CentraleSupélec

université PARIS-SACLAY

Rapport de projet

Génie logiciel orienté objet

Réalisation d'une application permettant de jouer à un jeu de type Sokoban

Réalisé par le groupe AV :
Virgile Rapegno
Amaury Blanpain

16 octobre 2023

Table des matières

Chapitre 1

Introduction

1.1 Contexte : Le jeu du *Sokoban*

Le *Sokoban* est un jeu de réflexion originaire du Japon. Le nom *Sokoban* signifie "gardien d'entrepôt" en japonais. Le jeu se déroule dans un entrepôt représenté par un plateau composé de cases, dont certaines sont bloquées par des murs. Le joueur contrôle un personnage et doit déplacer des caisses sur le plateau afin que toutes se trouvent sur un rangement. Il convient de planifier soigneusement l'ordre de ses déplacements, car le joueur peut facilement se retrouver piégé par ses propres mouvements, par exemple s'il pousse une caisse dans un coin sans pouvoir la déplacer à nouveau. Ainsi, le but du jeu est de pousser toutes les caisses sur les emplacements désignés tout en évitant d'être bloqué.

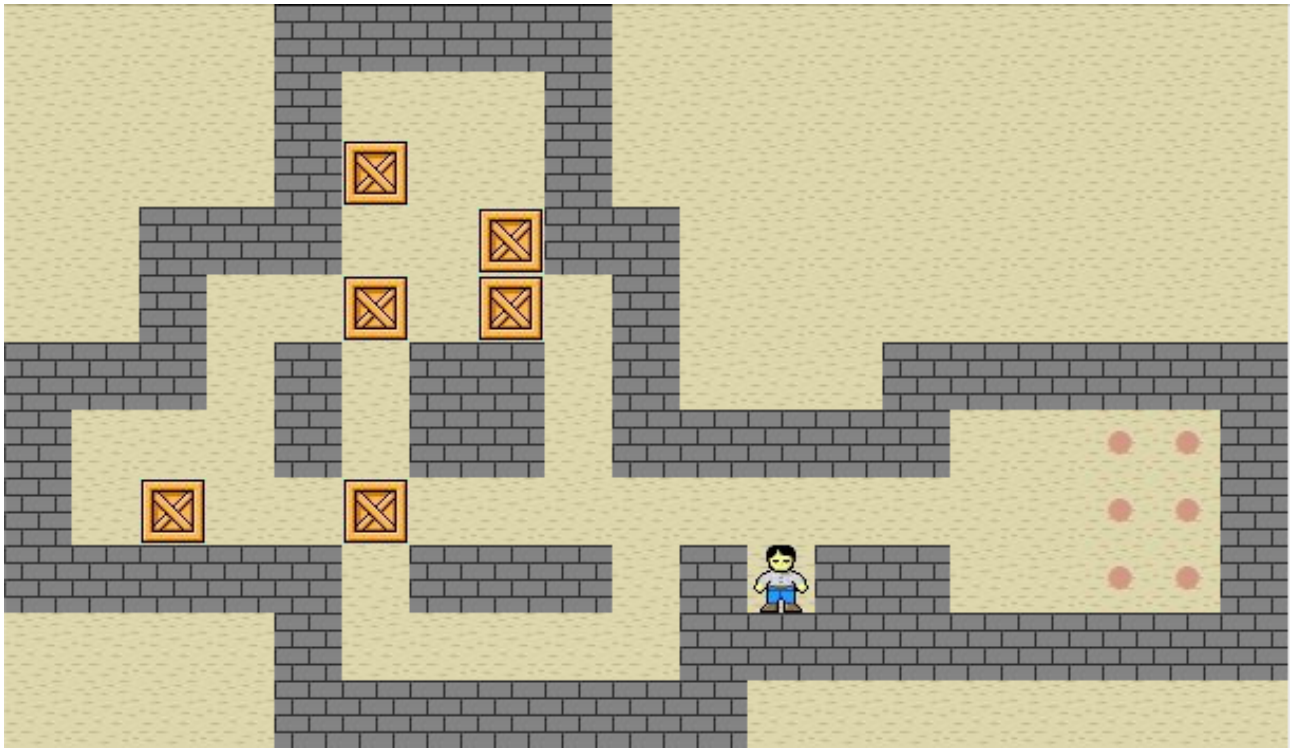


FIGURE 1.1 – Le premier niveau du *Sokoban* original, publié en 1982

1.2 Structure du rapport

Après cette rapide introduction, nous aborderons dans ce document les différentes étapes de réalisation de l'implémentation avec une interface graphique du jeu *Sokoban* en Java. Pour ce faire, nous commencerons dans le chapitre 2 par le cahier des charges que doit remplir le logiciel, puis l'expression des besoins dans le chapitre 3. Nous enchaînerons ensuite sur la phase de conception dans le chapitre 4 et sur la réalisation du projet dans le chapitre 5. Nous conclurons dans le chapitre 6.

Une archive des projets Modelio et Eclipse est également fournie avec le rapport, via Edunao.

Chapitre 2

Cahier des charges

Un jeu *Sokoban* typique présente les caractéristiques suivantes :

- Plateau de jeu : Le jeu se joue sur une grille rectangulaire ou carrée de cases, dont la taille est généralement comprise entre 5×5 et 20×20.
- Murs : Certaines des cases du plateau de jeu sont désignées comme des murs, que le joueur et les caisses ne peuvent pas traverser. En particulier, l'entrepôt est un espace fermé par des murs, afin que le joueur ne s'échappe pas du plateau.
- Joueur : L'utilisateur contrôle un personnage (appelé joueur par la suite) qui peut être déplacé sur le plateau de jeu à l'aide des flèches ou des touches ZQSD. Il y a un unique joueur contrôlable sur le plateau. Il a la capacité de pousser des caisses, mais en aucun cas celle de les tirer.
- Caisses : Le joueur doit pousser les caisses présentes sur le plateau de jeu jusqu'à un rangement. Les caisses peuvent être poussées mais pas tirées, et elles ne peuvent être déplacées qu'une par une (pousser une caisse contre une autre revient à la pousser contre un mur).
- Rangement : Chaque niveau comporte une ou plusieurs cases désignés sur le plateau de jeu, sur lesquelles le joueur doit pousser les caisses. Le joueur et les caisses peuvent simplement passer dessus.
- Objectif : L'objectif du jeu est de placer toutes les caisses sur les rangements, tout en évitant de se faire piéger ou de bloquer le mouvement du personnage.
- Difficulté : Elle varie d'un niveau à l'autre, certains seront simples et d'autres complexes, nécessitant plus de logique et de planification. Un nombre de coups maximum peut aussi intervenir dans certaines versions du jeu.

Certaines variantes du jeu incluent des fonctionnalités supplémentaires telles que différents types de caisses et rangements, la possibilité de contrôler plus d'un joueur, ou encore la possibilité d'annuler une action pour revenir en arrière.

Il convient de noter qu'il existe différentes implémentations du jeu, et que tous les jeux n'ont pas les mêmes caractéristiques, néanmoins les caractéristiques de ce cahier des charges sont suffisantes pour un *Sokoban* classique.

Chapitre 3

Expression des besoins

3.1 Lancer une partie de *Sokoban*

Afin de profiter du jeu, il suffit de l'importer dans un IDE capable d'utiliser Java 17. Puis pour jouer à chaque niveau il faudra :

- Lancer dans l'IDE le projet, puisque celui-ci contient une unique fonction *main*.
- L'utilisateur est invité à saisir un nom de niveau dans la console de l'IDE, il a le choix entre les niveaux originaux avec niveau1 à niveau10 ou les niveaux de tests avec test1 à test9 (attention à bien coller le numéro).
- Une fenêtre graphique s'ouvre après la saisie. L'entrepôt se dévoile et l'utilisateur peut dès lors contrôler le joueur et pousser les cases vers leurs rangements.
- Pour le contrôle, il est possible d'utiliser les flèches directionnelles du claviers ainsi que les touches ZQSD.
- Le joueur peut se déplacer sur toutes les cases vides et les rangements, mais il sera bloqué par les murs. Il peut se mettre à côté d'une caisse et essayer d'aller dans sa direction pour la pousser. Si la case après la caisse dans la direction désirée est une case vide ou un rangement, alors la caisse y ira et le joueur prendra l'ancienne case de la caisse. Si ce n'est pas le cas, donc si le joueur essaye de pousser la caisse vers un mur ou une autre caisse, alors l'action est impossible et rien ne se passe.

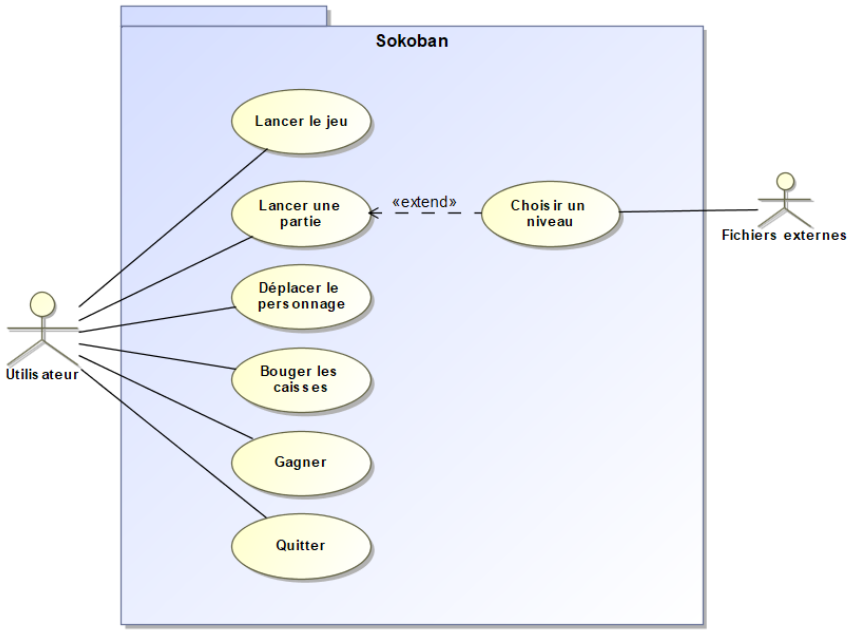


FIGURE 3.1 – Diagramme de cas d'utilisation

3.2 Les éléments du jeu

Il y a tout d'abord les éléments fixes de l'entrepôt. Le mur est impénétrable tandis que les cases vides et les rangements peuvent accueillir le joueur ou une caisse.

FIGURE 3.2 – Les éléments fixes du plateau : mur, case vide, rangement

Il y a ensuite les éléments mobiles. Le joueur est le seul élément contrôlable par l'utilisateur tandis que les caisses sont poussées par l'interaction du joueur. Une caisse rangée change de couleur pour indiquer visuellement sa progression à l'utilisateur.

FIGURE 3.3 – Les éléments mobiles du plateau : joueur, caisse, caisse rangée

Enfin voici les situations où le joueur ne peut pas déplacer une caisse.

FIGURE 3.4 – Situations où une caisse est bloquée

Chapitre 4

Conception

4.1 Choix effectués

L'implémentation utilise l'IMH fournie par *Dominique Marcadet*[1], qui a pu être adaptée par endroits.

Le code respecte un découpage en Modèle-Vue-Contrôleur, et les principes de la programmation orientée objet (attributs privés, getters, setters et héritage notamment).

Seules les règles classiques du *Sokoban* ont été ajoutées, mais le code a été rédigé afin de permettre facilement l'implémentation de nouvelles fonctionnalités, notamment l'ajout de nouveaux type de cases ou de modification des comportements des objets mobiles (voir les perspectives :6.2).

4.2 Diagramme de classes du modèle métier

Pour ce qui de l'implémentation, on propose ce choix de diagramme de classes :

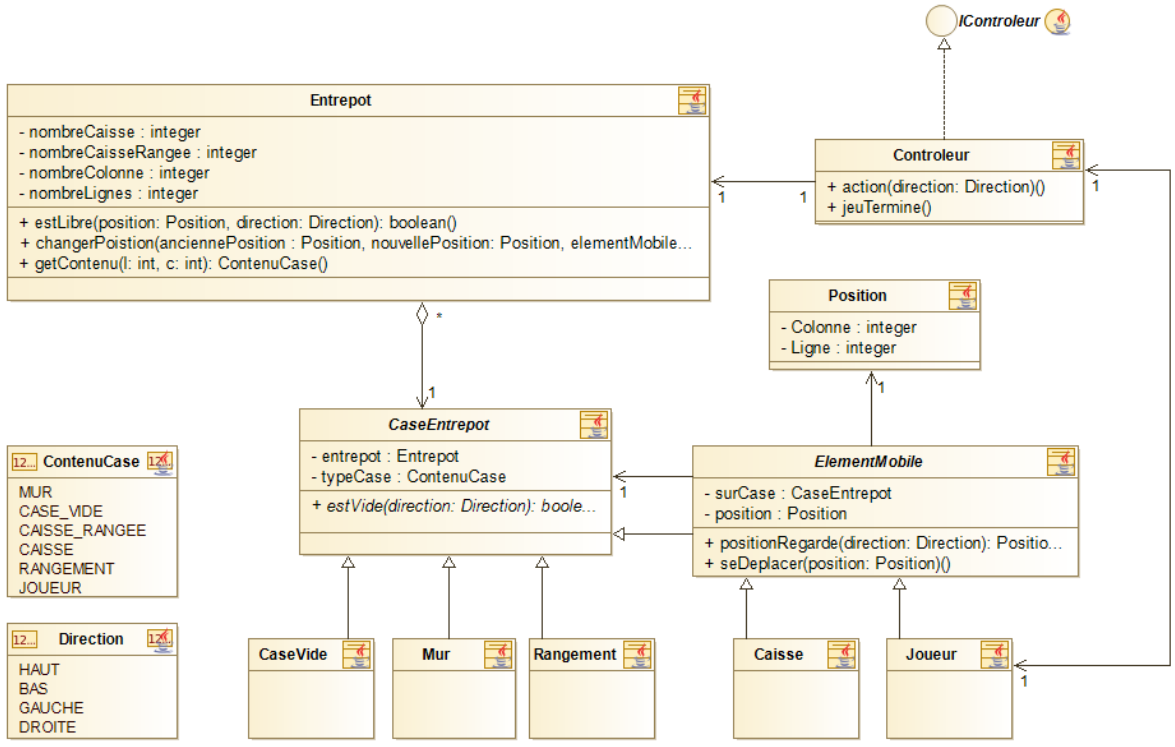


FIGURE 4.1 – Diagramme de classes (les getters et setters ne sont pas présentés)

Ainsi le contrôleur interagit avec un `Entrepot` qui est une agrégation de `CaseEntrepot`. Une case en particulier se distingue, la case `Joueur`. Il y a une utilisation de deux classes abstraites (`CaseEntrepot` et `ElementMobile`) afin de regrouper au mieux des fonctionnalités communes des cases puis des éléments mobiles.

1. L'IHM est disponible ici `https://wdi.centralesupelec.fr/2EL1520FR/Projet2022`

4.3 Description des classes

4.3.1 Entrepot

Un `Entrepot` est principalement une agrégation de `CaseEntrepot` stockées dans une matrice. Des informations sur l'état de l'`Entrepot` nécessaires au bon fonctionnement du `Controleur` sont aussi accessibles.

Puisque `Entrepot` dispose de l'ensemble des éléments du plateau, c'est à lui que revient la tâche d'indiquer le contenu d'une case pour l'IHM et si une case est libre pour les `ElementMobile`.

Lorsqu'un `ElementMobile` souhaite se déplacer, l'`Entrepot` est invité à mettre à jour sa matrice en cohérence avec l'action désirée.

L'`Entrepot` se construit en lisant la map passée par le `Controleur`. Pour ce faire, il la découpe en lignes puis remplit sa matrice en fonction des caractères présents.

4.3.2 CaseEntrepot, CaseVide, Mur et Rangement

Une `CaseEntreport` est une classe abstraite contenant `ContenuCase` et indiquant la dépendance avec `Entrepot` avec un attribut référençant celui-ci.

Ses sous-classes directes sont `CaseVide`, `Mur` et `Rangement`. Ces sous-classes sont utilisées pour faciliter la création de cases, en renseignant directement le `ContenuCase` désiré et le retour de la méthode `estLibre`.

4.3.3 ElementMobile, Caisse et Joueur

Un `ElementMobile` est une sous classes abstraite de `CaseVide`. Son intérêt est encore de regrouper des méthodes communes, ici pour `Caisse` et `Joueur`.

En effet ces deux sous-classes ont un comportement quasi-identique, si ce n'est que le `Joueur` agit à la suite d'une `action` envoyée par le `Controleur`, quand une `Caisse` agis à la suite d'une demande envoyée par le `Joueur`.

On retrouve dans le classe `ElementMobile` en particulier la méthode `positionRegardee` qui retourne la `Position` un attribut pour accéder rapidement aux coordonnées (on choisit de dupliquer cette information par rapport à la matrice de `Entrepot` pour plus de facilité) dans la `Direction` où l'élément souhaite aller. Il y a aussi l'ajout de l'attribut `surCase` de type `CaseEntrepot` car un `ElementMobile` se superpose à un élément fixe. Enfin il faut savoir si l'élément est sur un rangement, pour adapter le `ContenuCase` d'une `Caisse` et avoir le bon affichage.

4.3.4 Position

La classe `Position` se rapproche plus d'un type ou même d'un tuple, elle permet de stocker les coordonnées de façon naïve en attribut pour chaque `ElementMobile`.

4.4 Diagrammes de séquence

4.4.1 Déplacement sur une case bloquées

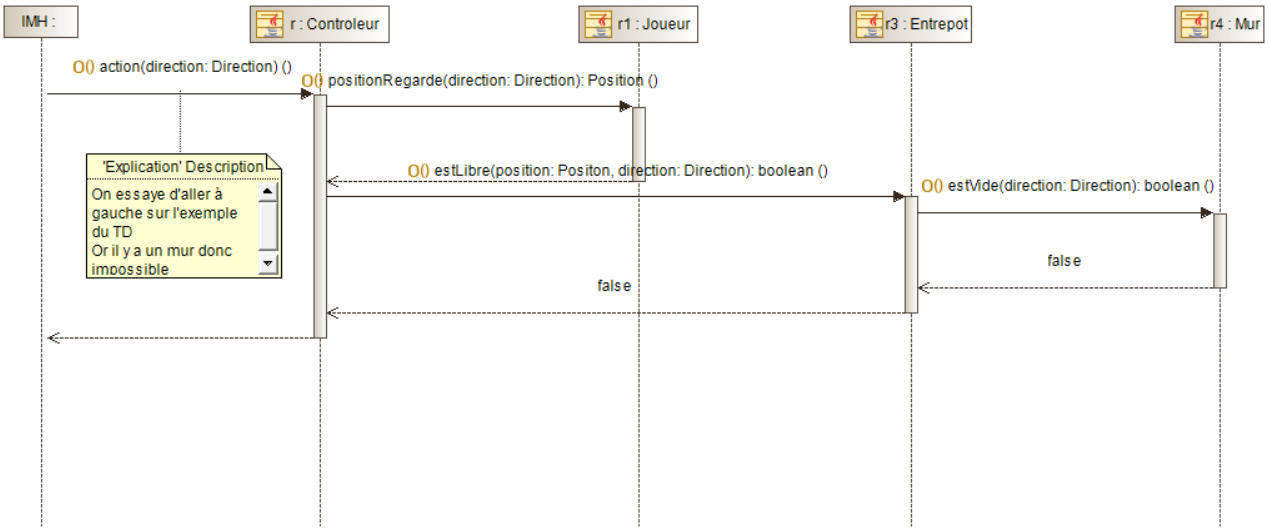


FIGURE 4.2 – Diagramme de séquence : déplacement sur une case bloquée

L'IHM est le premier intervenant dans la réalisation d'un déplacement car c'est elle qui surveille les entrées du clavier et qui va ordonner l'appel d'`action` du `Controleur`. La première étape est de demander au `Joueur` sur quelle case il arrivera, puis à `Entrepot` si cette case est libre.

Dans ce premier cas 4.2, il y a un mur qui n'est donc pas une case libre, le joueur ne peut pas y aller et aucune autre action n'est entreprise.

4.4.2 Déplacement sur une case vide

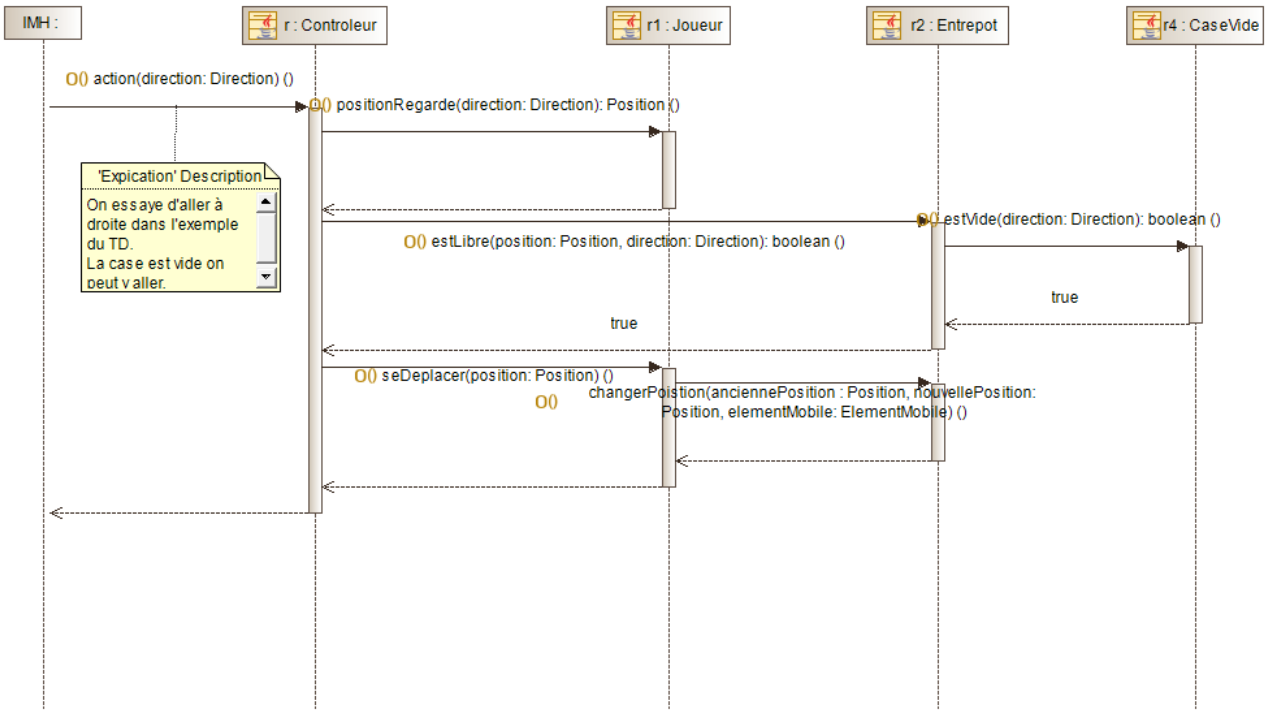


FIGURE 4.3 – Diagramme de séquence : déplacement sur une case vide

Dans ce second cas 4.3, la case où souhaite arriver le `Joueur` est libre. Le `Controleur` va ordonner au `Joueur` de se déplacer vers cette case. Il commence par modifier la valeur de sa position, puis demande à `Entrepot` de le déplacer. `Entrepot` va donc modifier sa matrice interne, et vérifier si le joueur arrive sur un `Rangement` afin de ne pas perdre l'information.

4.4.3 Pousser une caisse

Dans cet exemple 4.4, il y a une succession d'étapes supplémentaires afin de déplacer la `Caisse` puis le `Joueur`. Lorsque `Entrepot` demande si la case est vide, `Caisse` va essayer de se déplacer dans la bonne direction avant de répondre. Il faut donc recommencer le même dialogue avec `Entrepot`, à savoir vers quelle case la `Caisse` souhaite se déplacer, si cette case est libre, et enfin réaliser le déplacement de la `Caisse` en remettant proprement un `Rangement` ou une `CaseVide` mais surtout en mettant à jour la valeur de `surCase` et donc le `ContenuCase` associé pour l'affichage.

Ici il est possible de pousser la `Caisse` donc ce déplacement est effectué, puis le `Joueur` a l'indication que la case est libre et on retourne sur un déplacement classique vers une case vide.

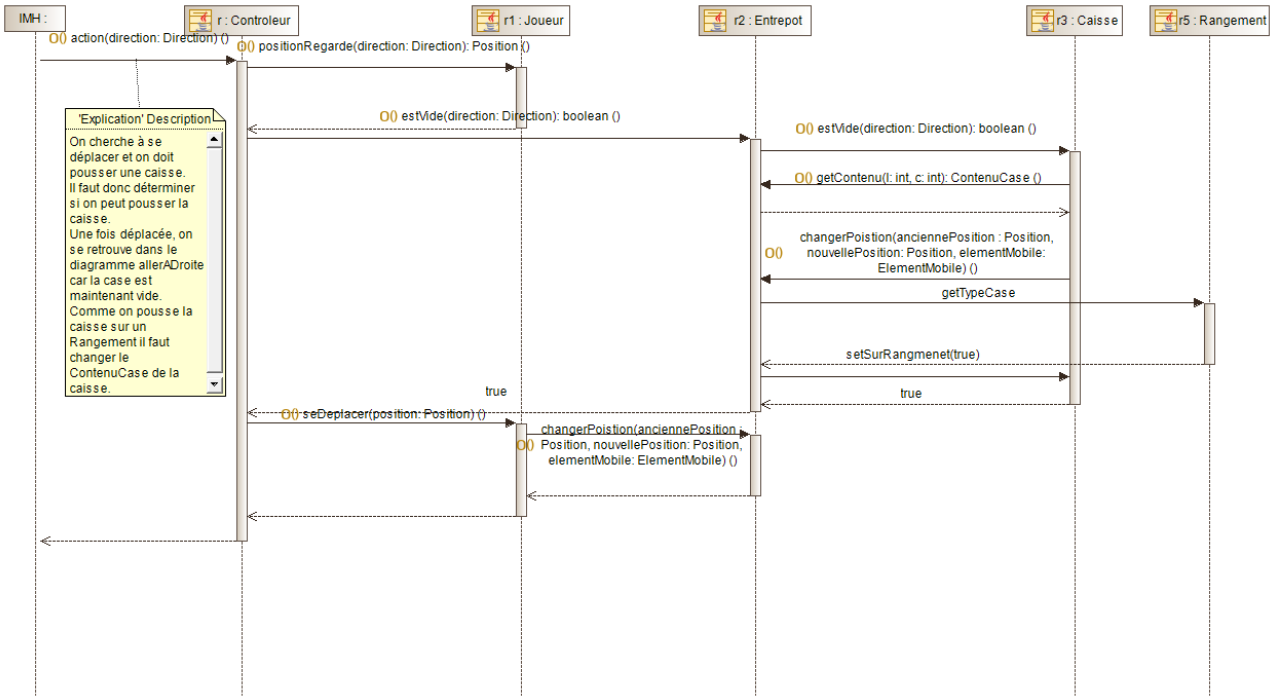


FIGURE 4.4 – Diagramme de séquence : pousser une caisse

Chapitre 5

Réalisation

5.1 Passage du modèle UML au code Java

Le modèle a été repris de zéro après le TD, même s'il s'en inspire. Il a ensuite été augmenté par incrémentation au fur et à mesure du développement des diagrammes de séquence.

Après un premier export depuis Modelio nous avions le squelette de notre projet. Cependant, il a été impossible de mettre à jour le projet Modelio après les modifications du code, et ainsi le diagramme de classe pourrait ne pas être complet car il a été complété par la suite à la main.

La majorité du code a été écrit en une longue session :

- Amaury s'est occupé de la lecture de carte dans un format standard de partage. Il a ensuite retranscrit les niveaux du Sokoban original et créé les cartes de test.
- Virgile s'est d'abord appliqué à réaliser l'ensemble des éléments disponibles dans l'entrepôt, puis à implémenté progressivement les déplacements des éléments mobiles.

À la fin de cette session, les bases du jeu étaient posées et il était déjà possible de lancer un premier niveau. Plusieurs tests ont été effectués 5.2 et les incohérences révélées ont été corrigées.

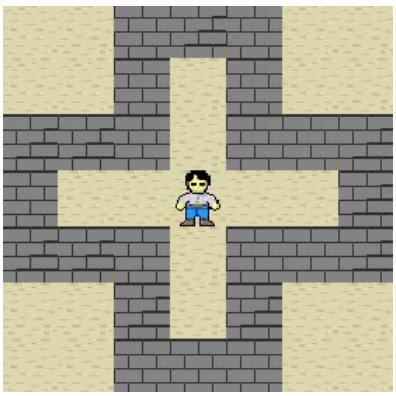
Nous avons investi notre temps par la suite dans la simplification du code. Une étude des améliorations et incrémentations possibles nous a permis de repenser une partie du code. En effet, initialement chaque déplacement supprimait la case sur laquelle le joueur avançait et recréait la case adéquate derrière. Or, dans l'hypothèse de nouvelles cases traversables, il semblait plus logique de prendre la case sur laquelle on se situe comme attribut et de la redéposer après le déplacement (rangements colorés ou téléporteur par exemple). De nombreux autres petits refactorings ont finalement été entrepris.

5.2 Tests effectués

Afin de vérifier le bon fonctionnement du logiciel, nous avons créé plusieurs cartes de test (qui sont toujours disponibles en choisissant les niveaux test1 à test9).

FIGURE 5.1 – Test 1 : Niveau simple

Le premier test que nous avons effectué consiste en un niveau basique, avec 4 caisses et 4 rangements. Cela nous a permis de vérifier que la victoire se déclenchait uniquement quand l'ensemble des caisses est rangé. Nous avons ensuite essayé de fournir au logiciel des cartes défectueuses pour vérifier son comportement.



```
Exception in thread "main" java.lang.IllegalArgumentException: Il n'y a aucun rangement
        at ProjetSokoban/modele.Entrepot.<init>(Entrepot.java:110)
        at ProjetSokoban/projetSokoban.TestIHM.<init>(TestIHM.java:36)
        at ProjetSokoban/projetSokoban.TestIHM.main(TestIHM.java:45)
```

FIGURE 5.2 – Test 2 : Juste le joueur, pas d'objectif

Dans le cas où il n'y a ni caisses ni rangements, le logiciel détecte qu'il n'y a pas de rangements et que le niveau n'est donc pas valide. Il envoie une exception.

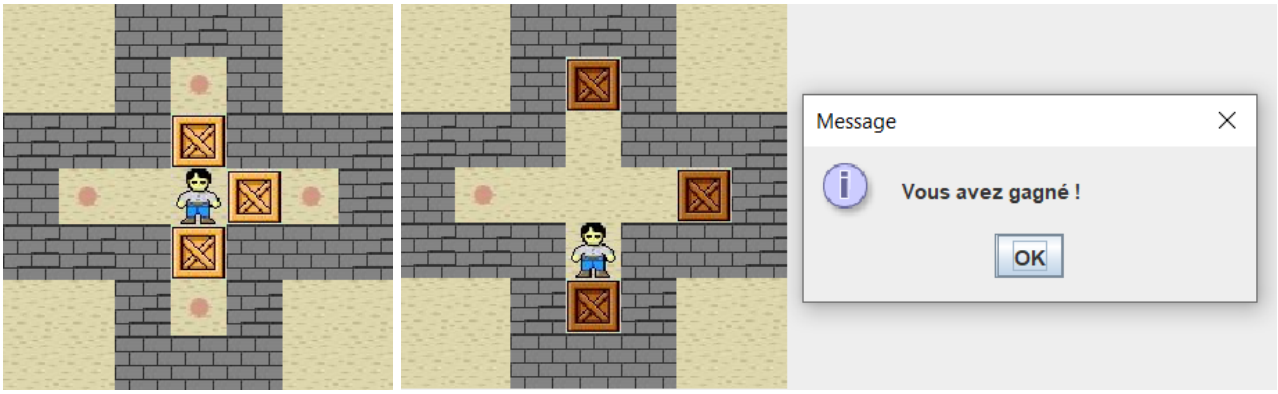
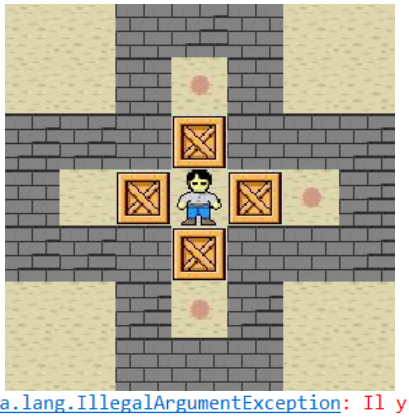


FIGURE 5.3 – Test 3 : Plus de rangements que de caisses

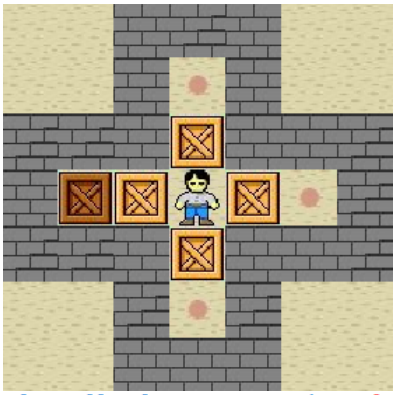
Dans le cas où il y a plus de rangements que de caisses, le niveau reste jouable. En effet, un entrepôt où il reste de la place une fois toutes les caisses stockées n'est pas embêtant. La victoire se déclenche bien quand toutes les caisses sont rangées. Ce type de niveau est donc valide.



```
Exception in thread "main" java.lang.IllegalArgumentException: Il y a des caisses sans rangement
        at ProjetSokoban/modele.Entrepot.<init>(Entrepot.java:106)
        at ProjetSokoban/projetSokoban.TestIHM.<init>(TestIHM.java:36)
        at ProjetSokoban/projetSokoban.TestIHM.main(TestIHM.java:45)
```

FIGURE 5.4 – Test 4 : Plus de caisses que de rangements

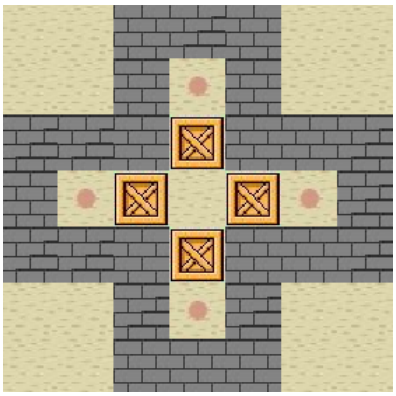
Dans le cas où il y a plus de caisses que de rangements, le niveau n'est pas résolvable. Le logiciel envoie donc une exception.



```
Exception in thread "main" java.lang.IllegalArgumentException: Il y a des caisses sans rangement
        at ProjetSokoban/modele.Entrepot.<init>(Entrepot.java:106)
        at ProjetSokoban/projetSokoban.TestIHM.<init>(TestIHM.java:36)
        at ProjetSokoban/projetSokoban.TestIHM.main(TestIHM.java:45)
```

FIGURE 5.5 – Test 5 : Plus de caisses que de rangements avec caisse rangée

Dans ce cas similaire où il y a plus de caisses que de rangements tout en ayant dès le départ une caisse rangée, la même exception est envoyée. C'est bien le comportement attendu.



```
Exception in thread "main" java.lang.IllegalArgumentException: Il n'y a pas de joueur sur la carte
        at ProjetSokoban/modele.Entrepot.<init>(Entrepot.java:102)
        at ProjetSokoban/projetSokoban.TestIHM.<init>(TestIHM.java:36)
        at ProjetSokoban/projetSokoban.TestIHM.main(TestIHM.java:45)
```

FIGURE 5.6 – Test 6 : Pas de joueur

Dans le cas où il n'y a pas de joueur, l'utilisateur n'aurait aucun moyen de déplacer les caisses pour résoudre le niveau. Le niveau n'est donc pas résolvable et le logiciel envoie une exception.

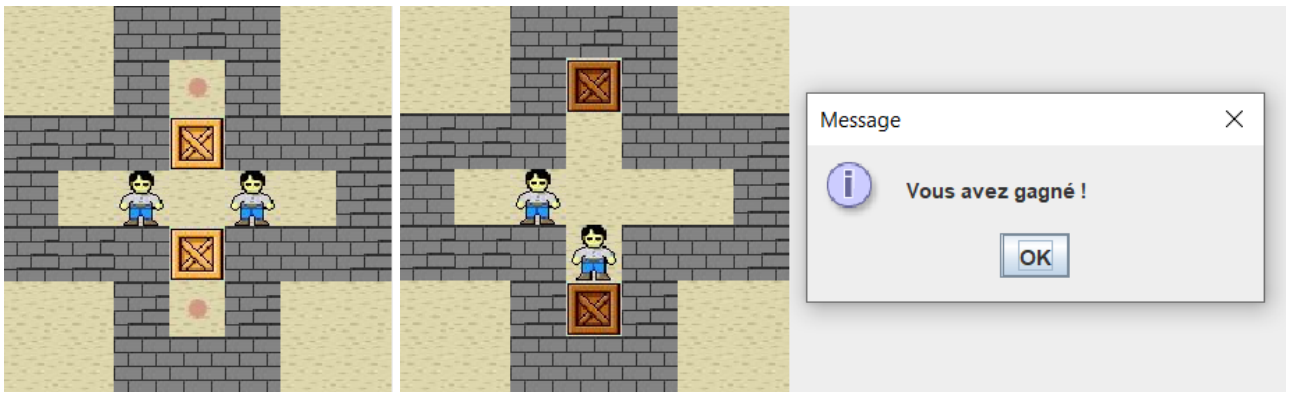


FIGURE 5.7 – Test 7 : Trop de joueurs

Dans le cas où il y a plus d'un joueur, c'est le joueur sur la ligne la plus basse qui sera contrôlable (ou le plus à droite si les joueurs sont sur la même ligne) à cause du sens de lecture de la carte par le logiciel. Les joueurs additionnels sont des objets immobiles qui se comportent comme des murs et le niveau reste valide. Nous n'avons pas implémenté d'erreur pour ce type de carte car l'implémentation d'un mode multijoueur est une perspective de fonctionnalité additionnelle (cf 6.2).

Les derniers tests portent sur les déplacements du joueur ainsi que ceux des caisses en fonction de la case de destination.

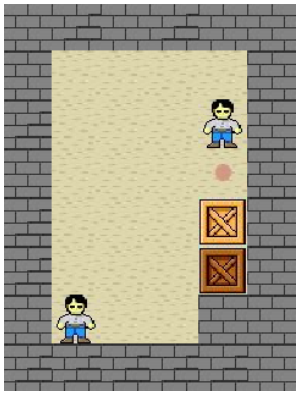


FIGURE 5.8 – Test 8 : Déplacement du joueur

Le joueur peut se déplacer sur une case vide ou un rangement, mais pas sur un autre joueur ni sur une caisse ou une caisse rangée. C'est le comportement attendu des déplacements du joueur.

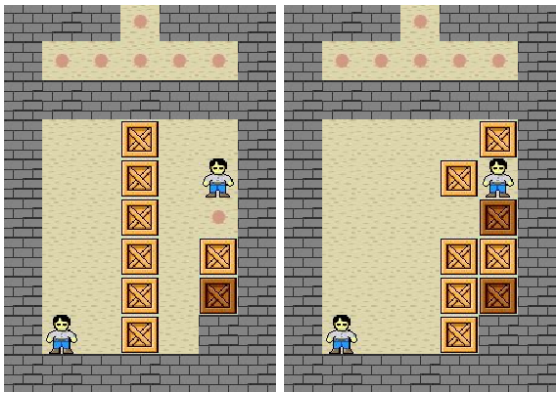


FIGURE 5.9 – Test 9 : Déplacement des caisses

Une caisse peut être poussée sur une case vide ou un rangement, mais pas sur un autre joueur ni sur une caisse ou une caisse rangée. C'est le comportement attendu des déplacements d'une caisse.

Chapitre 6

Conclusion

6.1 Bilan du projet

En conclusion, notre projet de jeu Sokoban a été une expérience stimulante et gratifiante. Nous avons pu appliquer nos connaissances en programmation Java et nos compétences de résolution de problèmes pour créer un jeu fonctionnel et convivial. Notre implémentation des mécanismes de jeu, notamment les déplacements des joueurs et des caisses, a été réussie et a satisfait nos exigences. Le projet nous a également permis de travailler en équipe et de collaborer efficacement pour produire un produit final. En outre, nous avons utilisé Git en tant que système de contrôle de version tout au long du processus de développement. Cela nous a permis de collaborer facilement et de suivre les modifications apportées par les deux membres de l'équipe.

Nous sommes fiers du résultat final et nous sommes également amusés à résoudre les premiers niveaux du jeu. Dans l'ensemble, ce projet a été une excellente occasion d'améliorer nos compétences de programmation et nous avons hâte de continuer à développer notre expertise dans ce domaine.

6.2 Perspectives envisageables

Nous avons envisagé de nombreuses perspectives pour notre projet. Certaines sont plutôt simples à implémenter, alors que d'autres demanderaient plus de modifications en profondeur. Toutefois dans l'ensemble, notre code semble facilement adaptable pour l'ajout de nouvelles fonctionnalités.

Les améliorations les plus évidentes sont des ajouts d'éléments en jeu afin de complexifier les niveaux. On pourrait ainsi imaginer avoir des caisses de différentes couleurs ne pouvant être rangées que sur les emplacements correspondants, des murs déverouillables avec des clés ou à casser avec des bombes, des tunnels pour relier différents endroits de la carte ou encore un passage à la 3D du jeu en ajoutant différents étages aux niveaux.

Nous pourrions également ajouter des améliorations au niveau de l'interface du joueur, par exemple en ajoutant le nombre de déplacements et le temps depuis le début de la partie afin de renforcer le sentiment de challenge. Nous avons également envisagé d'assigner des touches permettant de retourner en arrière, de relancer la partie ou encore pour activer un godmode permettant de traverser les murs.

Une autre facette de notre programme à remanier est la sélection de niveau. Nous pourrions ajouter un écran de sélection de la carte afin de remplacer le prompt actuel, permettant de sélectionner des niveaux de difficulté variables ou un tutoriel par exemple. Il serait aussi possible d'ajouter un menu permettant au joueur d'entrer son propre niveau (format textuel ou xsb) puis d'y jouer.

Enfin, il serait aisé d'implémenter des améliorations graphiques, que ce soit une amélioration de la résolution des textures, un changement de sprite du joueur en fonction de ses actions ou de son orientation, différentes ambiances ou encore un choix entre light et dark mode pour satisfaire au maximum les utilisateurs.

En somme, il y a de nombreuses perspectives envisageables pour continuer à développer ce projet. Toutes ces améliorations potentielles n'ont pas le même niveau de priorité, mais chacune pourrait modifier de manière importante le jeu ou la manière dont l'utilisateur le perçoit.