Le génie pour l'industrie

Département de génie logiciel et des TI

Rapport de laboratoire

N° de laboratoire	Laboratoire **2**
Étudiant(s)	Bakaleinik, Yulia Fily, Donatien Palashev, Valentin Quezada,Jonnie klein Racanelli, Anthony
Code(s) permanent(s)	BAKY30539705,FILD92110006,PALV04049901, QUEJ19129809,RACA28079504
Cours	LOG121
Session	Automne 2021
Groupe	04
Professeur	Benoit Galarneau
Chargés de laboratoire	Bianca Popa
Date de remise	Le 09 novembre 2021

1 INTRODUCTION

La programmation informatique nous ouvre beaucoup de possibilités en ce qui concerne une conception orientée objet. Elle encourage la réutilisation et l'extensibilité du programme. En effet, c'est le cas lors du développement d'un cadriciel. Celui-ci consiste à implémenter un noyau commun à des applications d'un domaine particulier en utilisant une forme d'application générique.

Les objectifs de ce laboratoire ciblent la création et l'implémentation d'un cadriciel pour un jeu de dés. Ce concept peut être utilisé par une variété d'applications de jeux nécessitant des dés contentant plusieurs variantes.

Pour atteindre ce résultat, nos objectifs s'adaptent au fonctionnement et a la logique du jeu de Bunco 21 points et a ses variantes. De plus, il est possible d'utiliser d'autres règles au jeu de dé qui accentuer le fonctionnement du logiciel.

Sous ce rapport, nous débuterons avec la présentation de nos choix et responsabilités de nos classes nécessitant au fonctionnement de notre logiciel. Notre conception sera expliquée à l'aide d'un diagramme de classe UML. Ensuite, une analyse des faiblisses de conception sera décrite avec de potentiels solutions. La présence de diagrammes de séquence en illustrant et en expliquant la dynamique du patron stratégie et un exemple du patron itérateur, sera fourni par la suite. Par la suite, nous poursuivrons avec l'explication de nos décisions concernant l'itération et le triage des données Dé et joueur, ainsi que la gestion d'entrées de l'utilisateur. Pour finir, nous conclurons avec un résumé des points forts et des points faibles de notre conception pour une possibilité de développement futurs.

2. CONCEPTION

2.1 CHOIX ET RESPONSABILITÉS DES CLASSES

Pour débuter, nous remarquons que nous avons besoin de deux composantes essentielles afin de commencer un jeu; un joueur et un dé. Les participants sont représentés par leur nom, leur nombre de points et leur nombre de victoires. Quant au dé, il sera représenté par la valeur du nombre obtenue ainsi que le nombre de faces qu'il contient.

Pour ce qui est de la classe Bunco, celle-ci est abstraite et contient deux répertoires; dé et joueur. Le calcul des points sera effectué par un calcul de ScoreClassique ou ScoreBlitz grâce à l'interface stratégie, nommé ICalculScore. De plus, uniquement la sous-classe BuncoFlex utilisera cette interface pour le patron stratégie, car l'utilisateur pourra dynamiquement changer la façon de calcul des scores en précisant si le calcul sera classique ou blitz. Les sous-classes BuncoClassique

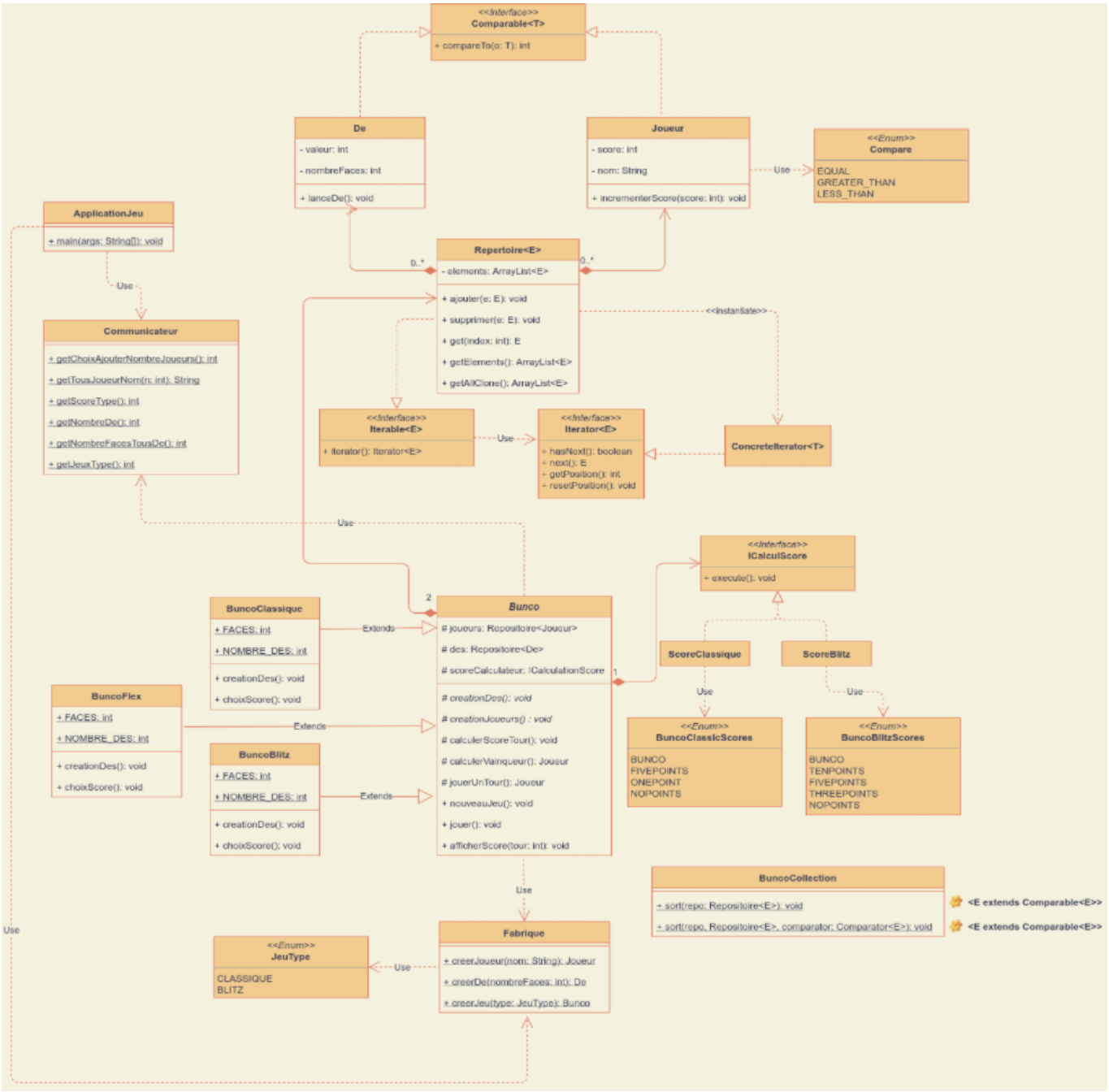
et BuncoBlitz possèdent chacun dans leur constructeur une instanciation de score classique ou blitz pour leur classe parent Bunco. Nous envisageons également l'implémentation du patron méthode Template dans cette classe.

Pour ce qui est de la classe Fabrique, celle-ci est une classe d'assistance afin de créer des joueurs, créer un dé ou créer un nouveau type de jeu (Classique, Blitz ou Flex).

Classe	*Responsabilités*	*Dépendances*
- De	– Obtenir le nombre sur le dé correspondant sur l'une de ses faces.	– Comparable – Répertoire
- Joueur	– Participer au jeu et tenir compte de son pointage.	– Comparable – Répertoire
- Répertoire	– Classe récipient de joueur et dé.	– De – Joueur
- Bunco	– Représenter le jeu dans son ensemble – Déterminer le vainqueur du tour et de la partie – Affichage des points.	– Répertoire – BuncoClassique – BuncoBlitz – BuncoFlex – ICalculScore – Communicateur
- BuncoClassique	– Utiliser la règle classique d'un jeu Bunco.	– ICalculScore
- BuncoBlitz	– Utiliser la règle blitz d'un jeu Bunco, c'est-à-dire, avec plus de dés.	– ICalculScore
- BuncoFlex	– Détermine le nombre de dés et le nombre de faces que le joueur a	– ICalculScore

	défini. – Choisir de quel façon le pointage sera calculé.	
- Fabrique	– Création d'un jeu, de dés et de joueurs.	– Bunco – ApplicationJeu
- ICalculScore	– Implémente la stratégie sélectionnée par l'utilisateur.	– Bunco
- ScoreClassique	– Le joueur peut augmenter son score s'il obtient un point, cinq points et grâce à un bunco.	– ICalculScore
- ScoreBlitz	– Le joueur peut augmenter son score s'il obtient trois points, cinq points, dix points et grâce à un bunco.	– ICalculScore

2.2 DIAGRAMME DES CLASSES



Pour débuter une partie de Bunco, le lancement se fait à partir de la classe ApplicationJeu, c'est ici que nous déterminons quel type de jeu nous voulons jouer parmi ceux qui ont été mentionnées dans la section précédente. À la suite du choix effectuée par l'utilisateur, celui-ci

devra déterminer le nombre de joueurs présents dans la partie ainsi que leurs noms. Si l'utilisateur choisi une partie de type Bunco Flex, il devra préciser le nombre de dés qui seront utilisés, ainsi que le nombre de faces que chacune contient. Ces décisions sont directement liées aux classes Fabrique et Communicateur, dont leur but est de créer un jeu, créer les joueurs ainsi que de créer les dés grâce à leurs mutateurs.

La classe Joueur permet de tenir en compte les points des joueurs présents dans la partie. Leur nombre de points incrémentera selon le résultat qu'il a obtenu lors de son lancer. Après chaque tour, le pointage des joueurs est comparé entre eux afin de déterminer le gagnant. La classe Dé a pour but d'effectuer un lancer dont le résultat aura une valeur aléatoire selon le nombre de faces présent. Cette valeur est ensuite comparée afin de déterminer si le joueur a obtenu des points lors de ce tour. Dans cette perspective, nous avons ajouté la classe Répertoire qui prend les éléments de type dé ou joueur sous forme d'ArrayList avec une seule implémentation. Celle-ci fait partie du parton Itérateur, qui lui s'assure de garder et de changer le pointage des joueurs, après chaque tour.

2.3 Faiblesses de la conception

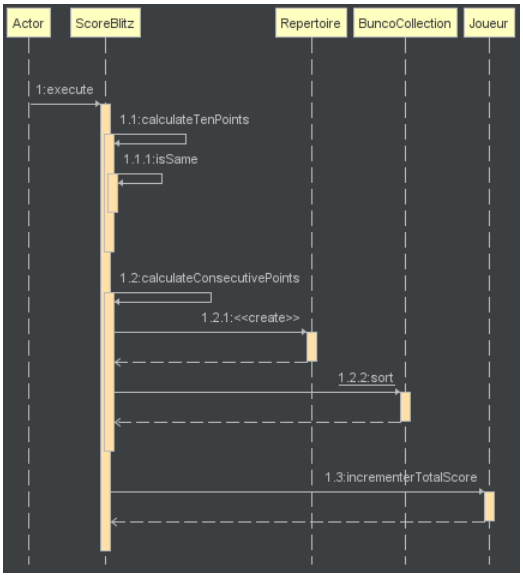
Tout d'abord, l'utilisation de classes Enum qui regroupent des variables constantes et peuvent avoir une bonne utilité lors de la conception d'un jeu de dés. Dans notre cas nous utilisons ce concept dans plusieurs classes, par exemple, nous avons besoins de ces variables constantes afin de définir le score, donc le nombre de points attribué ne changera pas. Par conséquent, il sera possible d'éviter de les utiliser lors de la création d'un jeu Bunco. Dans notre conception d'un jeu Bunco, nous utilisons le Switch/Case afin que l'utilisateur puisse choisir quel façon le jeu Bunco sera joué. En conséquence, ce concept est représenté de façon statique. Mais, nous avons pensés à une solution qui serai de rendre la création de façon dynamique. En effet, ça nous évitera de créer des nouveaux types de pointage ou même de nouvelles variantes du jeu Bunco, car il faudra à chaque fois les rajouter dans les classes Enum. Idéalement nous aurions à enter la valeur une seule fois dans ces classes sans avoir à les modifier à chaque fois.

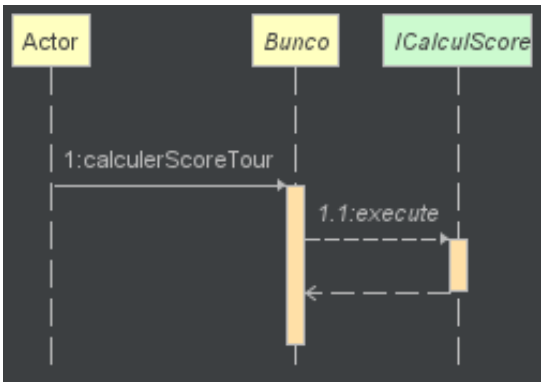
Ensuite, notre conception ne respecte pas la totalité du principe de la responsabilité unique des classes pour le fonctionnement du jeu Bunco. En effet, il y a des instances dans notre logiciel qui respectent cette conception. Cependant, nous surchargeons des classes et des méthodes avec plusieurs responsabilités. Les classes ScoreClassique et ScoreBlitz sont un exemple de cette faiblesse de conception, car cela a été fait intentionnellement pour que cela fonctionne avec Bunco. La méthode execute() dans ces classes, prend deux rôles dans lesquels elle fait une incrémentation si un utilisateur gagne la manche. De plus, elle compte et retourne un booléen disant s'il a un prochain tour, donc elle prend deux responsabilités à la fois. Donc, une solution pour ce problème serai de séparer celles-ci en deux méthodes distinctes.

2.4 Diagramme de séquence (UML)

2.4.1. Exemple qui illustre la dynamique du patron stratégie (Bunco Blitz)

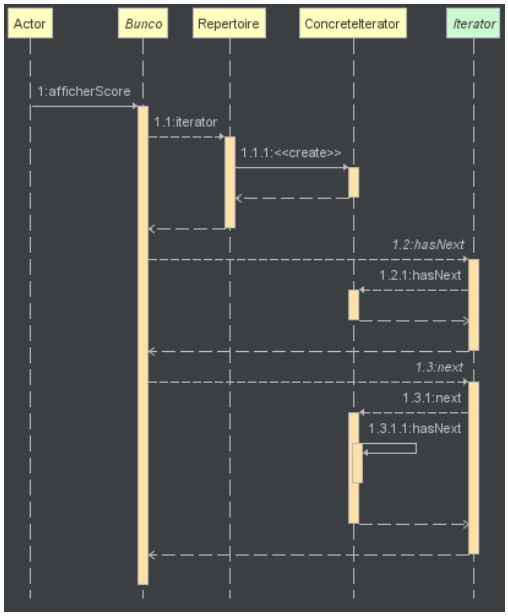
La méthode calculerScoreTour() est appelée à chaque tour afin de déterminer le nombre de points que le joueur a obtenu lors de cette manche. Cette méthode est exécutée dans l'interface ICalculScore qui est associé à la stratégie sélectionnée par l'utilisateur. Par la suite, la classe ScoreBlitz attribuera le nombre de points obtenu au joueur selon les dés qu'il a lancé pendant ce tour. En effet, selon les valeurs obtenues sur les dés, il sera attribué soit; un bunco (21 points), dix points, cinq points, trois points ou aucun point. Les points que le joueur a obtenu lors de chaque tour sera incrémenté et additionné dans son score global. Ce fonctionnement continue jusqu'au dernier tour afin de déterminer le joueur qui a le plus de point et le déclarer gagnant.





2.4.2. Exemple qui illustre la dynamique du patron itérateur

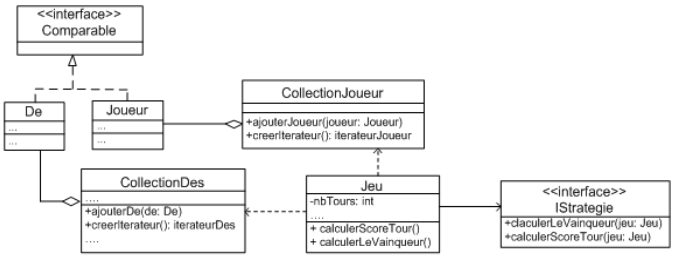
La méthode afficherScore() est appelée depuis la classe abstraite Bunco, son but est d'afficher le pointage des joueurs après chaque tour de la partie jusqu'à ce que celle-ci se termine. Par la suite, en passant par la classe répertoire, l'itération des joueurs et des dés est effectuée. Celle-ci permet de changer le score actuel du joueur selon son lancer de dés après chaque manche. Ensuite, le pointage des joueurs est affiché par l'application à la fin de chaque tour. D'ailleurs, tant que la fonction hasNext() est appelée par la classe Bunco et que l'itérateur de joueur donne vrai, la classe ConcreteIterator reçois le prochain joueur dont le pointage devra être affiché. Cette dynamique aide aussi à déterminer qui sera le potentiel gagnant de la partie.



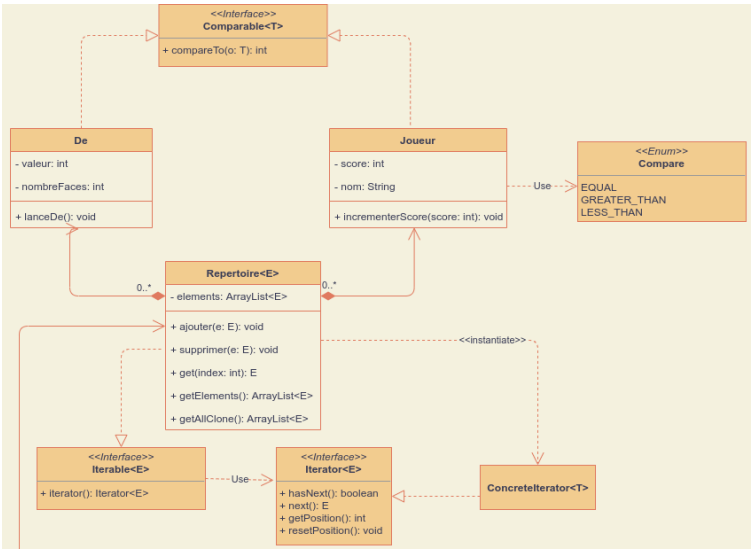
3 DÉCISIONS DE CONCEPTION/D'IMPLÉMENTATION

3.1 DÉCISION 1 : ITÉRATION ET TRIAGE DES DONNES DÉ ET JOUEUR

- **Contexte**: Pour un jeu Bunco, il est important de faire une itération et un triage pour les données des dés et des joueurs. Notre but est de trouver une solution qui aura la meilleure efficience.
- **Solution 1**: il est possible de séparer chacune de ces classes afin qu'elles aillent leur propre collection, c'est-à-dire, d'implémenter des classes CollectionDé et CollectionJoueur qui avait pour but de contenir les éléments dé et les éléments joueur. Nous pouvons considérer un avantage à cette solution qui est la facilité du débogage. D'autre part, il sera nécessaire d'implémenter un itérateur pour chaque collection, ce qui diminue l'efficience de l'application.



- **Solution 2** : l'implémentation d'une classe repositoire qui prend les éléments de type dé ou joueur sous forme d'ArrayList avec une seule implémentation. C'est un moyen plus simple car nous aurions besoin d'un seul itérateur. L'avantage à utiliser ce moyen est une architecture du logiciel plus soignée et une bonne efficience du code.



- **Choix:** Grâce à la solution 2, nous avons décidé qu'il est plus simple d'utiliser une seule implémentation qui peut accepter plusieurs types d'éléments (dé et joueur). Donc, l'itération se fait à partir de la classe répertoire, qui elle peut être de type joueur ou de type dé. De cette façon, l'architecture du logiciel est moins chargée et nous permet d'utiliser un seul itérateur pour deux classes différentes.

3.2 Décision 2 : Gestion d'entrées de l'utilisateur

- **Contexte**: Pour débuter un jeu Bunco, nous avons besoin de trouver le meilleur moyen pour récupérer la saisie des donnes de l'utilisateur lorsqu'il spécifie la variante de Bunco qui sera joué en plus du nombre de joueurs présent dans la partie et le nombre de dés ainsi que le nombre de faces dont il contient, si c'est le cas.
- **Solution 1** : il est possible de créer une classe (communicateur) qui manipule tous les saisies possibles de l'utilisateur à l'aide d'un scanner. Cette classe comportera toutes les méthodes scanner des données que l'utilisateur devra identifier. L'avantage de cette démarche c'est que son applique le principe de responsabilité unique. Il sera également plus simple d'effectuer les tests unitaires du logiciel.

```java
public final class Communicateur {
    private static Scanner scan = new Scanner(System.in);

    private Communicateur() {}

    public static int getChoixAjouterNombreJoueurs() {
        System.out.print("\nAjouter combien de joueur? (2 ou plus): ");
        return Integer.parseInt(scan.nextLine());
    }

    public static String getNomDesJoueurs(int n) {
        System.out.print("Nom du joueur #" + n +": ");
        return scan.nextLine();
    }

    public static int getTypeScore() {
        System.out.print("\n(1) Classique");
        System.out.print("\n(2) Blitz");
        System.out.print("\nSélectionnez le calcul du score en tapant le numéro correspondant: ");
        return Integer.parseInt(scan.nextLine());
    }

    public static int getNombreDe() {
        System.out.print("\nAjouter combien de dés?: ");
        return Integer.parseInt(scan.nextLine());
    }

    public static int getNombresFacesDes() {
        System.out.print("Nombre de faces pour tous les dés: ");
        return Integer.parseInt(scan.nextLine());
    }

    public static int getTypeJeu() {
        System.out.print("\n(1) Bunco Classique");
        System.out.print("\n(2) Bunco Blitz");
        System.out.print("\n(3) Bunco Flex");
        System.out.print("\nSélectionnez le jeu en tapant le numéro correspondant: ");
        return Integer.parseInt(scan.nextLine());
    }
}
```

```java
public class ApplicationJeu {
    public static void main(String[] args) {
        int position = 0;
        Bunco jeu = null;
        while (position < 1 || position > 3 ){
            position = Communicateur.getTypeJeu();
        }

        jeu = switch(position) {
            case 1 -> Fabrique.creerJeu(JeuType.CLASSIQUE);
            case 2 -> Fabrique.creerJeu(JeuType.BLITZ);
            case 3 -> Fabrique.creerJeu(JeuType.FLEX);
            default -> throw new IllegalStateException("Unexpected value: " + position);
        };
        jeu.nouveauJeu();
    }
}
```

- **Solution 2** : appliquer un scanner dans toutes les classes qui nécessitent cette utilisation. Grâce a cette méthode il sera possible de minimiser le nombre de classes dans l'architecture du logiciel. Par conséquent, les tests unitaires seront très compliqués à implémenter. De plus, il sera difficile de savoir d'où proviennent toutes les entrées effectuées par l'utilisateur.

```java
public class ApplicationJeu {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);

        int position = 0;
        Bunco jeu = null;
        while (position < 1 || position > 3 ){
            System.out.print("\n(1) Bunco Classique");
            System.out.print("\n(2) Bunco Blitz");
            System.out.print("\n(3) Bunco Flex");
            System.out.print("\nSélectionnez le jeu en tapant le numéro correspondant: ");

            position = Integer.parseInt(scan.nextLine());
        }

        jeu = switch(position) {
            case 1 -> Fabrique.creerJeu(JeuType.CLASSIQUE);
            case 2 -> Fabrique.creerJeu(JeuType.BLITZ);
            case 3 -> Fabrique.creerJeu(JeuType.FLEX);
            default -> throw new IllegalStateException("Unexpected value: " + position);
        };
        jeu.nouveauJeu();
    }
}
```

- **Choix :** nous avons opté pour la première solution, car il est plus simple de visualiser d'où proviennent les entrées de l'utilisateur. En effet, la création de la classe Communicateur permet de rassembler toutes les entrées nécessaires pour toute l'application. Une centralisation comme celle-ci applique le principe de responsabilité unique et simplifie la création de tests unitaires.

4 CONCLUSION

Les objectifs de ce laboratoire étaient de concevoir et d'implémenter un cadriciel pour un jeu de dés. Celui-ci devait avoir la possibilité d'être utilisé par une variété jeu de dés qui contient différentes possibilités de jouer. Avec la conception de ce logiciel, nous avons entièrement atteint ces objectifs.

Pour ce qui est des points forts de notre conception, nous pouvons préciser que la décision d'implémenter une classe répertoire nous a permis de faciliter l'itération et le triage des donnes de dé et de joueur. En effet, avec une seule implémentation des éléments de type dé ou joueur sous forme d'ArrayList, nous avions besoin d'implémenter un seul itérateur, chose qui n'étais pas le cas si nous avions opté pour des classes Collections. De plus, la création de la classe Communicateur, qui respecte le principe de responsabilité unique, nous a permis de réaliser les tests unitaires beaucoup plus rapidement, car cette classe comporte toutes les méthodes scanner des données que l'utilisateur devra identifier. Alors elle avantage sur la gestion d'entrées de l'utilisateur.

Pour ce qui est des points faibles de notre conception, celle-ci se situe à la conception partielle du principe de la responsabilité unique des classes. En effet, même si nous avons pu l'intégrer a sous certains points, nous avons beaucoup de méthodes qui on multiples responsabilités. Par conséquent, avec notre conception, nous surchargeons plusieurs classes, ce qui n'est pas une bonne pratique. Un second point faible serait l'utilisation des Enums de façon statique. Le fait d'utiliser cette conception peut nous compliquer les taches si nous devions rajouter une nouvelle variation de jeu.

Pour conclure, les futurs projets que nous envisageons sont la création d'autres types de jeux de cartes comme le Rami, le poker ou le blackjack, en ajoutant plus de flexibilité dans notre logiciel. De cette façon il sera possible de fusionner les deux ensembles pour créer un tout nouveau jeu, utilisant des dés et des cartes.