abstraite qui sera héritée

par des usines pour chaque règle de jeu créée. Dans un bundle de jeu, nous aurons donc la stratégie, définissant les règles, le déroulement d'un tour ainsi que la désignation du vainqueur du jeu, mais aussi une usine, qui s'occupera de créer le jeu avec un nombre prédéfini de dés et/ou tours et/ou joueurs par exemple.

De cette manière, aucune erreur ne peut être commise et si l'utilisateur souhaite faire un changement alors il faudrait créer un deuxième bundle similaire en changeant juste les valeurs qu'il souhaite changer.

4 CONCLUSION

Le but de ce laboratoire était de réaliser un cadriciel pour un jeu de dés, utilisant les patrons Méthode Template, Stratégie et Itérateur. Ce cadriciel est ensuite repris par l'utilisateur pour qu'il puisse implémenter les règles de son jeu et les moduler à sa guise. Nous avons pu réaliser ces objectifs pour ce logiciel.

Pour rendre le cadriciel le plus modulable possible, nous laissons à l'utilisateur le soin de régler certains détails dans la création du jeu, qui peuvent changer selon les règles (un certain nombre de dés…) ainsi que les passages de tour et désignation du vainqueur : les détails sont réglés dans la classe fille de Factory et le reste sera réalisé dans la classe fille de Stratégie (ces deux classes étant uniques pour chaque package de règle). Notre conception empêche donc à l'utilisateur de faire une erreur dans la création d'un jeu si certaines règles sont fixes, mais cela veut aussi dire qu'il n'est pas possible pour l'utilisateur de tester un jeu en changeant certaines options (par exemple, Bunco avec 4 dés au lieu de 3). Pour pouvoir changer ces options, il faut le faire directement dans le code ou bien réaliser un autre package utilisant ces options.

Enfin, la réalisation d'un cadriciel de jeu peut permettre à des utilisateurs de créer les jeux auquel ils jouent, ou bien encore en inventer en testant les différentes fonctionnalités. Les méthodes de création du jeu peuvent être améliorées en les rendant plus modulables et en enlevant les paramètres de la méthode lorsque ceux-ci sont prédéfinis dans les règles.