Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire **2**
Étudiant(s)	Pierre Amar ABDELLI
	Mihai ARMASARU
	Michael COUET
	Dave VOUMA-GAGNON
Code(s) permanent(s)	
Cours	LOG121
Session	Automne 2020
Groupe	1 - B
Professeur	**Vincent Lacasse**
Chargés de laboratoire	**Eduardo Furtado-Sa-Correa**
Date de remise	11-11-2020

1 INTRODUCTION

Le projet consiste à créer un cadriciel (framework) pour des jeux de dés, ainsi que le jeu Bunco+. Le cadriciel contiendra des classes pour l'initialisation des dés, des joueurs, des tours, des scores, etc. Trois patrons minimums seront utilisés pour la création de ce projet : stratégie, itérateur et méthode template. Un des objectifs de ce projet est d'utiliser l'interface Comparable de l'API Java, avec laquelle on pourrait comparer les scores des différents joueurs ou encore comparer les dés entre eux (par exemple, pour calculer le score d'un brassage dans Bunco+). Nous allons aussi créer des tests unitaires pour le cadriciel et pour le jeu, dans l'intérêt d'en comprendre l'utilité et de savoir comment les utiliser avec l'outil JUnit. Dans ce rapport, on verra la conception du code, les décisions d'implémentation et, une fois le code terminé, une synthèse du projet.

2. CONCEPTION

2.1 CHOIX ET RESPONSABILITÉS DES CLASSES

– *Classe*	– *Responsabilités*	– *Dépendances*
Main	- Afficher le menu du jeu - Débuter le code	Aucune
Jeu	- Contient les actions principales à exécuter pour n'importe quel jeu de dés (ajouter des joueurs, calculer le vainqueur, etc)	Interface IScore
JeuFactory	- Contient le choix des jeux et de leurs règles	Aucune
Joueurs	- Création des joueurs et leurs attributs - Comparer les joueurs	Interface Comparable
CollectionJoueurs	- Ajouter et aller à travers les joueurs	Interface Container Classe Itérateur
Des	- Initialiser un dé	Comparable
CollectionDes	- Ajouter et aller à travers les dés - Comparer des dés	Container Itérateur
IScore	- Calculer le score d'un tour et calculer le vainqueur	Aucune
Container	- Cherche l'itérateur	Aucune

Iterateur	- Gère l'itération des objets	aucune
TestDe	- Contient les tests unitaires pour les dés	aucune
TestJeu	- Contient des tests de jeu (test pour Bunco, test des tours, test des joueurs, etc)	aucune
TestJoueur	- Teste la situation quand 2 joueurs ont le même score	aucune
BuncoPlus	- Initialise le jeu BuncoPlus	Jeu
ScoreBuncoPlus	- Gère le score du jeu BuncoPlus	IScore

2.2 Diagramme des classes

***Voir Annexe**

2.3 Faiblesses de la conception

Dans la classe BuncoPlus et Beetle, il y a deux méthodes qui sont exactement pareilles. On a la méthode initilizeDes et la méthode initilizeJoueurs. On aurait dû les mettre dans la classe Jeu. On s'est rendu compte de ce problème après avoir implémenté un deuxième jeu. Il y a un problème de cohésion, si l'on n'avait pas implémenté un autre jeu on ne s'en serait pas rendu compte du problème de cohésion.

La solution à ce problème de conception aurait été de mettre les méthodes dans la classe mère de manière à rendre les méthodes accessibles à toutes les classes. Si jamais celle-ci avait besoin de faire l'instanciation différemment, elles pourraient simplement « override » les méthodes nécessaires à leur bon fonctionnement.

2.4 Diagramme de séquence (uml)

2.4.1. Exemple qui illustre la dynamique du patron stratégie

***Voir annexe**

2.4.2. Autre diagramme de séquence

***Voir annexe**

3 DÉCISIONS DE CONCEPTION/D'IMPLÉMENTATION

3.1 DÉCISION 1 : MÉTHODE RUN()

Une des décisions de conception auquel nous avons fait face était celle de trouver où implanter la méthode principale qui roule le jeu.

Solution 1: La méthode « run() » dans la classe BuncoPlus.

```
public void run() {

    Iterateur iterJ = super.getCollectionJoueurs().getIterator();
    Boolean finPartie = false;

    while (iterJ.hasNext() && !finPartie) {
        System.out.println("Tour : " + super.getNbTour());
        Joueurs j = (Joueurs) iterJ.next();

        System.out.println("------ JOUEUR ----- [" + j.getNumeroJoueurs() +
                + "pour jouer votre tour");
        scanner.nextLine();
        super.getCollectionDes().brasser();
        scoreTour = scoreBuncoPlus.calculerScoreTour( jeu: this);
        j.setScoreJoueur(scoreTour);

        System.out.println("Le score total du joueur " + j.getNumeroJoueurs(
        finPartie = checkFinPartie(iterJ);
        iterJ = addTour(iterJ);

    }
}
```

Cette solution me semblait être une bonne solution car elle permettait d'abord d'être implantée de manière abstraite dans la super classe abstraite et elle permettait d'obtenir une certaine logique dans l'appel des méthodes. Nous faisions d'abord un appel d'une méthode d'instanciation « initialize() » puis nous appelons la méthode « run() » et finalement « calculerLeVainqueur() ». Tout cela convient très bien à la logique du programme et aussi de comment nous jouons à un jeu. Le problème avec cette méthode, c'est que le programme devient très dépendant de cette dernière et il devient difficile d'en faire une méthode simple et d'en extraire les facteurs importants et de sous diviser les tâches.

Solution 2 : Implanter toute la structure de tour dans la méthode « calculerScoreTour » de la classe ScoreBuncoPlus.

Dans cette option, nous avons tenté de faire l'appel original décrit dans le scénario du TP, soit : l'appel des méthodes « calculerScoreTour() » et « calculerLeVainqueur() ». En choisissant de ne pas découpler la méthode principale qui roule le jeu de la méthode du score du tour, cela aurait créé une énorme dépendance à cette méthode et un grand problème de cohésion car celle-ci n'effectuerait pas juste le calcul du score, mais aussi toute la logique de la notion de tour … ce qui n'était pas très cohésif. Un autre problème de cette solution était l'impossibilité de la tester car elle effectuait trop de choses, il était donc impossible pour nous de tester si le score était bien calculé.

Choix de la solution et justification

Nous avons opté pour la première solution qui était de faire l'utilisation d'un méthode « run() » qui s'occuperait de faire l'appel de la méthode de calcul du score et qui s'occuperait de la notion de tour. Malgré tout, nous trouvons que la cohésion de cette méthode n'est pas optimale. Nous avons passé beaucoup de temps pour essayer de découpler cette fonction et de la rendre plus cohésive mais nous n'y somment pas parvenus.

3.2 Décision 2 : titre de la décision

La deuxième décision d'implémentation fut l'utilisation ou non du patron Factory.

Solution 1: L'instanciation des objets de type Jeu dans la méthode main() » de la classe Main.

Lors de la première itération, nous avons opté pour une solution qui instancie le jeu approprié dans la méthode main(). Cette décision d'implémentation nous sembla, à première vue, comme étant appropriée puisque cette méthode interagie avec l'utilisateur afin de connaître son choix de jeu. Cependant, cette solution crée un couplage élevé avec les différents jeux supportés.

```
public static void main(String[] args) {

    JeuFactory jeuFactory = new JeuFactory();

    printMenu();

    Scanner scanner = new Scanner(System.in);
    int choix = scanner.nextInt();

    Jeu jeu = null;

    switch(choix) {
        case BUNCO:
            jeu = new BuncoPlus();
        case SNAKEEYES:
            snakeEyesInfo();
            jeu = new SnakeEyes();
        case BEETLE:
            beetlePrintInfo();
            jeu = new Beetle();
    }

    jeu.jouer();

}
```

Solution 2 : Déplacer l'instanciation des objets dans une classe intermédiaire.

Le choix de déplacer l'instanciation des objets dans la classe JeuFactory.java a permis de réduire le couplage entre la classe Main.java et les différents jeux. De plus, cette solution augmente la cohésion puisqu'elle donne à la classe JeuFactory.java une responsabilité unique, soit la création des objets. Ainsi, la classe Main.java utilise la logique qui est encapsulée dans la méthode getJeu() de la classe JeuFactory.java afin de délégué l'instanciation des objets.

```
public class JeuFactory {

    private final int BUNCO = 1;
    private final int SNAKEEYES = 2;
    private final int BEETLE = 3;

    /**
     * Methode qui retourne l'instance du jeu choisie par l'utilisateur
     * @param jeu integer du choix de jeu
     * @return instance du jeu choisie
     */
    public Jeu getJeu(int jeu) {

        switch(jeu) {
            case BUNCO:
                return new BuncoPlus();
            case SNAKEEYES:
                snakeEyesInfo();
                return new SnakeEyes();
            case BEETLE:
                beetlePrintInfo();
                return new Beetle();
        }
        return null;
    }
```

Choix de la solution et justification

Par conséquent, nous avons opté pour l'utilisation du patron Factory qui nous demandait de déplacer la logique de l'instantiation des objets dans une classe intermédiare. Cette décision nous a permis de réduire les responsabilités de la classe Main.java et par le fait même de réduire la quantité de lignes d'instructions dans la méthode main(). Cela a eu pour objectif d'améliorer la lisibilité de la méthode, d'augmenter la cohésion et de diminuer le couplage de la classe Main.java avec les différents jeux. Donc, présentement la responsabilité de la méthode main() est

uniquement d'instancier un objet de la classe JeuFactory() afin d'obtenir l'instance du jeu souhaité.

```java
public static void main(String[] args) {

    JeuFactory jeuFactory = new JeuFactory();

    printMenu();

    Scanner scanner = new Scanner(System.in);
    int choix = scanner.nextInt();

    Jeu jeu = jeuFactory.getJeu(choix);
    jeu.jouer();

}
```

4 CONCLUSION

Les objectifs ont été atteints avec succès. Nous avons réussi à utiliser les patrons obligatoires (stratégie, iterateur, méthode template), à utiliser appropriement l'interface Comparable et à bien implémenter des tests unitaires tout en utilisant JUnit. Le cadriciel fonctionne adéquatement, la preuve : deux autres jeux avec des règles complétement différentes de Bunco+ programmés grâce à ce cadriciel, SnakeEyes et Beetle. Un des nos points faibles a été notre solution d'implémentation, la méthode « run() », car elle ne respecte pas vraiment le principe de cohésion.

Ce projet présente un grand potentiel pour de futurs projets. Par exemple, on pourrait implémenter une interface graphique afficher les dés, où encore implémenter un mode « joueur contre joueur », où de vraies personnes jouent à différents jeux de dés.

5.RÉFÉRENCES ET ANNEXES

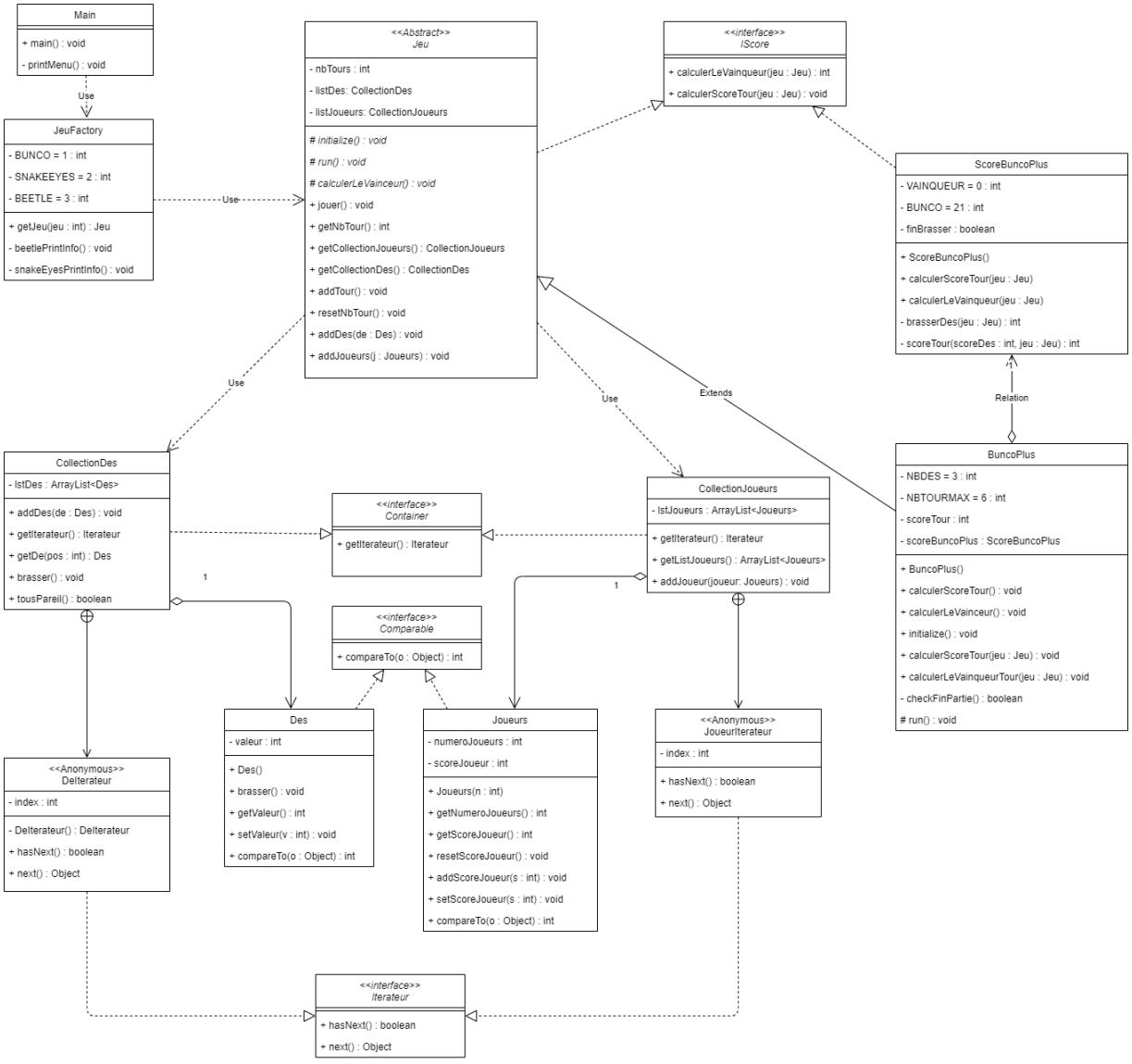

illustration 2.2.1

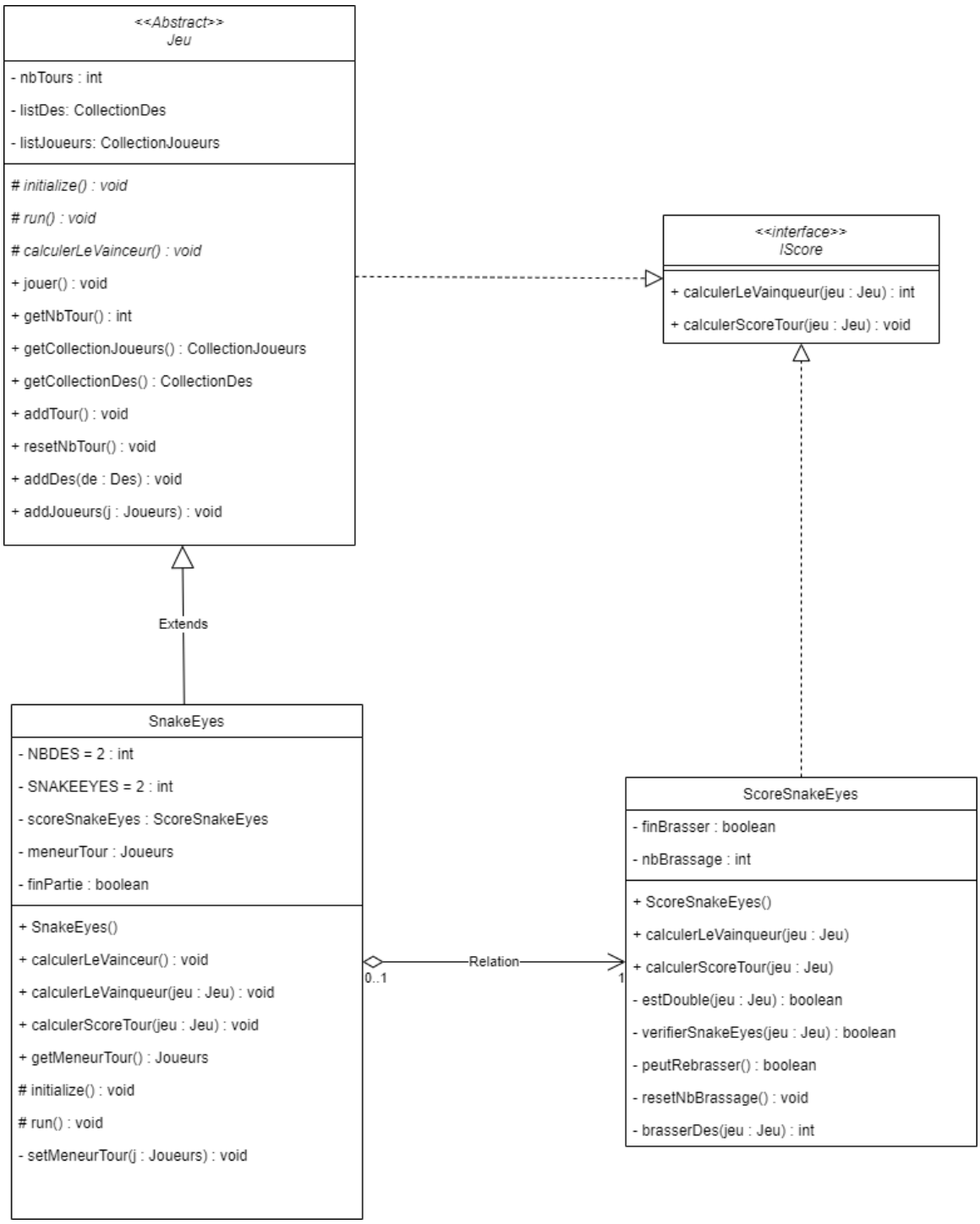

illustration 2.2.2

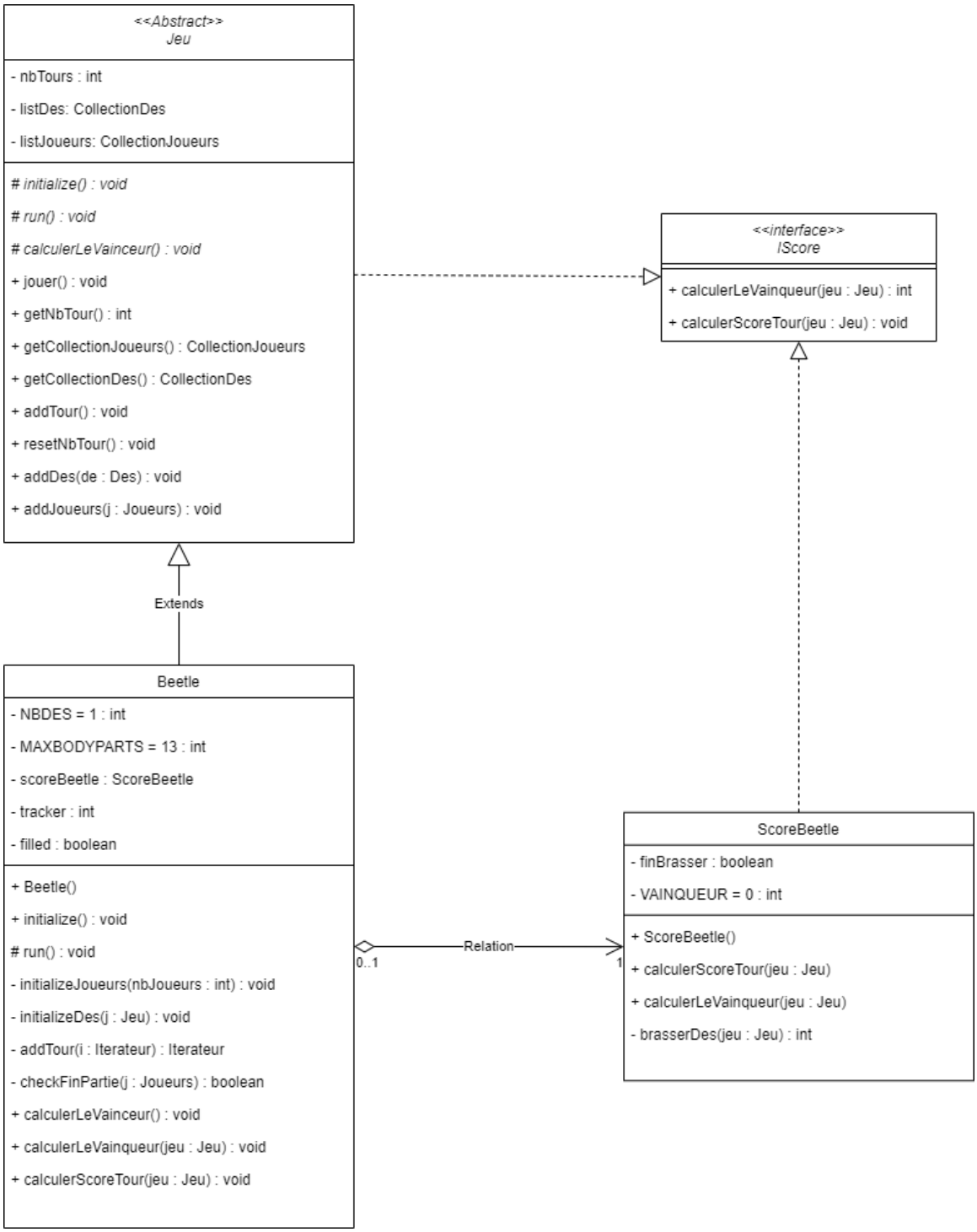

illustration 2.2.3

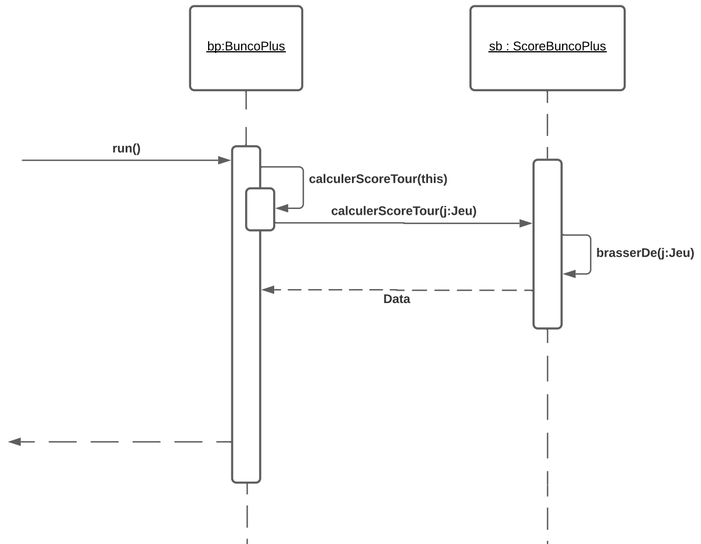

illustration 2.4.1 a

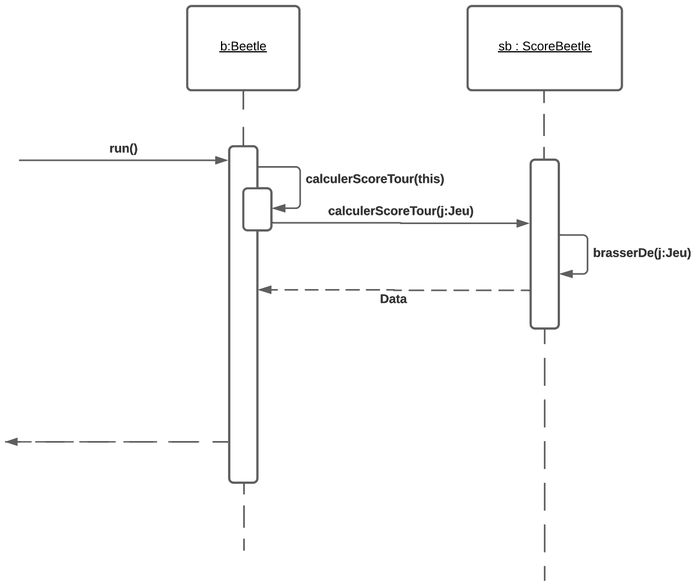

illustration 2.4.1 b

Main Sequence Diagram

Abdelli, Pierre Amar | November 9, 2020

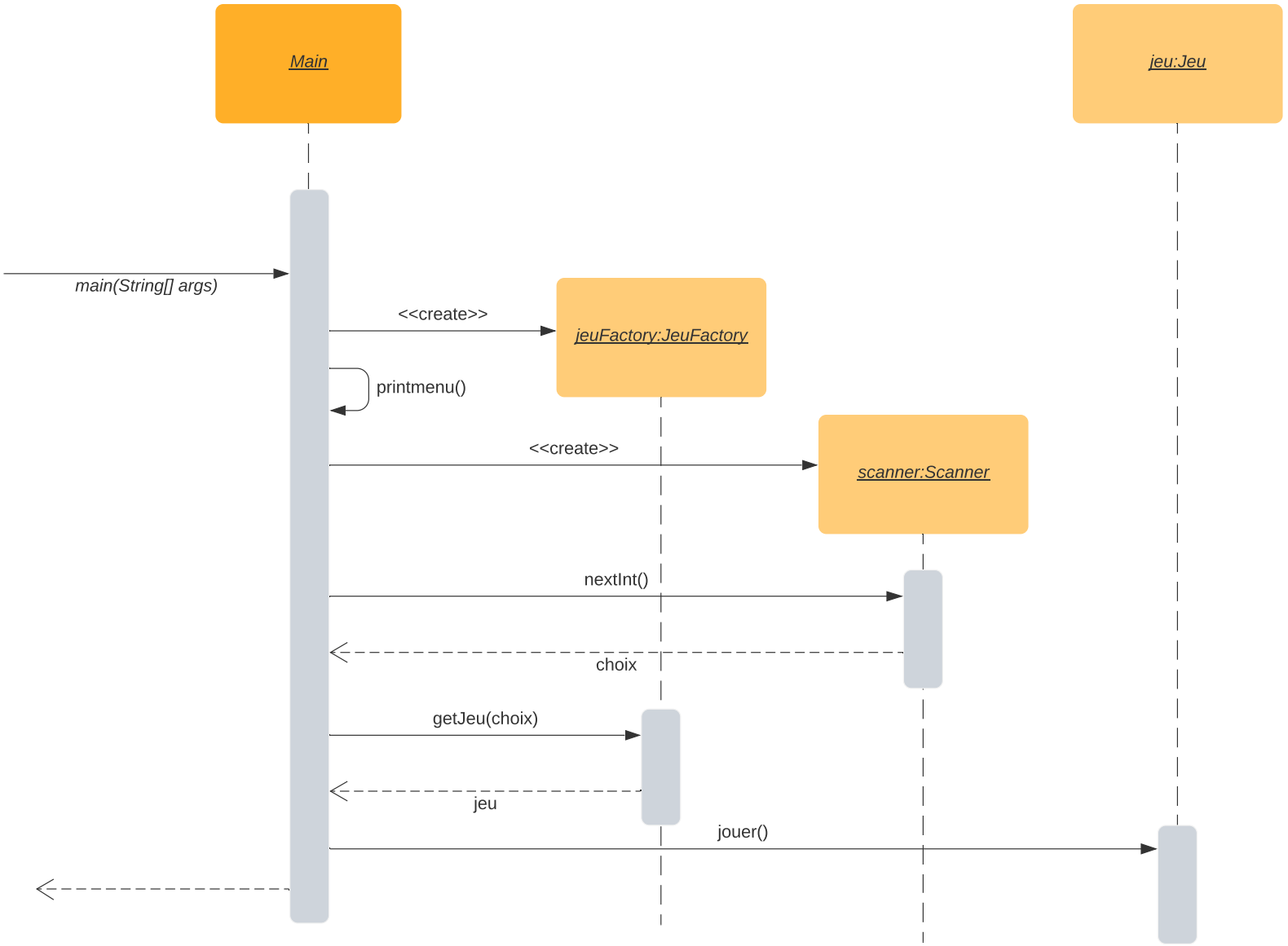

illustration 2.4.2