CentraleSupélec

université PARIS-SACLAY

Sokoban Java

Génie logiciel orienté objet

04/02/2024 PARIS

Étienne BONNAND, Milo HIVERT

Professeur référent : Dominique MARCADET

Sommaire

1 Introduction

Le Sokoban, connu sous le nom de "l'homme qui pousse les boîtes", est un casse-tête japonais classique créé dans les années 80. Le jeu se déroule dans un labyrinthe où les joueurs doivent déplacer des boîtes vers des emplacements spécifiques en évitant les obstacles. Les règles simples impliquent des déplacements stratégiques, chaque niveau réussi nécessitant une planification minutieuse pour anticiper les conséquences de chaque mouvement. Le Sokoban offre un défi intellectuel captivant avec plusieurs solutions possibles pour chaque niveau, ajoutant une touche de créativité. Notre projet Java vise à développer notre propre version de ce classique, dans un objectif d'apprentissage du java.
Le document débutera par le "Cahier des charges", définissant formellement les objectifs globaux du logiciel. Basés sur une version spécifique du jeu de Sokoban choisie comme référence, ces éléments nous guideront tout au long du projet en délimitant précisément les attentes en termes de fonctionnalités.
Suivant cette introduction, l'"Expression des besoins" se concentrera sur la description du comportement escompté du logiciel, du point de vue de l'utilisateur.
La section "Conception" approfondira l'architecture interne du logiciel du point de vue des concepteurs. Elle expliquera les choix d'implémentation, présentera un diagramme de classes, et détaillera chaque classe, incluant ses responsabilités et attributs.
La partie "Réalisation" détaillera la transformation concrète du modèle UML en code Java. Elle abordera l'organisation temporelle de la réalisation, la répartition des tâches au sein du groupe, et exposera la méthodologie de développement choisie.

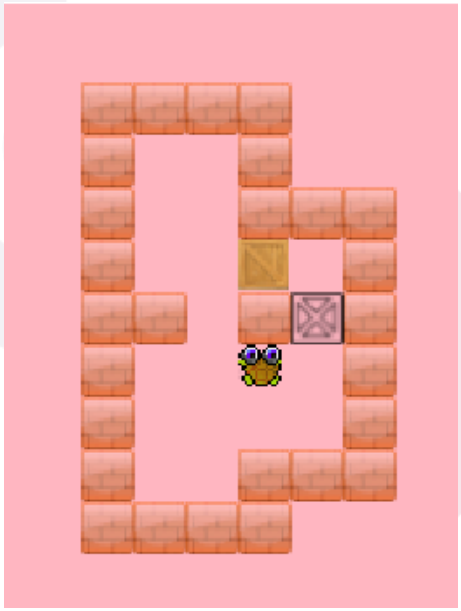

Figure 1: Sokoban

2 Cahier des charges

Le cahier des charges du projet Sokoban en Java vise à créer une application offrant une expérience de jeu immersive et divertissante. L'objectif principal est de reproduire fidèlement le Sokoban classique. L'application devra permettre aux utilisateurs de jouer au Sokoban avec des commandes intuitives, incluant des fonctionnalités telles que le déplacement du personnage principal, le déplacement des boîtes, la résolution de niveaux de complexité croissante, ainsi que la possibilité de recommencer le niveau actuel en appuyant sur la touche "R", tandis que le déplacement se fera de manière intuitive à l'aide des touches fléchées. La conception du logiciel reposera sur une architecture orientée objet, avec un diagramme de classes détaillant les relations entre les différentes entités. La réalisation consistera en la transformation du modèle UML en code Java, avec une organisation temporelle de la réalisation et des tests pour assurer le bon fonctionnement du logiciel.

3 Expressions des besoins

3.1 Fonctionnalités Principales

3.1.1 Déplacement du Joueur

L'utilisateur doit être en mesure de déplacer le joueur (représenté par une tortue) sur le plateau de jeu à l'aide des touches fléchées du clavier. Le déplacement doit être limité par la présence de murs et ne doit pas permettre de traverser les caisses.

3.1.2 Déplacement des Caisses

Le joueur doit pouvoir pousser les caisses sur le plateau pour les déplacer vers les zones cibles. Les caisses ne peuvent pas être traversées, et le joueur doit planifier ses mouvements pour éviter de bloquer le passage. Les caisses ne doivent pas traverser les murs.

3.1.3 Réinitialisation du Niveau

L'utilisateur doit avoir la possibilité de recommencer le niveau actuel à tout moment en appuyant sur une touche dédiée (la touche "R"). Cela permet au joueur de réessayer un niveau en cas d'erreur stratégique. Les caisses et le joueur doivent revenir à leur position initiales au début du niveau.

3.1.4 Changement de Niveau

Une fois qu'un niveau est réussi, le jet doit passer au niveau suivant. La progression à travers les niveaux doit être clairement indiquée, et le jeu doit s'ajuster dynamiquement à la nouvelle configuration du plateau.

3.2 Scénarios d'Utilisation

3.2.1 Déplacement Réussi

L'utilisateur déplace la tortue joueur avec les touches fléchées, réussissant à placer toutes les caisses dans les zones cibles sans bloquer le passage. Le niveau est marqué comme réussi, et le joueur peut passer au niveau suivant.

3.2.2 Recommencer un Niveau

En cas d'erreur stratégique, l'utilisateur appuie sur la touche "R" pour recommencer le niveau actuel sans avoir à relancer l'ensemble du jeu.

3.2.3 Passage au Niveau Suivant

Après avoir terminé un niveau avec succès, le jeu passe lui même au niveau suivant.

4 Conception

4.1 présentation des classes

4.1.1 Présentation de l'architecture UML

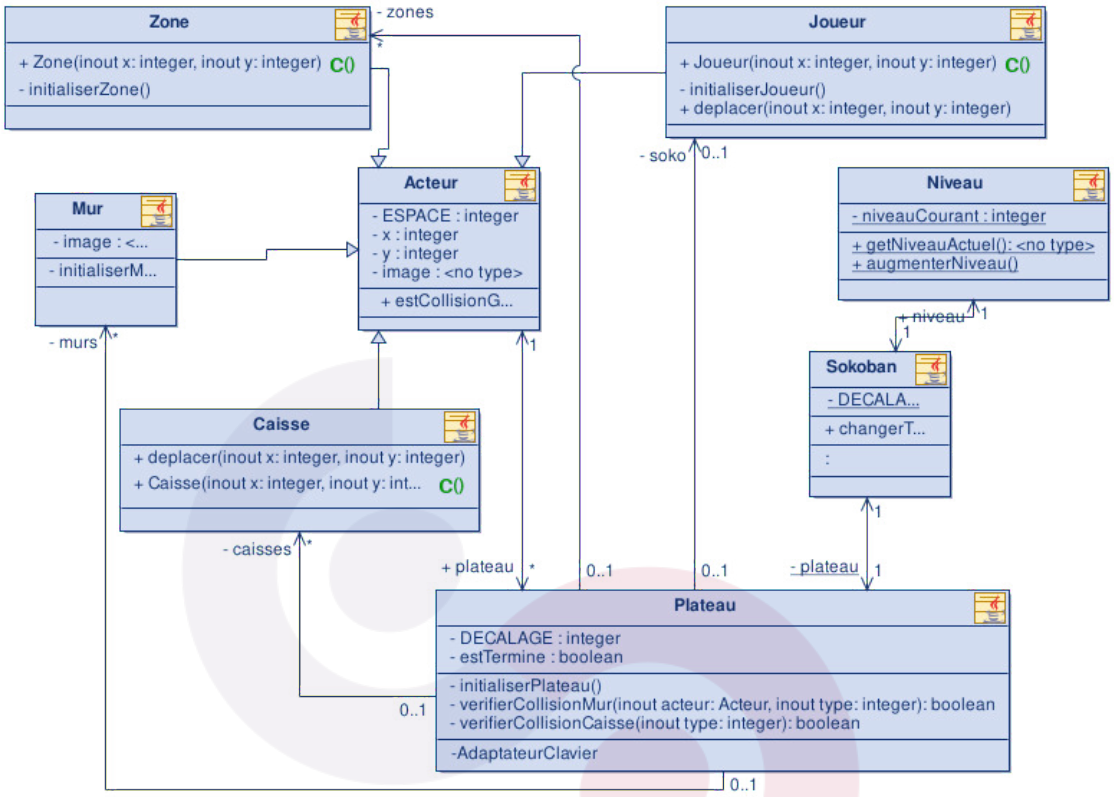

Figure 2: Architecture UML du projet

4.1.2 Class: Plateau

La classe `Plateau` constitue le cœur de l'implémentation du jeu Sokoban, responsable de la gestion complète du plateau de jeu. Étendant la classe `JPanel`, elle assure l'initialisation du plateau, la gestion des collisions, la construction graphique du monde, et la gestion des événements clavier pour permettre au joueur d'interagir avec le jeu. Les éléments clés tels que les murs, les caisses, les zones et le joueur sont stockés dans des listes, et des méthodes sophistiquées sont mises en œuvre pour vérifier et éviter les collisions entre ces éléments. La classe interne `AdaptateurClavier` est dédiée à la gestion des événements clavier, offrant une interaction fluide avec le jeu.
La méthode `estTermine` évalue si le niveau est terminé en comparant les positions des caisses avec celles des zones, tandis que la méthode `recommencerNiveau` permet de redémarrer le niveau actuel en réinitialisant les positions des éléments. La possibilité de passer au niveau suivant est également intégrée avec la méthode `niveauSuivant`.
Il est à noter que le code utilise des images pour représenter le joueur dans différentes directions, nécessitant des ressources graphiques appropriées. En appuyant sur la touche "R", le joueur peut recommencer le niveau, et une fois le jeu terminé, un message de félicitations est affiché. La variable `estTermine` guide le comportement du jeu en indiquant si le niveau est actuellement terminé. L'ensemble de ces fonctionnalités fait de la classe `Plateau` un composant central dans l'expérience de jeu Sokoban.

4.1.3 Class: Acteur

La classe `Acteur` représente la classe parente pour les différents composants du jeu Sokoban, tels que les murs, les caisses, les zones et le joueur. Elle détient les positions `x` et `y` de l'acteur, ainsi que l'image associée. Le constructeur permet l'initialisation des positions de l'acteur.
Les méthodes d'accès telles que `getImage`, `x`, et `y` facilitent l'obtention des informations de l'acteur. Les méthodes de modification `setX` et `setY` permettent d'ajuster les positions de l'acteur.
Les méthodes de détection de collision, telles que `estCollisionGauche`, `estCollisionDroite`, `estCollisionHaut`, et `estCollisionBas`, évaluent si une collision a lieu avec un autre acteur en considérant un espacement spécifié (`ESPACE`).
Cette classe joue un rôle fondamental dans la modélisation des entités du jeu Sokoban, offrant des fonctionnalités génériques de gestion de position et de collision pour les acteurs du jeu.

Class: Zone La classe `Zone` représente une zone dans le jeu Sokoban où les caisses doivent être déplacées. Elle étend la classe `Acteur`, héritant ainsi de ses propriétés et méthodes.
Le constructeur de la classe prend les coordonnées `x` et `y` de la zone en paramètres, appelant le constructeur de la classe parente `Acteur` pour initialiser les positions.
La méthode privée `initialiserZone` est utilisée pour charger l'image de la zone depuis le fichier "zone.png" situé dans le répertoire "src/resources". Cette image est ensuite associée à l'objet `Zone` en utilisant la méthode `setImage`, héritée de la classe `Acteur`.
La classe `Zone` participe à la modélisation des entités spécifiques du jeu Sokoban, en représentant les zones cibles où les caisses doivent être placées.

Class: Joueur La classe `Joueur` représente le joueur dans le jeu Sokoban et étend la classe `Acteur` du même package `com.zetcode`. Le constructeur de la classe prend les coordonnées `x` et `y` du joueur en paramètres, appelant le constructeur de la classe parente `Acteur` pour initialiser les positions.
La méthode privée `initialiserJoueur` est utilisée pour charger l'image du joueur depuis le fichier "joueurHaut.png" situé dans le répertoire "src/resources". Cette image est ensuite associée à l'objet `Joueur` en utilisant la méthode `setImage`, héritée de la classe `Acteur`.
La classe `Joueur` possède également une méthode publique `deplacer` permettant de déplacer le joueur en fonction des coordonnées fournies en paramètres. Cette méthode calcule les nouvelles positions `dx` et `dy` en ajoutant les valeurs `x` et `y` aux positions actuelles, puis met à jour les positions du joueur avec ces nouvelles valeurs.
La classe `Joueur` participe ainsi à la modélisation du personnage principal du jeu Sokoban, avec des méthodes pour initialiser son image et gérer son déplacement.

Class: Caisse La classe `Caisse` représente une caisse dans le jeu Sokoban et étend la classe `Acteur` du même package `com.zetcode`. Le constructeur de la classe prend les coordonnées `x` et `y` de la caisse en paramètres, appelant le constructeur de la classe parente `Acteur` pour initialiser les positions.
La méthode privée `initialiserCaisse` est utilisée pour charger l'image de la caisse depuis le fichier "caisse.png" situé dans le répertoire "src/resources". Cette image est ensuite associée à l'objet `Caisse` en utilisant la méthode `setImage`, héritée de la classe `Acteur`.
La classe `Caisse` possède également une méthode publique `deplacer` permettant de déplacer la caisse en fonction des coordonnées fournies en paramètres. Cette méthode calcule les nouvelles positions `dx` et `dy` en ajoutant les valeurs `x` et `y` aux positions actuelles, puis met à jour les positions de la caisse avec ces nouvelles valeurs.
Ainsi, la classe `Caisse` participe à la modélisation des éléments mobiles du jeu Sokoban, avec des méthodes pour initialiser son image et gérer son déplacement.

Class: Mur La classe `Mur` représente un mur dans le jeu Sokoban et hérite de la classe `Acteur` du package `com.zetcode`. Le constructeur de la classe prend les coordonnées `x` et `y` du mur en paramètres, appelant le constructeur de la classe parente `Acteur` pour initialiser les positions.

La méthode privée `initialiserMur` est utilisée pour charger l'image du mur depuis le fichier "mur.png" situé dans le répertoire "src/resources". Cette image est ensuite associée à l'objet `Mur` en utilisant la méthode `setImage`, héritée de la classe `Acteur`.
Ainsi, la classe `Mur` participe à la modélisation des éléments statiques du jeu Sokoban, fournissant une représentation visuelle des murs sur le plateau de jeu.

4.1.4 Class: Niveau

La classe `Niveau` représente les niveaux du jeu Sokoban et contient une liste de chaînes de caractères représentant différents niveaux du jeu. Chaque niveau est défini sous forme de chaîne, où chaque caractère représente un élément du plateau de jeu (mur, caisse, zone, joueur).
La méthode `getNiveauActuel()` renvoie le niveau actuel sous forme de chaîne de caractères. La méthode `getNiveauCourant()` renvoie l'indice du niveau courant dans la liste. La méthode `augmenterNiveau()` permet d'augmenter l'indice du niveau courant, permettant ainsi de passer au niveau suivant.
La classe `Niveau` joue un rôle crucial dans la gestion des niveaux du jeu Sokoban, fournissant une structure organisée pour définir et manipuler les différents niveaux du jeu.

4.1.5 Class: Sokoban

La classe `Sokoban` est la classe principale du jeu Sokoban et hérite de `JFrame` pour gérer l'affichage de l'interface utilisateur. Elle contient une instance de la classe `Plateau`, qui représente le plateau de jeu.
Le constructeur de la classe `Sokoban` initialise l'interface utilisateur en créant une instance de `Plateau` et en l'ajoutant à la fenêtre principale. La méthode `initialiserUI()` configure le titre de la fenêtre, sa taille et sa position.
La méthode `changerTaille()` est utilisée pour ajuster dynamiquement la taille de la fenêtre en fonction du niveau courant. Elle utilise un thread séparé pour éviter de bloquer l'interface utilisateur pendant les ajustements de taille.
La méthode `main` est la méthode principale du programme, elle crée une instance de `Sokoban`, rend la fenêtre principale visible, et lance un thread pour ajuster dynamiquement la taille de la fenêtre en fonction du niveau courant.

5 Réalisation

5.1 répartition des tâches

Conception UML sur Modelio Nous avons partagé la conception du diagramme UML

Développement Java sur Eclipse Suite à la conception du diagramme UML, nous avons conjointement travaillé sur le développement Java, utilisant l'environnement Eclipse. nous avons implémenté les méthodes et fonctionnalités clés du jeu, allant de la logique du plateau de jeu à la gestion des niveaux, en passant par les interactions clavier.

Class: Plateau - `estTermine`: Étienne a implémenté la logique pour évaluer si le niveau est terminé. - `recommencerNiveau`: Étienne a développé la fonction pour réinitialiser les positions des éléments. - `niveauSuivant`: Etienne a intégré la possibilité de passer au niveau suivant. - Construction graphique du monde: Etienne a pris en charge la création graphique du monde en utilisant des images pour représenter le joueur dans différentes directions. - Gestion des événements clavier (`AdaptateurClavier`): Milo a développé la classe interne dédiée à la gestion des événements clavier.

Class: Acteur - Méthodes d'accès (`getImage`, `x`, `y`): Étienne a mis en place les méthodes pour obtenir les informations de l'acteur. - Méthodes de modification (`setX`, `setY`): Étienne a développé les méthodes pour ajuster les positions de l'acteur. - Méthodes de détection de collision (`estCollisionGauche`, `estCollisionDroite`, `estCollisionHaut`, `estCollisionBas`): Étienne a conçu les méthodes pour évaluer les collisions avec d'autres acteurs.

Class: Zone / Joueur / Caisse / Mur - Milo c'est occupé de mettre en place les différentes métodes de ces class en partant de la classe parent acteur.

Class: Sokoban - `main`: Milo a mis en place la méthode principale du programme, créant une instance de `Sokoban` et lançant le thread pour ajuster la taille de la fenêtre en fonction du niveau courant. - `changerTaille`: Milo a implémenté la fonction pour ajuster dynamiquement la taille de la fenêtre en fonction du niveau courant en utilisant un thread séparé.

Class: Niveau - Méthodes `getNiveauActuel`, `getNiveauCourant`, `augmenterNiveau`: Milo a travaillé sur ces méthodes pour gérer les niveaux du jeu Sokoban.

Ajustement Dynamique de la Taille de la Fenêtre - Étienne: Idées et suggestions. - Milo: Implémentation de la solution dans le code, notamment la création d'un thread séparé.
Cette approche collaborative a assuré une progression efficace du projet, alliant la rigueur de la conception à la mise en œuvre pratique, avec une cohésion globale dans la réalisation du jeu Sokoban en Java.

5.2 interface graphique

Lors de la conceptualisation de notre version du jeu Sokoban, notre objectif était d'injecter une dose supplémentaire de plaisir et de divertissement, tant pour les joueurs que pour notre propre amusement. L'un des aspects clés de cette personnalisation résidait dans la création de notre propre interface graphique, où chaque composant du jeu arborait un design simple mais amusant à réaliser et dont nous pourrions être fier.
Nous avons décidé de mettre en œuvre des designs simple et épurés pour tous les éléments du jeu, optant pour une approche visuelle qui serait à la fois attrayante et accessible. Dans cet esprit, chaque élément, des caisses aux murs, a été dessiné simplement. ajoutant ainsi une esthétique soignée à l'expérience globale du joueur.
Un choix inhabituel et ludique que nous avons fait a été de représenter le protagoniste du jeu, le joueur, sous la forme d'une tortue. Cette décision a été motivée par notre désir d'apporter une touche de fantaisie et d'originalité au jeu. La tortue, souvent associée à la tranquillité et à une allure délibérée, a ajouté une dimension ludique à l'expérience du joueur, créant une connexion visuelle unique avec le personnage principal.

Cette personnalisation visuelle visait à créer une atmosphère engageante, où les joueurs pourraient apprécier non seulement les défis du jeu, mais aussi son esthétique charmante et originale. Ainsi, notre choix délibéré de concevoir notre propre interface graphique a permis de donner une identité visuelle distinctive à notre version de Sokoban, ajoutant une couche de plaisir visuel à chaque mouvement de la tortue joueur.

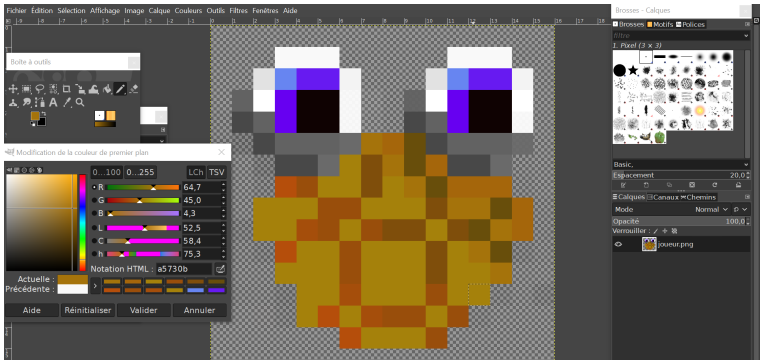

Figure 3: design de la tortue en cours de réalisation

Figure 4: design final de la tortue

5.3 difficultés rencontrées

5.3.1 Gestion Dynamique de la Taille de la Fenêtre

Au cours du développement de notre version personnalisée du jeu Sokoban, nous avons été confrontés à un défi majeur lié à la gestion de la taille de la fenêtre lorsque le joueur passe d'un niveau à un autre. Dans la première version, où un seul niveau était disponible, la taille de la fenêtre était bien encapsulée et adaptée à ce niveau spécifique.
Cependant, lors de l'extension du jeu pour inclure plusieurs niveaux, nous avons constaté que la taille de la fenêtre ne se mettait pas à jour automatiquement pour s'ajuster aux nouvelles dimensions du plateau de jeu. Ce problème découlait de l'encapsulation efficace utilisée dans notre architecture, qui rendait difficile la modification dynamique de la taille de la fenêtre en réponse à un changement de niveau.
Pour surmonter cette limitation sans compromettre l'encapsulation existante, nous avons opté pour une solution moins optimisée. Nous avons introduit un nouveau thread qui actualise en permanence la taille de la fenêtre. Ce thread s'assure que la fenêtre est redimensionnée de manière adéquate chaque fois que le joueur passe à un niveau suivant. Bien que cette approche ait résolu le problème, elle est moins efficace en termes de ressources système.
Idéalement, la meilleure solution aurait été de repenser l'architecture du jeu de manière à permettre une mise à jour plus facile et optimisée de la taille de la fenêtre en fonction des spécificités de chaque niveau. Cependant,

l'encapsulation préexistante a présenté un défi majeur pour cette modification.
Cette expérience souligne l'importance d'une conception flexible dès le début du développement d'un jeu, afin d'anticiper les évolutions futures et de permettre des ajustements efficaces. Malgré la suboptimalité de notre solution, elle a permis de maintenir la jouabilité du jeu tout en travaillant dans les contraintes de l'architecture existante.

6 Conclusion

En conclusion, notre jeu Sokoban en Java présente une fonctionnalité de base solide, permettant aux joueurs de naviguer à travers les niveaux avec succès. Cependant, l'achèvement total du jeu n'est pas encore parfaitement implémenté, et des ajustements sont nécessaires pour améliorer cette expérience.
Ce projet a été une expérience d'apprentissage précieuse en matière de programmation Java. Il a souligné l'importance de la planification réaliste des défis, mettant en lumière les difficultés rencontrées lors de la mise en œuvre de nombreuses fonctionnalités ambitieuses dans des délais stricts. Les ajustements des exigences en cours de projet ont été nécessaires pour assurer la livraison dans les délais impartis.
L'une des principales leçons tirées est qu'il est essentiel de trouver un équilibre entre l'ambition du projet et la réalité du calendrier. Pour les futures itérations du jeu Sokoban, nous envisageons plusieurs améliorations qui pourraient enrichir l'expérience utilisateur. Parmi celles-ci figurent l'ajout de nouveaux niveaux pour prolonger le plaisir du jeu, l'intégration d'un bouton dédié pour relancer un niveau, et la mise en place d'un menu défilant pour sélectionner facilement le niveau désiré.
En outre, nous envisageons d'implémenter un système de score, lié soit au temps mis pour terminer un niveau, soit au nombre de coups nécessaires pour atteindre l'objectif. Ces fonctionnalités apporteraient une dimension compétitive au jeu, stimulant l'engagement des joueurs et ajoutant une couche stratégique supplémentaire.
En résumé, malgré les défis rencontrés, ce projet a été une opportunité d'acquérir des compétences précieuses et de poser les bases d'une version améliorée de notre jeu Sokoban. Nous sommes impatients de continuer à développer et à améliorer cette expérience ludique dans le futur.