Piece Out

1. Introduction

Nous avons développé un jeu de puzzle intitulé Piece Out » en utilisant le langage C++ et la bibliothèque SFML. Le principe du jeu est de manipuler des pièces sur un plateau pour atteindre des objectifs précis, comme sortir une pièce d'une zone ou la placer sur une position cible. À partir d'une ébauche de code fournie par notre professeur, nous avons amélioré une structure initiale qui manquait de séparation claire entre les composantes Modèle, Vue et Contrôleur (MVC) et d'utilisation de patrons de conception.

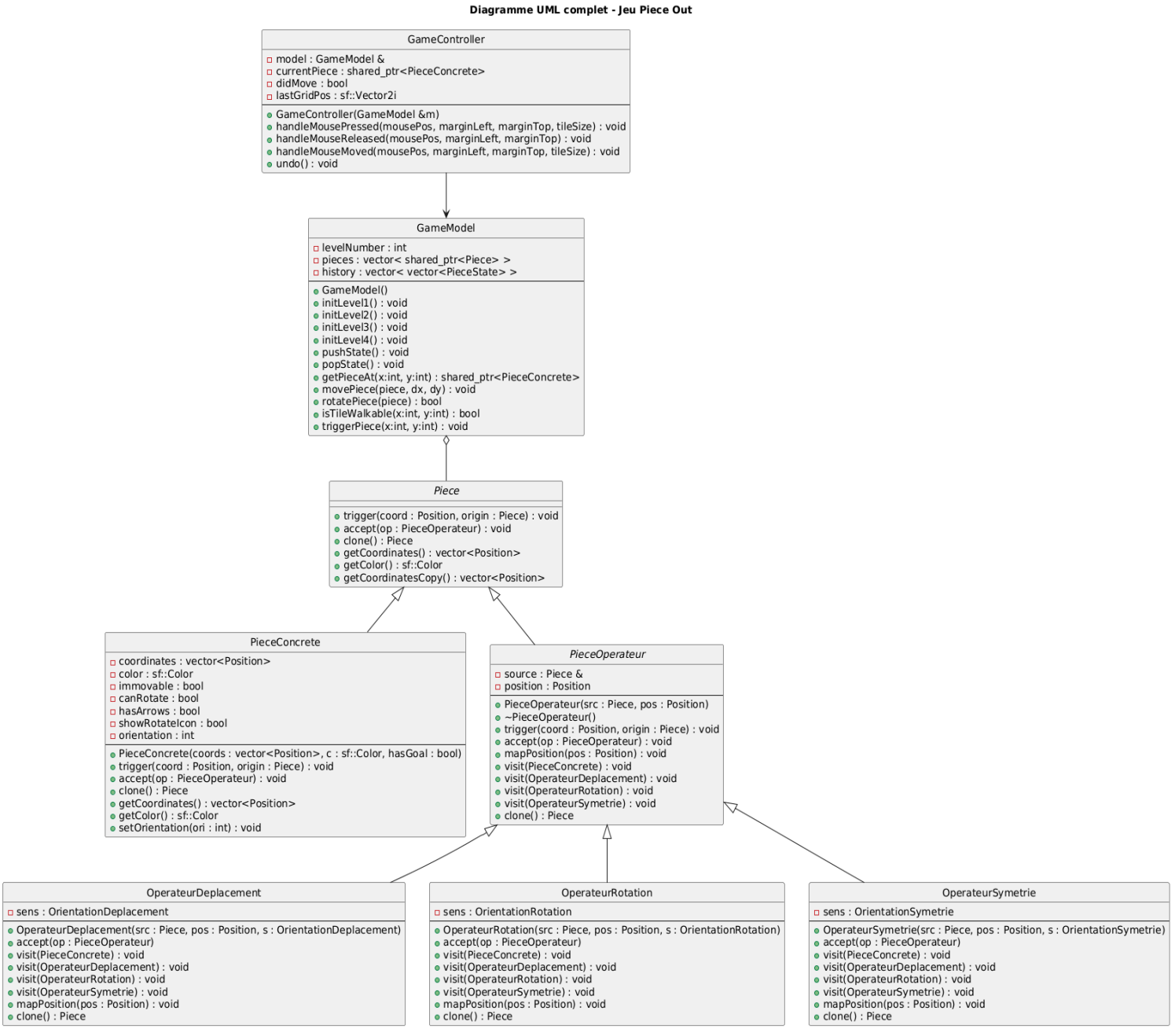
Pour rendre le projet plus maintenable et évolutif, nous avons restructuré le code en trois couches : le Modèle (logique du jeu), le Contrôleur (gestion des interactions utilisateur) et la Vue (affichage via SFML). Nous avons introduit le patron de conception Décorateur pour permettre des enchaînements d'effets (déplacement, rotation, symétrie) déclenchés par des clics. Le patron Visiteur a également été utilisé pour déléguer les actions aux classes adaptées.

Les pièces du jeu possèdent des propriétés comme des coordonnées, des couleurs, et parfois des caractéristiques spécifiques (immobilité, capacité de rotation). Certaines pièces comportent des flèches indiquant des directions possibles, ce qui permet au contrôleur de déclencher les actions correspondantes. Une gestion d'historique avec une fonction « Undo » a également été implémentée grâce à un système de pile permettant de revenir à un état précédent.

Le rapport de projet se divise en cinq parties : introduction, diagramme UML détaillé, architecture et fonctionnement du code, problèmes rencontrés et solutions, et conclusion. Chaque section vise à clarifier nos choix de conception et les stratégies adoptées.

2. Diagramme UML Complet

Afin de représenter clairement l'architecture du projet, nous avons réalisé un diagramme UML détaillé qui inclut toutes les classes majeures du projet : GameModel, GameController, Piece (classe abstraite), PieceConcrete, PieceOperateur(classe abstraite de décorateur), et ses sous-classes OperateurDeplacement, OperateurRotation, OperateurSymetrie, ainsi que d'autres éléments secondaires. Le but est de montrer les relations d'héritage, de composition ou d'agrégation entre ces entités.



La classe GameModel gère un vecteur de `shared_ptr<Piece>` représentant toutes les pièces présentes sur le plateau. Elle propose aussi des méthodes de niveau comme `initLevel1()`, `initLevel2()`, etc., et un mécanisme d'historique (`history`) pour permettre l'annulation.

La classe GameController tient une référence vers `GameModel` et interagit avec la souris (dans SFML) pour déterminer les actions à effectuer sur les pièces.

Piece est une classe abstraite dont PieceConcrete et PieceOperateur dérivent. PieceConcrete représente une vraie pièce (couleurs, coordonnées), tandis que

PieceOperateur représente un décorateur (pattern Décorateur) encapsulant un objet `Piece`.

Les sous-classes OperateurDeplacement, OperateurRotation, OperateurSymetrie héritent de PieceOperateur et permettent de transformer la position ou l'orientation de la pièce (ou d'autres opérateurs). Chaque opérateur applique son algorithme (ex. `mapPosition()` pour le déplacement ou la rotation) et propage la visite via un pattern Visiteur (méthodes `visit(PieceConcrete&)`, `visit(OperateurDeplacement&)`, etc.).

4. Problèmes Rencontrés et Solutions

Dans ce projet, nous avons rencontré plusieurs défis techniques et conceptuels qui ont nécessité des ajustements et des corrections. Voici un résumé des principaux problèmes et des solutions apportées :

Maintenir la cohérence lors de l'annulation (Undo) : Initialement, nous ne sauvegardions que les coordonnées d'une pièce dans l'historique. Cependant, une pièce ne se résume pas à sa position : son orientation et d'autres attributs comme *canRotate* ou *hasArrows* sont également essentiels. Nous avons donc créé une structure appelée *PieceState* qui inclut ces informations. Les fonctions *pushState()* et *popState()* gèrent désormais ces structures, garantissant ainsi une restauration complète et fidèle de l'état d'origine.

Éviter les collisions entre pièces : Avec le déplacement libre (cliquer-glisser), il arrivait que des pièces se superposent. Nous avons corrigé cela en modifiant la méthode *movePiece()* dans *GameModel*. Celle-ci vérifie maintenant si les nouvelles coordonnées sont valides (*walkable*) et s'il n'y a pas de collision. En cas de conflit détecté par la méthode *isColliding()*, le déplacement est annulé. Cela empêche efficacement les superpositions non autorisées.

Gestion des pivots lors des rotations : Les rotations utilisent un pivot (souvent la première coordonnée de la pièce). Cependant, lorsque des décorateurs de déplacement avec des pivots différents étaient appliqués, cela causait des incohérences. Nous avons utilisé le patron Visiteur pour que la rotation prenne en compte le décorateur et ajuste le pivot en conséquence. Ainsi, toutes les coordonnées intermédiaires restent alignées.

Ajout des flèches directionnelles : Pour indiquer les déplacements possibles, nous avons intégré des icônes de flèches autour du pivot ou aux extrémités de la pièce (par exemple, gauche/droite pour une orientation horizontale). Nous avons positionné ces flèches de façon à ne pas gêner la visibilité tout en rendant les clics plus intuitifs.

Ordre de destruction des objets : Initialement, le code contenait des vecteurs d'objets *Drawable* alloués de manière dynamique ou locale, ce qui provoquait des erreurs de double-destruction. Nous avons résolu ce problème en utilisant des *shared_ptr<Piece>* dans *GameModel*, permettant une gestion claire du cycle de vie des pièces et éliminant les risques de fuite ou de destruction multiple.

En conclusion, ces difficultés nous ont permis de rendre notre code plus robuste et clair. Grâce à des tests intensifs (déplacements extrêmes, rotations successives, annulations multiples), nous avons validé la fiabilité des solutions mises en œuvre

Conclusion :

En définitive, ce projet a atteint son objectif principal : proposer un jeu PieceOut, modulable et maintenable, tout en illustrant des principes importants de conception objet (MVC, Décorateur, Visiteur, gestion de l'historique, etc.). Les problèmes rencontrés, qu'il s'agisse de collisions, de gestion de pivot, de sauvegarde d'états ou de cohérence entre Vue et Modèle, ont été l'occasion d'implanter des solutions plus ou moins effectives. L'ensemble forme une base correcte.