UNIVERSITÉ
CAEN
NORMANDIE

Méthode de conception

Rapport explicatif

Jeu Tetris

Groupe

KACED	Louheb	22111744
DJEHA	Wassim	22208244
KABAR	Abderrahmene	22309225
MANSOUS	Younes	22110931

● Introduction :

Ce rapport documente la réalisation d'une version repensée du jeu Tetris en Java, se concentrant spécifiquement sur l'implémentation habile de divers design patterns. Guidés par l'approche méthodique du projet "Fil Rouge," notre objectif est de détailler comment ces motifs de conception ont été appliqués pour résoudre des défis particuliers, tels que la manipulation des pièces et la gestion des états de jeu. Chaque décision de conception est scrupuleusement examinée dans le contexte de son impact sur la structure modulaire, la maintenabilité, et l'extensibilité du code, visant ainsi à créer un Tetris fonctionnel, où l'essence même du projet réside dans la mise en œuvre judicieuse des design patterns.

● Les parties du projet :

1. Fonctionnalités principales :

- Collision :

La détection de collision est gérée par la méthode `collision`. Cette méthode prend les coordonnées (`x` et `y`), un objet `PiecePuzzle` (`p`) et un indicateur booléen (`bool`). Elle renvoie une valeur booléenne indiquant si une collision se produirait à la position spécifiée. La méthode `collision` crée d'abord une copie de la grille actuelle (`copy`) pour simuler l'état sans la pièce. Ensuite, elle vérifie si la pièce entrerait en collision avec des cellules occupées dans la grille. Si `bool` est `true`, elle met à jour la copie en supprimant les cellules occupées de la pièce pour simuler l'état après que la pièce a bougé. La collision est le dernier maillon de notre chaîne de responsabilité, c'est-à-dire qu'après avoir vérifié si la prochaine action (mouvement, rotation ou création) ne fait pas sortir la pièce de la grille, nous vérifions ensuite si cela n'entraîne pas de collision avec une pièce déjà existante.

- Rotation :

La rotation est gérée par les méthodes `rotateLeft` et `rotateRight`. Ces méthodes vérifient d'abord les collisions à l'aide de la méthode `collisionRotation`. S'il n'y a pas de collision, la rotation de la pièce est ajustée en conséquence en utilisant les méthodes `leftRotation` et `rightRotation` de la classe `PiecePuzzle`. Les méthodes `leftRotation` et `rightRotation` de `PiecePuzzle` changent l'orientation de la pièce sélectionnée vers sa prochaine forme parmi les 4 qu'elle possède, propre à chaque degré (0°, 90°, 180°, 270°).

- Mouvement :

Le mouvement est géré par la méthode `movePiece`. Elle prend une direction (1 pour bas, 2 pour droite, 3 pour gauche et 4 pour haut) et un objet `PiecePuzzle` (`piece`). La méthode vérifie d'abord les collisions à l'aide de l'objet `handler`. S'il n'y a pas de collision, le centre de la pièce est mis à jour et la grille est actualisée en appelant la méthode `allocatePieces`. Si une collision est détectée, une exception `OutOfMapException` est levée avec le message "impossible place already taken" (emplacement impossible déjà pris).

- Allocation des piéces dans la grille :

La méthode `allocatePieces` met à jour la grille pour refléter les positions actuelles des pièces. Elle itère à travers la liste de formes (`listOfShapes`) et définit les cellules correspondantes dans la grille comme occupées. Les méthodes `refreshGrid` et `refreshGrid(PiecePuzzle piece)` sont utilisées pour réinitialiser l'état de la grille avant d'allouer de nouvelles pièces.

2. Design Pattern :

Dans le cadre du développement de notre jeu, nous avons opté pour l'utilisation de plusieurs design patterns pour garantir une architecture logicielle flexible et modulaire. Trois de ces patterns se démarquent particulièrement : la Factory, la Strategy, et la Chain of Responsibility. Chacun de ces concepts joue un rôle spécifique dans la création d'objets, la gestion dynamique de la difficulté du jeu, et l'orchestration des requêtes au sein du système. Ce rapport explore l'application pratique de ces design patterns, démontrant ainsi leur utilité dans la conception d'un système de jeu cohérent et évolutif.

- Factory :

Cette section traite de la création d'objets variés à l'aide d'une classe dédiée, en l'occurrence la classe `PieceFactory`. Trois méthodes spécifiques, `createShapeL()`, `createShapeT()`, et `createShapeU()`, ont été mises en place pour créer différentes formes de pièces.

Dans la phase d'utilisation, notamment dans la méthode `initializeGame()`, un mécanisme conditionnel est introduit pour déterminer le type de forme à créer en fonction d'un paramètre spécifié (`forme`). Chaque branche conditionnelle utilise la `PieceFactory` pour créer la forme respective, en fonction de la valeur de `forme`.

```
Piece piece;

if (forme == 1) {
    piece = pieceFactory.createShapeL(new PointTetris(row, col), pieceSize, pieceSize, pieceOrientation);
} else if (forme == 2) {
    piece = pieceFactory.createShapeT(new PointTetris(row, col), pieceSize, pieceSize, pieceOrientation);
} else if (forme == 3) {
    piece = pieceFactory.createShapeU(new PointTetris(row, col), pieceSize, pieceSize, pieceOrientation);
}
```

Il convient de noter que préalablement à l'utilisation de la `PieceFactory`, il est impératif de l'instancier et de définir les valeurs nécessaires telles que `row`, `col`, `pieceSize`, et `pieceOrientation`. Cette approche modulaire facilite l'extension future du jeu avec de nouvelles formes de pièces.

- Strategy :

L'objectif de cette section est de détailler la mise en place d'une stratégie dynamique de modification de la difficulté du jeu. Pour ce faire, nous avons introduit une interface `InitialConfigurationStrategy` comportant une méthode `initialConfiguration()`. Cette méthode est ensuite implémentée par les classes `MediumDifficulty`, `HardStrategy`, et `DefaultStrategy`, chacune redéfinissant de manière distincte la façon dont l'initialisation de la grille doit être effectuée en fonction de la difficulté choisie.

Dans l'utilisation pratique de cette stratégie, le programme principal instancie un plateau de jeu, puis utilise la méthode `setStrategy(InitialConfigurationStrategy strat)` pour choisir la stratégie de configuration initiale souhaitée. Enfin, le lancement de la méthode `plateau.initializeGame()` effectue l'instanciation du plateau selon la stratégie sélectionnée.

```
Plateau plateau = new Plateau();
plateau.setStrategy(new MediumDifficulty());
plateau.initializeGame();
```

- Chain Of Responsability :

Le pattern Chain of Responsibility a été utilisé dans notre projet Tetris pour simplifier la vérification des différentes conditions liées au placement des pièces sur le plateau de jeu.

Cette approche permet de créer une chaîne de gestionnaires (handlers) qui se succèdent pour traiter une demande spécifique. Chaque handler a la responsabilité de traiter la demande ou de la transmettre au handler suivant dans la chaîne.

- ## Interface TetrisHandler

L'interface TetrisHandler définit la méthode `handlePiece` qui sera implémentée par chaque handler de la chaîne. Cette méthode prend en paramètre les coordonnées (x, y) de la pièce, la pièce elle-même, le plateau de jeu, et un paramètre booléen qui peut être utilisé pour des conditions spécifiques.

- ## Classe AbstractTetrisHandler

La classe abstraite AbstractTetrisHandler implémente l'interface TetrisHandler et sert de base pour les handlers concrets. Chaque handler contient une référence vers le handler suivant dans la chaîne. Si le handler actuel ne peut pas traiter la demande, il la transmet au handler suivant.

- ## Classe BoundaryHandler

La classe BoundaryHandler est un handler concret qui vérifie si la pièce se trouve à l'intérieur des limites du plateau. Si tel est le cas, il transmet la demande au handler suivant dans la chaîne. En revanche, si la pièce se trouve à l'extérieur des limites, le traitement de la demande s'effectue au sein même de ce handler, évitant ainsi la transmission à un handler suivant en retournant false

- ## Classe PieceCollisionHandler

La classe PieceCollisionHandler est un autre handler concret qui vérifie s'il y a une collision entre la pièce et une autre déjà présente sur le plateau. Il utilise la méthode `collision` du plateau pour effectuer cette vérification si y’a pas de colision il renvoie true sinon false

- ## Utilisation dans le code

Dans le code d'utilisation, la méthode handlePiece est appelée sur le handler initial avec les coordonnées de la pièce. Chaque handler dans la chaîne va alors traiter la demande ou la transmettre au suivant. La logique spécifique à chaque type de vérification est encapsulée dans les handlers, ce qui rend le code plus modulaire et lisible.

En résumé, l'utilisation du pattern Chain of Responsibility nous permet d'organiser et d'écrire le code de manière plus propre en évitant l'imbriquement de multiples conditions. Au

lieu de cela, nous remplaçons cette approche par la séquence de lancement de handlers, améliorant ainsi la lisibilité, la modularité et la maintenabilité du code.

- ## Commandes de compilation et d'exécution:

Pour lancer le jeu on peut directement taper la commande : **ant** dans le terminal qu'on ouvrira dans le répertoire racine **/livraison** , cette commande permet directement de lancer le fichier **build .xml** , ou on peut utiliser la manière classique comme suit :

suivez ces étapes pour la compilation et l'exécution du code.

1. Tout d'abord, ouvrez un terminal dans le répertoire source /**livraison** de votre projet.

2. Ensuite, pour compiler toutes les classes nécessaires, utilisez la commande suivante:

javac -d build src/model/*/*.java src/controleur/*.java src/vue/*.java

3. Une fois la compilation réussie, vous pouvez exécuter soit en mode console ou graphique :

- Mode console :

java -cp build model.plateauPuzzle.Demo

- Mode graphique:

java -cp build vue.StartWindow