Jeu des 13 allumettes

1 Description fonctionnelle

Le jeu des 13 allumettes se joue à deux joueurs qui, à tour de rôle, prennent 1, 2 ou 3 allumettes d'un tas qui en contient initialement 13. Le joueur qui prend la dernière allumette perd.

L'application à développer doit permettre à deux joueurs de s'affronter. C'est l'ordinateur qui est l'arbitre de la partie. Il affiche le nombre d'allumettes restant en jeu, donne la main à tour de rôle à chaque joueur et vérifie que les joueurs respectent les règles du jeu. Ainsi, si un joueur veut prendre trop ou trop peu d'allumettes, le coup est refusé et le joueur doit rejouer. Un joueur doit retirer entre 1 et 3 allumettes. En fin de partie, l'arbitre affiche le résultat.

Chaque joueur choisit le nombre d'allumettes à prendre en fonction d'une stratégie. Une stratégie *humain* consiste à demander à l'utilisateur ce nombre. Ceci permet d'avoir un joueur humain. Les autres stratégies consistent à faire jouer l'ordinateur et définissent son niveau de jeu. Une liste non exhaustive des niveaux de jeu, et donc des stratégies correspondantes, inclut :

— niveau *rapide* : l'ordinateur prend le maximum d'allumettes possible (de manière à ce que la partie se termine le plus rapidement possible),

— niveau *naïf* : l'ordinateur choisit aléatoirement un nombre entre 1 et 3,

— niveau *expert* : l'ordinateur joue du mieux qu'il peut (s'il peut gagner, il gagnera).

Le listing suivant constitue une copie des informations affichées à l'écran lors du déroulement d'une partie opposant un humain (Xavier) à l'ordinateur (Ordinateur) en niveau naïf.

```
1  Allumettes restantes : 13
2  Xavier, combien d'allumettes ? 2
3  Xavier prend 2 allumettes.
4
5  Allumettes restantes : 11
6  Ordinateur prend 1 allumette.
7
8  Allumettes restantes : 10
9  Xavier, combien d'allumettes ? X
10 Vous devez donner un entier.
11 Xavier, combien d'allumettes ? 5
12 Xavier prend 5 allumettes.
13 Impossible ! Nombre invalide : 5 (> 3)
14
15 Allumettes restantes : 10
16 Xavier, combien d'allumettes ? 0
17 Xavier prend 0 allumette.
18 Impossible ! Nombre invalide : 0 (< 1)
19
20 Allumettes restantes : 10
21 Xavier, combien d'allumettes ? 3
22 Xavier prend 3 allumettes.
23
```

```
Allumettes restantes : 7
Ordinateur prend 1 allumette.

Allumettes restantes : 6
Xavier, combien d'allumettes ? 2
Xavier prend 2 allumettes.

Allumettes restantes : 4
Ordinateur prend 2 allumettes.

Allumettes restantes : 2
Xavier, combien d'allumettes ? 1
Xavier prend 1 allumette.

Allumettes restantes : 1
Ordinateur prend 2 allumettes.
Impossible ! Nombre invalide : 2 (> 1)

Allumettes restantes : 1
Ordinateur prend 1 allumette.

Ordinateur perd !
Xavier gagne !
```

2 Architecture

L'analyse du problème a été commencée et un diagramme de classe d'analyse, partiel, est proposé sur la figure 1. La classe `Jeu` modélise le plateau du jeu des 13 allumettes, y compris les règles sur la prise des allumettes. Le jeu est caractérisé par le nombre d'allumettes encore présentes, initialement 13.

Il est possible de retirer du jeu un nombre d'allumettes compris entre 1 et 3. 3 est le nombre maximal d'allumettes que peut prendre un joueur par tour. Bien entendu, il ne peut pas prendre plus d'allumettes qu'il n'en reste en jeu. Le non respect de ces règles sera signalé par une exception, contrôlée par le compilateur, appelée `CoupInvalideException`.

Le nombre maximal d'allumettes que peut prendre un joueur sera représenté par une constante de la classe Jeu. Cette constante est la même pour tous les jeux.

La classe `Joueur` modélise un joueur. Un joueur a un nom. On peut demander à un joueur le nombre d'allumettes qu'il veut prendre pour un jeu donné (`getPrise`). Un joueur détermine le nombre d'allumettes à prendre en fonction de sa stratégie : naïf, rapide, expert ou humain. Il pourrait y avoir d'autres stratégies.

La classe `Arbitre` fait respecter les règles du jeu aux deux joueurs. Cette classe possède un constructeur qui prend en paramètre les deux joueurs `j1` et `j2` qui vont s'affronter. Une (et une seule) partie peut alors être arbitrée entre ces deux joueurs. L'arbitre fait jouer, à tour de rôle, chaque joueur en commençant par le joueur `j1`. Celui qui prend la dernière allumette a perdu. Faire jouer un joueur consiste à afficher le nombre d'allumettes encore en jeu, lui demander le nombre d'allumettes qu'il souhaite prendre, afficher ce nombre, puis à retirer les allumettes du

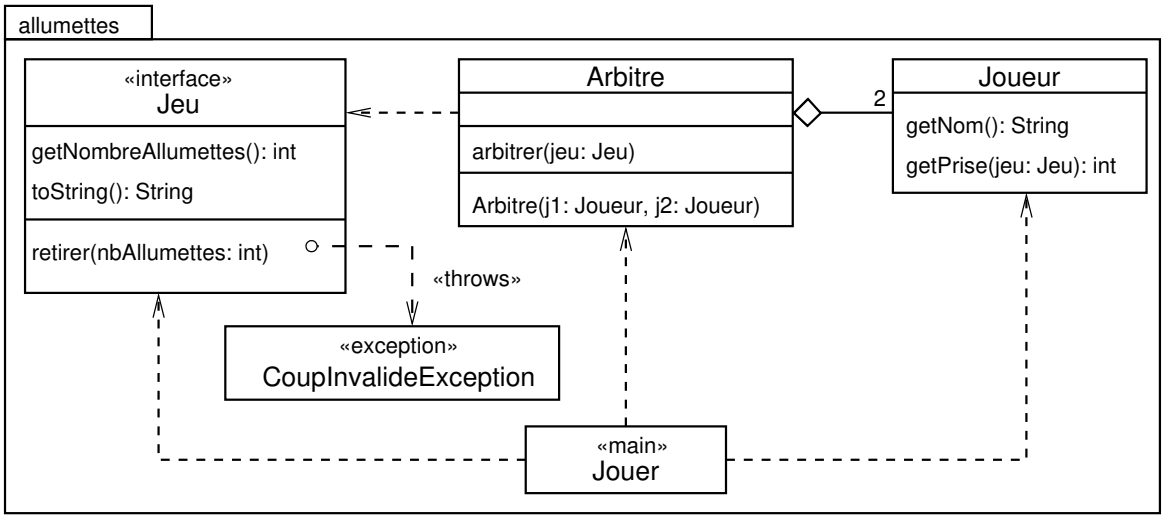

FIGURE 1 – Diagramme de classe d'analyse pour le jeu des 13 allumettes

jeu. Si ceci provoque une violation des règles du jeu (exception `CoupInvalideException`), le même joueur devra rejouer.

Dans la modélisation proposée, on constate que l'arbitre communique le jeu aux joueurs. Un joueur a ainsi accès au jeu. Il serait possible de programmer une stratégie consistant à retirer toutes les allumettes du jeu sauf 2. En proposant alors de ne prendre qu'une seule allumette, le joueur indélicat est sûr de gagner. Bien sûr, il y a triche. Dans ce cas la partie devrait être immédiatement arrêtée et le joueur désigné comme tricheur !

On définira une stratégie de jeu qui correspond à un joueur qui triche. Voici un exemple de déroulement de partie entre un joueur Ordinateur qui applique une stratégie rapide et un joueur Xavier qui applique la stratégie tricheur. Xavier dit quand il triche : `[je triche...]`.

```
1 Allumettes restantes : 13
2 Ordinateur prend 3 allumettes.
3
4 Allumettes restantes : 10
5 [Je triche...]
6 Abandon de la partie car Xavier triche !
```

En l'état actuel de la modélisation, l'arbitre ne peut pas détecter [1] ce type de triche. Une solution consiste à s'appuyer sur le patron de conception Procuration (aussi appelé Mandataire, Proxy, etc.) dont l'architecture est donnée à la figure 2. Au lieu d'accéder au sujet réel, le client accède à l'objet procuration qui relaie l'opération vers le sujet réel. Ici, le client est le joueur et le sujet réel est le jeu. Nous allons nous servir de la procuration pour interdire au joueur [2] de

1. En fait, avec un jeu aussi simple que les 13 allumettes, il suffirait de 1) mémoriser le nombre d'allumettes encore en jeu avant de communiquer le jeu au joueur et 2) vérifier que le nombre d'allumettes en jeu n'a pas changé après que le joueur a indiqué le nombre d'allumettes choisi. Pour un jeu plus compliqué, il peut être difficile de vérifier qu'il n'y a pas eu de changement apporté au jeu. Cette solution **ne doit pas** être mise en œuvre.

2. En fait, un tricheur connaissant bien Java pourrait quand même arriver à ses fins, même avec cette solution.

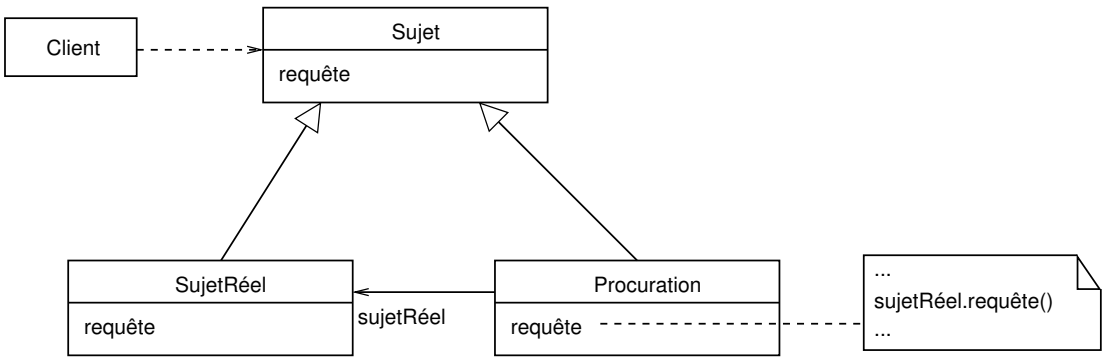

FIGURE 2 – Diagramme de classe du patron de conception Procuration

modifier le jeu. Si le joueur appelle la méthode `retirer` de la procuration, la procuration lèvera une exception `OperationInterditeException`. Pour les autres méthodes, la procuration appelle simplement l'opération correspondante du sujet réel. On note que l'utilisation de la procuration se fait sans aucune modification du client.

Enfin, la classe `Jouer` permet de lancer un jeu des 13 allumettes entre deux joueurs. C'est la classe principale de l'application. Elle exploite les arguments de la ligne de commande pour configurer la partie à lancer. Les deux arguments désignent respectivement le premier et le second joueur et respectent le même format : `nom@stratégie`, où `nom` est le nom du joueur et `stratégie` est le nom de la stratégie à utiliser pour ce joueur. Les valeurs possibles pour la stratégie sont `humain`, `naif`, `rapide`, `expert` et `tricheur`. L'exemple d'exécution donné en début de sujet correspond au lancement suivant :

```
java allumettes.Jouer Xavier@humain Ordinateur@naif
```

L'exemple de triche correspond à :

```
java allumettes.Jouer Ordinateur@rapide Xavier@tricheur
```

Un argument optionnel, forcément en première position, permet de rendre l'arbitre confiant. L'arbitre transmet alors directement le jeu au joueur sans utiliser le proxy. Cet argument optionnel est "-confiant". Voici comment jouer avec un arbitre confiant.

```
java allumettes.Jouer -confiant Ordinateur@rapide Xavier@tricheur
```

Le résultat est alors le suivant

```
1  Allumettes restantes : 13
2  Ordinateur prend 3 allumettes.
3
4  Allumettes restantes : 10
5  [Je triche...]
6  [Allumettes restantes : 2]
7  Xavier prend 1 allumette.
8
9  Allumettes restantes : 1
10 Ordinateur prend 1 allumette.
```

```
11
12 Ordinateur perd !
13 Xavier gagne !
```

Que s'est-il passé ?

L'arbitre demande au joueur Ordinateur combien il prend d'allumettes. Ordinateur répond 3 et l'arbitre les retire du jeu. Il en reste 10.

L'arbitre demande à Xavier combien il prend d'allumettes. Xavier commence par retirer des allumettes du jeu pour n'en laisser que deux (c'est un tricheur !). Il répond ensuite à l'arbitre qu'il en prend une. L'arbitre retire une allumette du jeu. Il en reste une seule.

Ordinateur est obligé de la prendre : Ordinateur perd et Xavier gagne !

Si on exécute sans l'option "-confiant", l'arbitre détecte la triche et sanctionne le tricheur. On retrouve bien l'exécution précédente avec abandon de la partie parce que Xavier triche.

L'humain peut aussi tricher. Par exemple, si l'utilisateur répond « triche » quand le programme lui demande le nombre d'allumettes qu'il souhaite prendre, on retirera discrètement du jeu une allumette. Ceci permettra au joueur indélicat de gagner même fasse à l'expert ! Bien sûr, il faut que l'arbitre soit confiant sinon la partie sera abandonée. Voici un exemple entre Xavier qui commence (un humain) et un Ordinateur expert.

```
1  Allumettes restantes : 13
2  Xavier, combien d'allumettes ? 1
3  Xavier prend 1 allumette.
4
5  Allumettes restantes : 12
6  Ordinateur prend 3 allumettes.
7
8  Allumettes restantes : 9
9  Xavier, combien d'allumettes ? 3
10 Xavier prend 3 allumettes.
11
12 Allumettes restantes : 6
13 Ordinateur prend 1 allumette.
14
15 Allumettes restantes : 5
16 Xavier, combien d'allumettes ? triche
17 [Une allumette en moins, plus que 4. Chut !]
18 Xavier, combien d'allumettes ? 3
19 Xavier prend 3 allumettes.
20
21 Allumettes restantes : 1
22 Ordinateur prend 1 allumette.
23
24 Ordinateur perd !
25 Xavier gagne !
```

Pour obtenir ce résultat, la commande lancée est :

```
java -confiant Xavier@humain Ordinateur@expert
```

3 Contraintes de réalisation

Les contraintes de réalisation à respecter *impérativement* sont les suivantes :

- C_1 Ce projet est un travail individuel. *Toute triche sera sanctionnée par la note minimale.*
- C_2 Le projet doit fonctionner sans aucune modification sur les machines Unix des salles de TP du bâtiment C de l'ENSEEIHT.
- C_3 Les principes énoncés en cours et étudiés en TD et TP doivent être respectés.
- C_4 La documentation des classes n'est pas explicitement demandée. Elle peut cependant être utile à la compréhension du travail fait.
- C_5 Les lettres accentuées ne seront pas utilisées dans les identifiants.
- C_6 Toutes les classes de l'application seront placées dans un unique paquetage appelé `allumettes`.
- C_7 La solution proposée doit respecter le diagramme de classe de la figure 1 : les éléments présents sur le diagramme de classe doivent être présents dans votre solution. Il est possible d'ajouter d'autres éléments (classes, méthodes, constructeurs...). En particulier, toutes les classes de l'application ne sont pas représentées sur ce diagramme.
- C_8 Pour demander à l'utilisateur le nombre d'allumettes qu'il souhaite prendre, on utilisera la classe `java.util.Scanner`.
- C_9 Même si ici nous utiliserons toujours 13 allumettes, le jeu doit permettre d'avoir un nombre initial d'allumettes quelconque.
- C_{10} Pour écrire la classe Jouer, on utilisera la méthode `split` de `String` pour récupérer le nom et la stratégie de chacun des joueurs.
- C_{11} Pour traiter la robustesse lors de l'interprétation des arguments de la ligne de commande, il est conseillé d'utiliser le mécanisme d'exception.
- C_{12} Pour le niveau *naïf*, on utilisera la classe `java.util.Random`.
- C_{13} On doit pouvoir ajouter de nouvelles stratégies de jeu sans modifier aucune des classes existantes à l'exception de la classe `Jouer`. Par exemple, on pourrait ajouter une stratégie *lente* qui consiste à toujours prendre une seule allumette.
- C_{14} La solution retenue doit permettre de changer en cours de partie la stratégie suivie par un joueur. Il n'est cependant pas demandé d'implanter ce changement de stratégie.
- C_{15} Les classes de tests unitaires utiliseront le suffixe Test et doivent s'appuyer sur JUnit.
- C_{16} Les tests unitaires pour la stratégie *rapide* doivent être complets. Ils doivent aussi utiliser le minimum de classes de l'application.
- C_{17} Les tests unitaires pour les autres éléments de l'application ne sont pas demandés (mais sont certainement utiles pour détecter et éliminer les erreurs des classes écrites).
- C_{18} L'IHM (interface homme machine) doit respecter les exemples donnés dans ce sujet. Ceci est impératif car votre programme sera testé en boîte noire, donc en s'appuyant seulement sur les entrées qu'il reçoit et les sorties (affichages) qu'il produit.

  En particulier, les exemples présentés dans ce sujet doivent pouvoir être reproduits.

C_{19} Les interfaces et les classes d'exception fournies ne doivent pas être changées. Les autres classes fournies peuvent être complétées mais les méthodes et constructeurs présents sur le diagramme de classe doivent être implantés.

C_{20} On veillera à n'avoir que des méthodes courtes.

C_{21} Il est interdit d'avoir des répétitions imbriquées dans une même méthode.

C_{22} La méthode `System.exit` ne doit pas être utilisée [3].

4 Livrables

Vous devez rendre sur le SVN (voir descriptif en ligne du module) :

L_1 Le code source de vos programmes (et seulement le code source, ni les .class, ni la documentation au format html).

L_2 Le code source des programmes de test réalisés.

L_3 Le fichier texte `LISEZ-MOI.txt` qui contient des informations sur les choix réalisés et les informations jugées utiles pour comprendre le travail fait.

5 Principales dates

Les principales dates concernant ce projet sont :

— 19 février 2024 : mise en ligne (et mise à disposition) du sujet. Le descriptif en ligne du module explique comment récupérer les fichiers fournis.

— 9 mars 2024 : remise des livrables (version complète pour première évaluation) ;
Attention : Cette première version n'est pas notée. Cependant, vous pouvez avoir des points de pénalité (qui ne pourront donc pas être rattrapés avec la deuxième version) si le travail n'est pas fait sérieusement.

— avant le 16 mars 2024 : retour de la première évaluation ;

— 23 mars 2024 : remise de la deuxième version.

6 Conseils

Voici quelques conseils...

1. Dans le fichier LISEZ-MOI.txt il faudra indiquer le temps passé sur le projet. C'est une information purement indicative qui n'est bien sûr pas prise en compte dans la notation. Il est intéressant pour vous de savoir combien de temps vous passez sur une activité. Pensez

3. Le seul appel autorisé est celui déjà présent dans la classe `Jouer` Son but n'est pas d'arrêter le programme : c'est la dernière instruction exécutée, il s'arrêterait même sans lui. Son but est de définir le code de retour du programme (différent de zéro en cas de problème).

à le noter au fur et à mesure sinon vous n'en aurez qu'une vague idée, certainement loin de la réalité. Pour vous, et même si ce n'est pas demandé, il peut être intéressant de distinguer le temps passé à comprendre le sujet, à concevoir une solution, à la programmer, à mettre au point le programme, etc.

2. Lire complètement et attentivement l'ensemble du sujet. Commencer à programmer le projet sans avoir bien compris ce qui est demandé est le meilleur moyen de passer beaucoup plus de temps que nécessaire sur ce projet.
3. Ne pas avoir de retard sur les cours/TD/TP. Il est important d'avoir compris les concepts objets (jusqu'aux exceptions) pour mener à bien ce projet. Le but de ce projet est de les mettre en pratique sur un exemple réaliste mais simplifié. Il ne s'agit pas seulement d'écrire un programme qui reproduit les exemples du sujet.
4. Ne pas attendre le dernier moment ! Il est important de pouvoir prendre un peu de recul sur votre travail. Il faut donc le faire sur une durée suffisante pour regarder avec un œil différent ce qui a été fait.
5. Il est utile de discuter de la compréhension du sujet et des solutions possibles. En revanche, chacun doit écrire sa solution. Partager du code n'a d'intérêt pour personne, ni pour celui qui le donne, ni pour celui qui le récupère. Parler d'une solution est profitable pour les deux. Ceci oblige à expliquer, à trouver les bons mots, à comprendre et se faire comprendre. C'est plus long que de copier/coller mais beaucoup plus rentable à moyen et long terme.

   Savoir faire sans savoir expliquer ou faire sans comprendre n'a que peu d'intérêt.
6. Ne pas hésiter à poser des questions sur le forum Moodle.