TP Architecture Logicielle / INF4043

Par Alexandra RAMRAMI & Nirosan NARATNAM

Tetris Multijoueur Java

1. Description

En 1985 Alexey Pazhitnov, s'inspira du casse-tête appelé pentomino pour inventer le célèbre Tetris. Notre projet reprend les fondamentaux de ce jeu, tout en ajoutant quelques features modernes. Le jeu consiste à diriger une pièce (aussi appelée tetromino) en mouvement constant vers le bas dans une grille de jeu.

Le tetromino peut être tourné (à 90°) ou déplacé (à gauche ou à droite) par le joueur. Le but du jeu étant de remplir des lignes de la grille en posant les tetrominos, afin d'augmenter son score. Le jeu s'arrête si on ne peut plus insérer de pièce en haut de la grille.

2. Les tâches demandées

Le but de ce tp est de coder un jeu créé en juin 1984 par Alekseï Pajitnov : _TETRIS_

- ✓ Notre tetris pourra être joué seul mais aussi en mode multijoueur (2 personnes sont suffisantes mais vous pouvez faire plus).
- ✓ Le mode multijoueur devra être effectué à travers le réseau (une instance de l'application démarré par joueur).
- ✓ En mode multijoueur, chaque fois qu'un joueur casse 10 lignes, il envoie un malus à son adversaire. Un minimum de deux types de malus différents est requis.
- ✓ Un minimum de trois types de piéces différentes est requis. Le joueur doit au minimum pouvoir tourner la pièce et la déplacer de droite à gauche.
- ✓ Les 5 meilleurs scores (nombre de lignes détruites) doivent être enregistrés dans un fichier pour pouvoir être persistés et ainsi pouvoir être affichés a posteriori.
- ✓ Une simple interface en ligne de commande est acceptée et suffisante. Nous pouvons aussi utiliser des librairies graphiques si vous le souhaitez.
- ✓ Une grande importance sera attachée à la qualité du code, à la conception objet et au découpage par fonctionnalités avec des contrats clairs.
- ✓ Nous allons utiliser des analyseurs de code statiques (npm, findbugs, ...). Nous les utiliserons pour corriger.
- ✓ Nous utiliserons aussi une approche TDD sur le projet. L'utilisation des librairies mockito (Tasty mocking framework for unit tests in Java) et AssertJ est fortement conseillée.

Un ou plusieurs paragraphes nous sont demandés pour présenter et justifier notre architecture.

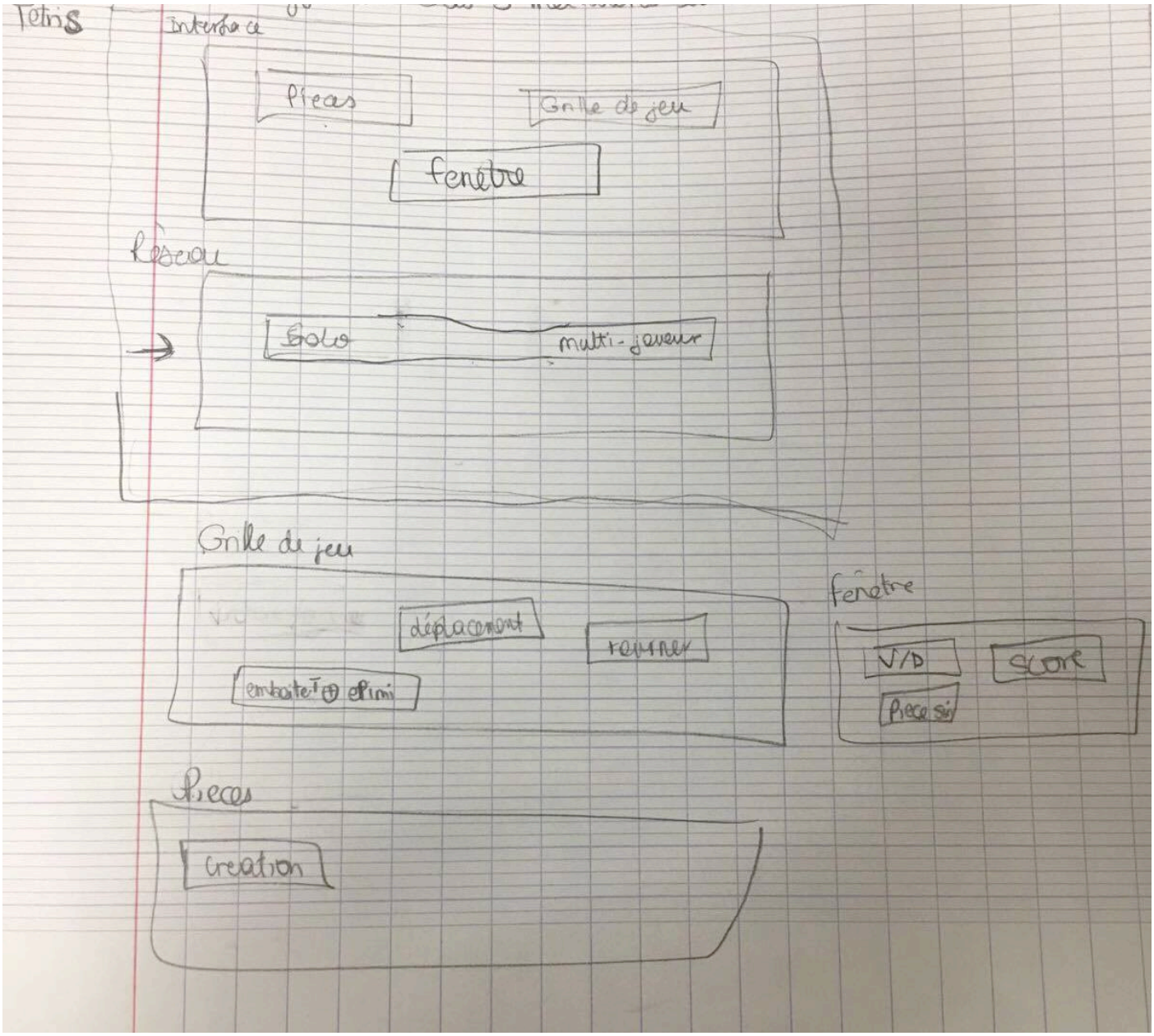
TP Architecture Logicielle / INF4043

Par Alexandra RAMRAMI & Nirosan NARATNAM

Tetris Multijoueur Java

3. Choix de développement

Dans un premier temps , nous avons effectuer un brain storming en essayant d'adopter une approche de l'architecture que nous souhaitions pour notre jeu Tetris :



TP Architecture Logicielle / INF4043

Par Alexandra RAMRAMI & Nirosan NARATNAM

Tetris Multijoueur Java

Nous avons ensuite implémenté le jeu Tetris en se basant sur la référence suivante:

https://en.wikipedia.org/wiki/Tetris

Ensuite nous avons implémenté le serveur et récrire le code pour un fonctionnement multi-joueurs.

Nous avons travaillé sur l'architecture en produisant les diagrammes suivants:

- ✓ activité
- ✓ séquence
- ✓ cas d'utilisation

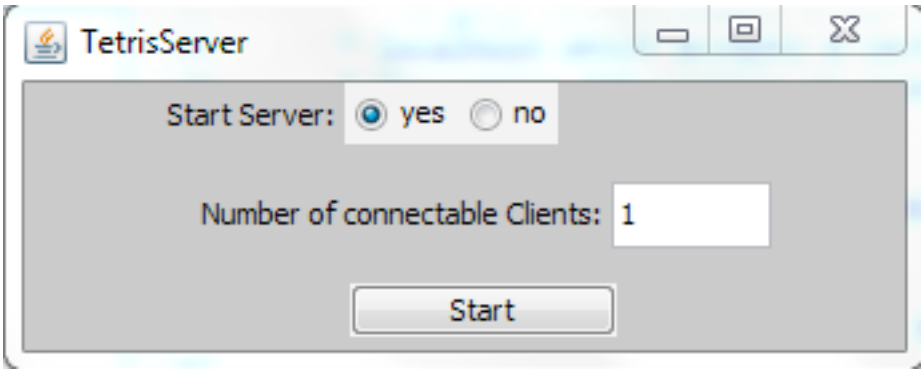
Nous avons utilisé Eclipse Mars 2.0 avec les plugins suivants :

- Findbugs

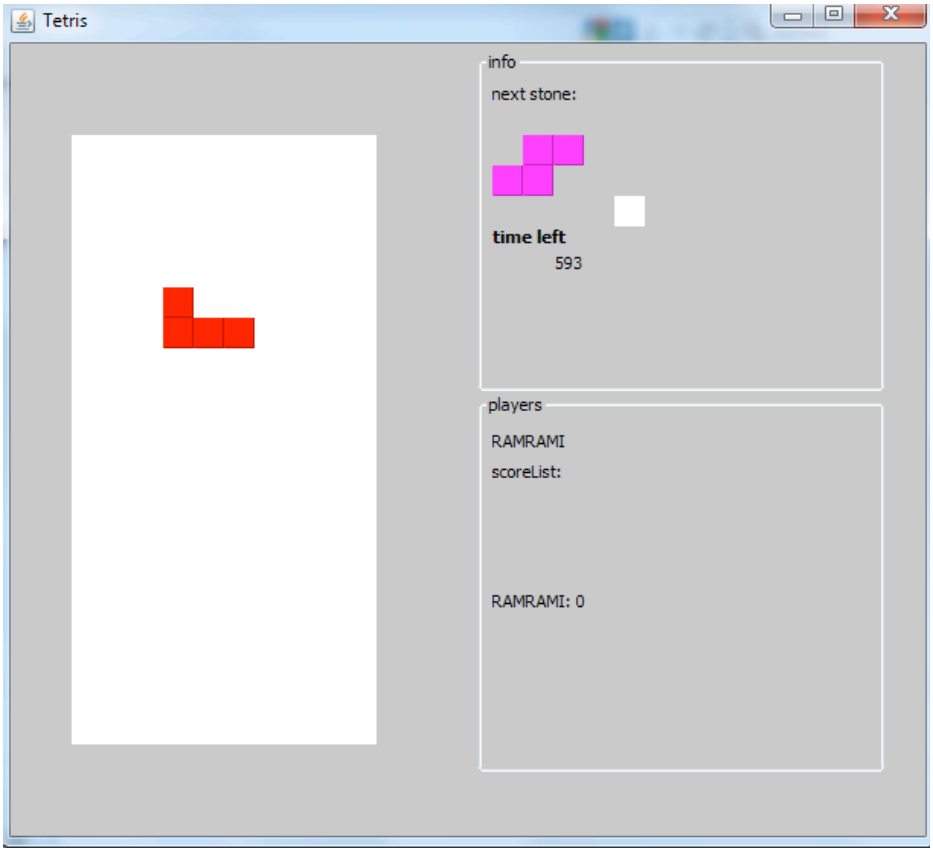
Le code a été testé avec Findbugs cependant à cause du manque de temps nous n'avons pas pu corriger les bugs trouvés.

Le manque de temps ne nous a pas permis de faire les tests unitaires et les assertions dans le code.

Quand on lance le jeu on a :



Ensuite nous cliquons sur start et nous obtenons :



4. Architecture et Design Patterns

Nous avons pris la décision de faire les designs patterns avec UML 2.0.

Nous avons essayé de respecter les principes SOLID suivants :

- ✓ S : Chaque classe implémente une seul responsabilité. Une chose à la fois.
- ✓ O : Chaque classe est fermé à la modification de son code source mais ouverte à l'extension
- ✓ L : Le principe de substitution de Liskov a été respecté au maximum
- ✓ I : Le couplage inutile entre les classes est supprimé en respectant le principe « Interface segregation »
- ✓ D : Nous avons essayé de respecter autant que faire se peu le principe « Dependency Inversion Principle ».
  Ci dessous les patterns utilisés :

a. Patterns de Création:

- Singleton : Assure qu'une seule instance d'une classe existe
- Factory : Remplace les constructeurs pour permettre
- Builder : permet de créer un objet complexe, éviter les constructeurs multiples et/ou avec trop de paramètres

- Prototype : Remplace l'utilisation d'un constructeur par la copie d'une instance existant
- Object Pool : Limite le nombre d'allocation/désallocation mémoire

b. Patterns de Structure

- Adapter : Permet d'adapter une implémentation à une interface donnée. Typique lors de l'intégration de librairie tierce dans un système existant
- Decorator : Encapsule un objet avec une interface commune pour modifier ou ajouter des fonctionnalités
- Proxy : Contrôle l'accès à une implémentation Virtual Proxy (Lazy initialization) Remote Proxy (implémentation locale, liée à un objet réel distant) Protection Proxy (autorise ou refuse l'accès à certaines méthodes/données) Switch Proxy (possède plusieurs implémentations, et change laquelle est utilisée) Composite : Un objet implémentant une interface, qui est composé d'un ensemble d'objets implémentant cette même interface
- Facade : Encapsule un groupe de fonctionnalités pour offrir une interface simplifiée
- Marker : Interface vide qui sert simplement d'information sur une classe
- Twin : Deux (ou plus) objets sont étroitement liés et doivent fonctionner ensemble
  Permet de simuler de l'héritage multiple

c. Patterns de Comportement

- Chain of responsability : Permet de déléguer le traitement d'une requête si une condition n'est pas validée
- Command : Encapsule les informations nécessaires à l'exécution d'une commande, avant qu'elle soit déclenchée plus tard

- Target : l'objet sur lequel la commande est appliquée (Receiver)
- Invoker : l'objet qui crée la commande (la stock) et l'exécute
- Iterator : Permet de visiter tous les éléments d'un conteneur
- visitor : Permet de visiter une structure complexe et d'agir sur chacun des éléments de ce e structure
- Memomento : Enregistre l'état d'un objet afin de le restaurer plus tard
  Observer : Permet à un objet de notifier ses changements d'état
- Listener : Permet à un objet de no er d'événements (changement d'état, actions utilisateurs, détection d'événements)
- Servant : Permet de définir un comportement commun à plusieurs classes sans dupliquer le code
  Utile lorsqu'une classe parent commune n'est pas possible Souvent classe utilitaire statique

d. Pattern de Concurrence:

- Lock/ Mutex : Méchanisme de synchronisation
  Un lock doit être obtenu avant de pouvoir lire ou modifier une donnée synchronisé

5.1. Diagramme de classe du modèle

Voici le diagramme de classe de notre jeu Tetris multi-joueurs.

La classe principale Jeu, est la classe maîtresse qui met en relation chacunes des instances nécessaires et les manipule afin de gérer les collisions entre la grille et la pièce courante, les calculs de score, et met à jour les lignes pleines si nécessaire.

Ce diagramme a été choisi car on a une vision globale du jeu. Le modèle UML 2.0 (Vous trouverez une version améliorée sur le github

TP Architecture Logicielle / INF4043

Par Alexandra RAMRAMI & Nirosan NARATNAM

Tetris Multijoueur Java

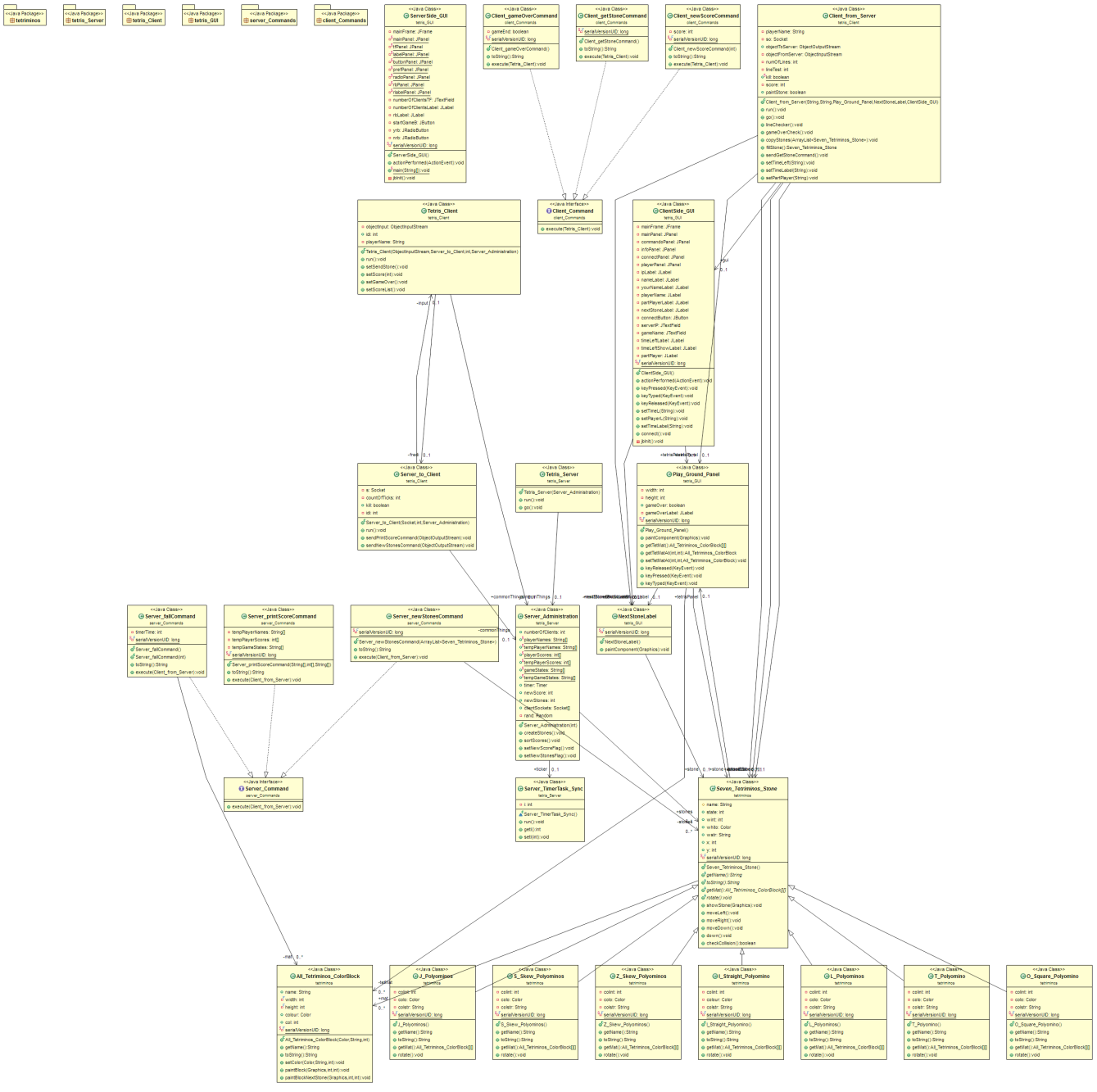
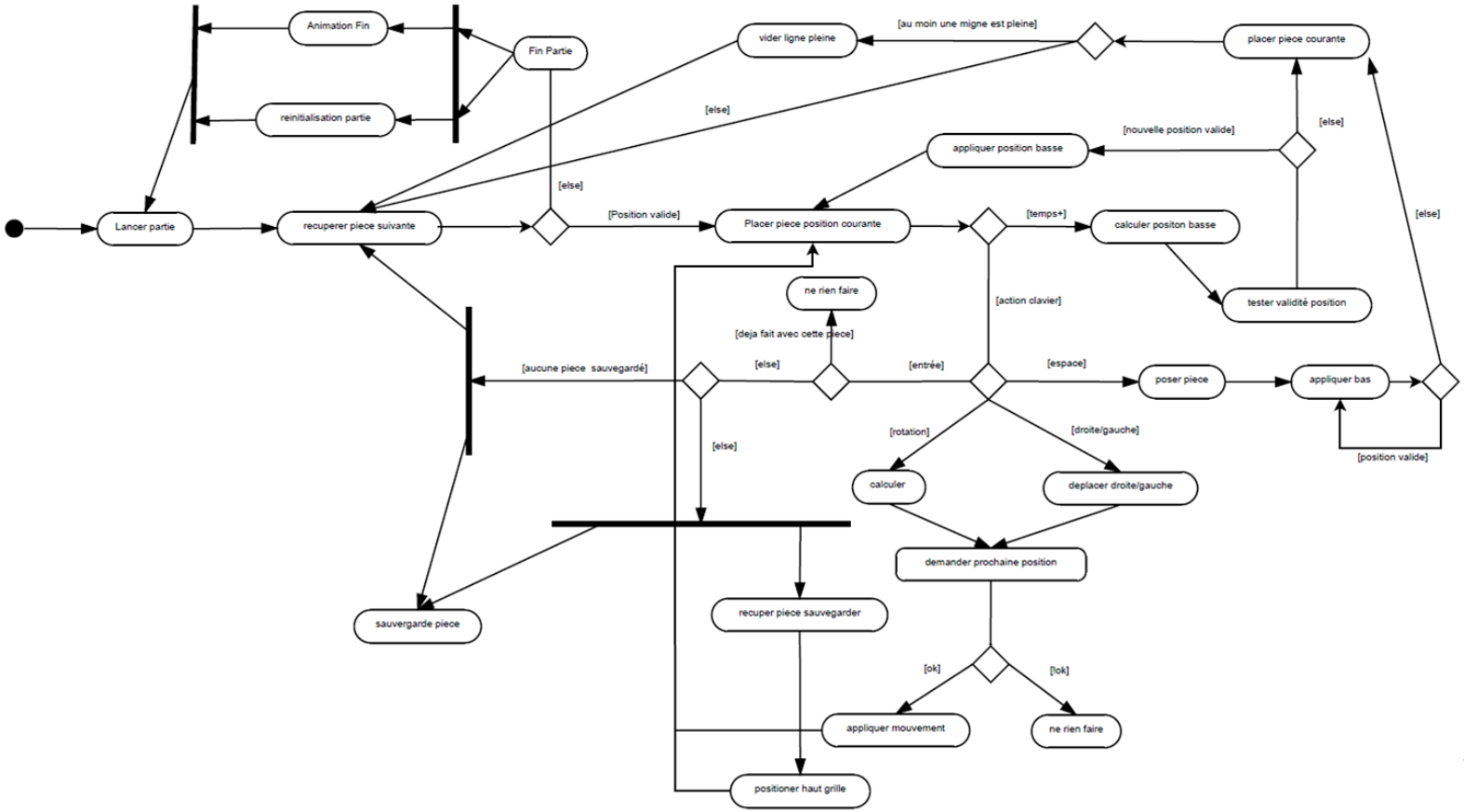


Diagramme de classe du modèle

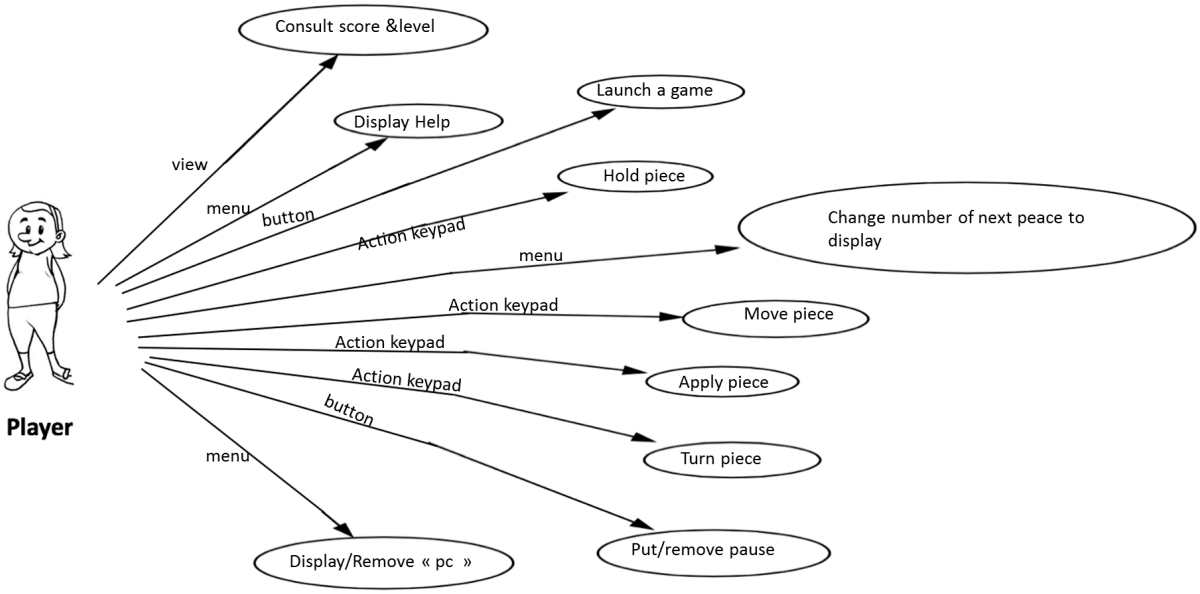
5.2. Diagramme de séquence d'une partie

Le diagramme d'activités complet du jeu. Ce diagramme a été choisir pour expliquer le déroulement du jeu Tetris.



5.3. Diagramme de cas d'utilisation

Voici la liste de fonctionnalités sous forme d'un diagramme d'utilisation :



Conclusion : Pour ce projet nous avons essayer de respecter au mieux les règles de l'architecture logiciel en choisissant les patterns qui nous semblait les mieux adaptés . Le diagramme UML à été générer avec un plugin additionnel sur Eclipse et le diagramme de séquences ont été créer grâce à power point.