Patchwork

Rendu Intermédiaire

Architecture et choix de programmation

Programmation en Java

RADONIAINA Gabriel
NGUYEN Hervé
3 décembre 2022
L2 Informatique

Table de matières

Introduction

Dans ce rapport nous allons expliquer nos choix jusqu'à maintenant pendant le développement du jeu Patchwork dans le cadre du projet de programmation en Java.

Structure des données principales du jeu

Dès le début du projet, nous avons d'abord commencé à penser à la manière dont les données du jeu étaient stockés.

Dans cette étape, nous avons tenté de visualiser comment on manipulera les données plus tard dans le développement afin de limiter les retouches majeures qui pourront nous coûter du temps.

Prenons exemple sur le record `Patch` , nous avons décidé d'utiliser un record car on sait que ses champs ne changent pas drastiquement au fil du jeu pour un patch donné. En effet, la forme, le prix, la valeur de temps, ni le nombre de bouton varie pour un patch spécifique.

Également il y un certain avantage d'utiliser un record plutôt qu'une classe, les accesseurs et les méthodes comme `equals`, `hashCode` sont bien implémentés par défaut.

Au niveau de la représentation des joueurs , nous avons d'abord eu recours à un record pour avoir les bénéfices des accesseurs. Mais au fil du

développement, nous avons constaté que cette décision était peu convenable parce que les champs des joueurs peuvent changer au fil de la partie. Ce qui nous forcerait à écrire des méthodes qui renvoient un nouveau joueur à chaque modification. Notre choix actuel est alors d'employer une classe `Player` qui aurait ses champs modifiable (donc pas `final`).

Pour les champs du joueur, il a été décidé d'associer toutes les données propre au joueur (lors de la partie) dans ses champs. Par exemple le nombre de boutons en possession, le quiltboard à construire par le joueur lors de la partie (qui sera donc modifiable), sa localisation dans le time board, son identifiant (pour le différencier avec d'autre joueurs) et enfin un booléen représentant le fait que le joueur a terminé de parcourir le time board.

Pour le quiltboard des joueurs, nous nous sommes mis d'accord qu'un tableau de booléen pour représenter les zones occupées sur le quiltboard était pertinent. Cela permet de developper des méthodes simple pour vérifier si un patch était possible à placer ou non, des méthodes de calcul de score.

En prévision pour le développement de la version graphique, nous avons ajouté à la classe `QuiltBoard` un champs `patchplaced` qui est une liste de couple de `Patch` et `Point (java.awt.Point)`. Afin de pouvoir accéder à la localisation exacte de chaque `Patch` placés.

Pour représenter le time board, qui est le plateau de déplacement des pions des joueurs. Nous avons choisi de représenter cela à travers un tableau de `TimeBoardElement` (énumération des différents type d'object qu'on pourrait retrouver sur ce plateau tel que les boutons, les patch spéciaux).

Architecture

Pour déterminer comment les différents objects vont être utilisés dans l'algorithme du jeu, le choix a été d'abord d'essayer de développer le plus de petites méthodes d'instances dans les classes principales (`Player`, `QuiltBoard`, `Patch`, `TimeBoard`) qui pourront être utile à l'élaboration de l'algorithme.

Après que cela soit fait à cette échelle, un premier algorithme à grande échelle dans le main avec un nombre conséquent de ligne a été écrit (la raison étant que nous n'avons pas réussi à visualiser de manière précise et efficace des méthodes intermédiaire pertinentes) dans la perspective de refactoriser certaines parties.

Finalement, nous avons pu refactoriser l'algorithme et le projet fonctionne de cette manière :

- La classe Displays contient des méthodes permettant d'afficher dans le terminal des objets tel que les `Patch`, les `QuiltBoard`, le `PatchCircle`, le TimeBoard (et les positions des joueurs sur celui-ci)
- Les classes `Patch`, `PatchCircle`, `Player`, `QuiltBoard`, `TimeBoard`, `TimeBoardElement` représentent les éléments du jeu Patchwork
- La classe Main prend en charge les arguments passés par l'utilisateur pour déterminer quel mode de jeu lancer, puis lance `GameAlgorithm.terminalMode` avec les arguments appropriés
- La classe `GameAlgorithm` contient l'algorithme de jeu principal (pour l'instant la seule méthode est `terminalMode` qui appelle des méthodes de `TerminalAlgorithm`)

- La classe `TerminalAlgorithm` qui contient des méthodes reprenant certaines partie du déroulement d'un tour du jeu. Étant donné que le

comportement des entrées sorties dans la phase 1 et 2 est très différente du jeu lors de la phase 3 et au delà, nous estimons qu'il était nécéssaire de déporter toutes méthodes qui avaient été trop dépendantes de l'utilisation de `scanner` et `println` (IO sur le terminal) vers cette classe.

- La classe `Tools` contient des méthodes « outils » qui ne peuvent pas être associées « philosophiquement » aux classes précédemment énoncées. Ils sont utilisés dans `TerminalAlgorithm` et `GameAlgorithm`

Quand on lance le jeu, on fait appel à `Main.main` puis à `GameAlgorithm.terminalMode` se sert des méthodes de `TerminalAlgorithm` qui lui se sert des méthodes des objets tangibles du jeu (`Player`, `PatchCircle`, `Patch`...).

Remarques

Nous avons eu des difficultés jusque'à maintenant à trouver des cas où il est pertinent d'utiliser des `Stream` et des `Lambda`.

Nous avons envisagé de créer une interface pour Player afin de possiblement avoir `PlayerTerminal` et `PlayerGraphical` qui implémente Player mais selon le type de jeu (affichage graphique ou sur terminal) auront leur propre implémentation (en parlant de méthodes sont équivalents aux méthodes de `TerminalAlgorithm`). Cette proposition a été écartée dû au fait que cela aurait rendu Player beaucoup plus complexe car on y ajoute des méthodes qui sont au delà de la représentation d'un joueur.

Perspectives d'évolutions

À l'état actuel du projet, afin d'implémenter le mode graphique sous Zen. Nous estimons qu'il est nécéssaire de fournir un équivalent graphique qui fonctionne de manière graphique de `TerminalAlgorithm` (qui nécessitait beaucoup de scanner, et utilise beaucoup de message avec println), de modifier légèrement `GameAlgorithm` pour utiliser ces équivalents.

La classe `Displays` aura possiblement son équivalent direct, mais sans savoir connaissance au préalable le fonctionnement de Zen, nous ne pouvons pas encore trancher.

Le fonctionnement dans le fond des algorithmes de jeu ne va probablement pas changer drastiquement mais des ajustements au niveau de l'IO pour recevoir les informations et afficher les données devront être fait.

`Patch` va subir probablement quelques modifications mineurs pour ajouter des informations pour l'affichage cohérent des boutons sur un patch.

Mais pour l'instant nous pensons que les objets tangibles du jeu ne nécessitent plus pour l'instant des modifications majeures (mise-à-part l'ajout d'un champs ou deux si cela est nécéssaire).