Université Gustave Eiffel

Rapport POO Java:

Projet:

Baba Is You

SOMMAIRE :

1 Développeur :

1.1 Changement depuis la soutenance bêta :

Nos erreurs étaient un mauvais choix de la structure pour représenter le plateau de jeu et la non-séparation de la partie graphique et la partie « jeu ». C'est donc sur ces points que nous nous sommes focalisés.

1.1.1 Ajouts :

- **Rule** : Classe qui gère les règles du jeu.
- **Movement** : Classe qui gère les déplacements.
- **Board** : Classe qui représente le jeu (contient un champ Rule, Movement…)
- **Parser** : Classe qui sert à analyser un fichier.
- **Decor** : Classe enum pour les éléments décoratifs.
- **BlockInter** : Interface hérité par tous les types d'Enum (Name, Property, Text…)

1.1.2 Modification :

- **Block** : Classe Abstraite -> **Record**.
- Position -> **Coordinates** (Classe -> **Record**)
- Names -> **Name** (Enum)
- Properties -> **Property** (Enum)
- Operators -> **Operator** (Enum)
- **Draw** : Interface -> **Classe**
- **Level :** séparation de l'affichage et du jeu.

1.1.3 Suppression :

- ListBlock (Classe)
- Window (Interface)
- (Anciennement) Name (Classe)
- (Anciennement) Property (Classe)
- (Anciennement) TextBlock (Classe)
- (Anciennement) Operator (Classe)

1.2 Architecture :

1.2.1 Block :

Comme montré dans les modifications apportés, Block est désormais un Record possédant un champ « name » de type BlockInter.

BlockInter est une Interface (d'où le Inter…) héritée par toutes les classes enum désignant des types de Block (Name, Property…) chacune de ces classes enum implémente la méthode getType() de l'interface, qui renvoie le type du Block (définit dans BlockType).

Nous avons supprimé plusieurs classes (voir Développeur :Suppression :) pour factoriser le code de cette manière.

1.2.2 Board :

Nous avons changé notre structure pour représenter le jeu comme recommandé lors de la soutenance bêta :

-LinkedList<Block>[][] **to** **HashMap<Coordinates, TreeSet<Block>>**

Notre board est aussi constitué de règles (champ : private Rule rules) représentant les règles du board, de déplacements possible (champ : private Movement movements), d'actions (champ : private Action actions).

On a choisi d'utilisé un TreeSet pour rendre l'affichage plus élégant de sort à pouvoir trier les Block et d'afficher les images les plus « grosses » en premier afin qu'elle ne cache pas les plus petites. (Par exemple : si le BABA se trouve sur la même case qu'un ROCK alors le ROCK est en arrière-plan pour ne pas cacher le BABA, pareille pour les propriétés…).

1.2.3 Déplacements :

Pour les déplacements, nous avons choisi d'utiliser une **LinkedHashMap<Coordinates, Set<Block>>**, pour garder l'ordre d'insertion des Blocks (utile savoir quelle Block bouger en premier et éviter des bugs).

Cette structure de données nous sert à stocker les coordonnées et les Block qui doivent être déplacés, pour savoir si un Block peut bouger on utilise la méthode « canMove » de façon récursive si un Block à déplacé possède la propriété PUSH ou YOU.

Une fois la Map remplie, on bouge tous les Block précédemment ajoutés (nous avons essayé de gérer le plus de cas particuliers possibles, par exemple si deux Block avec la propriété PUSH se trouve sur la même case alors on bouge les deux en même temps).

1.2.4 Règles :

Pour les règles, nous avons choisi d'utiliser une **Map<Property, Set<Name>>,** pour chaque propriété est associé un dictionnaire de Name (Block jouable (WALL, BABA...)) qui possède la propriété.

Nous avons aussi ajouté une **Set<Map<Name, Property>>** pour les règles activées avec l'option –execute (voir Arguments (Options...).

A chaque mouvement on clear chaque Set de la Map, et on parcourt le board pour affecter les règles (horizontaux/verticaux), nous avons essayé de faire le moins de « new » possible ce qui donne lieu à des streams parfois peu lisibles…

1.2.5 Affichage :

L'affichage se fait dans la Classe Draw, le plus gros changement concernant l'affichage a été l'utilisation d'une **Map<Block, Image>** pour affecté à chaque Block son image et chargé les images qu'une seule fois plutôt que de faire un new à chaque fois que l'on rencontre un Block lors du parcours du board (ce que nous faisions avant).

1.2.6 Main :

Dans le main, on itère sur la liste de level (voir Arguments (Options...), et on change de niveau si on perd ou gagne, nous avons utilisé « Thread.sleep(400); » pour mettre un petit délai pour voir le déplacement qui a provoqué la victoire/défaite, au lieu de quitter instantanément.

1.2.7 Arguments (Options...) :

1.2.7.1 --execute :

Pour l'option –execute on utilise une **Set<Map<Name, Property>>**, on parcourt les arguments, et pour chaque commande –execute valide on ajoute ses valeurs à la structure de données, les règles utilisées avec l'option –execute dure tout le temps, si on tape --execute ROCK IS PUSH, alors ROCK sera pushable tout le temps.

1.2.7.2 --level et --levels :

Comme pour –execute, on parcourt les arguments et pour chaque arguments valide, on construit le chemin menant au fichier/dossier spécifié, et on initialise un Parser avec le contenu du/des fichier, on crée et remplit ensuite un Board avec le contenu du/des fichiers.

Et on crée et ajoute le level avec son Board associé à la liste des levels.

Pour ouvrir les fichiers nous avons utilisé « java.nio.file.Files » comme recommandé par M.Forax.

Nous avons utilisé System.getProperty("user.dir") pour obtenir le chemin du répertoire de travail actuel à cause de problème rencontré lors de la recherche des chemins des fichiers image et level lors de l'exécution de l'exécutable jar.

1.2.8 Parser :

Pour les fichiers représentant un level on a décidé d'utiliser un format simple :

24 18 ← 1ère ligne : taille du board (colonne) (ligne)
Name BABA 13 14 ← (Type du Block) (Nom du Block) (ligne) (colonne)

On split donc chaque ligne avec un espace comme délimiteur, et on construit notre board avec le contenu de la ligne.

1.2.9 Level :

Chaque level contient un board, ses dimensions (levelHeight, levelWidth) ainsi qu'une taille de case adaptée (calculée de sorte à avoir des cases carrées et non rectangulaires), un context et un champ draw (pour l'affichage).

1.3 Niveau Exclusif : Le Sacrifice.

Le choix était limité pour ajouter des nouveaux mots exclusifs tout semblait déjà fait dans le vrai jeu.

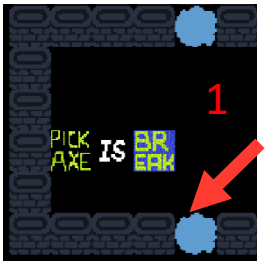
Alors nous avons décidé d'ajouté la propriété BREAK et l'élément PICKAXE (pour faire le lien avec BREAK).

La propriété BREAK comme son nom l'indique confère le pouvoir de détruire n'importe quel Block ne possédant pas cette propriété (un peu similaire à SINK mais en plus puissante), la propriété a le dessus sur les autres propriétés.

1.3.1 Solution :

Si vous lisez cette partie c'est que vous n'êtes vraiment pas très doué…x)

Pour résoudre le niveau exclusif, il faut détruire les Blocs WATER à droite.



Puis il faut détruire les 2 Blocs WATER à gauche.

Puis enlever les propriétés FLAG IS PUSH et PICKAXE IS BREAK.
Et retourner pousser la PICKAXE jusque dans l'eau (sacrifié la pioche).
Descendre désactiver la propriété WALL IS STOP et monter activer la propriété FLAG IS WIN.
Et voilà c'est terminer pour la solution.

2 Problèmes Rencontrés :

2.1 Image .jar :

Nous avons eu un problème pour accéder aux images depuis le .jar (lors de son exécution), et nous n'avons pas réussi à ajouter les ressources à l'exécutable, nous avons donc résolu le

problème en utilisant ***System.getProperty("user.dir")*** pour spécifier le chemin à partir du dossier courant et non à partir du .jar.