Rapport de projet
-
Génie Logiciel Orienté Objet

MAHAUT Raphaël - MULLER Thibault

I - Introduction :

Ce projet avait pour but de coder en java le jeu Sokoban qui est un jeu d'arcade avec une mécanique simple, mais demandant de la logique pour parvenir à résoudre les niveaux sans se retrouver bloqué. Notre but est non seulement d'obtenir un jeu fonctionnel, comportant les mécaniques du jeu original, mais également d'appliquer les principes de bonne conception vus en cours. Dans ce rapport, nous aborderons les choix effectués, les étapes de la conception et l'organisation du code que l'on a produit au cours de ce projet.

II - Cahier des charges :

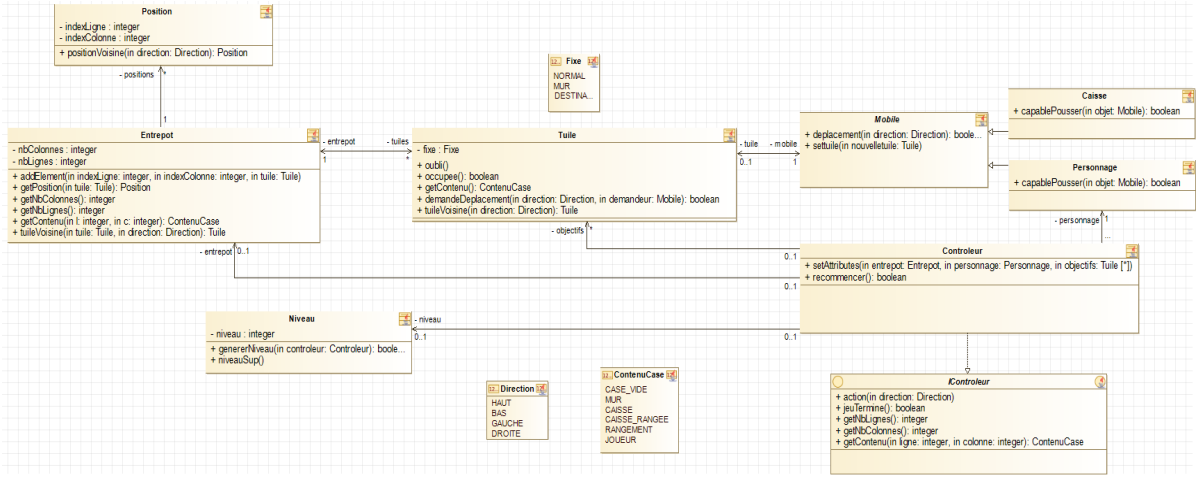
Le jeu se déroule dans un entrepôt représenté par des cases carrées, que nous appellerons tuile dans notre code pour ne pas interférer avec le mot clé "case" de Java. L'entrepôt est délimité par des murs contraignants les mouvements possibles. Nous devons déplacer des caisses à des positions particulières, marquées dans l'entrepôt par un point, à l'aide d'un gardien d'entrepôt représentant le joueur. Le gardien se déplace de manière verticale ou horizontale par bond de case en case et peut déplacer les caisses en les poussant, mais il ne peut pousser qu'une seule caisse à la fois et ne peut pas les tirer.

III - Expression des besoins :

Nous souhaitons que le joueur puisse contrôler le gardien de l'entrepôt à l'aide des flèches du clavier et qu'il y ait une interface homme-machine graphique pour que le jeu soit plaisant à voir. Nous souhaitons également que le joueur puisse accéder à un niveau différent lorsqu'il complète un niveau. De plus, il faut pouvoir recommencer le niveau lorsque l'on est bloqué sans être obligé de relancer le jeu, nous avons choisi de donner cette possibilité en appuyant sur la touche R du clavier.

IV - Conception :

Nous avons organisé notre code, sous la forme du patron MVC, dont voici le diagramme de classe pour les parties modèle métier et contrôleur:



Modèle métier:

- Entrepot :

La classe Entrepot permet de faire le lien entre les tuiles qui composent l'entrepôt et leurs positions représentées par le numéro de ligne et le numéro de colonne. Le point (0, 0) est situé en haut à gauche. L'entrepôt possède en attributs deux listes, tuiles et positions, ordonnées de manière correspondante ce qui permet de faire le lien entre les tuiles et leurs positions.

C'est aussi cette classe qui contient l'information sur la taille de l'entrepôt avec les attributs nbLignes et nbColonnes. **Elle permet de trouver les tuiles adjacentes**.

- Tuile :

Cette classe représente les parties de l'entrepôt, elles possèdent une propriété fixe parmi (NORMAL, MUR et DESTINATION) et peuvent contenir des objets mobiles à certains moments. Les tuiles ont une position fixe, ce sont les éléments qu'elles contiennent qui peuvent bouger.

Cette classe est chargée de **contenir les éléments et par conséquent connaît le contenu à afficher**.

- ContenuCase:

Enumération des différents état possible d'une tuile, **utile pour définir l'affichage**.

- Position :

Représentation de la position au sein de l'entrepôt sous forme de ligne et colonne. Cette classe est capable de **donner la position située à côté dans les quatres directions**.

- Mobile :

Classe abstraite regroupant les différents types d'objets mobiles. Cet objet connait la Tuile dans laquelle il est contenu. Elle **fournit la méthode pour se déplacer**.

- Personnage :

Classe qui hérite de Mobile, représentant le gardien d'entrepôt, il est **capable de pousser une caisse**.

- Caisse :

Classe qui hérite de Mobile, représentant les caisses de l'entrepôt, cette classe est in**capable de pousser un autre objet mobile.**

- Direction :

Enumération des mouvements possibles (HAUT, BAS, GAUCHE, DROITE).

- Fixe :

. Énumération des propriétés fixes d'une tuile (NORMAL, MUR et DESTINATION).

- Niveau :

Génère les différents objets représentant un niveau du jeu à partir de fichiers csv. Cette classe **connaît le niveau actuel du joueur** et est capable de **déterminer si un niveau suivant existe**.

Contrôleur :

La classe Controleur **connaît le personnage pour lui demander de se déplacer**, l'entrepôt pour pouvoir **fournir à l'ihm les informations sur la disposition de l'entrepôt à tout instant**, et les tuiles qui sont des destinations à atteindre pour les caisses ce qui permet de **vérifier si le niveau est complété ou non**.

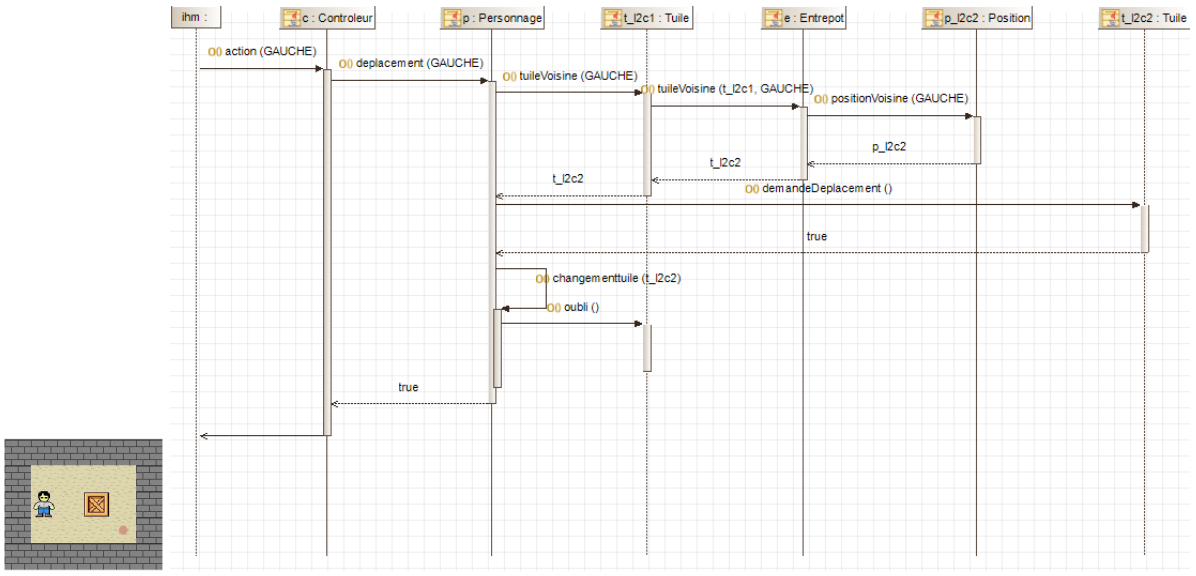
Ihm :

Pour l'ihm nous avons pris l'interface graphique fournie par Dominique Marcadet. Dans celle-ci nous avons deux classes: FenetreSokoban qui **écoute le clavier et communique avec le contrôleur** et PanneauSokoban qui contient les images et **affiche les différentes tuiles**.

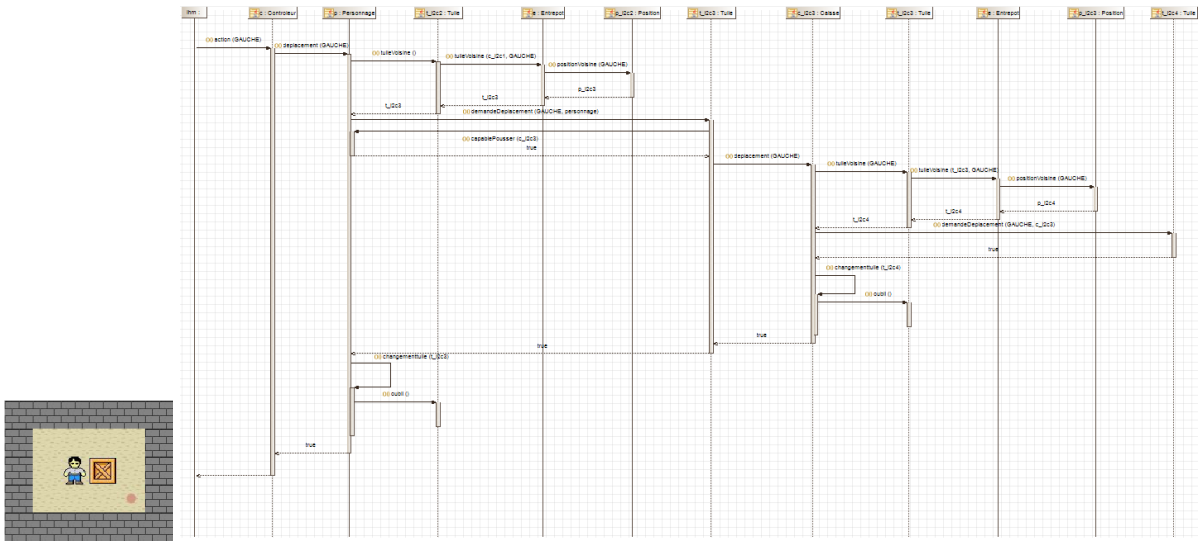
Fonctionnement des déplacements:

Lorsque l'on demande au personnage de se déplacer, il recherche d'abord la tuile vers laquelle, il doit essayer de se déplacer, pour cela il demande à sa tuile, qui demande à l'entrepôt, qui recherche la position de la tuile et demande quelle est la position d'à côté et renvoie la tuile situé à cette position. Une fois cela effectué, le personnage demande à la tuile vers laquelle il veut se déplacer si celle-ci peut l'accueillir. Dans le cas où un autre objet mobile est déjà sur cette tuile, la tuile demande au personnage s'il est capable de pousser l'objet auquel cas la tuile demande à l'objet de se déplacer (utilisant le même fonctionnement). Si la tuile peut finalement accueillir le personnage, la tuile change son attribut mobile et le personnage change son attribut tuile tout en signifiant à la tuile sur laquelle il se trouvait de l'oublier.

Ci-dessus, deux cas sont présentés sous forme de diagramme de séquence.



déplacement à gauche



déplacement à gauche

Génération des niveaux:

Les niveaux sont généré à partir d'une description dans un fichier csv, dans lequel on retrouvent en ligne et colonnes des lettres correspondant aux objets (M: mur, J: personnage, V: case vide, C: caisse, D: destination, R: caisse rangée, autres: aucun élément). On peut facilement ajouter des niveaux simplement en ajoutant un fichier nommé niveau_n.csv dans le dossier niveau. Si les niveaux précédents existent, lorsque tous auront été fini la classe essayera d'ouvrir ce fichier. Lorsque l'exception, FileNotFoundException est attrapée, le jeu considère que tous les niveaux ont été fait. Si un fichier ne décrit pas un niveau valide de manière évidente (absence de personnage ou nombre de caisse différent du nombre de destinations), le jeu essaie d'ouvrir le niveau suivant jusqu'à ne plus trouver de niveau. Il est cependant possible de générer un niveau impossible à finir ou un niveau où des cases non-référencées seraient accessibles (elles agissent alors comme des murs). Nous avons

considéré que c'est a celui qui crée le fichier du niveau de vérifier que son niveau soit valide de ce point de vue.

La classe niveau se charge de générer un objet *Entrepot* et les objets qu'il contient, à l'exception des positions qui sont générés lorsque l'on ajoute une tuile à l'entrepôt. Certaines classes ont des relations bilatérales (l'objet A connaît l'objet B et l'objet B connaît l'objet A), comme *Tuile* et *Mobile* dans ce cas l'une des deux classes possède un setter, nécessaire pour créer ce double lien.

La classe Entrepot possède un setter (addElement) qui permet de créer et d'ajouter une position dans la liste des positions en même temps que d'ajouter une tuile dans la liste des tuiles. Ainsi les deux listes sont toujours correspondantes. De plus, la taille de l'entrepôt s'ajuste automatiquement.

Lorsque l'on presse la touche R, le niveau se réinitialise en appelant la fonction de génération de niveau avec le niveau actuel. Les attributs du contrôleur sont remplacés par de nouveaux objets, les anciens n'étant plus accessible par aucune variable du programme seront supprimés par le ramasse miette. Cela ne devrait donc pas pouvoir saturer l'espace disponible d'objets inutiles.

V- Réalisation :

Nous avons donc d'abord conçu le modèle UML des parties modèle et contrôleur du patron MVC sur Modelio en reprenant les éléments que nous avions vu en cours. Ce travail nous a permis de définir les classes nécessaires à priori pour la conception de ce jeu ainsi que leurs attributs et les possibles interactions entre elles. Nous avons ensuite effectué un diagramme de séquence pour définir les méthodes à implémenter pour le fonctionnement des parties modèle et contrôleur. La réalisation du diagramme UML a été faite en commun en amont du codage Java, bien que nous ayons par la suite revu ce projet Modelio. Nous avons repris une grande partie de la réflexion qui avait été faite durant le TD dédié pour obtenir ce premier jet du travail.

A partir de cette première vision du projet, nous avons généré la trame du code à l'aide de la génération automatique de Modelio. Pour pouvoir coder à deux sur ce projet, nous avons ensuite créé un répertoire Gitlab et nous nous sommes réparti les différentes fonctions que nous avions prédéfinies.

Il nous a fallu ensuite ajouter la partie ihm, et pour se faire, nous avons utilisé les fichiers fournis.

Une fois les premières fonctions complétées, le besoin de certaines autres méthodes est apparu nécessaire. En effet, le schémas UML n'était pas assez complet pour être fonctionnel, et il a fallu aussi modifier certaines choses. Par exemple, nous avons eu besoin d'ajouter une méthode pour effacer la présence d'un objet mobile dans une tuile. De plus, certains liens entre classes étaient inutiles et il nous a fallu ajouter les constructeurs des classes.

Une fois cela fait, tout s'est très vite délié, et nous avons eu un premier jet fonctionnel. Dans un premier temps, nous avons testé que les interactions entre les objets convenaient en générant un petit entrepôt directement implémenté dans le contrôleur.

Cela nous a permis de tester le fonctionnement de la partie modèle métier. L'interface graphique permet de vérifier facilement que les déplacements fonctionnent, que le personnage peut se déplacer et pousser les caisses dans toutes les directions, que les murs empêchent bien les déplacements, et que deux caisses alignées ne peuvent pas être déplacées.

Il nous restait alors à implémenter la détection de la fin du jeu, pour cela, il nous a alors paru judicieux que le contrôleur connaisse les destinations des caisses pour faciliter cette tâche. Une fois cela fait, nous avons ajouté la classe Niveau pour gérer la génération des niveaux à partir de fichiers csv. Cela a permis dans le même temps de pouvoir recommencer le niveau depuis le début. Enfin, nous avons implémenté le passage d'un niveau à l'autre tant que de nouveaux niveaux existent et nous avons testé la réaction du logiciel aux différentes erreurs possibles dans le fichier csv qui décrit le niveau (absence de personnage, pas assez de caisse par rapport aux destinations, …).

Nous avions alors le jeu que nous souhaitions avec l'ensemble des fonctionnalités que nous avions choisi de faire au cours de ce projet. Toute cette première partie s'est étalée entre début décembre et mi-janvier.

À partir de ce moment-là, nous avons élagué le code dans le but d'en augmenter la qualité, notamment en retirant les liens inutiles, en essayant de respecter la loi de Déméter là où elle ne l'était pas. Par exemple, nous utilisons une classe Fixe et deux sous-classes Mur et Destination pour représenter les éléments fixes pouvant être sur une tuile. Cela était inutile et transformer la classe Fixe en énumération nous a paru mieux correspondre à l'utilisation que nous faisions de ces classes.

Gitlab nous a permis de travailler de manière conjointe, lorsque l'un faisait des ajustements sur une partie du code, nous demandions à l'autre de vérifier que tout semble correct.

VI - Conclusion :

Ce projet a été très intéressant pour s'exercer au codage orienté objet en Java. Nous en sommes restés aux fonctionnalités de base, mais nous sommes parvenus à ce que nous souhaitions initialement.

Pour améliorer le jeu, nous pourrions permettre le retour en arrière d'un coup, pour cela il serait nécessaire de sauvegarder la position des objets mobiles de l'entrepôt au coup précédent. Nous pourrions également ajouter un menu, pour choisir le niveau auquel on souhaite jouer sans avoir besoin de faire les niveaux précédents.