Mini Projet : Sokoban ®

Dossier de conception

Ce document présente le jeu de plateforme 2D Sokoban développé dans le cadre d'un projet libre pour l'école d'ingénieur 3il Rodez

Mourgues Xavier – Lormeau Axel
12/22/2013

Mourgues Xavier
Lormeau Axel

Introduction

Sokoban est un jeu vidéo de réflexion dont la première version a été mise au point par le japonais Hiroyuki Imabayashi dans les années 80. Signifiant « Magasinier » en japonais, ce jeu a remporté plusieurs concours de programmation au fil du temps.

C'est une bonne entrée en matière de programmation de jeux vidéo puisqu'il comporte plusieurs spécificités liées à cette branche de développement tel que la gestion des collisions.

Après avoir rappelé les règles du jeu, ce document présente la conception mise au point pour réaliser l'application. La phase de réalisation de l'application sera également présentée (outils de développement, organisation du code source) et illustrées de captures d'écrans. Enfin sera présenté l'application avec d'une part le package de niveaux standards du jeu et d'autre part l'éditeur graphique de niveaux.

Mourgues Xavier
Lormeau Axel

1. Sokoban

1.1. Règles du jeu

Le but du jeu de Sokoban est de ramener toutes les caisses sur leur emplacement final. Le personnage ne peut que pousser les caisses et non les tirer vers lui. Le niveau est fini quand toutes les caisses du niveau sont sur un emplacement final. Une fois un niveau terminé, l'accès au niveau suivant est ouvert mais attention, la difficulté est croissante et peut atteindre une complexité quasiment infinie.

La difficulté principale de Sokoban est donc de ne pas se bloquer soi-même, pour cela, il est nécessaire d'éviter plusieurs dangers :

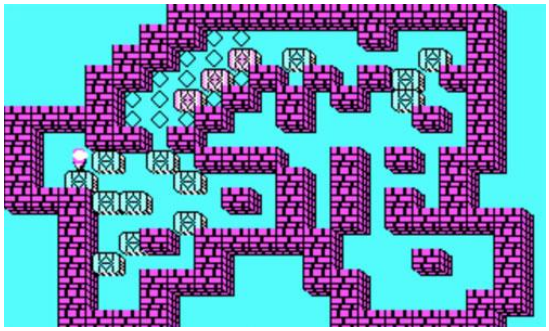
- mettre une caisse dans un angle fera perdre la partie à coup sûr (on ne peut pas tirer les caisses)
- aligner deux caisses fera souvent perdre la partie (on ne peut pousser qu'une seule caisse à la fois)
- faire attention aux positions des emplacements finaux car souvent la place pour manœuvrer les caisses impose un ordre précis pour l'occupation de ces emplacements

1.2. Anecdotes sur la complexité de Sokoban

La difficulté du Sokoban provient de son facteur de branchement (comparable à celui des échecs et du jeu de go), et également de la profondeur de son arbre de recherche. Ainsi, certains niveaux nécessitent plusieurs milliers de déplacements et plus de 1 000 « poussées ».

Les joueurs humains expérimentés se basent principalement sur des méthodes heuristiques (solution non optimale mais réalisable au coup par coup). Ils sont généralement capables d'éliminer rapidement les tactiques de jeu inutiles ou redondantes, et ils identifient les dispositions du jeu et les sous-buts, ce qui leur permet d'élaguer sensiblement l'étendue des recherches.

Plusieurs solveurs automatiques existent à l'heure actuelle cependant il existe encore aujourd'hui quelques niveaux (conçus spécialement à cet effet) qui tiennent en échec ces solveurs.



Par exemple : ce niveau est réalisé par les solveurs actuels (et par les « meilleurs » joueurs humain) en 1500 coups environ.

Figure 1 : Niveau 50 premier Sokoban

Mourgues Xavier
Lormeau Axel

2. Conception

2.1. Application

La description des classes est la suivante :

- Jsokoban : C'est la classe dite « point d'entrée » de l'application. Elle permet les niveaux au départ de l'application et de lancer l'interface.
- LevelEditor : Cette classe est une classe comportant des méthodes statiques qui gèrent la création de niveau par programmation, ainsi que les méthodes permettant de lire et sauvegarder les niveaux fabriqués par l'utilisateur dans des fichiers.
- Level : Classe qui contient toutes les informations du niveau (sa taille, la position du personnage, et la matrice de case). Elle permet aussi de gérer les déplacements du personnage et des blocs en gérant les collisions.
- Case : Cette classe représente une case d'un niveau, elle contient son type (mur, personnage, bloc, etc…) et implémente sa méthode de dessin pour s'afficher à l'écran.
- CaseType : Enumération des type de cases (bloc, mur, personnage, etc…)
- FenetreUI : Interface Utilisateur principale de l'application gérant le basculement (contrôleur) entre le menu, l'interface des niveaux et l'interface d'édition de niveau.
- LevelUI : C'est la classe qui gère l'affichage d'un niveau à l'écran. Elle a un attribut « level » et s'occupe de l'afficher en fonction de la taille de l'écran, de la taille du niveau et d'autres paramètres.
- MenuUI : Cette classe gère l'affichage de la liste des niveaux pour la navigation entre les menus.
- LevelEditorUI : Cette classe gère l'affichage de l'édition de niveau graphique.
- LevelEditorCreateDialog, ListCaseTypeUI, MarioLabel, MarioTextField, MenuBarUI : Ces classes sont des widget utilisés dans les précédentes classes d'interface utilisateur et permettent de séparer le code pour des raisons de maintenabilité.

Diagramme des classes de l'application disponible en annexes.

2.2. Gestion des collisions

Lorsque l'on programme un jeu vidéo, on s'intéresse aux collisions d'objets. Est-ce-que mon personnage touche un mur ? Est-ce qu'il touche une caisse? Tout ceci, ce sont des tests de collision. Les collisions sont un aspect fondamental de tout jeu d'action ou d'animation en général. Nous considèrerons une ou plusieurs fonctions « collisions » qui testent la faisabilité des mouvements exécutés dans le jeu. Et ces fonctions renverront simplement un booléen, ayant donc pour valeur true ou false, selon si ça touche ou non. Toute la gestion des collisions s'appuiera sur ces fonctions.

Dans le cadre de notre application, nous avons fait une fonction par mouvement possible (NORTH / SOUTH / EAST / WEST).

Note : nous aurions aussi pu faire une seule fonction prenant en paramètre la direction demandée, mais cette approche nous a paru plus difficile à maintenir et à faire évoluer. Notre solution implique de la redondance de code, mais pourra facilement être améliorée.

Voici un exemple de la gestion des collisions dans notre application (lorsque Mario essai d'aller vers la droite !).

```
    /**
     * Gère les collisions pour savoir si le personnage peut bouger à droite
     * @return true si le déplacement est possible, false sinon
     */
    private boolean checkMoveEast() {
        // Verification qu'on est dans les limites du tableau
        try {
            checkBoundaries(characterPosX + 1, characterPosY);
        } catch (ArrayIndexOutOfBoundsException e) {
            return false;
        }
        // Vérification qu'il n'y ai pas de mur
        if (cases[characterPosX + 1][characterPosY].getType().equals(
                CaseType.WALL)) {
            return false;
        }
        // Existance pierre ?
        if (cases[characterPosX + 1][characterPosY].getType().equals(
                CaseType.STONE)
                || cases[characterPosX + 1][characterPosY].getType().equals(
                        CaseType.FILLED_HOLE)) {
        // On verifie qu'elle va pas etre en dehors du tableau
            try {
                checkBoundaries(characterPosX + 2, characterPosY);
            } catch (ArrayIndexOutOfBoundsException e) {
                return false;
            }
            // On vérifie qu'il n'y ai pas une autre pierre ou un mur derriere.
            if (cases[characterPosX + 2][characterPosY].getType().equals(
                    CaseType.WALL)
                    || cases[characterPosX + 2][characterPosY].getType()
                            .equals(CaseType.STONE)
                    || cases[characterPosX + 2][characterPosY].getType()
                            .equals(CaseType.FILLED_HOLE)) {
                return false;
            }
        }
        return true;
    }
```

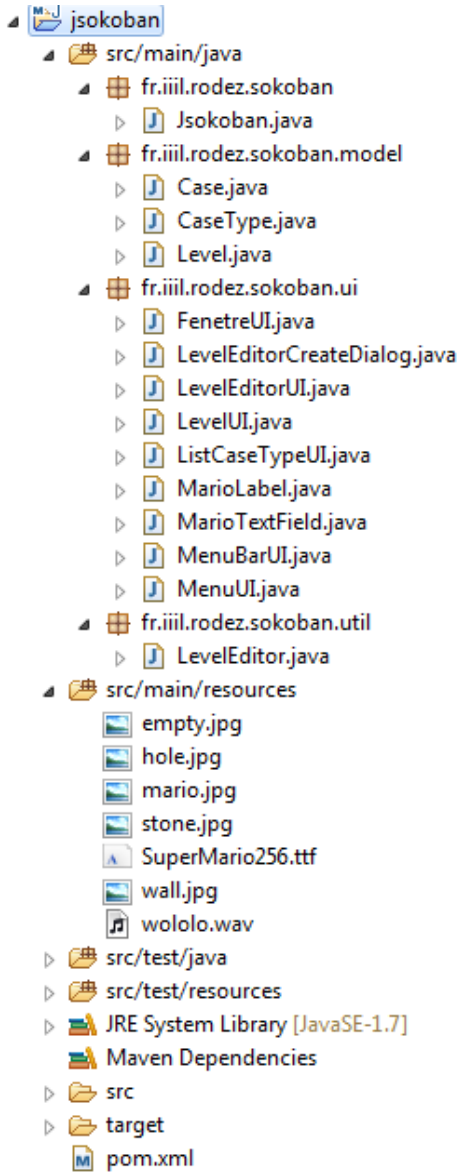
3. Réalisation

3.1. Arborescence du code source

Description de l'arborescence :

- Le package « fr.iiil.rodez.sokoban » est le package principale de l'application. Il contient la classe dite « point d'entrée » qui lance le jeu.
- Le package « fr.iiil.rodez.sokoban.model » contient les objets dit « POJO (Plain Old Java Object) » qui gère les données de l'application telle que les niveaux et leurs cases.
- Le package « fr.iiil.rodez.sokoban.ui » contient les classes destinées à gérer l'interface.
- Le package « fr.iiil.rodez.sokoban.util » contient les classes dites « utilitaires » permettant dans notre cas de créer des niveaux par programmation.
- Le dossier « src/main/resources » contient toutes les ressources nécessaires à l'application telles que les images représentant chaque type de case, la police « Mario » et le son joué lors de la complétion d'un niveau.

Ci-après les fichiers sources de l'application.



3.2. Outils utilisés

Eclipse : environnement de développement intégré pour coder en Java.

SVN : utilitaire de versionning. Indispensable pour une application codée en groupe.

Skype : pour que le travail avance pendant les vacances de Noël.

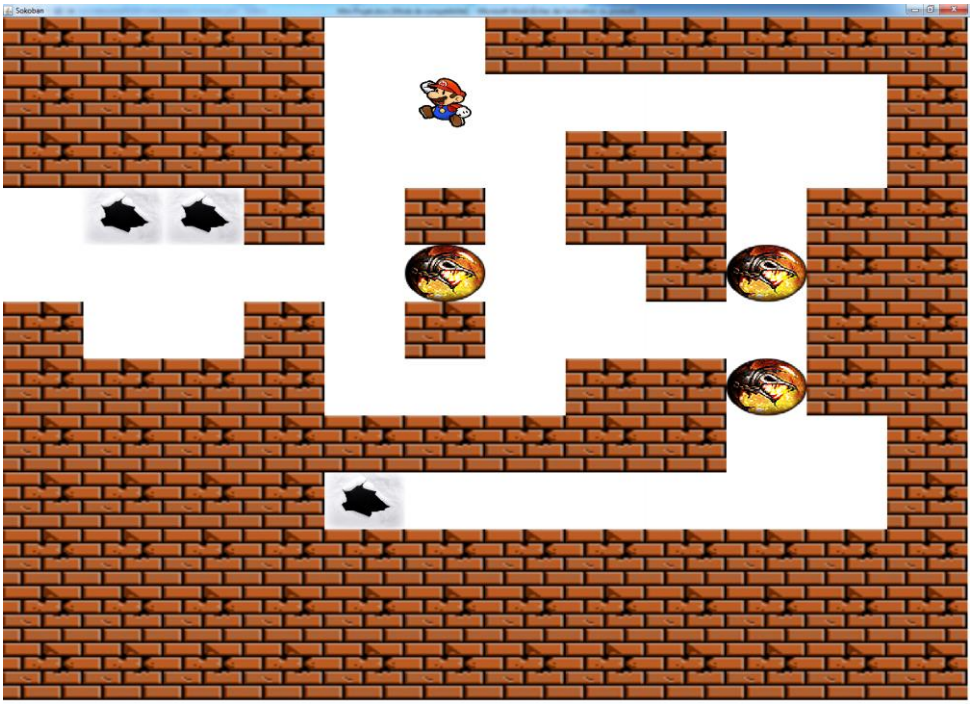
4. Présentation de l'application

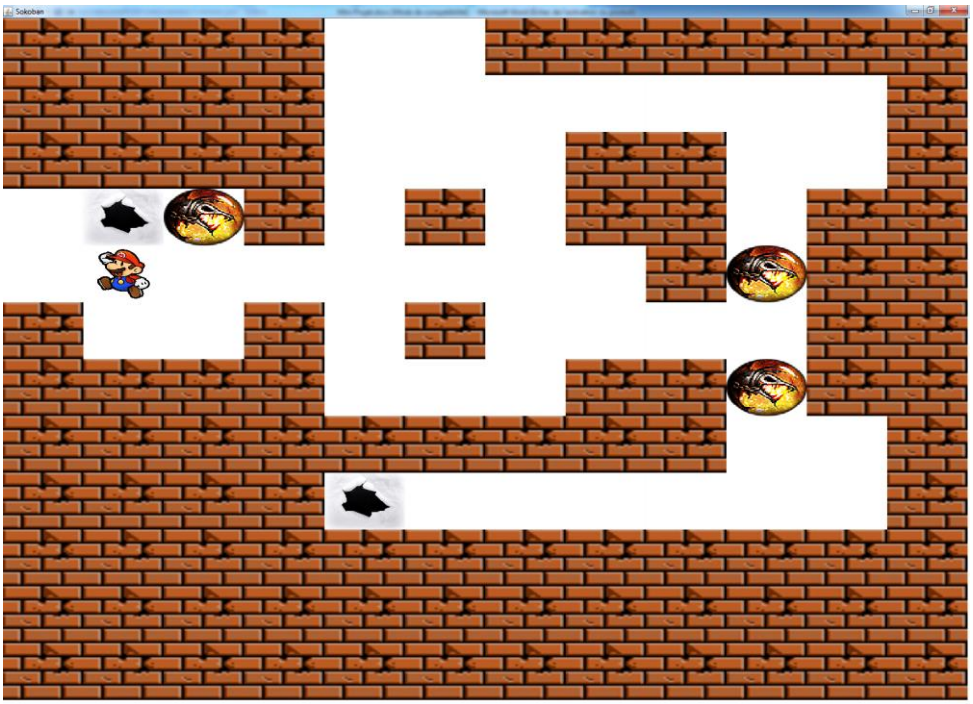
Le jeu se présente sous la forme d'une carte 2D avec plusieurs éléments :

- Mario : c'est le magasinier, il pousse les pièces de dragon sur les trous noirs.
- Pièces de dragon : ce sont l'équivalent des caisses dans la version originale du jeu.
- Trous noirs : ce sont les emplacements qui doivent accueillir les pièces de dragon.

Le niveau est terminé quand toutes les pièces de dragons sont positionnées sur des trous noirs.

4.1. La cartes de base





4.2. Edition de niveaux

Dans l'application, l'édition (entendre création) de niveau peut être effectuée de plusieurs façons. La première est faite par programmation, la seconde par un éditeur graphique.

4.2.1. Programmation des niveaux

Quelques niveaux d'exemples sont programmés en dur dans l'application, ils ne sont pas stockés sous forme de fichier texte.

Nous avons choisi cette option car aujourd'hui la mémoire d'un ordinateur est quasiment infinie (surtout quand il s'agit de faire tourner une application aussi peu gourmande que la nôtre). Nous avons donc jugé inutile de mettre en place une architecture de lecture / écriture dans des fichiers textes pour représenter nos niveaux.

En plus des textures que nous avons choisis, c'est sur ce point que notre version de Sokoban diffère le plus des versions existantes.

Le bémol de ce choix : nous ne pouvons échanger des niveaux avec la communauté au format .skb (qui est le format utilisé pour échanger des fichiers textes entre joueurs de Sokoban.

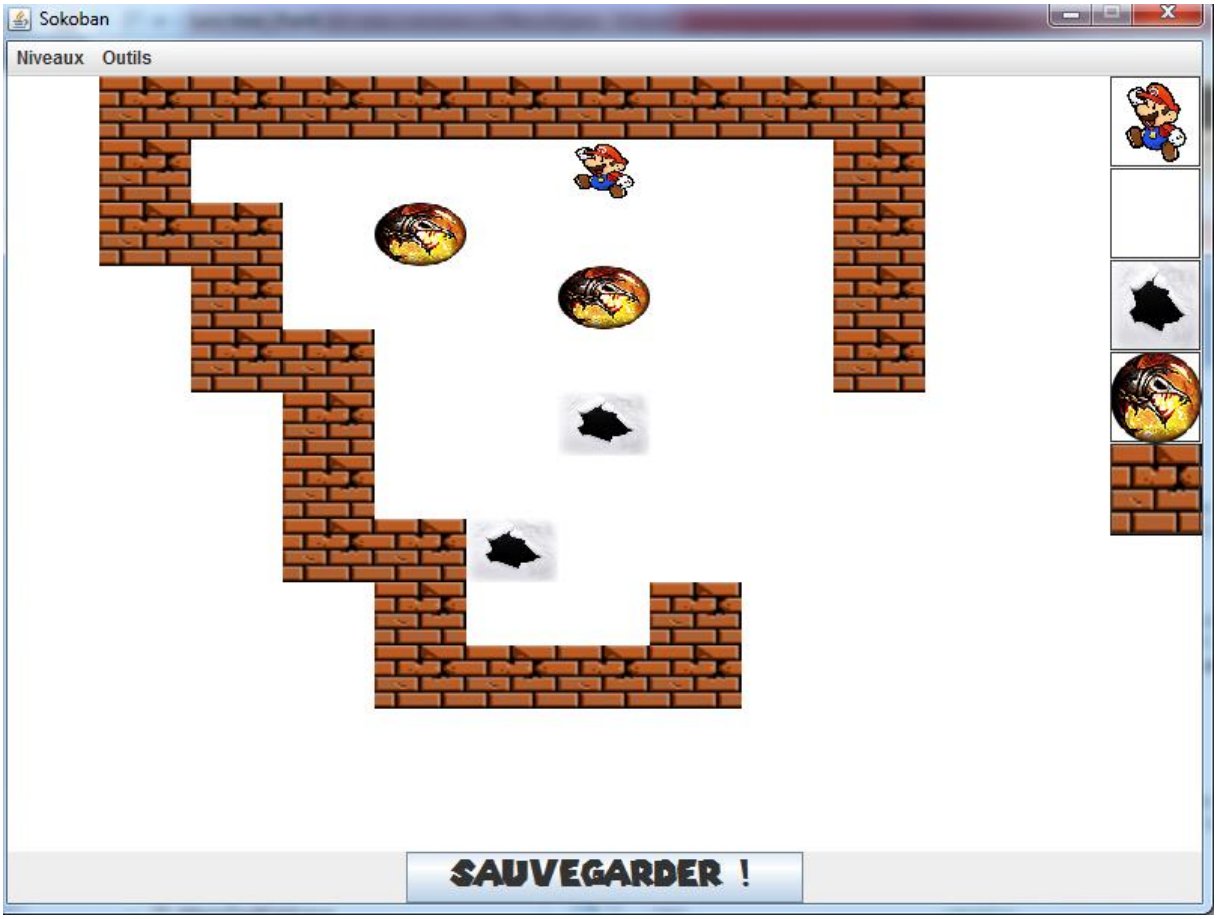
On pourrait cependant imaginer un module qui convertie notre représentation des niveaux au format .skb.

Voici un exemple de création de niveau par programmation.

```
/**
 * Céation du niveau 1 du jeu Sokoban
 * @return Le niveau créé
 */
public static Level createLevel1(){
	Level level = new Level(12, 12, 5, 1);
	level.addWall (0, 0 );
	level.addWall (1, 0 );
	level.addWall (2, 0 );
	level.addWall (9,11 );
	level.addWall (10,11);
	level.addWall (11,11);
	level.addHole (1, 3 );
	level.addHole (2, 3 );
	level.addHole (4, 8 );
	level.addStone (5, 4);
	level.addStone (9, 4);
	level.addStone (9, 6);
	return level;
}
```

4.2.2. Editeur graphique.

Nous rendant compte que la programmation des niveaux en dur est vite rébarbative et moins intéressante que de découvrir de nouveau point technique au sein de l'application, nous avons décidé de se donner un challenge en réalisant un éditeur de niveau graphique.



L'éditeur de niveau graphique nous permet ainsi de cliquer sur un type de case dans la liste de droite, et ainsi venir la placée sur le niveau en cours de création.

Lors de la sauvegarde, un fichier (extension .lvl) est créé dans le répertoire de l'application et contient la version « sérialisée » de l'objet « Level ». Ce qui nous permet de le relire lorsque le jeu est relancé.

Conclusion

Ce petit projet de développement libre nous a permis d'utiliser les concepts vus en cour de programmation / programmation avancé.

Nous avons pris la mesure de la difficulté de coder une application client lourd en groupe.

Nous avons également abordé de tous nouveaux concepts intimement liés à la programmation d'un jeu vidéo : la gestion des collisions. En plus d'être la partie la plus technique de l'application, c'est le cœur de tous les jeux vidéo où un personnage évolue dans un environnement (2D ou 3D).

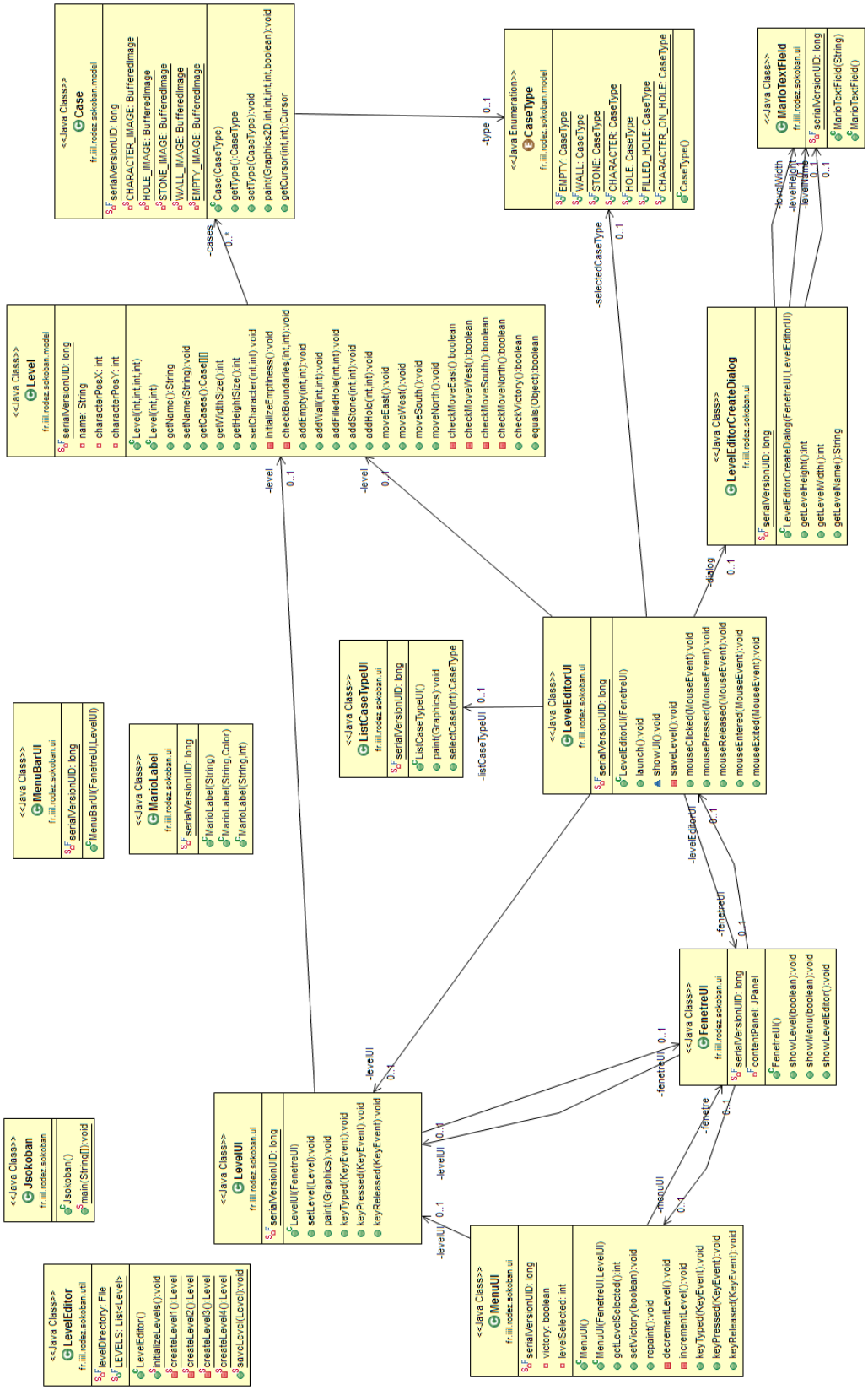
Nous avons dans nos recherches croisé des exemples de gestion des collisions en 3D et nous nous sommes rendu compte qu'en rajoutant une dimension au problème, la complexité explose complètement : Sokoban 3D aurait été une toute autre histoire !

Enfin, avoir codé ce projet en Java quelques semaines avant de passer la certification aura eu le mérite de nous faire revoir plusieurs concept élémentaire de ce langage de programmation.

Mourgues Xavier
Lormeau Axel

Annexes

Diagramme des classes



Mourgues Xavier
Lormeau Axel