DEV4 - Rapport Tetris Remise 2 Implémentation Console

Enzo RENARD - 60904 - D211 & Julien DELCOMBEL - 60189 - D212
Maître-assistant : Jonas Beleho

Introduction

Nous avons terminé la version console de notre projet. Dans ce rapport, nous détaillerons toutes les modifications apportées au jeu. Cela inclut les changements par rapport à ce qui était prévu initialement, les nouveaux ajouts fonctionnels et les ajustements pour améliorer la qualité de notre code. Nous discuterons également des problèmes rencontrés et des solutions que nous avons mises en place. Nous examinerons les bugs encore présents, ainsi que les avertissements émis par le compilateur, afin de proposer des pistes de solution. Enfin, nous fournirons une estimation du temps passé sur cette première partie du projet.

Modifications effectuées par rapport à l'analyse métier:

Model

Commençons tout d'abord par les modifications les moins notables du modèle.

Tetromino

Nous avons conservé la logique présentée lors de la modélisation. Cependant, nous avons ajouté un attribut permettant de savoir si cette pièce peut être tournée. En effet, pour des questions de cohérence des rotations, nous avons préféré ne pas effectuer de rotation lorsque les formes sont symétriques.
Tout au long de notre rapport, nous n'allons pas documenter nos ajouts de "getter", de "setter" ainsi que d'opérateurs, bien qu'ils soient effectivement présents.

Bag:

Il en va de même pour la classe Bag, nous avons conservé la logique présentée. Nous avons juste ajouté la méthode peekNext() nous permettant d'obtenir une référence constante vers le tetromino suivant sans que celui-ci ne soit retiré.

Continuons maintenant avec les plus grosses modifications du modèle.

Board:

La première modification notable dans la logique de notre jeu est que nous avons déplacé l'attribut gameOver de notre classe Game vers Board. En effet, nous avons compris que c'était le plateau de jeu qui donnait l'information de s'il était ou non en situation d'échec. Il va indiquer à la classe Game que le jeu se trouve dans une situation d'échec.
Nous avons également ajouté la méthode activeTetroIsBellow(), elle s'est révélée très utile lors de la vérification de la position du tetromino actif. A l'aide de cette méthode, nous

pouvons facilement réaliser un drop mais aussi et surtout nous pouvons vérifier la position d'un tetromino actif lors d'un mouvement. Nous pouvons désormais savoir si le tetromino est positionné en bas du plateau ou non.
La méthode isFree() à, entre autres, été renommée en isOccupied et permet désormais de vérifier si une case est occupée sur base d'une position absolue, mais aussi d'un tetromino.
Il n'est peut être pas utile de le mentionner mais nous avons également ajouté un getter pour le tetromino actif ainsi que pour l'ensemble des tetrominos du plateau. Elles nous permettent uniquement de pouvoir tester le modèle ou d'interagir avec la vue mais n'ont aucune dépendance envers les autres classes du modèle.

Game

Concernant notre classe Game, elle a subi bon nombre de modifications.
En premier lieu, nous avons modifié le comportement global de celle-ci. En effet, désormais le constructeur permet d'initialiser le jeu avec des paramètres par défauts. Il sera dès lors nécessaire de modifier les paramètres à l'aide des "setters" adaptés avant que le jeu n'ait démarré. Lorsque l'utilisateur est prêt, il peut faire appel à la méthode start() afin de démarrer le jeu. Cette classe a désormais toutes les responsabilités quant à la validité des paramètres du jeu.
Ceci m'amène donc à la deuxième modification notable : l'ajout de tous les attributs caractérisant la partie. Ils nous permettent à tout moment de connaître l'état de la partie en cours et de gérer correctement la fin du jeu et son état d'avancement.
Le reste des modifications ne sont donc exclusivement liées qu'à ces modifications.

Controller

- Nous avons gardé le Pattern Command. Nous avons cependant ajouté des classes pour d'autres commandes auxquelles nous n'avions pas pensé lors de l'analyse.

- StartGameCommand: Comme il était important de donner la possibilité au joueur de configurer sa partie, le jeu ne pouvait pas se lancer automatiquement au départ. Il était donc nécessaire d'ajouter cette commande pour permettre au joueur de lancer le jeu quand il était prêt.

- SettingsGameCommand: Dans la même logique, il fallait une commande permettant au joueur de lancer le configurateur de partie.

- RestartGameCommand et QuitGameCommand: A la fin d'une partie, pour éviter de "brutalement" quitter l'application, nous offrons, grâce à ces deux nouvelles commandes, la possibilité au joueur de choisir de relancer une partie ou de quitter.

GameController:

- C'est cette interface qui fait le lien avec le modèle et fait office de façade pour Game. Il était donc nécessaire d'y ajouter les méthodes adéquates dont les setters et getters.

Invoker:

- Nous avons pensé au cours de notre implémentation qu'il serait intéressant de pouvoir filtrer les commandes disponibles selon l'état du jeu. Pour cela, il fallait ajouter cet état aux commandes que nous enregistrions. Nous avons donc ajouté deux attributs à la classe Invoker: commandMap et currentState.
- commandMap est une multimap composé d'un string et d'une paire composé d'une commande et d'un vector de GameState. Ce choix permet d'attribuer un string correspondant à une commande et d'attribuer cette dernière à plusieurs états de jeu possible. En effet, certaines commandes comme les mouvements ne seront disponible que durant la partie mais une commande comme quit, par exemple, doit être disponible tout au long du jeu.
- Dans cette même logique, nous avons ajouté aux méthodes prévues, deux nouvelles méthodes setState(GameState state) et getState() const pour pouvoir gérer cela.

ApplicationTetris:

- Nous avons revu notre classe App, renommée ApplicationTetris afin qu'elle fonctionne comme le contrôleur principal de notre application. Nous avons sorti le Main de cette classe pour ne lui donner plus que le rôle de lanceur d'application. Nous avons ajouté différentes méthodes qui se chargent de faire le lien entre le modèle, la vue et les commandes. Cela nous semblait rendre plus lisible le fonctionnement de notre application et renforçait l'utilisation du Pattern MVC.

- ApplicationTetris(): L'ajout d'un constructeur qui initialise la vue et le GameController

- run(): la boucle du jeu. Indispensable à l'élaboration du développement d'un jeu.

- initializeCommands(): Bien que nous avons modélisé la méthode générale dans la classe Invoker dans notre diagramme de classe, nous n'avions pas précisé où les différentes commandes seraient concrètement enregistrées. Cela fait sens pour nous, que cela se fasse içi à la création d'une instance d'une partie.

- handleInput(): il nous manquait cette méthode pour gérer l'interaction de l'utilisateur avec le jeu.

- Bien que ces concepts étaient présents dans notre analyse, cela était peut être trop abstrait. La refonte de cette classe nous permet le fonctionnement concret du jeu.

View

BagView:

- Nous avons supprimé l'utilisation d'un attribut dans cette classe car nous pouvons plus simplement passer une référence en argument d'une méthode qui en aura besoin.

- Nous avons donc ajouté la méthode publique drawNextTetromino(const Bag& bag) qui se charge concrètement d'afficher le prochain tetromino depuis le bag passé en argument. Cette méthode est publique afin d'en permettre l'utilisation par le contrôleur chargé de faire le lien entre le modèle et la vue. L'argument est devenu une référence constante d'un bag plutôt qu'un vector de tetrominos comme annoncé car c'est le bag qui contient l'information que nous recherchons içi et ne pourra être modifié grâce au const.

- Nous avons également ajouté des méthodes privées createSymbolMap(), createAndFillGrid() et printGrid() nécessaires à drawNextTetromino() pour une meilleure répartition des tâches, une meilleure lisibilité du code et un entretien facilité.

BoardView:

- Cette classe étant responsable de l'affichage du plateau de jeu, nous avons içi aussi créé une méthode publique drawBoard(const Board& board) pour concrètement effectuer la tâche en lui passant un argument plutôt qu'utiliser un attribut de classe. L'argument est constant pour éviter de le modifier et le board contient l'état du jeu en cours nécessaire à son affichage.
- Même logique de séparation des tâches, lisibilité et entretien avec l'ajout des méthodes privées nécessaires à drawBoard() que sont createSymbolMap(), createAndFilGridl() et printGrid()

GameView:

- Les attributs de la classe ne sont plus directement des instances de bag, board et score mais une instance du GameController qui sert à faire le lien entre la vue et le modèle sans que ceux ci interagissent directement. Le GameController se chargera donc de fournir les informations du jeu à la vue sans que celle-ci puisse interagir directement avec.

- Il s'agit de notre classe principale pour la Vue du jeu. Elle est chargée de l'ensemble de l'affichage et il lui manquait donc quelques méthodes dans notre diagramme de classe initiale pour répondre à tous nos besoins. Nous avons donc ajouté des méthodes spécifiques à chaque élément à afficher tels qu'un menu principal avec displayMenu(), un message de fin avec displayGameOver(), etc.

- Nous avons également ajouté une méthode displaySettings() avec une série de méthodes privées chargées de recevoir les entrées de l'utilisateur. Nous avons fait le choix de réunir ces deux tâches en une pour une expérience utilisateur plus agréable et que le inputHandler de la partie controller soit spécialisé dans les commandes du jeu entré par l'utilisateur.

Bugs restants:

- ☐ Le niveau de l'utilisateur ne peut actuellement pas démarrer à 0. Il est nécessaire de le démarrer à 1. Ceci implique une erreur lors du calcul du premier niveau par rapport aux nombres de lignes supprimées

Warning restants:

N/A

Problèmes rencontrés et solutions:

- Des erreurs de segmentation dues à des méthodes trop longues. La segmentation en méthode plus petite nous à permis de trouver et corriger ces problèmes.

Estimation du temps de travail:

Nous nous sommes basés sur le nombre de commits. Certains, sur la logique du métier prenant du temps, et d'autres impliquants de petits correctifs plus rapide, nous prenons une moyenne de 30 minutes par commit et estimons donc le temps de travail nécessaire:

$$170 * 30\,min = 5100\,min = 85\,h$$