Alexandre Faure

Rapport de projet : Sokoban

Génie logiciel orienté objet

5 février 2024

Introduction

Ce rapport s'inscrit dans le cadre de l'électif : Génie logiciel orienté objet. Dans ce contexte, j'ai réalisé de décembre 2023 à janvier 2024 un projet de Sokoban en Java après modélisation en UML. Ce rapport permet de poser le cadre de cet exercice en présentant dans un premier temps le cahier des charges au regard de l'objectif du projet que représente la programmation du jeu. Dans un second temps, la description du comportement attendu du logiciel et de ses cas d'utilisation sera détaillée, par le biais de la représentation UML. Une fois l'expression des besoins achevée, une description formelle de la structure interne du logiciel sera faite. Ceci permettra, dans un dernier temps, d'évoquer les étapes et le processus de réalisation tout au long du projet.

Plan du rapport

I. Cahier des charges

Ce projet vise à mettre en pratique l'utilisation du langage UML et sa traduction en Java dans le cadre du paradigme de programmation orientée objet.

Pour se faire, ce projet s'intéresse tout particulièrement au jeu Sokoban (Wikipédia). Ce jeu consiste en un casse-tête. L'interface est composée de cellules délimitées par des murs. Sur certaines cellules sont représentées des caisses déplaçables qui doivent être déplacées sur des cellules dédiées. L'objectif du jeu est de pousser les caisses sur les cellules dédiées à l'aide d'un personnage pouvant se déplacer de cellule en cellule.

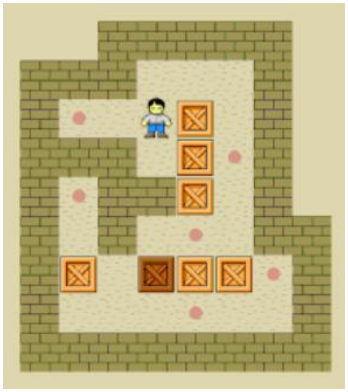


Figure 1. Capture d'écran d'une partie de Sokoban (Wikipédia)

Le logiciel doit donc permettre à un utilisateur de sélectionner le jeu à l'aide d'une interface dédiée. On pourra envisager selon l'avancement du projet de proposer différents niveaux, voire un module de création de niveau. Une fois la partie sélectionnée, l'utilisateur devra pouvoir interagir avec l'interface dédiée afin de commander le personnage jusqu'à réussir le niveau. Une façon de permettre de quitter la partie en cours ou de la réinitialiser pourra également être implémentée.

Dans le cadre du projet, l'interface réalisée consistera en une interface avec terminal, puis pourra être améliorée en une interface graphique Swing.

II. Expression des besoins

Les considérations ci-dessous concernent les besoins pour une interface avec un terminal. Les besoins additionnels pour une interface graphique seront développés ci-après.

Dans le cadre du jeu de Sokoban avec une interface sur terminal, le comportement attendu de la part de l'utilisateur est de pouvoir sélectionner un niveau à partir de son numéro parmi un ensemble de numéro fixé. Une fois le numéro saisi, le plateau de jeu doit être affiché dans le terminal avec des signes distinctifs permettant de décrire les différents éléments du jeu (caisse, cible, personnage, sol, mur). Dès lors, l'utilisateur devra pouvoir faire se déplacer le personnage avec des commandes claviers prédéfinies (l = left, r = right, u = up, d = down dans notre cas). A chaque commande valide saisie, le terminal devra regénérer le plateau de jeu, mis-à-jour à la suite de la précédente commande puis demander la commande suivante. Ce, jusqu'à ce que le joueur ait gagné ou qu'il ait saisi une commande lui permettant de réinitialiser la partie (n = new game) ou de quitter le jeu (q = quit). À tout moment de la partie, il sera également possible pour le joueur de redemander les commandes de bases en cas d'oubli par la commande h (help).

Dans le cadre du développement d'une interface graphique avec Swing, deux onglets distincts permettent de gérer la page d'accueil et la page de jeu. La page d'accueil permet à l'utilisateur de saisir le numéro du niveau qu'il souhaite jouer dans un champ de texte et de lancer le niveau à l'appui d'un bouton. La page de jeu est constituée en son centre du plateau de jeu. En haut à gauche une flèche permet de quitter la partie en cours et de retourner à la page d'accueil. En haut à droite, un second bouton permet de réinitialiser la partie (de la même façon que pour la commande 'n' précédemment). Le déplacement du player se fait désormais avec les flèches du clavier. Le jeu s'interrompt lorsque le joueur est parvenu à placer chaque caisse sur une cible du plateau. Une boîte s'affichera pour annoncer la victoire du joueur et lui permettant d'effectuer quelques actions (Retourner à l'accueil, Rejouer le niveau, Quitter le jeu).

Dans les deux cas, les cas d'utilisations du logiciel peuvent se résumer au travers du diagramme de cas d'utilisations ci-dessous.

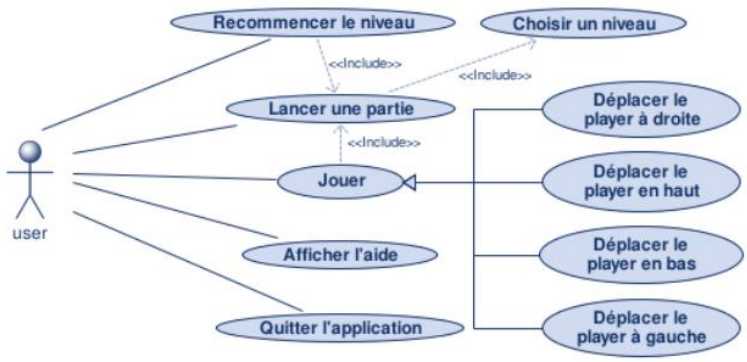


Figure 2. Diagramme de cas d'usage UML pour le jeu de Sokoban réalisé avec Modelio

III. Conception

La conception du logiciel java de Sokoban a été déployée sous le paradigme MVC (Modèle-Vue-Contrôleur). Ce modèle d'architecture vise à séparer distinctement les différentes responsabilités de l'application. La partie Modèle permet de gérer les données ainsi que la logique sous-jacente liant les différentes entités du modèle. La partie Vue est l'interface utilisateur qui permet à ce dernier d'interagir avec le modèle pour réaliser diverses actions. Enfin, le Contrôleur sert d'intermédiaire entre la partie Vue et la partie Modèle en traduisant les requêtes dans un sens et les réponses du modèle dans l'autre.

Dans le cadre du projet, un Modèle et un Contrôleur ont été définis et seront présentés successivement ci-dessous. Concernant l'interface utilisateur, deux versions distinctes ont été pensées pour permettre un affichage dans un terminal (viewConsole) et dans une interface graphique avec Swing (viewSing).

A. Modèle

Le modèle a été développé dans un package nommé « entity » pour représenter les différentes entités du modèle. Voici un diagramme de classe faisant apparaître dans le package « entity » les différentes classes implémentées. On peut déjà voir le package « control » qui sera évoqué plus longuement par la suite.

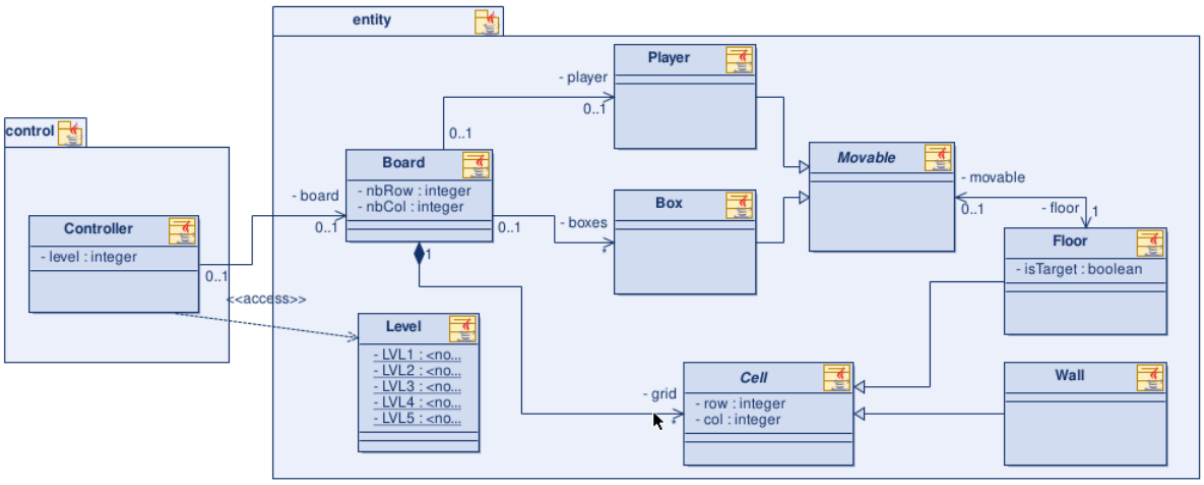


Figure 3. Diagramme de classe réalisé avec Modelio pour les packages control et entity

Pour chaque classe, le tableau ci-dessous vise à définir clairement leur responsabilité et leurs attributs respectifs.

Classe	Description et responsabilité	Attributs
Level	La classe Level permet de stocker en dur les différents plateaux de jeux pour chaque niveau. Elle ne contient qu'une fonction statique getBoardTemplate(int level) qui permet de récupérer le template du plateau du jeu pour le niveau donné à l'appel du contrôleur.	- LVL{i} (Tableau de chaine de caractères) : stocke en dur l'agencement des Board pour les niveaux i allant de 1 à 5.
Board	La classe Board est l'élément principal du package et représente le plateau de jeu. C'est cette classe qui va superviser les mouvements du joueur (Player) et des boîtes (Box) par la méthode move. C'est également cette classe qui à la charge d'envoyer l'état du plateau (autrement dit l'état des différentes cellules) afin de pouvoir l'afficher dans l'interface souhaitée à l'aide de l'énumérateur CellStatus. Cette classe a enfin la charge de vérifier si le joueur a gagné ou non.	- nbRow (entier) : nombre de lignes du plateau de jeu - nbCol (entier) : nombre de colonnes du plateau de jeu. - grid (tableau 2D de Cell) : ensemble des cellules qui le constituent sous forme d'un tableau à 2 dimensions - boxes (liste de Box) : stocke l'ensemble des Box du plateau pour accéder facilement aux Box et vérifier si elles sont sur les cibles (ce qui garantit la victoire du joueur). - player (Player) : stocke le joueur sur le plateau pour accéder facilement à sa position et gérer ses déplacements.
Cell	La classe Cell représente les cellules du plateau et est une classe abstraite. Elle n'est jamais directement appelée mais permet d'implémenter les attributs et méthodes communes à ses sous-classes Floor (pour le sol) et Wall (pour les murs). Elle définit notamment les méthodes moveInMovable et moveOutMovable qui permettent de déplacer ou retirer un objet Movable de cellules en cellules (cf. classe Movable). Elle implémente également la classe isTarget qui permet de savoir si la cellule est une cible ou non. Et enfin la méthode getCellStatus qui permet de renvoyer le statut de la cellule parmi l'enumérateur CellStatus en vue d'afficher son rendu dans l'interface.	- row (entier) : numéro de la ligne de la cellule en partant de 0. - col (entier) : numéro de la colonne de la cellule en partant de 0.
Floor	La classe Floor est une sous-classe de Cell qui décrit un sol accessible pour les Movable (Player et Box). Elle implémente les méthodes de la classe mère. Notamment la méthode getCellStatus interroge la présence ou non de Movable sur la cellule et le statut ou non de cible.	- isTarget (booléen) : décrit si la cellule est une cible. - movable (Movable) : stocke le movable présent sur la cellule s'il y en a un parmi Player ou Box.
Wall	La classe Wall est la deuxième déclinaison de Cell, sous-classe représentant donc les murs. Elle implémente les méthodes de la classe-mère, la plupart ne devant rien	Aucun attribut spécifique n'y est associé.

	renvoyer du fait du statut « neutre » des murs dans le jeu.	
Movable	La classe Movable est une classe abstraite qui décrit les éléments qui peuvent bouger sur le plateau au cours de la partie. Elle implémente la méthode abstraite getMovableStatus, sur le même modèle que getCellStatus, qui permet de décrire la nature du Movable.	- floor (Floor) : stocke la cellule sur laquelle le Movable se situe pour récupérer ensuite sa position et éviter de stocker en double cette information. Cela permet de récupérer ces informations à la fois pour le Player lorsqu'il doit se déplacer et pour les Box pour vérifier si elles se trouvent sur une cible.
Player	La classe Player est une sous-classe de Movable et décrit le joueur sur le plateau de jeu. Elle n'a pas de comportement spécifique si ce n'est qu'elle implémente la méthode getMovableStatus pour renvoyer son statut PLAYER.	Aucun attribut spécifique n'y est rattaché.
Box	La classe Box est une sous-classe de Movable et décrit une caisse sur le plateau de jeu. Comme Player, elle ne fait qu'implémenter la méthode getMovableStatus.	Aucun attribut spécifique n'y est rattaché.

Des énumérateurs ont par ailleurs été créés :

Enumérateur	Description et responsabilité	Valeurs possibles
Direction	L'énumérateur Direction permet de décrire les directions empruntables par le joueur sur le plateau de jeu. Il permet d'avoir des notations unifiées pour le modèle, et qui puisse servir également pour le contrôleur et l'interface visuelle.	LEFT : vers la gauche RIGHT : vers la droite UP : vers le haut DOWN : vers le bas
CellStatus	CellStatus permet de faire l'énumération des différents états dans lesquels peuvent se trouver une cellule à un instant donné. Cela permet d'une part de gérer l'affichage du plateau, mais également de faire des tests sur les statuts des cellules pour procéder à des déplacements ou vérifier la victoire ou non du joueur. Dans une première version, les valeurs PLAYER_TARGET et BOX_TRAGET n'avaient pas été implémentés. Puis pour distinguer ces deux cas de figures, ils ont été rajoutés.	WALL : mur FLOOR : sol nu TARGET : sol avec une cible PLAYER : sol avec un joueur PLAYER_TARGET : sol avec un joueur sur une cible BOX : sol avec une caisse BOX_TARGET : sol avec une caisse sur une caisse

B. Contrôleur

Le contrôleur, qui fait donc la jonction entre le package « view » et le package « entity » n'est constitué que d'une seule classe Controller. Cette classe traduit l'ensemble des requêtes de l'utilisateur

en des appels aux méthodes adéquates dans la partie entité. Ces deux attributs sont le niveau de la partie en cours et le plateau de jeu actuel, porte d'entrée sur la partie Modèle.

Différentes méthodes découlent ensuite des potentielles interactions de l'utilisateur avec le jeu :

- beginGame(level) permet de créer le niveau demandé et d'initialiser le plateau de jeu correspondant
- getBoardStatus() permet de récupérer l'état du plateau de jeu sous la forme d'un tableau 2D d'éléments de l'énumérateur CellStatus
- move(direction) permet de demander un déplacement dans la direction demandée parmi les direction de l'énumérateur Direction
- getLevel() est un getter permettant de récupérer le niveau actuel afin de pouvoir réinitialiser la partie en cours
- hasWon() permet de vérifier que toutes les caisses sont bien sur une cible, ce qui traduit la victoire du joueur

Un diagramme de séquence a été réalisé pour le niveau représenté ci-dessous. Les cellules sont nommées « c_l{i}c{j} » où i et j représentent respectivement le numéro de la ligne et de la colonne de la cellule. Quelques-unes des opérations élémentaires de la classe Contrôler y sont appelées, incluant la création du plateau, la récupération de son contenu, puis une tentative de déplacement à gauche (avortée) suivi d'un déplacement à droite. La méthode hasWon vérifie finalement si le joueur a gagné, ce qui est faux à la suite des deux mouvements demandés.

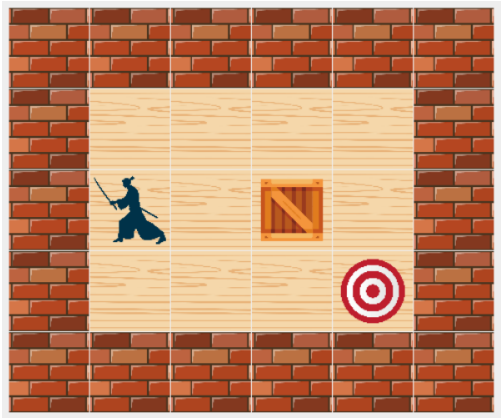


Figure 4. Plateau du niveau 1 après initialisation

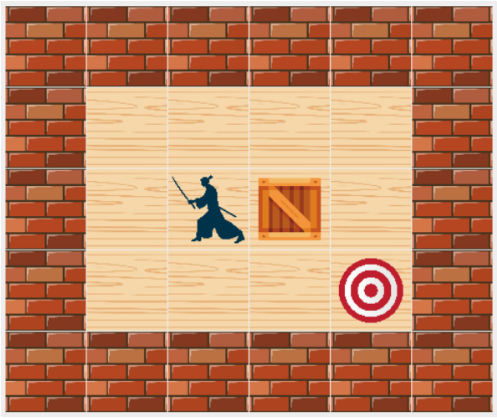
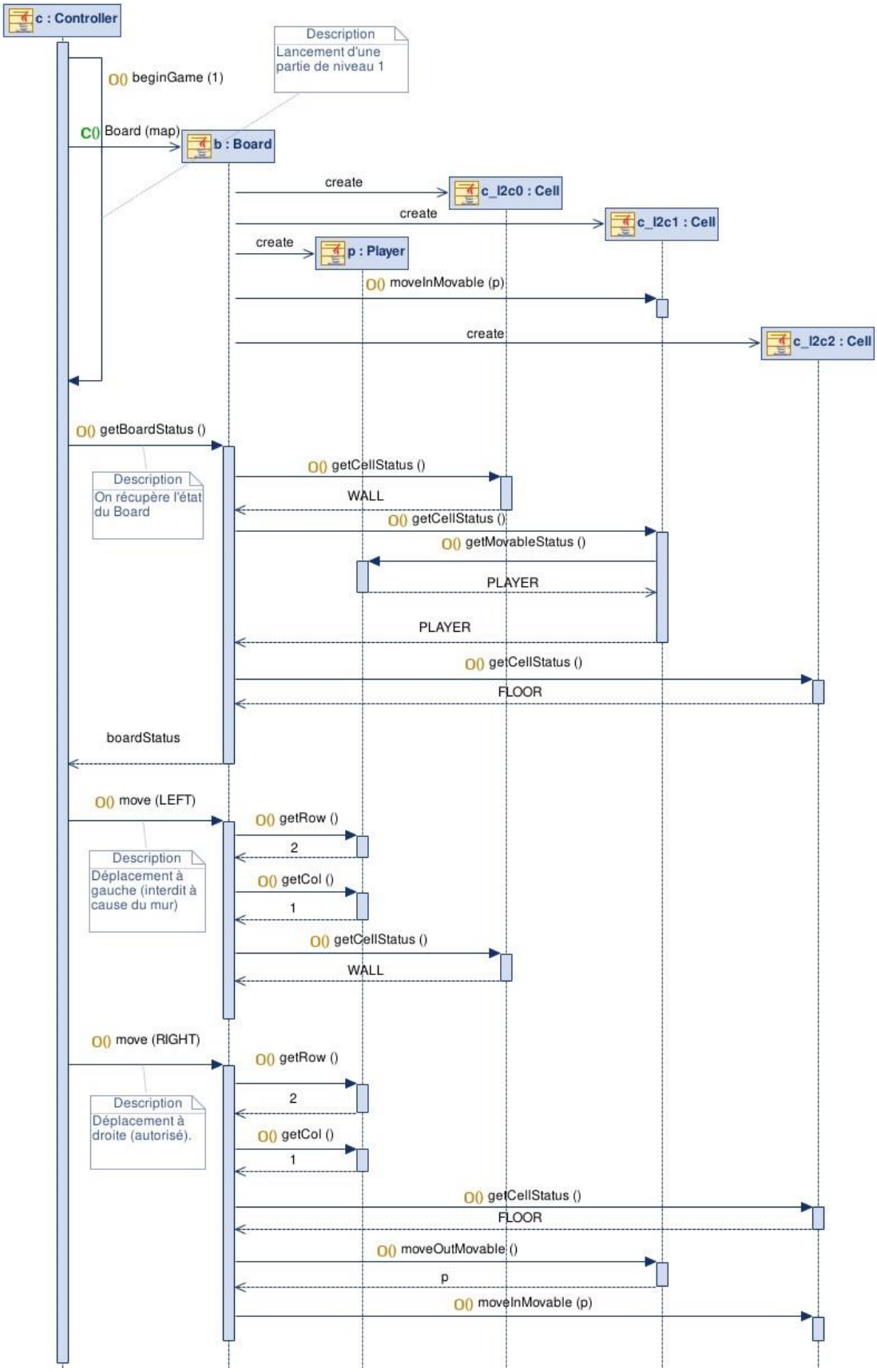


Figure 5. Plateau de jeu à l'issue des 2 mouvements demandés


c : Controller
Description
Lancement d'une partie de niveau 1
O() beginGame (1)
C() Board (map)
b : Board
create
c_l2c0 : Cell
create
c_l2c1 : Cell
create
p : Player
O() moveInMovable (p)
create
c_l2c2 : Cell
O() getBoardStatus ()
Description
On récupère l'état du Board
O() getCellStatus ()
WALL
O() getCellStatus ()
O() getMovableStatus ()
PLAYER
PLAYER
O() getCellStatus ()
FLOOR
boardStatus
O() move (LEFT)
O() getRow ()
2
Description
Déplacement à gauche (interdit à cause du mur)
O() getCol ()
1
O() getCellStatus ()
WALL
O() move (RIGHT)
O() getRow ()
2
Description
Déplacement à droite (autorisé).
O() getCol ()
1
O() getCellStatus ()
FLOOR
O() moveOutMovable ()
p
O() moveInMovable (p)

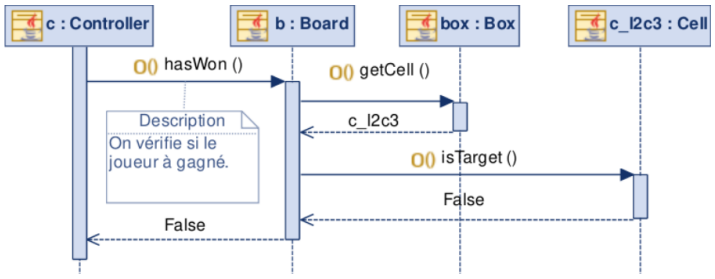


Figure 6. Diagramme de séquence UML modélisant l'interaction entre contrôleur et composants du modèle sur Modélio

C. Vue

a. Console View

Le package « viewConsole » a été le premier développé à la suite du développement des packages « entity » et « control ». Ce package a pour objectif de proposer une interface utilisateur par le biais du terminal afin de jouer au jeu de Sokoban sur la base des actions permises par le Controller selon le paradigme de programmation MVC.

Voici le diagramme de classe UML de ce package :

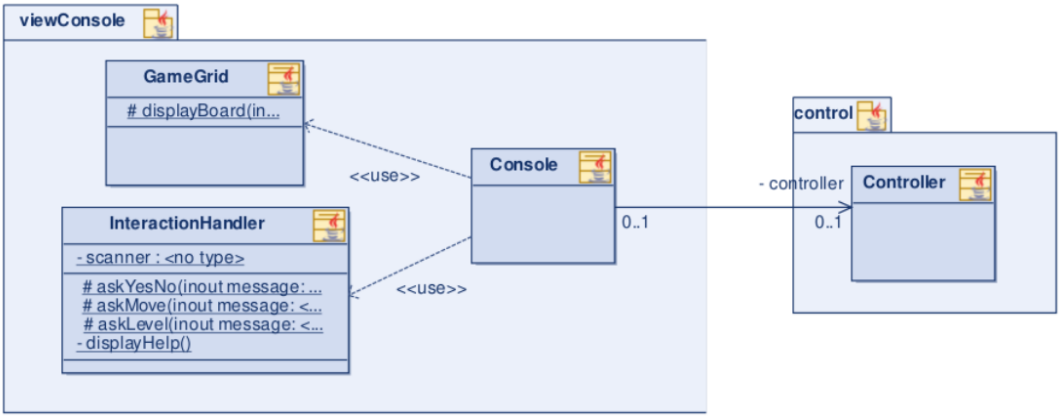


Figure 7. Diagramme de classe du package viewConsole et de son interaction avec le package control sur Modélio

Voici la description des classes et de leurs responsabilités respectives :

Classe	Description et responsabilité	Méthodes
Console	La classe Console est la classe principale contenant la fonction main pour lancer le jeu. Cette classe contient les fonctions principales et appellent les composants importants. Ces notamment cette classe qui implémente le contrôleur.	- playGame : permet de gérer le déroulement d'une partie une fois lancée en demandant le prochain mouvement jusqu'à ce que je le joueur gagne ou quitte/réinitialise sa partie. - wonGame : affiche un message de félicitations lorsque le joueur gagne - quitGame : affiche un message d'au revoir
InteractionHandler	La classe InteractionHandler est la classe qui a la responsabilité du Scanner qui permet de lire le contenu saisi par l'utilisateur dans le terminal. Cette classe ne contient que des	- readRequest : permet de traduire les messages saisis par l'utilisateur en une requête identifiée comme valide parmi les valeurs possibles de

	méthodes statiques qui permettent de gérer les commandes spéciales appelables à tout moment (pour quitter ou afficher l'aide) et les commandes valides seulement dans certains contexte (par exemple le déplacement pendant une partie).	l'énumérateur ConsoleRequest (cf. plus bas). - askYesNo : permet de poser une question à l'utilisateur dont la réponse attendue est oui ou non. - askMove : permet de demander à l'utilisateur son prochain mouvement - askLevel : permet de demander à l'utilisateur le niveau qu'il souhaite jouer - displayHelp : permet d'afficher l'aide lorsque la commande 'h' correspondante est détectée
GameGrid	GameGrid est une classe ne contenant qu'une unique méthode statique qui permet d'afficher le plateau à l'appel de la méthode. C'est ici que le choix des caractères pour afficher les différents états des cellules sont choisis.	- displayBoard : demande au contrôleur l'état du plateau puis affiche les cellules une par une selon leur statut.

Deux énumérateurs permettent de gérer le comportement du package « viewConsole ».

Enumérateur	Description et responsabilité	Valeurs possibles
ConsoleRequest	ConsoleRequest énumère les différentes requêtes que peut faire l'utilisateur par l'intermédiaire du terminal.	UNKNOWN : permet de caractériser une commande inconnue. Sert également de valeur par défaut dans l'attente d'une requête. QUIT : pour quitter HELP : pour afficher l'aide REPLAY : pour rejouer YES : pour oui NO : pour non LEVEL : pour une saisie correspondant à un niveau MOVE_UP : pour se déplacer vers le haut MOVE_DOWN : pour se déplacer vers le bas MOVE_RIGHT : pour se déplacer vers la droite MOVE_LEFT : pour se déplacer vers la gauche
QuestionStatus	QuestionStatus permet de déterminer le type de question qui a été posé au joueur afin de filtrer les commandes autorisées dans ce contexte.	YES_NO : pour les questions dont la réponse attendue est oui ou non MOVE : pour les questions pour lesquelles on attend un mouvement LEVEL : pour les questions pour lesquelles on attend un niveau

b. Swing View

Le package « viewSwing » a été développé dans un second temps avec pour objectif de proposer une alternative au package « viewConsole » afin de développer une interface graphique. Le principal changement réside dans

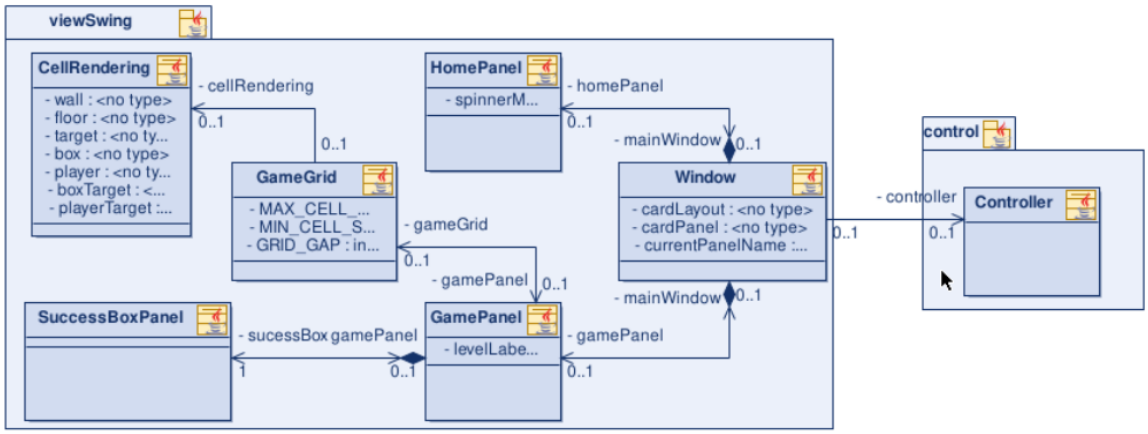


Figure 8. Diagramme de classe du package viewSwing et de son interaction avec le contrôleur avec Modélio

Voici les différentes classes y figurant :

Classe	Description et responsabilité	Attributs
Window	Window, de la même manière que la classe Console dans le package viewConsole est la classe principale du package. Elle sert de canvas principal et détecte notamment les interactions clavier. Cette classe permet également de gérer différents onglets, à savoir l'onglet d'accueil et l'onglet de partie et leur navigation de l'un à l'autre.	- cardLayout et cardPanel : permettent de gérer la navigation sous forme d'onglets - currentPanelName : permet de mémoriser l'onglet actuel. Cela permet notamment de savoir si l'onglet est celui du jeu (« game ») auquel cas les appuis sur les flèches doivent être détecté. - controller : permet de créer le contrôleur par lequel les commandes s'effecturont
HomePanel	HomePanel est la classe qui gère l'onglet de la page d'accueil. Elle contient notamment le champ numéraire permettant de fixer le niveau souhaité avant de lancer la partie.	- spinenrModel : permet de créer le champ numéraire borné entre 1 et 5 (qui correspondent aux niveaux existants). - mainWindow : pointe vers la fenêtre principale afin d'effectuer diverses opérations incluant des appels au contrôleur ou la navigation entre les onglets
GamePanel	GamePanel est le deuxième onglet qui permet de gérer l'affichage de la partie en cours. Il contient notamment deux boutons pour retourner à la page d'accueil ou relancer la partie en cours, un titre contenant le nom du niveau, le plateau de jeu et un popup pour féliciter le joueur de sa victoire, caché au début de la partie.	- levelLabel : permet de stocker le nom du niveau, affiché en haut de la fenêtre. Stocker ce champ permet de le mettre à jour pour chaque nouvelle partie. - mainWindow : pointe vers la fenêtre principale afin d'effectuer diverses opérations incluant des appels au

		contrôleur ou la navigation entre les onglets - sucessBox : stocke le panel affichant le message de succès - gameGrid : stocke le panel affichant le plateau de jeu
SucessBoxPanel	La classe SuccessBoxPanel permet d'afficher une boîte au premier plan du plateau de jeu dans l'onglet GamePanel lorsque le joueur parvient à gagner un niveau. Un message sur 2 lignes s'affiche, accompagné de 3 boutons permettant de retourner à l'accueil, rejouer la partie ou de quitter complètement.	Aucun attribut spécifique
GameGrid	La classe GameGrid est chargée d'afficher le plateau de jeu de la même façon que dans le package viewConsole. L'affichage fait appel à la classe CellRendering (cf. ci-dessous). Cette classe s'adapte dynamiquement à l'espace qui lui est accordé en respectant des contraintes de tailles des cellules du plateau grâce à des constantes (cf. attributs). Le plateau est lui-même centré dans la zone qui lui est accordée.	- MAX_CELL_SIZE : taille maximale d'une cellule du plateau - MIN_CELL_SIZE : taille minimale d'une cellule du plateau pour éviter que le plateau ne soit trop petit - GRID_GAP : espacement entre les cellules du plateau de jeu. Avec les 3 constantes précédentes, cela permet de fixer les caractéristiques du plateau - gamePanel : pointe vers l'onglet de jeu afin de pouvoir remonter au contrôleur, ce qui permet de récupérer le statut du plateau. - cellRendering : pointe vers la classe permettant d'afficher les images cellule par cellule correspondant à leur statut.
CellRendering	La classe CellRendering est une classe qui contient une méthode permettant de récupérer l'image correspondant au statut demandé. Ce module charge les images représentant les états possibles des cellules et les redimensionne à la taille demandée par la classe GameGrid.	- wall - floor - target - player - playerTarget - box - boxTarget Chaque attribut contient l'image représentant l'état de la cellule du même nom.

Voici un bref aperçu de l'interface graphique

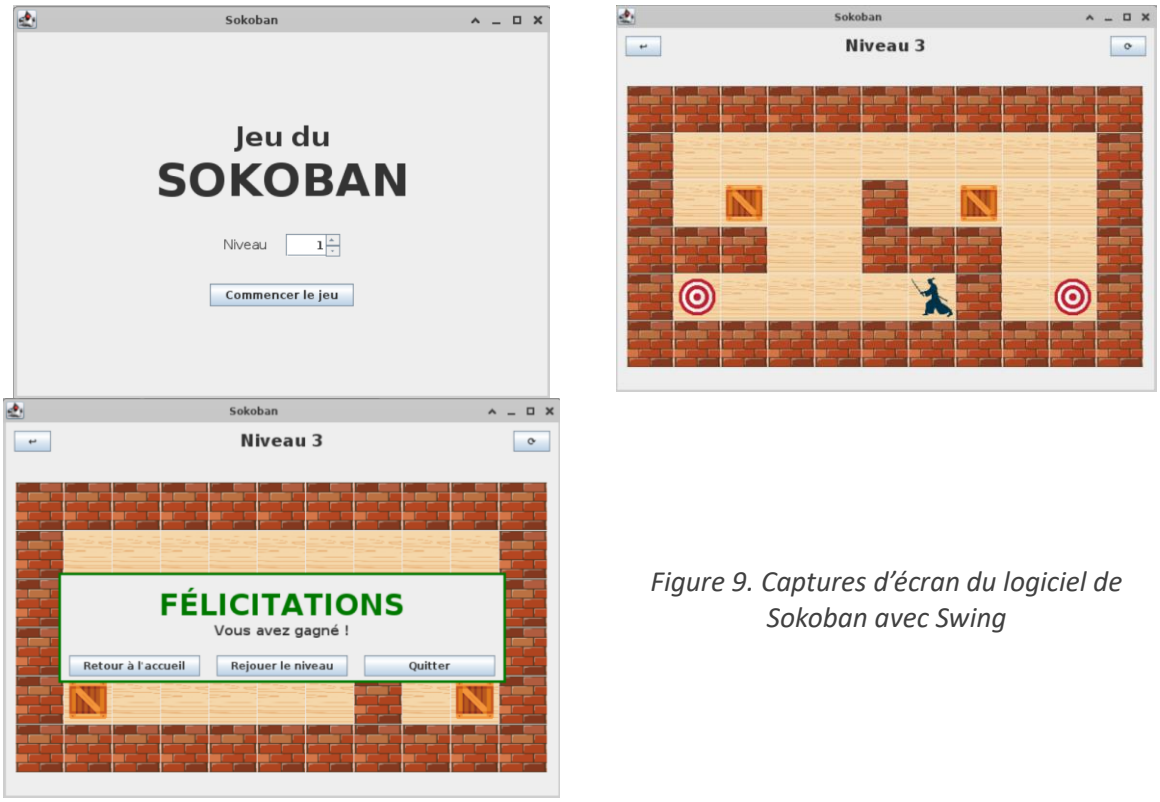


Figure 9. Captures d'écran du logiciel de Sokoban avec Swing

IV. Réalisation

A. Déroulement du projet

Le projet a débuté avec le premier TD de lancement du projet intégré. Un travail conséquent a été réalisé à cette occasion en ce qui concerne la représentation du module « entity » sous forme d'un diagramme de classe UML. A la suite du TD, j'ai pu commencer à réfléchir aux interactions entre les différents composants du module « entity » et aux méthodes utiles dans la classe Controller. A cette occasion, j'ai pu commencer à réaliser un diagramme de séquence qui figure dans le rapport (Figure 6.).

Une fois la section pour expliquer comment faire interagir le projet sur Modélio et Eclipse, j'ai pu transposer l'architecture générale de l'application Sokoban sur Eclipse en Java. Dès lors, j'ai pu implémenter les méthodes de chaque classe. Une fois ces classes réalisées, j'ai implémenté une classe de test dans le contrôleur pour s'assurer que le comportement du modèle en interaction avec le contrôleur est bien celui attendu. Une fois la partie modèle validée, j'ai pu commencer à implémenter le package « viewConsole » pour gérer la Vue dans un terminal. A la suite de quoi quelques tests à l'usage ont été réalisés pour valider la bonne implémentation du package.

Dans un second temps, après le TD sur l'utilisation du module Swing, je me suis attaché à développer une nouvelle version de la Vue dans le package « viewSwing » pour créer une interface graphique. De la même façon, et en s'inspirant grandement de l'architecture du package « viewConsole », j'ai implémenté le package. J'ai cependant souhaité distinguer à cette occasion les éléments Movable selon qu'ils soient sur une cible ou non pour pouvoir gérer un affichage différent. J'ai donc modifié le modèle à cette occasion ainsi que les tests sur le Modèle.

Tout au long du projet, diverses difficultés ont été rencontrées, notamment en ce qui concerne la gestion de la boîte d'affichage de victoire dans le cas du modèle viewSwing pour lequel la gestion de la superposition avec le plateau a posé de nombreux problèmes. Après de multiples allers retours entre

la documentation Java et le code, je suis parvenu à résoudre le problème. Ce qui m'a permis par la même occasion de mieux comprendre l'usage de la méthode repaint() et de découvrir la méthode setComponentZOrder() permettant de gérer la superposition de composants.

B. Tests effectués

Les tests effectués pour le Modèle ont été réalisés avec des tests boîte noir pour vérifier la satisfaction du comportement des éléments au cahier des charges. Ces tests sont directement accessibles dans la classe TestBoiteNoire dans le package Controller. Ils permettent de vérifier le bon fonctionnement de la création d'un niveau par le contrôleur, de vérifier que le plateau créé coïncide avec la carte du niveau, que les mouvements de base du joueur fonctionnent, que les mouvements spéciaux (avec des murs, des cibles ou des boîtes) sont correctement gérés, et enfin que le jeu détecte convenablement la victoire ou non du joueur.

En ce qui concerne la Vue, les tests boîte blanche à l'usage ont été réalisés pour vérifier que le comportement attendu est bien observé. Notamment en ce qui concerne les différentes commandes que peuvent saisir l'utilisateur à chaque étape du jeu. A cette occasion, plusieurs erreurs dans le code ont pu être corrigées notamment dans la partie « viewConsole » pour laquelle les boucles while demandant une commande pour se déplacer doivent continuer jusqu'à une condition d'arrêt qui est : l'utilisateur quitte ou saisit une commande de déplacement valide.

Conclusion

En conclusion, ce projet intégré a été une opportunité pour appliquer les bonnes pratiques dans le cadre de projets informatiques en intégrant les enjeux multiples qui accompagnent ces projets en entreprise. Ces enjeux sont tout d'abord la clarté et la lisibilité du code, mais également son évolutivité, sa facilité de maintenance et la nécessité de faire des tests pour valider le bon fonctionnement de l'application. Le projet a été l'occasion de travailler contre moi-même par moment, lorsque mon envie de coder avant de modéliser de façon formelle prenait le pas. J'ai ainsi eu l'occasion de devoir reprendre mon code après que la modélisation ait fait apparaître des redondances d'informations entre des classes, ce qui a illustré l'importance de la modélisation.

Du côté du logiciel, de multiples améliorations peuvent être imaginées. Parmi elles, l'ajout d'un bouton pour passer au niveau suivant après avoir fini une partie plutôt que de rejouer le même niveau, l'implémentation d'un module de génération automatique de niveau à partir d'une taille de grille donnée, la vérification que le joueur peut encore résoudre le Sokoban après chaque déplacement pour l'inviter à rejouer le cas échéant, l'implémentation d'un module de création de niveau, la rotation du personnage selon sa direction pour plus de réalisme… Puis de manière plus ambitieuse, on peut également imaginer de connecter le jeu à internet pour déployer des duels en ligne, des défis ou d'autres concours entre joueurs sur le web.