Rapport Projet Java

Conception et développement du jeu Sokoban
Modélisation UML et implémentation en Java

Zakaria Bheddar
Mohamed Cheikh Rouhou
Groupe P

Deuxième Année Cycle d'Ingénieur
Electif : Génie logiciel orienté objet
Semestre 7

Janvier 2025

Responsables des TD :
Joanna Tomasik
Francesca Bugiotti

CentraleSupélec | université PARIS-SACLAY

Résumé

Ce projet consiste en la conception et le développement d'une version du jeu Sokoban, un puzzle classique où le joueur pousse des caisses vers des emplacements cibles dans un environnement rempli de défis. L'objectif est de créer une implémentation robuste et modulaire, en utilisant l'architecture Modèle-Vue-Contrôleur (MVC) pour séparer clairement la logique du jeu, l'interface utilisateur et la gestion des entrées. Le projet démarre par une version textuelle pour valider les mécaniques de base, suivie d'une interface graphique intuitive développée en Java. Des diagrammes UML, incluant des diagrammes de classes et de séquence, guideront la conception pour assurer une structure claire et maintenable. Des scénarios de test seront implémentés pour vérifier les fonctionnalités essentielles, telles que les déplacements du joueur, les interactions avec les caisses, et la validation des niveaux. Ce document détaille les spécifications techniques, les exigences fonctionnelles, et les étapes clés du développement, garantissant un produit final à la fois fonctionnel et évolutif.

Table des matières

1 Introduction

Sokoban [2] est un jeu de puzzle classique qui a captivé des millions de joueurs à travers le monde depuis sa création en 1982. Le nom "Sokoban" vient du japonais et signifie "gérant d'entrepôt", ce qui reflète parfaitement l'objectif du jeu. Le joueur incarne un personnage chargé de pousser des caisses dans un entrepôt pour les placer sur des emplacements spécifiques. Bien que le concept semble simple, les niveaux deviennent rapidement complexes et nécessitent une réflexion stratégique et une planification minutieuse.

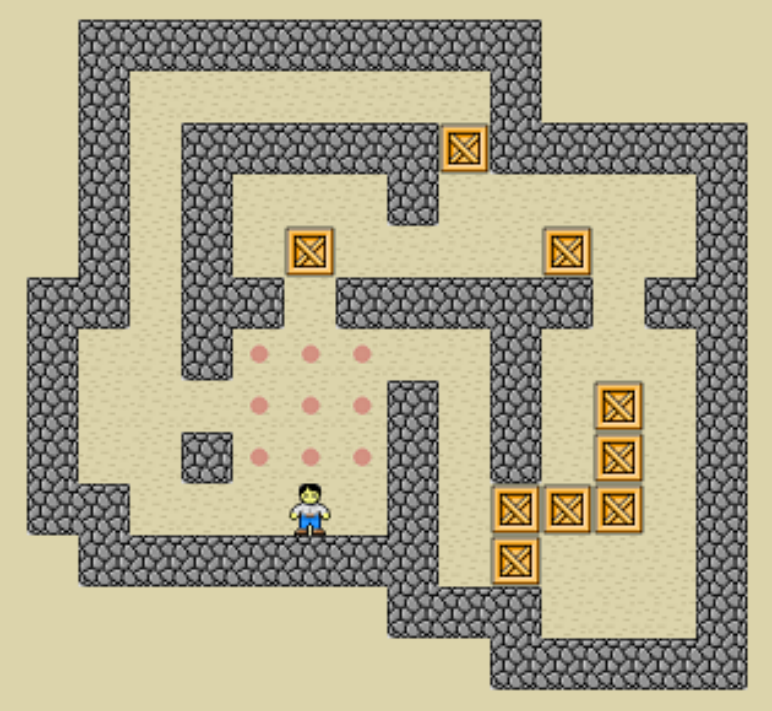

FIGURE 2 – Un exemple de niveau Sokoban. Le personnage doit pousser les caisses (représentées par des carrés) vers les emplacements cibles (marqués par des points).

Le jeu se déroule sur une grille en 2D, où le joueur peut se déplacer dans quatre directions : haut, bas, gauche et droite. Les caisses ne peuvent être poussées que dans une seule direction à la fois, et une fois qu'une caisse est contre un mur ou une autre caisse, elle ne peut plus être déplacée dans cette direction. Cela ajoute une couche de difficulté, car un mauvais mouvement peut bloquer une caisse et rendre le niveau insoluble, obligeant le joueur à recommencer.

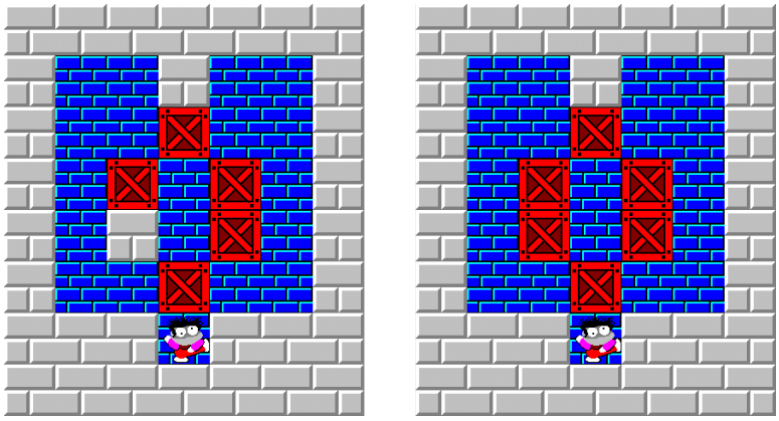

FIGURE 3 – Le personnage pousse une caisse vers un emplacement cible. Notez que les caisses ne peuvent pas être tirées, seulement poussées.

Sokoban est souvent considéré comme un excellent exercice pour le cerveau, car il encourage la pensée logique et la résolution de problèmes. Les niveaux varient en difficulté, allant de simples puzzles qui peuvent être résolus en quelques mouvements à des énigmes complexes qui nécessitent des dizaines, voire des centaines de mouvements pour être résolus.

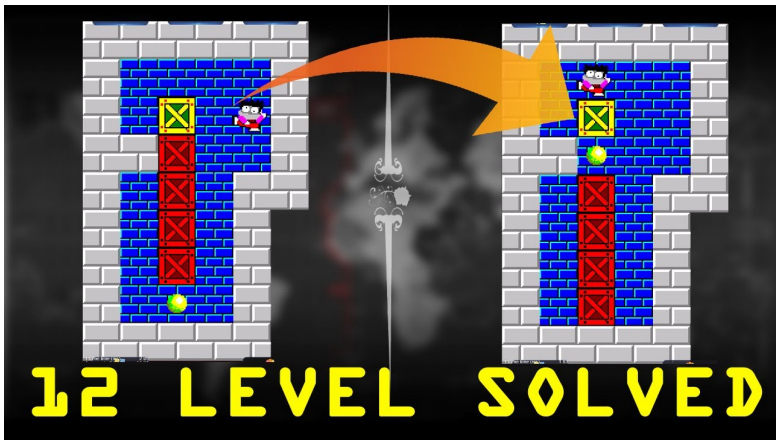

FIGURE 4 – Un niveau complexe de Sokoban nécessitant une planification minutieuse pour éviter de bloquer les caisses.

Avec son gameplay addictif et ses défis intellectuels, Sokoban reste un jeu intemporel qui continue d'inspirer de nouvelles générations de joueurs et de développeurs. Que vous soyez un novice cherchant à comprendre les bases ou un expert cherchant à maîtriser les niveaux les plus difficiles, Sokoban offre une expérience de jeu riche et gratifiante.

2 Cahier des charges

Ce document décrit les spécifications techniques et fonctionnelles pour la création d'une version du jeu Sokoban, incluant une interface graphique, une modélisation UML des classes, et une architecture respectant le design pattern MVC [1] (Modèle-Vue-Contrôleur).

2.1 Spécifications fonctionnelles

Le jeu doit respecter les règles suivantes concernant les objets et leurs interactions :

1. **Déplacements du joueur** :
   — Le joueur peut se déplacer dans les zones vides.
   — Le joueur ne peut pas traverser un mur ou une boîte.
2. **Interaction avec les boîtes** :
   — Le joueur doit être capable de pousser une boîte vers une zone vide.
   — Le joueur ne peut pas pousser deux boîtes en même temps.
3. **Validation des niveaux** :
   — Un niveau est réussi lorsque toutes les boîtes sont placées sur les emplacements cibles.

2.2 Scénarios de test pour le MVP

Pour valider un MVP (Minimum Viable Product) de notre projet, le modèle doit être capable d'exécuter les situations de jeu suivantes :

— Le joueur essaye d'aller à gauche et rencontre un mur (aucun déplacement n'est effectué).
— Le joueur se déplace à droite (déplacement valide).
— Le joueur se déplace une seconde fois à droite, ce qui provoque le déplacement d'une boîte.
— Le joueur tente de pousser deux boîtes en même temps (action impossible).

2.3 Version textuelle du MVP

Une version textuelle du jeu doit être développée en premier lieu pour valider la logique du jeu. Cette version doit inclure :

— Une représentation textuelle du niveau (grille avec des caractères pour le joueur, les murs, les boîtes, et les emplacements cibles).
— Des tests manuels pour vérifier les scénarios de test du MVP.
— Une sortie console indiquant l'état du jeu après chaque mouvement.

Exemple de représentation textuelle :

```
#######
#     #
#  P  #
#  B  #
#  X  #
#######
```

Où :
- `#` représente un mur.
- `P` représente le joueur.
- `B` représente une boîte.
- `X` représente un emplacement cible.

2.4 Interface graphique

L'interface graphique doit être intuitive et inclure les éléments suivants :
- Une grille représentant le niveau en cours.
- Des sprites pour le joueur, les boîtes, les murs, les emplacements cibles, et les zones vides.
- Un bouton pour réinitialiser le niveau en cours.

2.5 Modélisation UML

Le système doit être modélisé à l'aide de diagrammes UML, créés avec Modelio, incluant :
- Un diagramme de classes détaillant les relations entre les objets (joueur, boîtes, murs, etc.).
- Un diagramme de séquence pour illustrer les interactions entre les objets lors d'une partie.

2.6 Architecture MVC

Le système doit respecter l'architecture MVC (Modèle-Vue-Contrôleur) pour une séparation claire des responsabilités :
- **Modèle** : Gère la logique du jeu, y compris les règles de déplacement et l'état du niveau.
- **Vue** : Responsable de l'affichage graphique et de l'interface utilisateur.
- **Contrôleur** : Gère les entrées utilisateur (clavier, souris) et met à jour le modèle en conséquence.

2.7 Exigences techniques

- Langage de programmation : Java.
- Outil de modélisation : Modelio pour les diagrammes UML.
- Versioning : Utilisation de Git pour le contrôle de version.

2.8 Planning prévisionnel

- Conception des diagrammes UML et définition des classes.
- Implémentation de la logique du jeu (Modèle) et version textuelle.
- Développement de l'interface graphique (Vue).
- Intégration du contrôleur et tests finaux.

— Corrections de bugs et finalisation du projet.

3 Expression des besoins

Cette partie décrit le comportement attendu du logiciel et ses fonctionnalités du point de vue externe (point de vue utilisateur). Le besoin principal est de créer un jeu Sokoban fonctionnel, intuitif et esthétiquement plaisant, avec des fonctionnalités supplémentaires pour améliorer l'expérience utilisateur.

3.1 Fonctionnalités principales

- **Menu principal** :
  - Le joueur doit être accueilli par un menu principal convivial avec un bouton "Commencer" pour démarrer le jeu.
  - Le menu doit être visuellement attrayant, avec des animations ou des graphismes de qualité.
- **Déroulement du jeu** :
  - Le joueur commence à jouer les niveaux prédéfinis dans l'ordre.
  - Si le joueur réussit un niveau, il passe automatiquement au niveau suivant avec un message de félicitations.
  - Si le joueur est bloqué ou ne peut pas continuer, il doit avoir la possibilité de réinitialiser la carte actuelle pour recommencer.
- **Expérience utilisateur** :
  - Le jeu doit être visuellement beau, avec des spritesheets de haute qualité pour les éléments du jeu (joueur, boîtes, murs, etc.).
  - Les mouvements du joueur doivent être fluides et logiques, avec une réactivité immédiate aux commandes.
- **Niveaux personnalisés** :
  - Le joueur doit pouvoir ajouter des niveaux personnalisés en modifiant un fichier texte selon un format prédéfini.
  - Le jeu doit être capable de charger ces niveaux personnalisés et de les intégrer dans la liste des niveaux disponibles.
- **Script bash** :
  - Un script bash doit être fourni pour faciliter la compilation et le lancement du jeu.
  - Le script doit être simple à utiliser et documenté pour les utilisateurs non techniques.

3.2 Scénarios d'utilisation

- **Scénario 1 : Démarrage du jeu** :
  - L'utilisateur lance le jeu et est accueilli par un menu principal avec un bouton "Commencer".
  - L'utilisateur clique sur "Commencer" et le premier niveau se charge.
- **Scénario 2 : Réussite d'un niveau** :
  - L'utilisateur termine un niveau avec succès.
  - Un message de félicitations s'affiche, et le niveau suivant se charge automatiquement.

— **Scénario 3 : Réinitialisation d'un niveau** :
  — L'utilisateur est bloqué et ne peut pas terminer le niveau.
  — L'utilisateur utilise l'option "Réinitialiser" pour recommencer le niveau actuel.
— **Scénario 4 : Ajout de niveaux personnalisés** :
  — L'utilisateur modifie un fichier texte pour ajouter un niveau personnalisé.
  — Le jeu charge le niveau personnalisé et l'ajoute à la liste des niveaux disponibles.
— **Scénario 5 : Compilation et lancement du jeu** :
  — L'utilisateur exécute le script bash pour compiler et lancer le jeu.
  — Le jeu se lance sans erreur et le menu principal s'affiche.

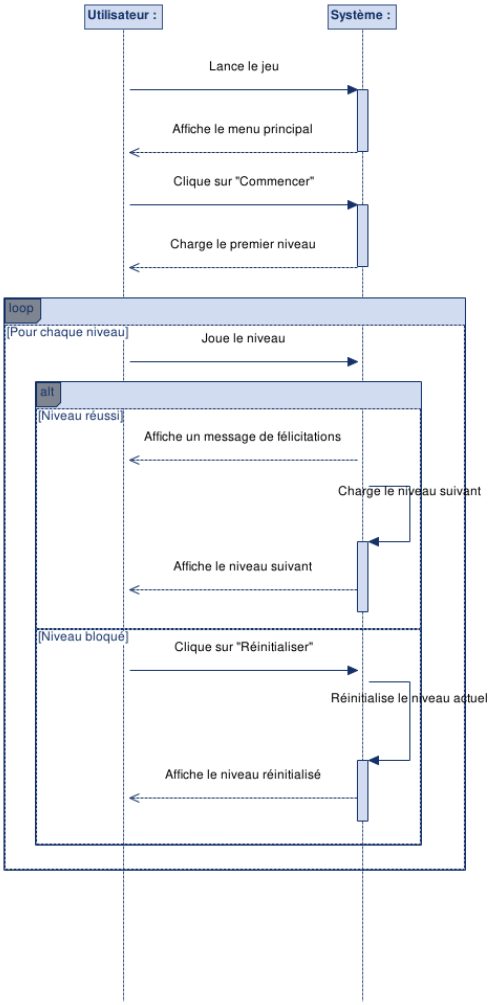

FIGURE 5 – Diagramme de séquence du jeu

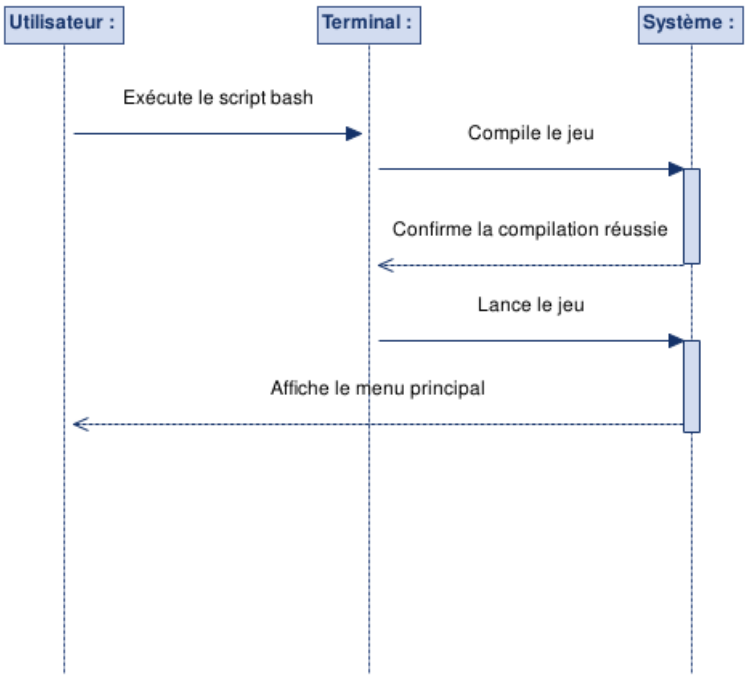

FIGURE 6 – Diagramme de séquence du script

Les diagrammes ci-dessus illustrent les interactions entre l'utilisateur et le système, ainsi que le déroulement des actions lors de l'exécution du script bash. Ces diagrammes permettent de visualiser les étapes clés du fonctionnement du jeu et du processus de compilation.

4 Architecture et conception

Notre conception repose sur une architecture **MVC (Modèle-Vue-Contrôleur)**, un modèle architectural largement utilisé pour séparer les responsabilités dans les applications logicielles. Cette architecture divise le système en trois composants principaux : le **Modèle**, la **Vue** et le **Contrôleur**. Cette séparation permet une meilleure organisation du code, une maintenance simplifiée et une extensibilité accrue.

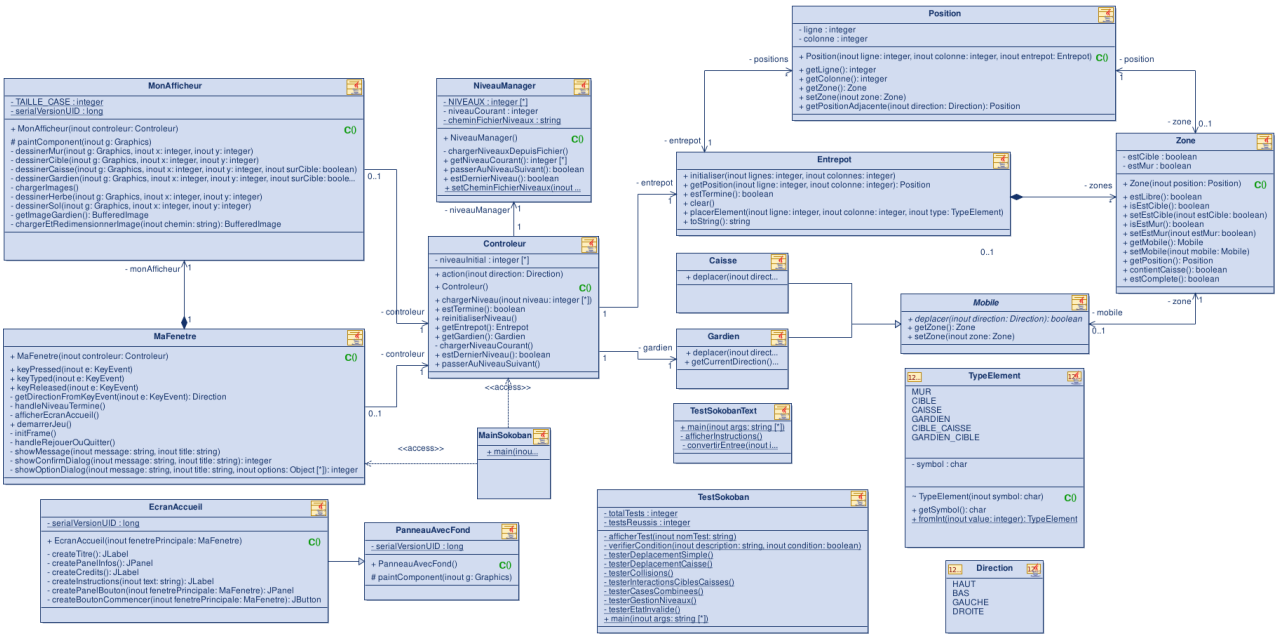

FIGURE 7 – Diagramme de classes illustrant l'architecture MVC du jeu

Dans les parties qui suivent, nous allons traiter chaque composant de cette architecture **partie par partie**, en détaillant son rôle, son implémentation et son interaction avec les autres éléments du système. Cette approche nous permettra de clarifier la structure globale du jeu et de montrer comment les différents modules travaillent ensemble pour offrir une expérience utilisateur fluide et cohérente.

4.1 Le Modèle

Nous nous sommes inspirés de la solution du TD en apportant des modifications à certaines relations et en ajoutant de nouveaux éléments, comme illustré dans la figure suivante :

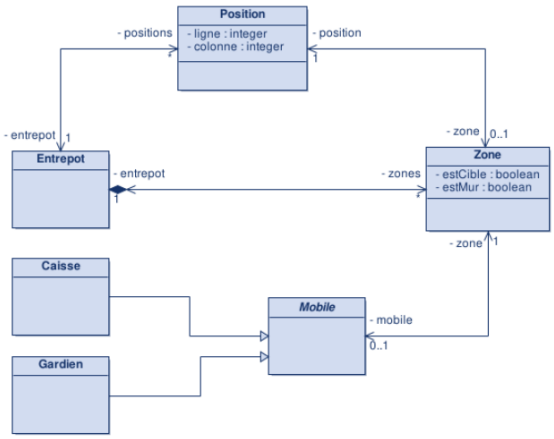

FIGURE 8 – Structure du modèle de TD avant modifications et ajouts

4.1.1 Les classes

Position Dans notre jeu, la classe Position a pour rôle de localiser les différentes entités sur l'espace de jeu.
Elle possède deux attributs principaux : la ligne et la colonne. Cela permet de déterminer les coordonnées précises des zones, des objets mobiles comme le gardien et les caisses.
De plus, cette classe permet aussi de calculer les coordonnées des zones adjacentes en fonction de la direction de déplacement souhaitée. Cela facilite la gestion des mouvements des éléments mobiles dans le jeu. En résumé, la classe Position est essentielle pour représenter la localisation des différentes composantes du jeu sur la grille de l'entrepôt.

Zone La classe Zone représente une position distincte dans l'entrepôt. Elle possède les attributs suivants :

- Position : les coordonnées de la zone
- estCible : indique si la zone est une cible
- estMur : indique si la zone est un mur

Chaque case peut contenir 0 ou 1 élément mobile. La classe Zone fournit les méthodes :

- contientCaisse() : détermine si la zone contient une caisse
- estComplete() : vérifie si la zone est une cible et contient une caisse indiquant l'achèvement du niveau

Mobile La classe Mobile représente un élément mobile, tel qu'un gardien ou une caisse, capable de se déplacer dans l'entrepôt. Cette classe permet de déplacer les éléments mobiles et de renvoyer la zone sur laquelle le mobile est situé.

Caisse+Gardien Les classes Caisse et Gardien héritent de la classe Mobile. Elles représentent respectivement les caisses à déplacer et le gardien chargé de les manipuler. Ces classes bénéficient des méthodes héritées de la classe mère. La classe Gardien contient notamment une méthode permettant de détecter sa direction, ce qui est particulièrement utile pour l'interface graphique.

Entrepôt La classe Entrepôt représente l'état global de l'entrepôt, incluant ses positions, ses zones et ses éléments. Elle dispose d'attributs tels que la liste des positions (positions) et la liste des zones (zones). Parmi ses méthodes principales, on trouve : initialiser, qui configure l'entrepôt avec une grille de dimensions données ; getPosition, qui retourne la position aux coordonnées spécifiées ; estTermine, qui vérifie si le jeu est terminé, c'est-à-dire si toutes les caisses sont sur les cibles ; clear, qui réinitialise l'état de l'entrepôt ; placeElement, qui permet de placer un type d'élément spécifique à une position donnée ; et toString, qui génère une représentation textuelle de l'entrepôt.

4.1.2 Vue d'ensemble

Le modèle est la partie centrale du jeu Sokoban. Il est conçu en suivant une approche orientée objet, où chaque classe a une responsabilité précise. Cette structure permet une séparation claire des tâches et facilite la maintenance du code. Un diagramme de classes illustre cette architecture (voir Figure 9).

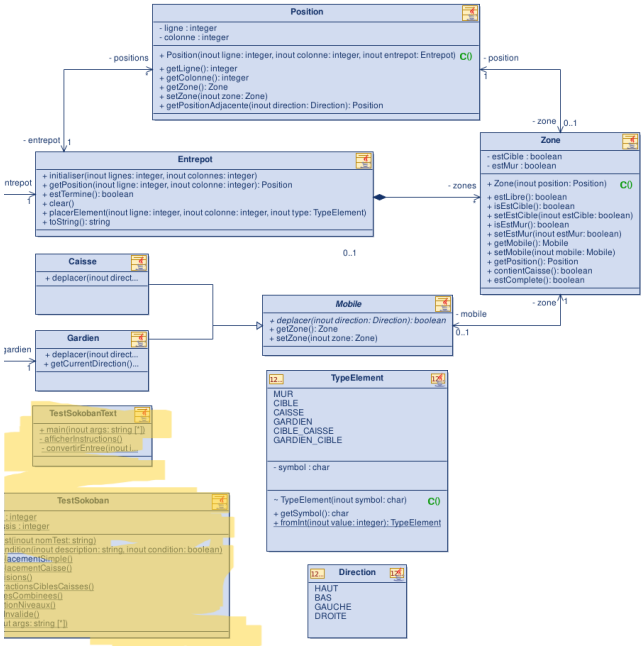

FIGURE 9 – Diagramme de classes du modèle

4.1.3 Mécanismes de jeu

Déplacement du gardien Le déplacement du gardien est illustré par le diagramme de séquence suivant :

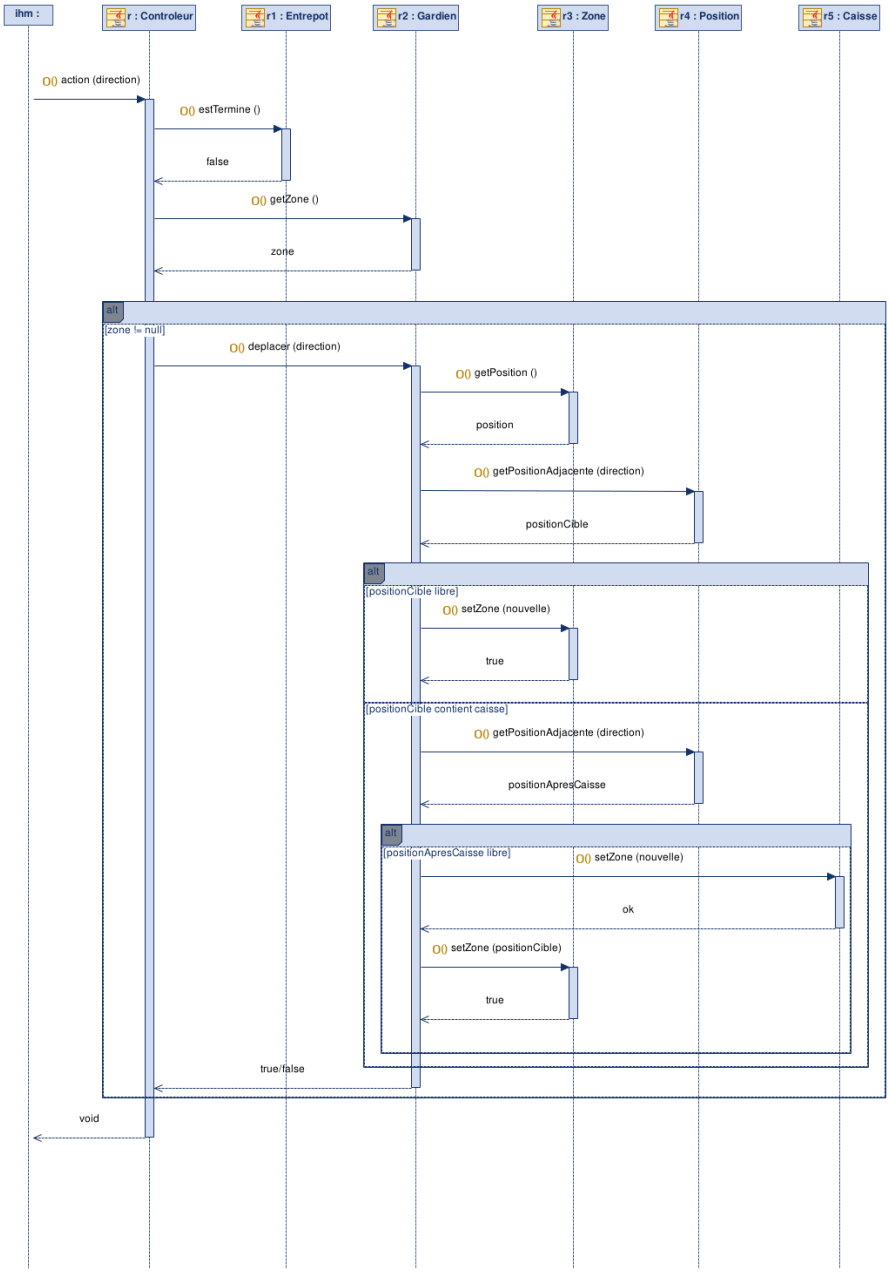

FIGURE 10 – Diagramme de séquence du mouvement du gardien

Le gardien suit un processus précis pour se déplacer. Tout d'abord, il reçoit une direction de déplacement. Ensuite, il vérifie si la position cible est libre. Si cette position contient une

caisse, il vérifie également si la position derrière la caisse est libre. Si toutes les conditions sont remplies, le gardien se déplace et pousse éventuellement la caisse. Ce mécanisme garantit que les règles du jeu sont respectées et que les déplacements sont fluides.

Déplacement d'une caisse Le déplacement d'une caisse est illustré par le diagramme de séquence suivant :

FIGURE 11 – Diagramme de séquence du mouvement d'une caisse

Le déplacement d'une caisse est déclenché lorsque le gardien pousse celle-ci. La caisse vérifie d'abord si la position cible est libre. Si cette position est occupée par une autre caisse, le système

vérifie si la position suivante est libre. Si les conditions sont satisfaites, la caisse se déplace vers la nouvelle position. Ce processus assure que les caisses ne se bloquent pas mutuellement et que les déplacements restent cohérents.

Calcul de la position adjacente Le calcul de la position adjacente est illustré par le diagramme de séquence suivant :

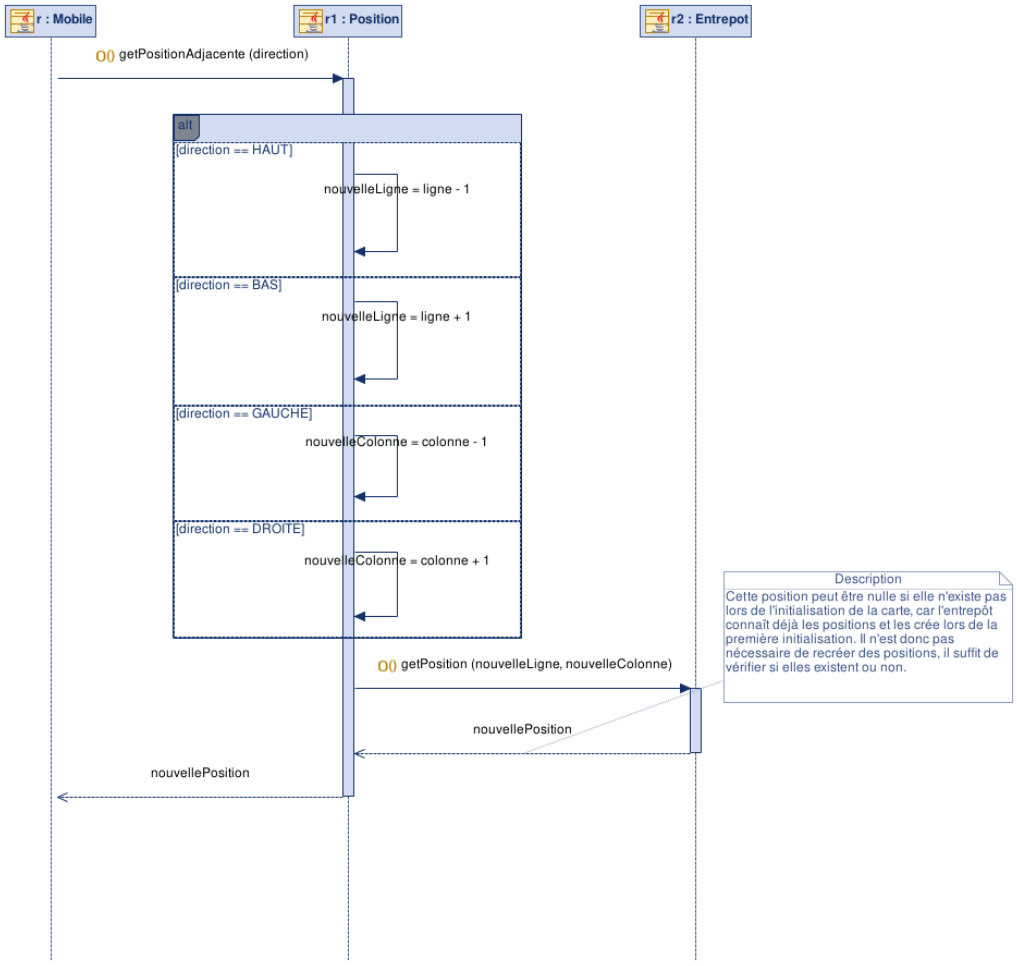

FIGURE 12 – Diagramme de séquence du calcul de la position adjacente

Le calcul de la position adjacente est une opération fondamentale pour les déplacements. En fonction de la direction donnée (haut, bas, gauche, droite), le système calcule les nouvelles coordonnées. Si la nouvelle position existe dans l'entrepôt, elle est retournée ; sinon, la position est considérée comme invalide. Ce mécanisme permet de déterminer rapidement les positions valides pour les déplacements, tout en évitant les erreurs liées aux positions hors limites.

4.1.4 Avantages du modèle

Cette architecture offre plusieurs avantages. La simplicité des classes permet une compréhension facile du code. La séparation des responsabilités facilite la maintenance et l'ajout de nouvelles fonctionnalités. Enfin, l'optimisation des opérations de déplacement et de vérification garantit une exécution fluide et efficace du jeu.

4.2 Le Contrôleur

4.2.1 Vue d'ensemble

Le contrôleur est le composant central qui gère les interactions entre l'utilisateur et le modèle. Il reçoit les entrées de l'utilisateur (comme les déplacements du gardien) et met à jour l'état du jeu en conséquence. Il agit comme un intermédiaire entre l'interface utilisateur et le modèle, garantissant que les règles du jeu sont respectées.

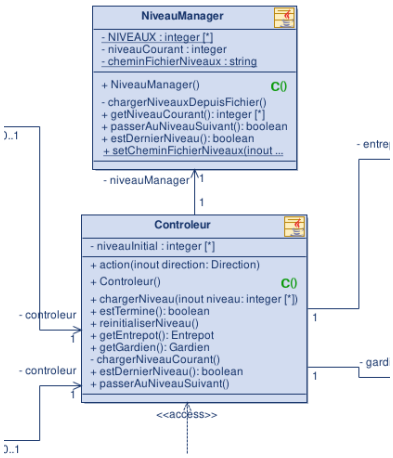

FIGURE 13 – Diagramme de classes du contrôleur

4.2.2 Fonctionnalités principales

Le contrôleur remplit plusieurs rôles clés :
— Il gère les déplacements du gardien en fonction des entrées de l'utilisateur.
— Il charge et initialise les niveaux du jeu.
— Il vérifie si un niveau est terminé et permet de passer au niveau suivant.
— Il offre la possibilité de réinitialiser un niveau en cas d'erreur.

4.2.3 Structure et fonctionnement

Le contrôleur est composé de plusieurs éléments essentiels :
— `Entrepot` : Représente le plateau de jeu et gère les éléments (caisses, murs, cibles, etc.).

— `Gardien` : Représente le personnage principal que l'utilisateur contrôle.
— `NiveauManager` : Gère les niveaux du jeu, leur chargement et leur progression.

Gestion des déplacements Lorsque l'utilisateur donne une direction (haut, bas, gauche, droite), le contrôleur vérifie si le déplacement est possible. Si la position cible est libre, le gardien se déplace. Si la position cible contient une caisse, le contrôleur vérifie si la caisse peut être poussée. Ce mécanisme garantit que les règles du jeu sont respectées.

Chargement des niveaux Le contrôleur charge les niveaux à partir d'un fichier texte. Chaque niveau est représenté par une grille de nombres, où chaque nombre correspond à un élément du jeu (mur, caisse, cible, etc.). Le contrôleur initialise l'entrepôt et place les éléments aux positions appropriées.

Réinitialisation et progression Le contrôleur permet de réinitialiser un niveau en cas d'erreur ou de blocage. Il gère également la progression entre les niveaux, en vérifiant si le niveau actuel est terminé et en passant au niveau suivant si nécessaire.

4.2.4 Avantages du contrôleur

Le contrôleur offre plusieurs avantages :
— **Modularité** : Il sépare clairement la logique de contrôle de la logique du modèle.
— **Flexibilité** : Il permet de charger et de gérer plusieurs niveaux de manière dynamique.
— **Maintenabilité** : Le code est structuré et facile à comprendre, ce qui facilite les modifications futures.

5 Version Console : MVP et Scénarios de Test

Nous souhaitons suivre un ordre logique en fonction de l'avancement de notre travail dans le projet. Ainsi, après avoir conçu le contrôleur et le modèle, l'étape suivante consiste à implémenter le jeu sous forme d'interface textuelle. Cette approche nous permet d'avoir un aperçu concret de l'avancement et de valider le travail accompli.

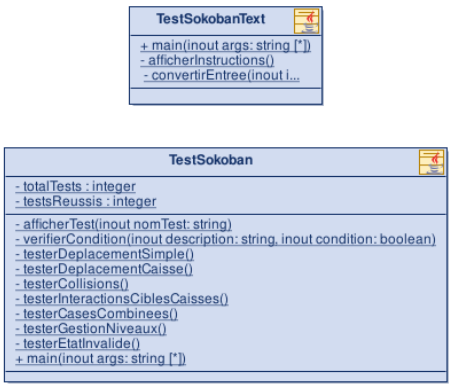

FIGURE 14 – Partie test

Cependant, avant de procéder à cette implémentation, nous avons réalisé des tests sans utiliser JUnit. Ces tests ont pour objectif de valider la logique du jeu et de s'assurer que son fonctionnement est correct.

5.1 Tests réalisés

5.1.1 Introduction aux tests

Pour valider le fonctionnement du jeu, nous avons implémenté une classe de test appelée `TestSokoban`. Cette classe contient une série de méthodes qui vérifient les différentes fonctionnalités du jeu. Chaque test est conçu pour vérifier une condition spécifique et affiche un message de succès ou d'échec en fonction du résultat. Les tests sont exécutés dans la méthode `main`, et un résumé est affiché à la fin.

5.1.2 Test de déplacement simple du gardien

Ce test vérifie que le gardien peut se déplacer dans les quatre directions (haut, bas, gauche, droite) sur une carte vide. Une grille de test est créée avec le gardien placé au centre. Le test vérifie que chaque déplacement est effectué correctement en appelant la méthode `deplacer` du gardien. Si le déplacement réussit, le test est marqué comme réussi.

5.1.3 Test de déplacement de caisse

Ce test vérifie que le gardien peut pousser une caisse si la zone derrière elle est libre. Une grille de test est créée avec une caisse placée à côté du gardien. Le test vérifie également que le gardien ne peut pas pousser une caisse si elle est bloquée par un mur ou une autre caisse. Cela valide les interactions entre le gardien et les caisses.

5.1.4 Test des collisions

Ce test vérifie que le gardien ne peut pas traverser les murs ou sortir des limites de la carte. Une grille de test est créée avec des murs et des bords, et le test vérifie que le gardien ne peut pas se déplacer dans ces cas. Cela valide la gestion des collisions.

5.1.5 Test des interactions cibles et caisses

Ce test vérifie que les caisses peuvent être poussées sur les cibles et que le niveau est marqué comme terminé lorsque toutes les caisses sont sur les cibles. Une grille de test est créée avec des cibles et des caisses, et le test vérifie que le niveau est correctement marqué comme terminé. Cela valide la logique de victoire.

5.1.6 Test des cases combinées

Ce test vérifie que le gardien peut se déplacer sur une cible et qu'une caisse peut être poussée depuis une cible. Une grille de test est créée avec des cases combinées (cible avec caisse), et le test vérifie que les déplacements sont correctement gérés. Cela valide les interactions complexes entre les éléments.

5.1.7 Test de gestion des niveaux

Ce test vérifie que les niveaux peuvent être chargés, réinitialisés et passés au niveau suivant. Une grille de test est créée, et le test vérifie que l'état du niveau est correctement réinitialisé et que le passage au niveau suivant fonctionne. Cela valide la gestion des niveaux.

5.1.8 Test des états invalides

Ce test vérifie que le jeu gère correctement les cas limites, tels qu'un niveau vide ou un niveau null. Une grille de test minimale est créée, et le test vérifie que le jeu ne plante pas dans ces cas. Cela valide la robustesse du code.

5.1.9 Résultats des tests

À la fin de l'exécution des tests, un résumé est affiché, indiquant le nombre de tests réussis et échoués. Voici un exemple de sortie des tests :

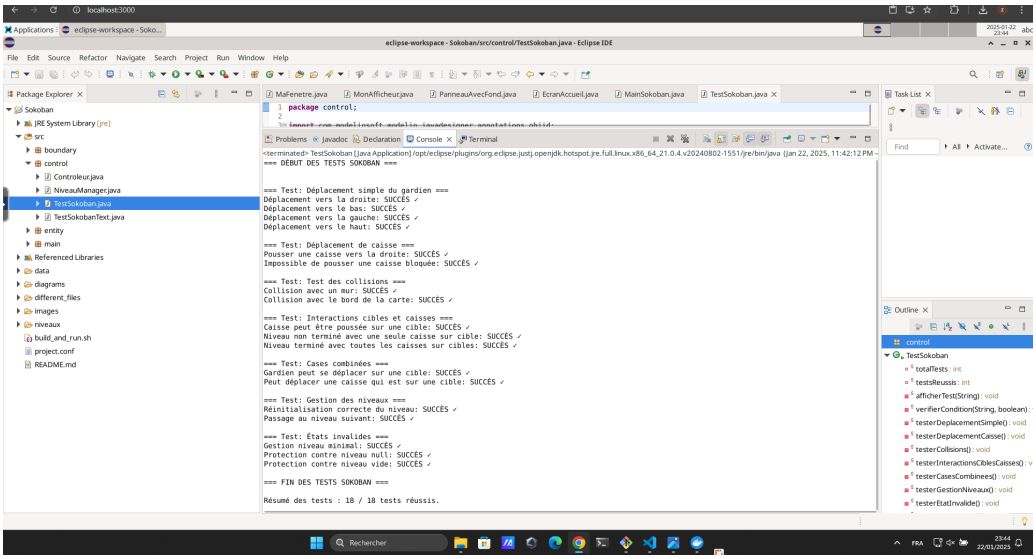

FIGURE 15 – Résultats des tests

Tous les tests ont été exécutés avec succès, confirmant que la logique du jeu fonctionne comme prévu. Chaque test a validé une fonctionnalité spécifique, garantissant ainsi la robustesse et la fiabilité du système. Ce résultat démontre que notre implémentation respecte les règles du jeu Sokoban et est prête pour une utilisation en production.

5.2 Version Console

La version console du jeu Sokoban permet de valider les mécanismes de base du jeu en utilisant une interface textuelle. Cette version est implémentée en Java et repose sur une boucle principale qui gère les entrées de l'utilisateur et met à jour l'état du jeu en conséquence. Voici une description détaillée de son fonctionnement.

5.2.1 Représentation des éléments du jeu

Les éléments du jeu (mur, caisse, cible, etc.) sont représentés à l'aide de symboles ASCII pour l'affichage dans la console. Par exemple :

— Le mur est représenté par le symbole `#`.
— La cible est représentée par le symbole `.`.
— La caisse est représentée par le symbole `$`.
— Le gardien (joueur) est représenté par le symbole `@`.
— Une caisse sur une cible est représentée par le symbole `*`.
— Le gardien sur une cible est représenté par le symbole `+`.

Ces symboles sont définis dans une énumération appelée `TypeElement`, qui associe chaque type d'élément à un caractère spécifique. Cette énumération permet de convertir facilement les valeurs numériques du niveau en symboles visuels.

5.2.2 Affichage de la grille

La grille du jeu est affichée dans la console à l'aide de la méthode `toString()` de la classe `Entrepot`. Cette méthode parcourt toutes les positions de la grille et détermine le symbole à afficher en fonction de l'élément présent à chaque position. Par exemple :

- Si une position contient un mur, le symbole `#` est affiché.
- Si une position contient une caisse sur une cible, le symbole `*` est affiché.
- Si une position est vide, un espace est affiché.

5.2.3 Fonctionnement de la boucle de jeu

La boucle de jeu fonctionne comme suit :

1. Le plateau de jeu est affiché dans la console.
2. L'utilisateur entre une commande (mouvement, recommencer, quitter).
3. La commande est convertie en direction (haut, bas, gauche, droite) ou en action (recommencer, quitter).
4. Le contrôleur met à jour l'état du jeu en fonction de la commande.
5. Si le niveau est terminé, un message de félicitations est affiché.

5.2.4 Instructions pour l'utilisateur

Au démarrage du jeu, les instructions suivantes sont affichées à l'utilisateur :

- `Z` : Déplacer le gardien vers le haut.
- `S` : Déplacer le gardien vers le bas.
- `Q` : Déplacer le gardien vers la gauche.
- `D` : Déplacer le gardien vers la droite.
- `R` : Recommencer le niveau.
- `X` : Quitter le jeu.

5.2.5 Exemple de niveau

Un niveau est représenté par une grille 2D, où chaque case correspond à un élément du jeu. Par exemple :

```
#########
#   $  .#
# @     #
#########
```

Dans cet exemple :

- `#` représente un mur.
- `$` représente une caisse.
- `@` représente le gardien.
- `.` représente une cible.

5.2.6 Exemple du jeu dans la console

Voici un exemple de ce à quoi ressemble le jeu dans la console. L'utilisateur interagit avec le jeu en entrant des commandes, et la grille est mise à jour en temps réel.

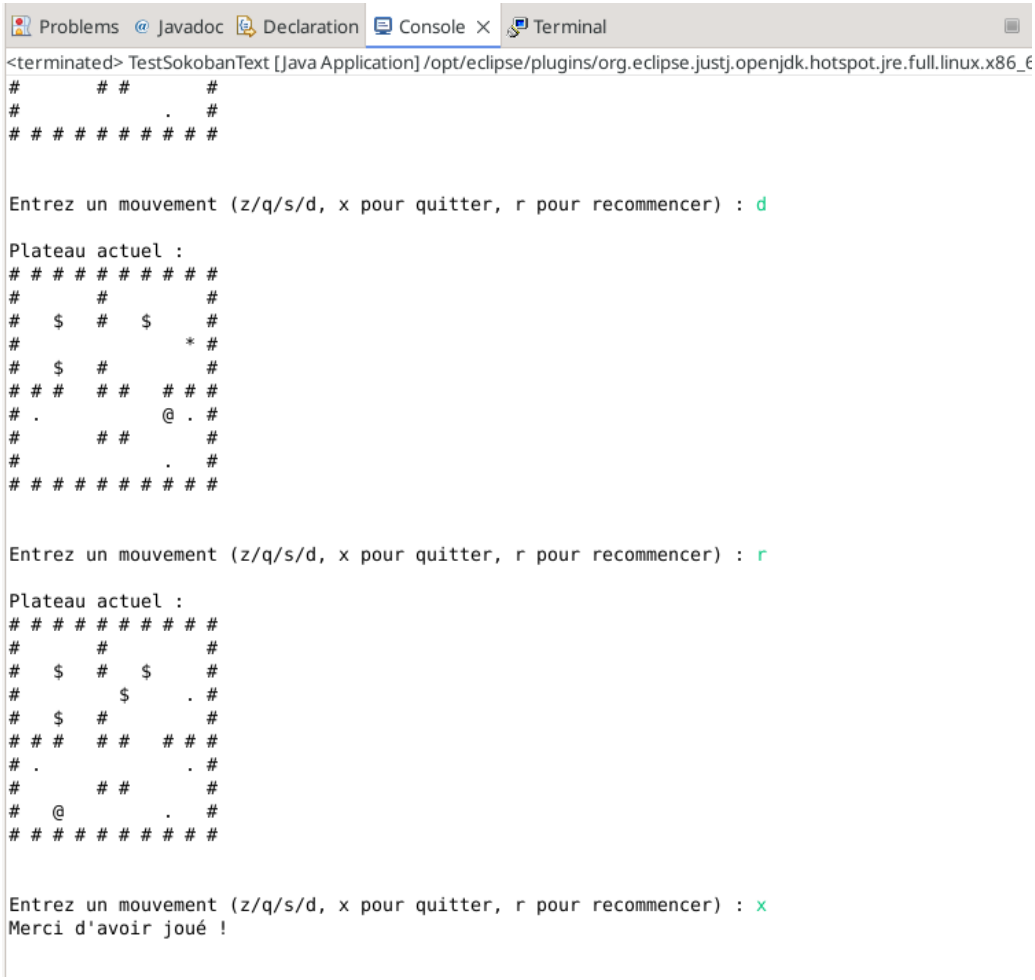

FIGURE 16 – Exemple du jeu Sokoban dans la console

Dans cet exemple :
- Le gardien (`@`) se déplace pour pousser les caisses (`$`) vers les cibles (`.`).
- Lorsqu'une caisse est sur une cible, elle est représentée par le symbole `*`.
- Les murs (`#`) délimitent les limites du niveau.

5.2.7 Fin du jeu

Lorsque l'utilisateur décide de quitter ou termine un niveau, un message de remerciement est affiché :

```
Merci d'avoir joué !
```

6 Interface Graphique et Expérience Utilisateur

À ce stade, après avoir vérifié que le modèle fonctionne correctement (via des tests ou en utilisant le jeu dans sa version initiale), il est temps d'implémenter la version graphique. Cette étape consiste à développer une interface visuelle, dont la structure est illustrée dans le diagramme de classes suivant.

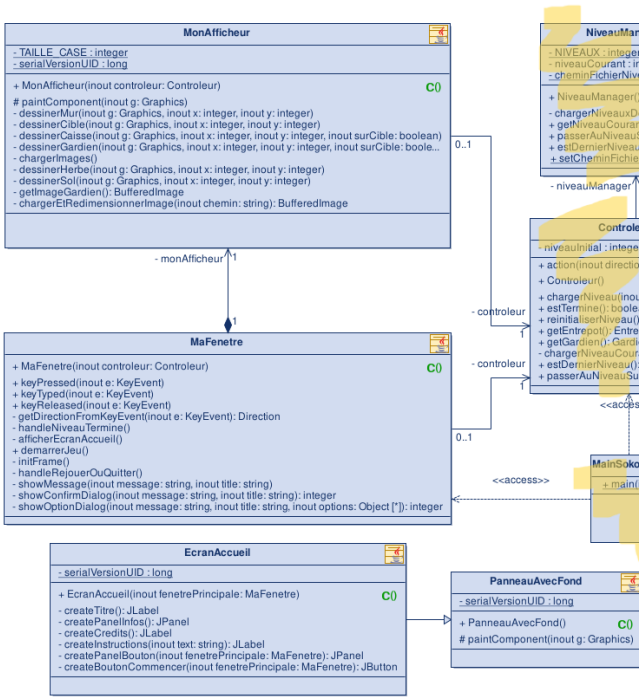

FIGURE 17 – Diagramme de classes représentant la structure de la version graphique.

6.1 Images de Tous les Éléments du Jeu

Pour assurer une cohérence visuelle et une immersion optimale, nous avons conçu des images distinctes pour chaque élément du jeu. Ces éléments incluent le joueur, le gardien, les objets interactifs et les décors. Chaque élément possède plusieurs images pour représenter ses différentes orientations ou états. Par exemple, le joueur dispose de quatre images correspondant aux directions haut, bas, gauche et droite, comme illustré dans la Figure 18. Cette approche permet de créer un univers visuel riche et dynamique.

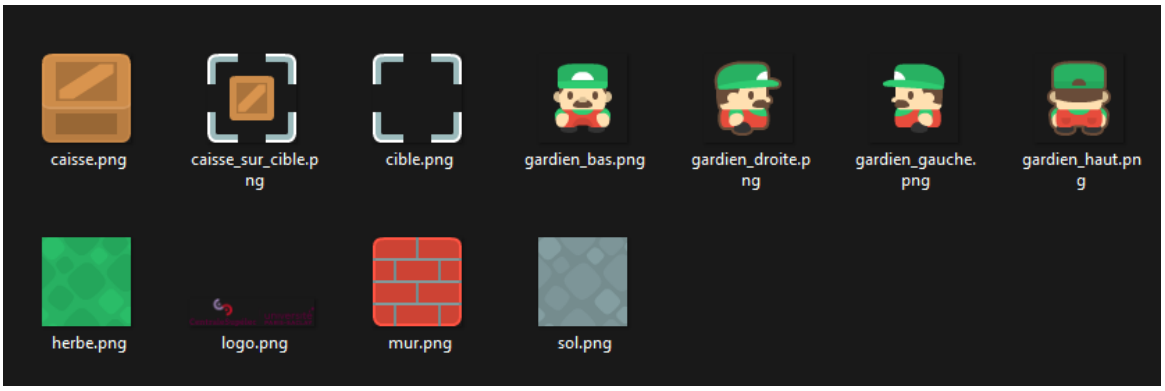

FIGURE 18 – Images de tous les éléments du jeu, y compris le joueur, le gardien et les objets.

6.2 Méthodes Essentielles

Voici les méthodes les plus importantes pour la gestion de l'interface graphique et des interactions utilisateur :

Méthode `chargerImages()`

Charge toutes les images du jeu (murs, sol, caisses, cibles, gardien) dès le démarrage. Assure que tout est prêt pour l'affichage.

Méthode `paintComponent()`

Dessine tous les éléments du jeu à l'écran. Met à jour l'affichage en temps réel après chaque action (mouvement du joueur, déplacement du gardien, etc.).

Méthode `getImageGardien()`

Retourne l'image du gardien en fonction de sa direction (haut, bas, gauche, droite). Permet un affichage réaliste des mouvements.

Méthode `keyPressed()`

Gère les entrées clavier. Détecte les touches pressées (flèches directionnelles, touche "R" pour réinitialiser) et met à jour le jeu en conséquence.

Méthode `handleNiveauTermine()`

Gère la fin d'un niveau. Propose de passer au niveau suivant, de rejouer ou de quitter le jeu. Affiche un message de félicitations si tous les niveaux sont terminés.

Méthode `createBoutonCommencer()`

Crée le bouton "Commencer" de l'écran d'accueil. Configure son apparence et son comportement pour démarrer le jeu.

7 Produit Final et Script d'Exécution

7.1 Script d'Exécution

Nous travaillons dans un environnement où l'un d'entre nous utilise un conteneur Docker, tandis que l'autre travaille directement sur le site de CS. Parfois, nous souhaitons voir le jeu fonctionner sur nos machines locales. Pour faciliter cela, nous avons créé un script Bash qui permet de supprimer les décorateurs et les éléments spécifiques à Modelio, de compiler le jeu, de l'exécuter, et d'enregistrer les fichiers dans un autre dossier. Ce script est particulièrement utile lorsque nous travaillons avec VSCode pour des tests.

Voici le script Bash :

```
#!/bin/bash

# Repertoires
WORKING_DIR="different_files"     # Repertoire principal pour les operations
SRC_DIR="src"                     # Repertoire original contenant les sources
CLEAN_DIR="$WORKING_DIR/src"  # Repertoire nettoye
BIN_DIR="$WORKING_DIR/bin"        # Repertoire des fichiers compiles

# Creation et nettoyage des repertoires
echo "Creation du repertoire de travail '$WORKING_DIR'..."
rm -rf "$WORKING_DIR"             # Supprimer le dossier precedent s'il existe
mkdir -p "$CLEAN_DIR" "$BIN_DIR" # Creer les repertoires necessaires

# Copier et nettoyer les fichiers Java
echo "Copie et nettoyage des fichiers Java dans '$CLEAN_DIR'..."
cp -r "$SRC_DIR"/* "$CLEAN_DIR"/
find "$CLEAN_DIR" -name "*.java" -exec sed -i '/import com\.modeliosoft\.modelio\.javadesigner\.annotations\.objid;/d' {} \;
find "$CLEAN_DIR" -name "*.java" -exec sed -i '/@objid/d' {} \;

# Compilation des fichiers nettoyes
echo "Compilation des fichiers Java dans '$BIN_DIR'..."
find "$CLEAN_DIR" -name "*.java" > "$WORKING_DIR/sources.txt"
javac -d "$BIN_DIR" @"$WORKING_DIR/sources.txt"

# Verification de la compilation
if [ $? -eq 0 ]; then
    echo "Compilation reussie ! Les fichiers compiles sont dans '$BIN_DIR'."
```

```
else
    echo "Erreur de compilation. Verifiez vos fichiers."
    exit 1
fi

# Lancement de l'application
MAIN_CLASS="main.MainSokoban"  # Remplace par le package complet de ta classe
echo "Lancement de l'application..."
java -cp "$BIN_DIR" "$MAIN_CLASS"

# Nettoyage des fichiers temporaires
echo "Nettoyage des fichiers temporaires..."
rm "$WORKING_DIR/sources.txt" # Supprime le fichier temporaire sources.txt
```

Listing 1 – Script Bash pour compiler et exécuter le jeu

Pour exécuter ce script, il suffit de lui accorder les permissions d'exécution avec la commande `chmod +x build_and_run.sh`, puis de l'exécuter avec `./build_and_run.sh`. Assurez-vous d'avoir Bash installé sur votre système.

7.2 Présentation du jeu

Le jeu Sokoban que nous avons développé offre une expérience utilisateur fluide et immersive. Voici les principales fonctionnalités, illustrées par des captures d'écran :

Écran d'accueil

L'écran d'accueil, illustré dans la Figure 19, présente un bouton "Commencer" pour démarrer le jeu. Il inclut également des instructions claires et les crédits des développeurs.

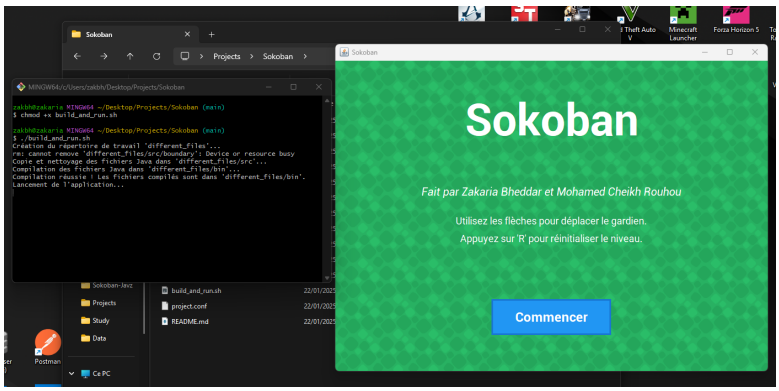

FIGURE 19 – Écran d'accueil du jeu Sokoban.

Interface graphique

La Figure 20 montre l'interface graphique du jeu, conçue pour être simple et intuitive. Elle inclut les éléments visuels tels que les murs, les caisses, les cibles et le gardien.

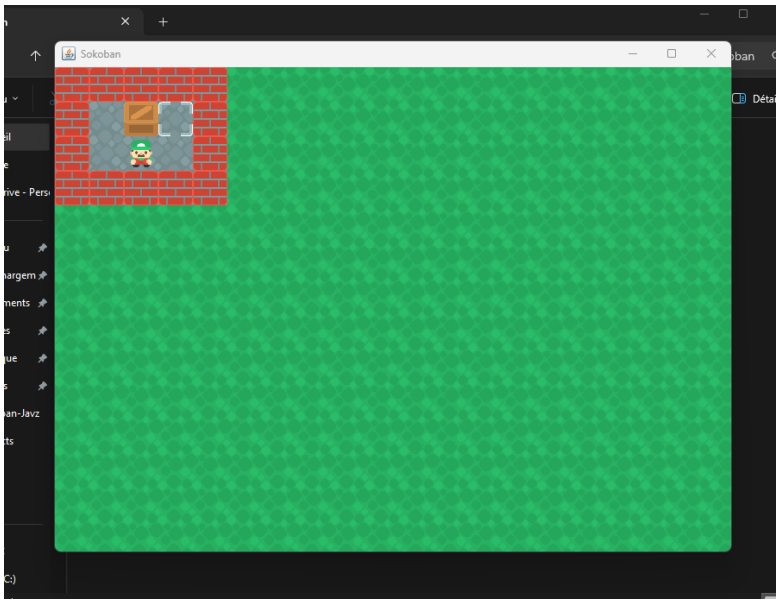

FIGURE 20 – Interface graphique du jeu Sokoban.

Fin d'un niveau

Lorsqu'un niveau est terminé, l'utilisateur a la possibilité de passer au niveau suivant ou de rejouer le niveau actuel. Cette interface est illustrée dans la Figure 21.

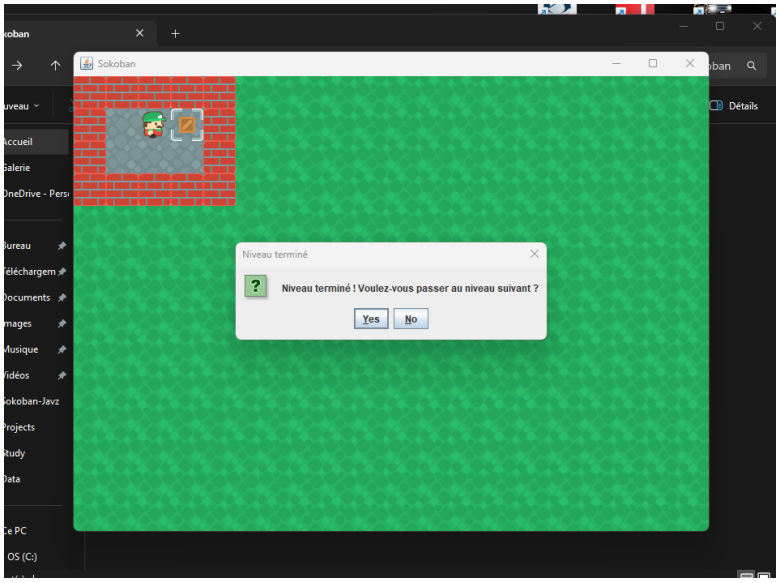

FIGURE 21 – Fin d'un niveau avec option pour passer au niveau suivant.

Choix après la fin d'un niveau

Si l'utilisateur choisit de ne pas passer au niveau suivant, il peut décider de rejouer le niveau ou de quitter le jeu. La Figure 22 montre cette interface.

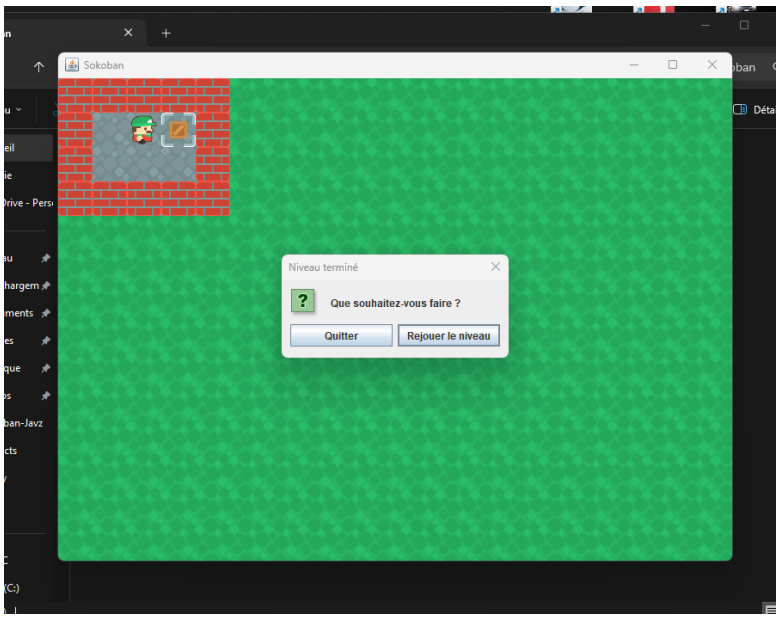

FIGURE 22 – Choix après la fin d'un niveau : rejouer ou quitter.

Fin du jeu

Lorsque tous les niveaux sont terminés, un message de félicitations est affiché, comme le montre la Figure 23.

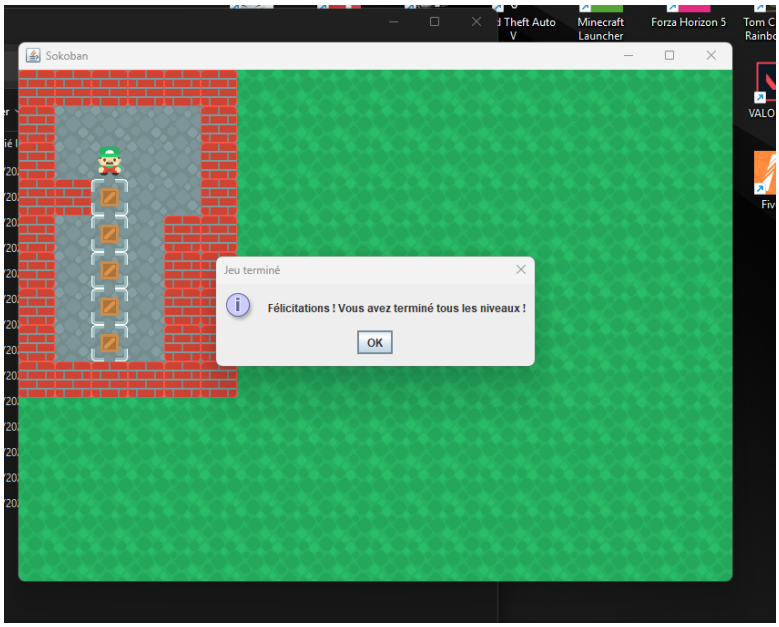

FIGURE 23 – Message de félicitations après avoir terminé tous les niveaux.

Exécution du jeu

La Figure 24 montre l'exécution du jeu via un script shell, illustrant le processus de démarrage et d'exécution du programme.

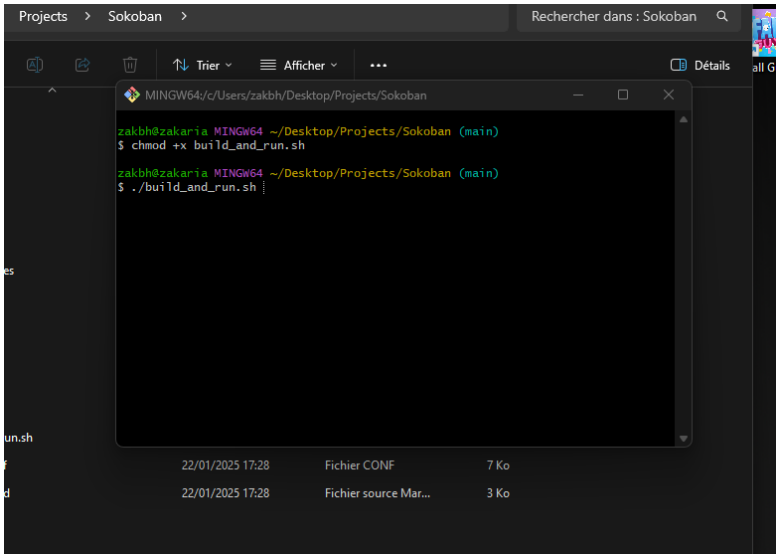

FIGURE 24 – Exécution du jeu via un script shell.

8 Charge de travail répartie entre les membres du groupe

La répartition des tâches au sein du groupe a été organisée de manière équilibrée pour assurer une collaboration efficace. Voici un aperçu détaillé de la répartition des tâches :

Membre	Tâches réalisées
Zakaria Bheddar	- Conception globale de l'architecture du jeu. - Implémentation des classes de base (Mobile, Caisse, Gardien). - Gestion des positions et des zones (Position, Zone). - Définition des types d'éléments (TypeElement) et des directions (Direction). - Implémentation du contrôleur principal (Controleur). - Gestion des déplacements, collisions et conditions de victoire. - Chargement des niveaux depuis un fichier (NiveauManager). - Conversion des niveaux en grille de jeu (Entrepot).
Mohamed Cheikh Rouhou	- Implémentation de l'écran d'accueil (EcranAccueil). - Gestion des interactions utilisateur (boutons, écouteurs d'événements). - Affichage dynamique des éléments du jeu (MonAfficheur). - Chargement et gestion des images (sprites). - Intégration des animations et des effets visuels. - Gestion des transitions entre les écrans. - Écriture des tests unitaires pour les déplacements et les interactions. - Écriture des tests pour l'interface graphique et les interactions utilisateur.

TABLE 1 – Répartition des tâches entre les membres du groupe

9 Dépôt GitHub du Projet

Notre projet est hébergé sur GitHub, une plateforme de gestion de versions et de collaboration. Vous pouvez accéder au dépôt à l'adresse suivante : `https://github.com/zikous/Sokoban`

Ce lien vous permet de consulter l'ensemble du code source, de la documentation, et des ressources associées au projet. N'hésitez pas à explorer et à contribuer si vous le souhaitez !

10 Conclusion

Ce projet nous a permis d'acquérir et de consolider des compétences essentielles en développement logiciel. En découvrant Java, nous avons renforcé notre compréhension de la programmation orientée objet et mis en pratique des concepts clés tels que l'encapsulation, l'héritage et le polymorphisme. L'utilisation d'outils de modélisation comme Modelio nous a également appris à structurer et concevoir efficacement une application, en adoptant une approche méthodique et rigoureuse.

Bien que notre jeu soit pleinement fonctionnel, plusieurs améliorations pourraient enrichir l'expérience utilisateur et la complexité du gameplay dans de futures versions :

- **Amélioration du menu et de la gestion des niveaux :** Un menu plus ergonomique avec un système de déblocage progressif des niveaux, ainsi qu'un panneau permettant aux joueurs de sélectionner les niveaux déjà terminés.
- **Ajout d'un timer :** Un chronomètre permettrait de mesurer les performances des joueurs, ajoutant ainsi un aspect compétitif et encourageant l'optimisation des stratégies.
- **Système de retour en arrière et d'annulation des mouvements :** Plutôt qu'un simple reset complet du niveau, permettre aux joueurs d'annuler quelques actions récentes pour ajuster leur approche sans devoir recommencer depuis le début.
- **Niveaux dynamiques :** L'intégration de plateformes mobiles ou de murs mouvants offrirait plus de variété et de défi.
- **Obstacles interactifs :** L'ajout d'objets manipulables, comme des caisses explosives ou des éléments déclenchant des réactions en chaîne, permettrait de diversifier les mécaniques du jeu.

Ces perspectives d'évolution ouvrent la voie à un perfectionnement continu du projet, nous encourageant à approfondir nos compétences en développement et à explorer de nouvelles approches en conception logicielle.

Références

[1] GeeksforGeeks. Mvc design pattern. https://www.geeksforgeeks.org/mvc-design-pattern/.

[2] Wikipédia. Sokoban. https://fr.wikipedia.org/wiki/Sokoban.