Faculté des Sciences et des Techniques Nantes
X3I0020 : Programmation objet
Groupe 302C
Irena Rusu

Projet de programmation objet n°1

Tetris

Rapport

fait par

Silvan Habenicht

1 novembre 2016

Table des matières

1 Introduction

Dans le cadre du module Programmation objet il nous a été demandé de réaliser la structure des classes pour un jeu de Tetris. Cela inclut classes représentantes pour le plateau, les pièces et des méthodes pour exécuter le jeu et l'influencer, comme par exemple par le mouvement des pièces ou la génération des mêmes.

Après réfléchir sur la langage pour mon rapport, j'ai finalement décidé de l'écrire en français (pas en anglais), alors que je ne suis pas encore très fort avec ça. C'est pourquoi je vous prie, en question de l'orthographie, de considérer ma situation comme étudiant étranger dans l'évaluation de ce rapport.

2 Description synthétique des classes

2.1 Classes imposées

2.1.1 Piece

Objectif

La classe `Piece` sert à la gestion d'un ensemble de quatre cellules qui représentent une de sept formes différentes sur le plateau. Elle permet la réalisation des mouvements comme spécifié dans l'API.

Variables d'instance

```
Cellule[]   cellules
Plateau     plateau
PieceType   pieceType
int         centre
```

Constructeur

```
Piece(Plateau plateau, PieceType pieceType)
```

API

`Piece versLeBas()`	Si insérable dans `plateau`, création d'une nouvelle instance `Piece` avec déplacement vers le bas. Sinon `this` est rendue sans modification.
`Piece versLaGauche()`	Si insérable dans `plateau`, création d'une nouvelle instance `Piece` avec déplacement vers la gauche. Sinon `this` est rendue sans modification.
`Piece versLaDroite()`	Si insérable dans `plateau`, création d'une nouvelle instance `Piece` avec déplacement vers la droite. Sinon `this` est rendue sans modification.
`Piece tourner()`	Si insérable dans `plateau`, création d'une nouvelle instance `Piece` avec 90° rotation à droite. Sinon `this` est rendue sans modification.

Description

`Piece` est une implémentation de `Forme` avec les fonctions ci-dessus. Chaque sous-classe a un constructeur individuel, comme détaillé dans 2.2.2. Le constructeur général crée un tableau de quatre cellules qui sont mémorisées dans la variable d'instance `cellules`. Dépendant du type de `Piece`, ces cellules sont arrangées d'une façon individuelle.

La variable `centre` contient la demie nombre de colonnes dans `plateau` pour le rajout d'une `Piece` dans le centre haut.

La variable `plateau` est nécessaire pour tester la permission d'un mouvement et pour pouvoir ajouter une `Piece` dans les fonctions de classe.

Le `PieceType` `pieceType` est nécessaire pour la fonction `tourner()`, car cette fonction n'est pas identique pour tous pièces. Par exemple le bâton (`PieceI`) n'a pas un pivot comme tous les autres types. La fonction `tourner()` ne pouvait pas être implémentée isolément dans chaque sous-classe parce que ça ne permet pas l'utilisation avec une `Piece` aléatoire (qui n'est pas d'une sous-classe spécifique).

Tous les méthodes dans `Piece` rendent une copie de `this`, qui est déplacée dans `plateau`. Pour le déplacement dans `tourner()`, les méthodes privées `tournerI()` ou `tourne(Cellule cellule, Cellule pivot)` – pour le déplacement d'une `Cellule` autour le pivot du pièce – sont utilisées. Avant ajouter la nouvelle `Piece` dans `plateau`, chaque méthode teste si c'est possible avec l'aide de la fonction `Plateau.accepter()`.

2.1.2 Cellule

Objectif

Une `Cellule` est la composante la plus élémentaire du programme et de la classe `Plateau`. Elle peut être libre ou, si part d'une `Piece`, occupé. Dépendant de son état, l'apparence d'une `Cellule` sur l'écran (comme élément du plateau) diffère.

Variables d'instance

```
int         x
int         y
boolean     estOccupe
ImageView   image
Image       gris
Image       vide
```

Constructeur

```
Cellule(int y, int x, boolean estOccupe)
```

API

`void retirer()`	Mettre `estOccupe false` et `vide` l'objet d'`image`.
`void ajouter()`	Mettre `estOccupe true` et `gris` l'objet d'`image`.

Description

Dans une `Cellule` on trouve ses coordonnées (dans un plateau), et l'information si elle est d'une `Piece` (alors occupée). Les coordonnées sont mémorisées dans les deux variables `x` et `y`. Le valeur `estOccupe` est `true` si la `Cellule` est d'une `Piece` et `false` sinon. Pour la représentation sur l'écran j'ai déclaré `Cellule` comme une sous-classe de `javafx.scene.layout.Pane` qui contient un `ImageView` avec deux images possibles. Ils sont insérées dépendant de l'état de `estOccupe`.

Les fonctions `retirer()` et `ajouter()` exécutent le changement de `estOccupe` et de `image` corrélativement. Ils sont utilisées par exemple dans `retirer()`, `ajouter()` et `testeEnlevement()` de la classe `Plateau`.

2.1.3 Plateau

Objectif

Le `Plateau` est la collection de tous les cellules disponibles, gardées dans un tableau bidimensionnel. Aucun mouvement d'une `Piece` peut être fait sans avoir confirmé, que la place dans le `Plateau` à la direction désirée est libre. La classe permet de retirer et ajouter des pièces ou, pour supprimer une ligne entière, des cellules séparées.

Variables d'instance

```
Cellule[][]  grille
int          xTaille
int          xTaille
```

Constructeur

```
Plateau(int xTaille, int yTaille)
```

API

`int centreX()`	Rend (`xTaille`/2).
`boolean accepter(Piece p)`	Rend `true` ssi `p` peut être ajoutée dans `this`.
`boolean accepter(Piece p, Direction direction)`	Rend `true` ssi `p` peut être déplacée dans `this` vers la direction dictée par `direction`.
`void retirer(Piece p)`	Supprime `p` dans `this`.
`void ajouter(Piece p)`	Ajoute `p` dans `this`.
`void testeEnlevement()`	Supprime une ligne complète dans `this`, si existant. Puis déplacement vers le bas des pièces au-dessus.

Description

Dans chaque jeu de Tetris il existe un seul `Plateau` qui est responsable pour la gestion des `Pieces`. Le `Plateau` d'habitude consiste de 200 cellules – 10 cellules dans 20 lignes. Avec `Cellule` comme sous-classe de `Pane` il était possible de déclarer `Plateau` comme `GridPane` de JavaFX. Au début, chaque `Cellule` dans `grille` n'est pas occupée et alors représentée par une image noire. Si une `Piece` est ajoutée sur le `Plateau`, les cellules correspondantes sont représentées par l'image d'un carré gris.

La fonction `centreX()` est utilisé dans le constructeur d'une `Piece` pour l'insérer dans le centre du `Plateau`.

Il y a deux méthodes `accepter()` avec un ou deux paramètres. La fonction avec un seul paramètre rend `false` si et seulement si une `Cellule` dans le paramètre est déjà occupée dans la `Cellule` correspondante du plateau. La fonction avec deux paramètres teste la même avec un déport vers la direction donnée dans le deuxième paramètre.

`retirer(Piece p)` et `ajouter(Piece p)` sont utilisées pour chaque déplacement d'une `Piece`. Une `Piece` est d'abord retirée du plateau et après une nouvelle `Piece` est ajoutée sur la position demandée.

La fonction `testeEnlevement()` n'a pas été demandée dans le cadre du projet. Elle permet de jouer/tester l'implémentation de Tetris complètement et est utilisée dans la fonction principale.

2.1.4 Fabrique

Objectif

Le but de la classe `Fabrique` est seulement de générer nouvelles `Piece`s sur le **`plateau`**. La méthode `creerAleatoire()` par exemple rend un des sept types de `Piece` par hasard.

Variables d'instance

```
Plateau  plateau
```

Constructeur

```
Fabrique(Plateau plateau)
```

API

`Piece` `creerAleatoire()`	Rend une nouvelle `Piece` de forme aléatoire.
`Piece` `creerO()`	Rend une nouvelle `Piece` de forme **`O`** (carré).
`Piece` `creerL()`	Rend une nouvelle `Piece` de forme **`L`**.
`Piece` `creerJ()`	Rend une nouvelle `Piece` de forme **`J`**.
`Piece` `creerI()`	Rend une nouvelle `Piece` de forme **`I`** (baton).
`Piece` `creerT()`	Rend une nouvelle `Piece` de forme **`T`**.
`Piece` `creerS()`	Rend une nouvelle `Piece` de forme **`S`**.
`Piece` `creerZ()`	Rend une nouvelle `Piece` de forme **`Z`**.

Description

Chaque méthode dans `Fabrique` rend une nouvelle `Piece` en appelant un constructeur avec **`plateau`** comme paramètre. Dans `creerAleatoire()`, avec l'aide de `Math.random()`, une des sept fonctions est appelée aléatoirement.

2.2 Classes ajoutées

2.2.1 Main

Objectif

La fonction `Main` sert à la visualisation et la contrôle du résultat de mon travail. En plus elle permet effectivement de jouer Tetris. Elle est en charge de lancer l'application, traiter l'actionnement des flèches, passer la musique et continuellement déplacer des `Pieces` vers le bas.

Variables d'instance

```
static final int  taille       //pixels pour chaque Cellule
final int         largeur      //nombre des colonnes
final int         hauteur      //nombre des lignes
Plateau           plateau
Fabrique          fabrique
Piece             actuel
boolean           run          //condition pour thread
int               attendre     //vitesse du jeu au début (ms)
MediaPlayer       mediaPlayer  //charger source de la musique
```

Description

La fonction principale permet de tester l'implémentation par une interface graphique réalisée avec les outils de JavaFX 8. Elle n'est pas élément des demandes dans ce projet mais elle démontre la fiabilité des classes implémentées. Dans la classe `Main` il est possible de changer la taille du plateau avec les variables **`largeur`** et **`hauteur`**. La variable taille change le nombre des pixels occupés dans chaque `Cellule` (longueur du côté).

Le déplacement et la génération automatique des `Pieces` sont exécutés dans un `Thread` qui contient une boucle retardée par le valeur de **`attendre`** millisecondes. Pendant le jeu, ce valeur est couramment diminué jusqu'à 200ms.

Pour finir j'ai ajouté deux pièces de musique de fond avec l'aide du `MediaPlayer`.

2.2.2 Les Pieces I, J, L, O, S, T et Z

Objectif

Les classes `PieceI, PieceJ, PieceL, PieceO, PieceS, PieceT` et `PieceZ` sont des représentations diverses de leur classe supérieure `Piece`. En raison de l'apparence et leur qualités différentes, et afin d'une gestion plus confortable, j'ai décidé de construire sept classes séparées.

Variables d'instance

Cf. classe supérieure **`Piece`**.

Constructeur

```
PieceT(Plateau plateau)
```

L'extrait suivant démontre le constructeur d'une sous-classe de `Piece`. Il enrichit le constructeur de `Piece` par l'initialisation de chaque `Cellule` dans **`cellules`** avec certains valeurs **`x`** et **`y`**. Après, la `Piece` est automatiquement ajoutée sur **`plateau`**.

```
class PieceT extends Piece {

   PieceT(Plateau plateau) {

      super(plateau, PieceType.T);
      cellules[0] = new Cellule(1, centre, true);
      cellules[1] = new Cellule(0, centre, true);
      cellules[2] = new Cellule(1, centre - 1, true);
      cellules[3] = new Cellule(1, centre + 1, true);
      if(plateau.accepter(this))
         plateau.ajouter(this));
   }
}
```

API

Cf. classe supérieure **`Piece`**.

Description

Les classes `PieceI, PieceJ, PieceL, PieceO, PieceS, PieceT` et `PieceZ` sont sous-classes de `Piece`. Chacune a un constructeur individuel ; cependant elles partagent tous les méthodes de leur classe supérieure.

2.3 Interfaces

2.3.1 Forme

Objectif

L'interface `Forme` contient tous les méthodes essentielles pour l'implémentation de `Piece` d'après les demandes. Il n'est pas nécessaire, mais judicieux de l'avoir.

Implémentation

```
interface Forme{

   Forme versLeBas();
   Forme versLaGauche();
   Forme versLaDroite();
   Forme tourner();

}
```

2.4 Enums

2.4.1 PieceType

Implémentation

```
   enum PieceType {I,J,L,O,S,T,Z}
```

Description

Dans la méthode `tourner()` de `Piece`, il est nécessaire d'identifier la sous-classe avec laquelle le `Piece` a été initialisée. Cette information est perdue après la conversion en le `Type Piece`. Avec une `PieceType` comme variable d'instance il est possible de la gérer.

2.4.2 Direction

Implémentation

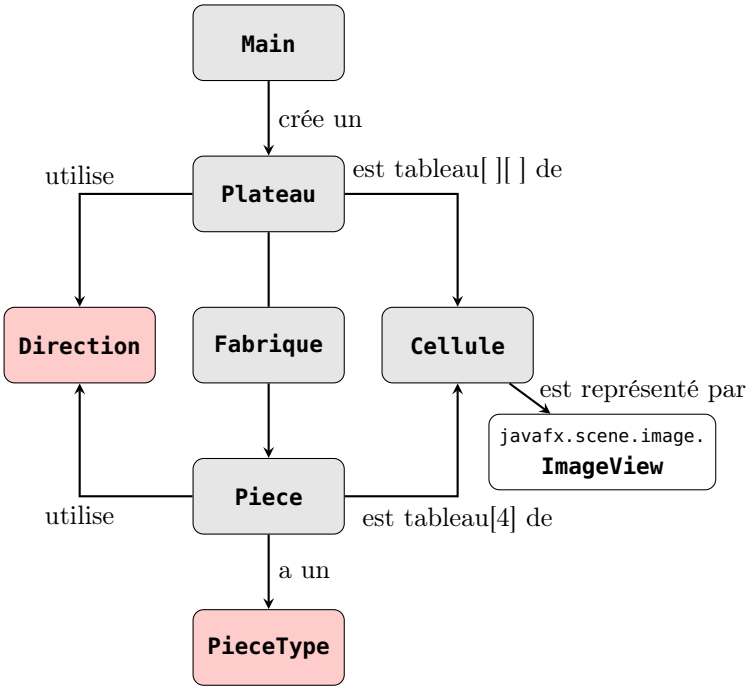
```
   enum Direction {BAS, GAUCHE, DROITE}
```

Description

La Direction `Direction` **`BAS`**, **`GAUCHE`** ou **`DROITE`** peut être un paramètre dans la méthode `accepter()` dans `Plateau`. Comme ça, la méthode teste si une `Piece` peut être bougée vers la direction donnée sans devoir le faire à l'avance.

3 Relations entre les classes

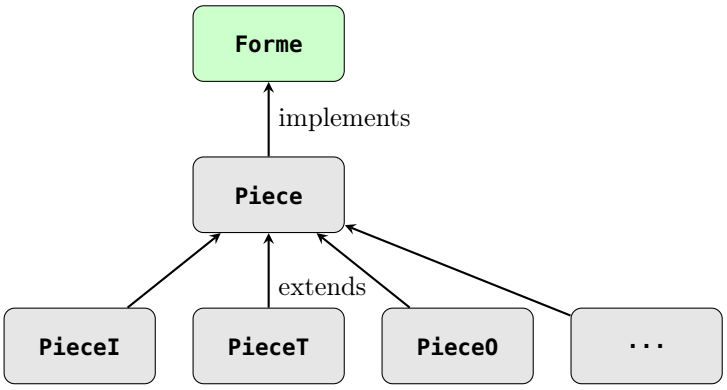
Les relations entre les classes peuvent être suivies dans le diagramme suivant :



Au début de la fonction principale un plateau est crée. Un `Plateau` consiste des `Cellules` vides et des `Pieces`, qui sont générées par la `Fabrique`. Chaque `Piece` consiste de quatre `Cellules` qui sont caractérisées comme élément d'une `Piece` par son image dans `ImageView`. On a besoin d'un `PieceType` comme identifiant pour la méthode `tourner()` dans `Piece`, et d'une `Direction` pour des méthodes entre `Piece` et `Plateau`.

Relation d'héritage

Avec sept types différents, la classe `Piece` est bien qualifiée pour avoir plusieurs sous-classes qui partagent les mêmes méthodes spécifiées dans une interface `Forme`. Voyez les sections correspondantes dans section 2 pour en savoir plus.



4 Jeux de tests

Réalisé par une interface graphique, cf. `Main` 2.2.1.

5 Conclusion

Mon implémentation du projet est finalement arrivée à un jeu complet de Tetris. Je sais que ce n'est pas demandé, mais j'ai profité des vacances pour perfectionner mon programme et c'est le résultat. Dans le fichier chargé sur Madoc, vous trouvez le code du programme et un fichier `Tetris.jar` pour le lancer immédiatement. Le jeu est contrôlé par les flèches avec le flèche supérieur pour tourner une pièce.

Rétrospectivement je ne suis pas sur, si la partition des `Piece` avec plusieurs sous classes et une interface est vraiment une solution raisonnable. Néanmoins je l'ai laissé comme ça. Peut-être ce n'est pas la façon la plus simple, mais elle permet de faire des changements sans beaucoup d'effort – par exemple si on veut ajouter des couleurs individuelles pour chaque type de `Pièce`.