UNIVERSITÉ CAEN NORMANDIE

UFR des Sciences

Université de Caen Normandie
Département mathématiques-informatique
2024–2025
Bruno Zanuttini

L2 informatique

TP fil rouge. Jeux

Unité SMINFL3A : Programmation Java orientée objet

L'objectif de ce TP fil rouge est de développer une application consistant en un ensemble de jeux de stratégie simples, accompagnés d'un algorithme d'intelligence artificielle capable de jouer à tous ces jeux, et d'un moteur de jeu permettant de faire jouer, à n'importe lequel de ces jeux, un joueur humain contre un autre, contre un joueur aléatoire, contre l'algorithme, ou encore l'algorithme contre lui-même, contre un joueur aléatoire, etc.

Une bibliothèque de tests (`gamestests.jar`) est fournie pour l'ensemble du fil rouge. Les tests disponibles sont décrits dans chaque partie de cet énoncé. Par ailleurs, tout le code correspondant à ce fil rouge devra être créé dans un *package* nommé `games` (et plus précisément dans des *subpackages* de `games`).

Note Les « indications » données pour certains exercices n'ont aucun caractère obligatoire. Il existe souvent d'autres façons d'écrire le code correspondant aux exercices.

Table des matières

1 Jeu de Nim

Le jeu de Nim est un jeu à deux joueurs, qui se joue traditionnellement avec des allumettes. Étant donnés deux nombres entiers positifs n et k ($k < n$), le jeu commence avec un tas de n allumettes ; chaque joueur, tour à tour, retire du tas un nombre d'allumettes de son choix, entre 1 et k (inclus) ; le joueur qui retire la dernière allumette perd.

L'objectif de cet exercice est d'écrire une classe, dont chaque instance représentera une partie de jeu de Nim. Une instance sera donc spécifiée par les joueurs impliqués dans la partie et par les paramètres n et k, et l'état de l'instance évoluera au fil de la partie.

De façon générale, on représentera les joueurs par de simples chaînes de caractères (`String`) correspondant à leur nom.

Les classes sont à écrire dans un *package* nommé `games.nim` (*subpackage* `nim` du *package* `games`). Un `main` complet de tests ressemblera à :

```
import gamestests.nim.NimTests;
[...]
boolean ok = true;
NimTests nimTester = new NimTests();
ok = ok && nimTester.testGetInitialNbMatches();
ok = ok && nimTester.testGetCurrentNbMatches();
ok = ok && nimTester.testGetCurrentPlayer();
ok = ok && nimTester.testRemoveMatches();
ok = ok && nimTester.testIsValid();
ok = ok && nimTester.testIsOver();
ok = ok && nimTester.testGetWinner();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

Exercice 1.1. *Écrire une classe* `Nim` *ayant pour attributs la taille initiale du tas (notée n ci-dessus), le nombre maximal d'allumettes qu'un joueur peut retirer (k), et deux noms de joueurs (de type* `String`*). Ajouter ensuite deux attributs, représentant le nombre courant d'allumettes et le joueur courant. La classe aura donc 6 attributs en tout.*

Exercice 1.2. *Munir la classe d'un constructeur prenant en arguments, dans cet ordre, la taille initiale du tas, le nombre maximal d'allumettes qu'un joueur peut retirer, et deux joueurs j_1 et j_2 (de type* `String`*) ; faire en sorte que ce constructeur initialise les attributs correspondants, puis initialise la taille courante du tas à la taille initiale, et le joueur courant au premier joueur passé au constructeur (j_1).*

Exercice 1.3. *Ajouter à la classe un accesseur* `getInitialNbMatches` *retournant la taille initiale du tas (à tout moment de la partie), un accesseur* `getCurrentNbMatches` *retournant la taille courante du tas, et un accesseur* `getCurrentPlayer` *retournant le joueur courant.*

Exercice 1.4. *Ajouter à la classe une méthode* `situationToString` *retournant une représentation de la situation courante comme une chaîne de caractères (par exemple la chaîne* `"Il reste 7 allumettes"`*).*

Les exercices suivants visent à implémenter les règles du jeu proprement dites.

Exercice 1.5. *Ajouter à la classe une méthode nommée* `removeMatches` *prenant un nombre d'allumettes en argument, retirant ce nombre au tas (pour le compte du joueur courant), et changeant le joueur courant.*

Exercice 1.6. *Ajouter des méthodes permettant de déterminer si un nombre d'allumettes représente un coup valide (c'est-à-dire s'il est strictement positif, inférieur ou égal au nombre maximal autorisé, et inférieur ou égal à la taille courante du tas — méthode* `isValid`*), si la partie est finie (c'est-à-dire si le nombre courant d'allumettes est nul — méthode* `isOver`*), et qui est le gagnant (en retournant* `null` *si la partie n'est pas finie — méthode* `getWinner`*).*

Pour terminer, on souhaite réaliser une petite interface (textuelle) permettant de jouer à deux joueurs.

Note Pour la saisie, on pourra utiliser la classe `Scanner` du *package* `java.util`, en s'inspirant du code suivant :

```
import java.util.Scanner;
[...]
Scanner scanner = new Scanner(System.in);
[...]
String inputStr = scanner.next();
int inputInt = Integer.parseInt(inputStr);
[...]
scanner.close();
```

L'expression `scanner.next()` attend que l'utilisateur saisisse une chaîne de caractère puis tape [entrée], et retourne lac chaîne saisie ; l'expression `Integer.parseInt(...)` permet de convertir, si possible, une chaîne de caractères en l'entier qu'elle représente, par exemple `"123"` (de type `String`) en `123` (de type `int`).

Attention, un *scanner* ouvert sur `System.in` ne peut pas être rouvert une fois fermé. De fait, si l'instance de *scanner* doit être utilisée dans d'autres méthodes que le `main`, il faut conserver l'ouverture et la fermeture seulement une fois, dans le `main`, et passer le *scanner* (ouvert) en argument de ces méthodes, qui n'auront alors qu'à appeler `scanner.next()`.

Exercice 1.7. *Écrire une classe exécutable dont la méthode* `main` *demande le nom des joueurs, les nombres n et k, puis, une fois que ces informations ont été saisies, permet de jouer une partie en demandant aux joueurs de saisir leurs coups au clavier.*

2 Jeu du morpion

Le jeu du morpion est un jeu à deux joueurs, qui se joue sur une grille 3×3, initialement vide. Les joueurs jouent tour à tour. À son tour, un joueur place une marque (qui lui est propre, « X » pour l'un et « O » pour l'autre) dans l'une des cases non encore marquées. Le premier joueur à placer sa marque dans trois cases alignées (en colonne, en ligne ou en diagonale) gagne la partie. Si la grille est pleine alors qu'aucun joueur n'y est parvenu, la partie est déclarée nulle.

Comme pour le jeu de Nim en partie 1, l'objectif de cet exercice est d'écrire une classe, dont chaque instance représentera une partie de jeu de morpion. Une instance sera donc spécifiée par les joueurs impliqués dans la partie, et l'état de l'instance évoluera au fil de la partie. On représentera à nouveau les joueurs par de simples chaînes de caractères correspondant à leur nom (ce qui n'est pas la même chose que leur marque).

Les classes sont à écrire dans un *package* nommé `games.tictactoe`. Un `main` complet de tests ressemblera à :

```
import gamestests.tictactoe.TicTacToeTests;
[...]
boolean ok = true;
TicTacToeTests ticTacToeTester=new TicTacToeTests();
ok = ok && ticTacToeTester.testGetCurrentPlayer();
ok = ok && ticTacToeTester.testExecuteAndIsValid();
ok = ok && ticTacToeTester.testGetWinner();
ok = ok && ticTacToeTester.testIsOver();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

On pourra ajouter la ligne

```
ok = ok && ticTacToeTester.testWins();
```

si l'on définit une méthode `wins` pour l'exercice 2.5.

Pour les exercices qui suivent, bien tester les méthodes au fur et à mesure.

Afin de faciliter la saisie des coups au clavier, on pourra utiliser un affichage tel que :

```
  0 1 2
0 . X O
1 X O X
2 . . .

C'est à Mike de jouer
Votre coup :
- rangée ? 2
- colonne ?
```

Exercice 2.1. *Créer une classe nommée* `TicTacToe`*, dont le constructeur prend deux noms de joueurs (de type* `String`*) représentant le premier et le second joueurs, respectivement ; ce constructeur devra stocker les deux joueurs en attributs de la classe, et initialiser deux autres attributs : un tableau à deux dimensions d'objets de type* `String` *pour stocker les coups joués, et le joueur courant (à qui c'est le tour de jouer).*

Exercice 2.2. *Ajouter à la classe un accesseur* `getCurrentPlayer`*.*

Exercice 2.3. *Ajouter à la classe une méthode* `execute`*, prenant en argument deux entiers (de type* `int`*, représentant une rangée et une colonne) et mettant à jour le plateau de jeu ainsi que le joueur courant ; on supposera que le coup est valide lorsque cette méthode est appelée.*

Exercice 2.4. *Ajouter à la classe une méthode* `isValid`*, prenant en argument deux entiers comme dans l'exercice 2.3, et retournant un booléen indiquant si le coup est valide (*`true`*) ou non (*`false`*) dans la situation courante.*

Exercice 2.5. *Ajouter à la classe une méthode* `getWinner`*, sans argument, retournant soit le joueur ayant gagné dans la situation courante, soit* `null` *(si la partie n'est pas terminée ou est un match nul) ; le type de retour sera donc* `String`*.*
Indication : *Pour réaliser cette méthode, on pourra réaliser préalablement une méthode* `wins`*, prenant en argument un joueur, un numéro de rangée* `row`*, un numéro de colonne* `column`*, une direction* `deltaRow` *(*$+1$*,* 0 *ou* -1*) et une direction* `deltaColumn`*, et retournant un booléen indiquant si l'alignement des cases de coordonnées* $(\texttt{row}, \texttt{column})$*,* $(\texttt{row} + \texttt{deltaRow}, \texttt{column} + \texttt{deltaColumn})$*,* $(\texttt{row} + 2 \times \texttt{deltaRow}, \texttt{column} + 2 \times \texttt{deltaColumn})$ *est entièrement occupé par le joueur donné. Cette méthode pourra ensuite être utilisée pour réaliser la méthode* `getWinner`*, en l'appelant pour chaque rangée, colonne et diagonale de la grille.*

Exercice 2.6. *Ajouter à la classe une méthode* `isOver`*, sans argument, retournant un booléen indiquant si la partie, dans sa situation courante, est terminée ou non.*
Indication : *On pourra utiliser la méthode* `getWinner`*, mais il faudra en plus distinguer, le cas échéant, si* `null` *représente un match nul, donc une partie terminée, ou une partie non terminée.*

Pour terminer, on souhaite réaliser une interface permettant de jouer au morpion dans une console.

Exercice 2.7. *Comme pour le jeu de Nim (exercice 1.7), écrire une classe exécutable permettant d'effectuer une partie de morpion à deux, avec une interface inspirée de l'illustration ci-dessus.*
Indication : *On pourra s'aider d'une méthode* `String situationToString()`*, retournant une représentation du plateau courant (sous la forme d'une chaîne de caractères multi-lignes, en utilisant* `chaine += System.lineSeparator()` *pour ajouter un retour à la ligne).*

3 Jeu du morpion avec indices (exercice optionnel)

Note Pour réaliser cet exercice, il faut nécessairement avoir intégralement réalisé la classe `TicTacToe` demandée en partie 2.

Il s'agit ici de réaliser une version du jeu de morpion qui puisse être utilisée pour l'apprentissage du jeu par des enfants (par exemple). Dans cette version, les règles sont inchangées, mais lorsque l'on affiche la situation de jeu, on indique également les « menaces », c'est-à-dire les cases sur lesquelles l'adversaire pourrait gagner immédiatement si c'était à son tour de jouer. Dans cet énoncé, nous appelons ces menaces des *hints* (indices/suggestions pour le joueur courant).

Les classes demandées doivent être créées dans le *package* `games.tictactoe` de la partie 2.

Exercice 3.1. *Créer une classe nommée* `TicTacToeWithHints`*, héritant de la classe TicTacToe. Munir la classe d'un constructeur prenant les mêmes arguments que celui de la classe* `TicTacToe`*, dans le même ordre.*

Exercice 3.2. *Ajouter à la classe une méthode* `hints`*, sans argument, qui retourne les* hints *pour le joueur courant, sous la forme d'une liste d'entiers représentant des coups (de type* `ArrayList<Integer>`*). Pour la représentation des coups comme des entiers, utiliser la base 3 : entier* $i = 3r + c$ *pour représenter le coup correspondant à la rangée* r *et à la colonne* c*.*
Indication : *Pour implémenter cette méthode, on pourra par exemple considérer chaque case vide, et pour chacune : simuler un coup de l'adversaire sur le tableau représentant le plateau, puis appeler la méthode* `getWinner`*, puis annuler le coup simulé.*

Exercice 3.3. *Redéfinir la méthode utilisée pour l'affichage du morpion (voir indications de l'exercice 2.7) pour qu'elle ajoute à la représentation de la situation la liste des* hints *pour le joueur courant. On pourra par exemple ajouter une ou plusieurs lignes listant ces* hints*, en-dessous de la représentation de la grille, et/ou indiquer un symbole tel que « ! » sur les cases concernées, dans la représentation de la grille.*
Indication : *Si une telle méthode n'a pas été réalisée à l'exercice 2.7, il sera utile (pour l'exercice 3.4) de commencer par l'ajouter à la classe* `TicTacToe`*, puis de la redéfinir effectivement dans la classe* `TicTacToeWithHints`*.*

Exercice 3.4. *Écrire une classe exécutable sur le modèle de celle déjà écrite pour gérer une partie de morpion (exercice 2.7), mais qui commence par demander si l'on veut jouer avec ou sans indices, puis qui fait jouer une partie avec un affichage en conséquence. On s'efforcera de ne pas dupliquer de code, en ajoutant simplement au début de la méthode main des lignes telles que :*

```
if (answer.equals("oui")) {
  game = new TicTacToeWithHints(...);
```

```
} else {
  game = new TicTacToe(...);
}
```

La classe pourra être testée avec le code suivant :

```
import gamestests.tictactoe.TicTacToeWithHintsTests;
[...]
boolean ok = true;
TicTacToeWithHintsTests tester = new TicTacToeWithHintsTests();
ok = ok && tester.testGetCurrentPlayer();
ok = ok && tester.testExecuteAndIsValid();
ok = ok && tester.testWins(); // si wins() est implementee
ok = ok && tester.testGetWinner();
ok = ok && tester.testIsOver();
ok = ok && tester.testHints();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

4 Factorisation des jeux

On constate que les implémentations des jeux de Nim et du morpion, telles que réalisées dans les parties 1 et 2, partagent un certain nombre de caractéristiques, en particulier la gestion des deux joueurs. On va donc chercher à factoriser ce code dans une classe abstraite.

Note importante Pour cette partie, l'idée est de modifier le code déjà implémenté, pour en factoriser des parties. Pour des raisons techniques liées aux tests, simuler cela en

- créant un nouveau *package* nommé `games.factoredgames`,
- y copiant les classes déjà implémentées dans les *packages* `games.nim` et `games.tictactoe`, à l'exception de la ou des classe(s) lançant les tests,
- changeant les déclarations de *package* de ces classes,
- effectuant tout le travail demandé dans cette partie dans ce nouveau *package*.

On aura donc désormais deux versions des jeux, la version non factorisée (dans les *packages* `games.nim` et `games.tictactoe`), et la version factorisée (dans le *package* `games.factoredgames`).

Avant de factoriser du code, on va tout d'abord adopter une représentation commune pour les coups dans les deux jeux, en choisissant le type `int`. Les coups sont déjà représentés ainsi pour le jeu de Nim, il reste donc à adapter le jeu du morpion. Pour cela, on représentera un coup constitué d'une rangée r et d'une colonne c par l'entier $3r + c$ (comme dans l'exercice 3.2).

Exercice 4.1. *Modifier la classe* `TicTacToe` *pour que les méthode* `execute` *et* `isValid` *prennent en argument, non plus deux entiers représentant une colonne et une rangée, mais un unique entier, représentant un coup comme ci-dessus. Si elle est implémentée, modifier de même la classe* `TicTacToeWithHints`*.*

Exercice 4.2. *Écrire une classe abstraite nommée* `AbstractGame`*, munie d'un constructeur prenant deux joueurs (de type* `String`*) en arguments. Par convention, le joueur passé en premier argument sera toujours le premier à jouer.*

Exercice 4.3. *Ajouter à la classe une méthode abstraite* `void doExecute(int)`*, de visibilité* `protected`*, permettant d'exécuter un coup sans changer le joueur courant.*

Exercice 4.4. *Ajouter à la classe une méthode concrète* `void execute(int)`*, permettant d'exécuter un coup et de changer le joueur. La définir en appelant la méthode abstraite* `doExecute`*.*

Cette méthode concrète, définie au niveau de la classe `AbstractGame`, a vocation à remplacer les méthodes `removeMatches` et `execute` des classes `Nim` et `TicTacToe`, respectivement.

Exercice 4.5. *Ajouter à la classe une méthode concrète* `getCurrentPlayer()`*, qui retourne le joueur courant (de type* `String`*).*

On pourra tester cette classe `AbstractGame` avec le code suivant :

```
import gamestests.factoredgames.AbstractGameTests;
[...]
boolean ok = true;
AbstractGameTests abstractGameTester = new AbstractGameTests();
ok = ok && abstractGameTester.testGetCurrentPlayer();
ok = ok && abstractGameTester.testExecute();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

Note pour les curieux Ces tests utilisent une classe, correspondant à jeu purement imaginaire, qui hérite de la classe `AbstractGame` de la façon la plus simpliste possible ; c'est ce qu'on appelle en anglais un objet *mock*, ici il s'agit donc de la classe `MockGame`, dont la méthode `doExecute` ne fait rien.

On va maintenant adapter les autres classes pour qu'elles tirent parti de la factorisation.

Exercice 4.6. *Modifier les classes* `Nim` *et* `TicTacToe` *pour qu'elles héritent de la classe* `AbstractGame`*, en utilisant au maximum la factorisation. Penser à supprimer les attributs et les définitions de méthodes devenus inutiles dans ces classes, et à annoter les redéfinitions par* `@Override`*.*

On pourra tester les classes ainsi modifiées avec le code suivant :

```
import gamestests.factoredgames.NimTests;
import gamestests.factoredgames.TicTacToeTests;
[...]
boolean ok = true;

NimTests nimTester = new NimTests();
ok = ok && nimTester.testExtends();
ok = ok && nimTester.testGetCurrentPlayer();
ok = ok && nimTester.testExecute();
ok = ok && nimTester.testIsValid();
ok = ok && nimTester.testIsOver();
ok = ok && nimTester.testGetWinner();

TicTacToeTests ticTacToeTester = new TicTacToeTests();
ok = ok && ticTacToeTester.testExtends();
ok = ok && ticTacToeTester.testGetCurrentPlayer();
ok = ok && ticTacToeTester.testExecuteAndIsValid();
ok = ok && ticTacToeTester.testWins(); // si wins() est implementee
ok = ok && ticTacToeTester.testGetWinner();
ok = ok && ticTacToeTester.testIsOver();

System.out.println(ok ? "All tests OK" : "At least one test KO");
```

Exercice 4.7. *Adapter les classes permettant de lancer les deux jeux (classes réalisées dans les exercices 1.7 et 2.7).*

Si la partie 3 a été traitée, les tests peuvent être réalisés avec le code suivant :

```
import gamestests.factoredgames.TicTacToeWithHintsTests;
[...]
boolean ok = true;
TicTacToeWithHintsTests tester = new TicTacToeWithHintsTests();
ok = ok && tester.testGetCurrentPlayer();
ok = ok && tester.testExecuteAndIsValid();
ok = ok && tester.testWins(); // si wins() est implementee
ok = ok && tester.testGetWinner();
ok = ok && tester.testIsOver();
ok = ok && tester.testHints();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

5 Abstraction des jeux et des joueurs

On souhaite désormais généraliser les jeux, de façon que l'on puisse écrire un seul « orchestrateur » pour gérer la boucle de jeu d'une partie de jeu de Nim comme d'une partie de jeu du morpion. On va également en profiter pour abstraire les joueurs (pour l'instant représentés par des chaînes de caractères), de façon à en avoir plusieurs types à disposition : « humains » (saisissant leurs coups au clavier), « aléatoires » (automatiques, choisissant leur coup au hasard, (pseudo-)uniformément), etc.

Note importante Comme pour la partie 4, on va modifier une dernière fois le code déjà implémenté. Simuler cela en

— créant un nouveau *package* nommé `games.genericgames`,
— y copiant les classes déjà implémentées dans le *package* `games.factoredgames`, à l'exception de la ou des classe(s) lançant les tests,
— changeant les déclarations de *package* de ces classes.

On aura donc désormais trois versions des jeux, la version non factorisée (dans les *packages* `games.nim` et `games.tictactoe`), la version factorisée (dans le *package* `games.factoredgames`), et la version générique (dans le *package* `games.genericgames`).

Tests Les classes et interfaces de cette partie seront à créer dans trois *packages* : `games.genericgames`, `games.players` et `games.plays`. Elles pourront être testées avec le code suivant :

— pour le *package* `games.genericgames` :

```
import gamestests.genericgames.AbstractGameTests;
import gamestests.genericgames.NimTests;
import gamestests.genericgames.TicTacToeTests;
import gamestests.genericgames.TicTacToeWithHintsTests;
[...]
boolean ok = true;

AbstractGameTests abstractGameTester = new AbstractGameTests();
ok = ok && abstractGameTester.testGetCurrentPlayer();
ok = ok && abstractGameTester.testExecute();

NimTests nimTester = new NimTests();
ok = ok && nimTester.testGetInitialNbMatches();
ok = ok && nimTester.testGetCurrentNbMatches();
ok = ok && nimTester.testGetCurrentPlayer();
ok = ok && nimTester.testExecute();
ok = ok && nimTester.testIsValid();
ok = ok && nimTester.testValidMoves();
ok = ok && nimTester.testIsOver();
ok = ok && nimTester.testGetWinner();
ok = ok && nimTester.testCopy();

TicTacToeTests ticTacToeTester = new TicTacToeTests();
ok = ok && ticTacToeTester.testGetCurrentPlayer();
ok = ok && ticTacToeTester.testExecuteAndIsValid();
ok = ok && ticTacToeTester.testValidMoves();
ok = ok && ticTacToeTester.testWins(); // si wins() est implementee
ok = ok && ticTacToeTester.testGetWinner();
ok = ok && ticTacToeTester.testIsOver();
ok = ok && ticTacToeTester.testCopy();

// Si la classe TicTacToeWithHints existe (exercice optionnel)
TicTacToeWithHintsTests ticTacToeWithHintsTester
  = new TicTacToeWithHintsTests();
ok = ok && ticTacToeWithHintsTester.testGetCurrentPlayer();
ok = ok && ticTacToeWithHintsTester.testExecuteAndIsValid();
ok = ok && ticTacToeWithHintsTester.testValidMoves();
ok = ok && ticTacToeWithHintsTester.testWins(); // si wins() est implementee
```

```
ok = ok && ticTacToeWithHintsTester.testGetWinner();
ok = ok && ticTacToeWithHintsTester.testIsOver();
ok = ok && ticTacToeWithHintsTester.testHints();
ok = ok && ticTacToeWithHintsTester.testCopy();

System.out.println(ok ? "All tests OK" : "At least one test KO");
```

— pour le *package* `games.players` :

```
import gamestests.players.HumanTests;
import gamestests.players.RandomPlayerTests;
[...]
boolean ok = true;

HumanTests humanTester = new HumanTests();
// Change argument to true in next call to reactivate printing
ok = ok && humanTester.testChooseMove(false);

RandomPlayerTests randomTester = new RandomPlayerTests();
ok = ok && randomTester.testChooseMove();

System.out.println(ok ? "All tests OK" : "At least one test KO");
```

— pour le *package* `games.plays` :

```
import gamestests.plays.OrchestratorTests;
[...]
boolean ok = true;

OrchestratorTests tester = new OrchestratorTests();
// Change argument to true in next call to reactivate printing
ok = ok && tester.testPlay(false);

System.out.println(ok ? "All tests OK" : "At least one test OK");
```

À noter, les méthodes `testChooseMove` et `testPlay` désactivent temporairement l'affichage sur la sortie standard, ce qui évite que les affichages effectués par les méthodes testées ne « polluent » les tests. Toutefois, pour débogguer, il peut être nécessaire d'afficher les valeurs de certaines variables (par exemple) à l'exécution ; il faudra alors (temporairement) passer `true` au lieu de `false` en argument des appels aux méthodes de test.

Exercice 5.1. *Créer une interface vide nommée* `Game` *dans le package* `games.genericgames`*, et une interface vide nommée* `Player` *dans le package* `games.players`*.*

L'idée est que l'interface `Game` déclare toutes les méthodes utiles pour « arbitrer » une partie, et que l'interface `Player` déclare une unique méthode, permettant à l'« arbitre » de demander au joueur quel coup il veut jouer.

Exercice 5.2. *Déclarer dans l'interface* `Game` *les méthodes* `getCurrentPlayer`*,* `isValid`*,* `execute`*,* `isOver` *et* `getWinner`*, avec la même signature que dans les classes* `AbstractGame` *et* `TicTacToe`*, à l'exception des méthodes* `getCurrentPlayer` *et* `getWinner` *qui devront désormais retourner un objet de type* `Player`*. Déclarer également une méthode* `validMoves`*, ne prenant aucun argument et retournant un objet de type* `List<Integer>` *(du* package `java.util`*) contenant tous les coups valides dans la situation courante (pour le joueur courant).*

Exercice 5.3. *Ajouter à l'interface* `Game` *la déclaration d'une méthode* `String moveToString`*, prenant un coup (de type* `int`*) en argument, et en retournant une représentation naturelle, et d'une méthode* `String situationToString` *permettant d'obtenir une représentation de la situation courante.*

Exercice 5.4. *Ajouter à l'interface* `Game` *une méthode* `Game copy()`*. Cette méthode sera utilisée par le joueur automatique (voir partie 6) : elle devra retourner une nouvelle instance du jeu sur lequel elle est appelée, représentant exactement la même situation.*

Exercice 5.5. *Déclarer dans l'interface* `Player` *une méthode* `int chooseMove(Game)`*.*

On va désormais adapter les classes du *package* `games.genericgames` représentant les jeux pour qu'elles implémentent l'interface `Game`. À noter, les classes exécutables des précédentes versions ne pourront plus être adaptées à ces changements tant que l'on n'aura pas écrit au moins une classe implémentant l'interface `Player` (exercice 5.9).

Exercice 5.6. *Déclarer dans la classe* `AbstractGame` *qu'elle implémente l'interface* `Game`*, en l'adaptant pour que les joueurs soient désormais représentés par des instances de* `Player`*.*

Exercice 5.7. *Adapter la classe* `Nim` *pour qu'elle implémente désormais correctement l'interface* `Game`*, en changeant le type des joueurs (de* `String` *à* `Player`*) et en définissant les méthodes manquantes.*
Indication : *Pour la méthode* `copy`*, veiller à copier non seulement les arguments passés au constructeur, mais aussi les autres attributs de la classe et de la classe mère* `AbstractGame`*, en s'inspirant par exemple du code suivant :*

```
res = new Nim(this.initialNbMatches, this.maxNbMatches, this.player1...);
res.currentNbMatches = this.currentNbMatches;
res.currentPlayer = super.currentPlayer;
return res
```

Exercice 5.8. *Adapter de même la classe* `TicTacToe`*.*
Indication : *Si un tableau est utilisé pour stocker la situation courante, veiller à en effectuer une copie « profonde », c'est-à-dire à le copier case par case (et non en utilisant seulement* `res.tableau = this.tableau`*, qui ne copierait que l'adresse du tableau) ; pour cela, créer un nouveau tableau et y copier chaque case de* `this.tableau`*.*

On va désormais définir deux types de joueurs (dans le *package* `games.players`).

Exercice 5.9. *Écrire une classe nommée* `Human` *et implémentant l'interface* `Player`*. La munir d'un constructeur prenant en arguments, dans cet ordre, un nom de type* `String` *et une instance de la classe* `java.util.Scanner`*. Pour la définition de la méthode* `chooseMove`*, afficher la liste des coups valides en utilisant les méthodes de la classe* `Game`*, et demander (via l'instance de* `Scanner`*) le coup choisi par le joueur ; redemander un coup tant que celui saisi n'est pas valide.*

Exercice 5.10. *Redéfinir la méthode* `toString` *dans la classe* `Human`*, de sorte qu'elle retourne le nom du joueur.*

Pour la définition des joueurs « aléatoires », on pourra utiliser la classe `java.util.Random` ; une instance `rand` de cette classe représente un générateur pseudo-aléatoire, et l'appel `rand.nextInt(n)` retourne un entier tiré (pseudo-)uniformément dans l'intervalle $[0, n[$.

Exercice 5.11. *Écrire une classe nommée* `RandomPlayer`*, avec un constructeur prenant en argument une instance de* `java.util.Random`*.*

Exercice 5.12. *Faire implémenter l'interface* `Player` *à la classe* `RandomPlayer`*, en faisant en sorte que la méthode* `chooseMove` *retourne un coup tiré (pseudo-)uniformément parmi l'ensemble des coups valides dans la situation donnée ; pour cela, on pourra tirer un indice aléatoire, et retourner le coup situé à cet indice dans la liste retournée par* `validMoves`*.*

Exercice 5.13. *Redéfinir la méthode* `toString` *dans la classe* `RandomPlayer`*, pour qu'elle retourne la chaîne* `"Joueur aléatoire n° " + this.hashCode()` *(ceci donnera un affichage du type « Joueur aléatoire n°320635 », le* hash code *permettant en pratique de distinguer plusieurs éventuels joueurs aléatoires.*

Enfin, on va écrire une classe permettant d'« arbitrer » une partie. L'un des objectifs est que cette classe n'utilise que les interfaces `Game` et `Player`, et ne dépende aucunement d'un type de jeu ou de joueur particulier.

Exercice 5.14. *Dans le* package `games.plays`*, écrire une classe nommée* `Orchestrator`*. La munir d'un constructeur prenant en argument une instance de* `Game`*.*

À noter, une instance de la classe `Orchestrator` ne pourra gérer qu'une partie du jeu en question ; une fois celle-ci jouée, il ne servira à rien de réutiliser cette instance (impossible de « rembobiner » la partie).

Exercice 5.15. *Ajouter à la classe* `Orchestrator` *une méthode* `play`*, sans argument et ne retournant rien, qui gère la partie : boucle de jeu en affichant la situation, en récupérant le coup retourné par la méthode* `chooseMove` *du joueur courant, et en exécutant ce coup puis, une fois la partie terminée, affichage du résultat.*

Exercice 5.16. *Écrire une classe exécutable permettant de jouer une partie en utilisant toutes les classes et interfaces écrites, en s'inspirant par exemple du code suivant :*

```
Random rand = new Random(123);
Scanner scanner = new Scanner(System.in);
Player player1 = new Human("Moi", scanner);
Player player2 = new RandomPlayer(rand);
TicTacToe game = new TicTacToe(player1, player2);
Orchestrator orchestrator = new Orchestrator(game);
orchestrator.play();
scanner.close();
```

La classe devra permettre de choisir le jeu et le type de chaque joueur, ainsi que le nom pour des joueurs humains ; le choix pourra se faire au clavier ou via *des arguments de la classe exécutable.*

Exercice 5.17. *Tester avec différentes configurations (joueur aléatoire contre joueur aléatoire, jeu de Nim, etc.).*

6 Algorithme négamax

L'objectif de cette partie est d'implémenter un joueur utilisant un algorithme « négamax » pour choisir un coup dans une situation de jeu donnée, et ceci de façon générique, c'est-à-dire un joueur capable de jouer à tout jeu implémentant l'interface `Game`.

On pourra implémenter l'algorithme en s'inspirant du pseudo-code donné dans les notes de cours. Un point important, du fait que l'on simule plusieurs coups dans une même situation, est de copier la situation courante autant que nécessaire, précisément, avant chaque appel à la méthode `execute` : créer une copie de la situation (de type `Game`) et appeler `execute` sur cette copie. C'est alors la copie qui sera donnée en argument à l'appel récursif (où elle sera à nouveau copiée autant que nécessaire).

La classe pourra être testée avec le code suivant :

```
import gamestests.players.NegamaxPlayerTests;
import gamestests.players.NegamaxPlayerWithCacheTests;
[...]
boolean ok = true;
NegamaxPlayerTests negamaxTester = new NegamaxPlayerTests();
ok = ok && negamaxTester.testEvaluate();
ok = ok && negamaxTester.testChooseMove();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

Exercice 6.1. *Ajouter au* package `games.players` *une classe nommée* `NegamaxPlayer` *implémentant l'interface* `Player`*, avec un constructeur sans argument et ne faisant rien.*

Exercice 6.2. *Ajouter à la classe* `NegamaxPlayer` *une méthode* `evaluate`*, prenant en argument la situation courante (de type* `Game`*) et retournant un entier (*`int`*) représentant la valeur de cette situation. Note : il est inutile de passer le joueur en argument, car il peut être récupéré en appelant* `getCurrentPlayer` *sur la situation.*

Exercice 6.3. *Implémenter la méthode* `chooseMove` *de l'interface* `Player` *dans la classe* `NegamaxPlayer`*.*

Exercice 6.4. *Adapter le programme principal de l'exercice 5.16 pour qu'il permette d'utiliser ce nouveau type de joueur.*

Exercice 6.5. *Tester sur le jeu de Nim avec 13 allumettes (et entre 1 et 3 à retirer à chaque coup) ; l'algorithme doit toujours gagner s'il est le second joueur. Tester sur le même jeu mais avec 14 allumettes (l'algorithme doit toujours gagner s'il est le premier joueur). Tester également sur le jeu du morpion (il doit toujours obtenir au moins une partie nulle, qu'il soit premier ou second joueur).*

Exercice 6.6. *Tester sur le jeu de Nim an augmentant le nombre initial d'allumettes au fur et à mesure ; que constate-t-on ?*

7 Algorithme négamax avec cache (exercice optionnel)

L'objectif de ce TP est de réaliser un joueur utilisant un algorithme « négamax », mais plus efficace que la version de base en termes de temps de calcul. Pour cela, l'idée est de mettre en cache les situations déjà évaluées, c'est-à-dire de les stocker en mémoire avec la valeur calculée afin que, si on tombe à nouveau sur ces situations lors du calcul, on n'ait pas besoin de redévelopper tout l'arbre de jeu pour les réévaluer.

On pourra tester les méthodes développées dans cette partie avec le code suivant :

```
import gamestests.players.NegamaxPlayerTests;
import gamestests.players.NegamaxPlayerWithCacheTests;
[...]
boolean ok = true;
NegamaxPlayerWithCacheTests negamaxWithCacheTester
  = new NegamaxPlayerWithCacheTests();
ok = ok && negamaxWithCacheTester.testNimEquals();
ok = ok && negamaxWithCacheTester.testNimHashCode();
ok = ok && negamaxWithCacheTester.testTicTacToeEquals();
ok = ok && negamaxWithCacheTester.testTicTacToeHashCode();
ok = ok && negamaxWithCacheTester.testEvaluate();
ok = ok && negamaxWithCacheTester.testChooseMove();
System.out.println(ok ? "All tests OK" : "At least one test KO");
```

Un préliminaire nécessaire à la réalisation du joueur consiste à se donner les outils pour reconnaître quand deux situations sont en réalité les mêmes.

Exercice 7.1. *Redéfinir les méthodes* `boolean equals(Object o)` *et* `int hashCode()` *dans les classes* `Nim` *et* `TicTacToe`*, de sorte qu'elles considèrent comme égales les instances qui représentent la même situation de jeu (même état du jeu, même joueur courant).*
Indication : *Pour le jeu du morpion, veiller à comparer les plateaux case par case (et non avec, par exemple,* `this.board == otherAsTicToe.board`*, qui ne reconnaîtrait que le cas où les deux tableaux sont au même emplacement mémoire). On pourra utiliser* `Arrays.deepEquals(tableau, autreTableau)` *et, pour le hachage,* `Objects.hash(..., Arrays.deepHashCode(tableau))`*.*

On peut désormais écrire le joueur.

Exercice 7.2. *Dans le* package `games.players`*, créer une classe nommée* `NegamaxPlayerWithCache`*, héritant de la classe* `NegamaxPlayer`*. Munir la classe d'un constructeur sans argument.*

Exercice 7.3. *Ajouter à la classe un attribut de type* `Map<..., ...>`*, destiné à stocker les situations déjà évaluées (comme clefs), avec les valeurs trouvées. Redéfinir la méthode* `evaluate` *pour qu'elle utilise ce cache. Pour cela, avant d'évaluer une situation, elle vérifiera tout d'abord si cette situation est déjà dans le cache, et si elle l'y trouve, elle renverra simplement la valeur associée. Par ailleurs, après avoir calculé une valeur pour une situation qui n'était pas en cache, elle ajoutera cette valeur dans le cache.*

Exercice 7.4. *Adapter le programme principal de l'exercice 6.4 pour qu'il permette d'utiliser ce nouveau type de joueur.*

Exercice 7.5. *Tester avec le jeu de Nim en augmentant progressivement le nombre initial d'allumettes.*