



TER FMIN200

—

Développement d'un jeu de type Bomberman en réseau  
sous Android et iOS

BONVILA Olivier

COUSEIN Kilian  
TARDIEU Benjamin

PITOT Ludovic

Nous tenons à remercier tout particulièrement :

- Notre tuteur : M. Deruelle pour ses conseils et son soutien technique.
- Les différents éditeurs des logiciels qui nous ont servis pour le développement mais aussi pour l'écriture de ce rapport.
- La faculté des Sciences pour les nombreuses connaissances qu'elle nous a permis d'acquérir, bien évidemment transmises par les professeurs mais également par les échanges inter-étudiants.

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Présentation</b>	<b>5</b>
2.1	Projet . . . . .	5
2.2	Plate-formes de développement . . . . .	6
2.2.1	Android . . . . .	6
2.2.2	iOS . . . . .	6
2.3	Organisation . . . . .	7
<b>3</b>	<b>Analyse</b>	<b>9</b>
3.1	Cahier des charges . . . . .	9
3.1.1	Menus . . . . .	9
3.1.2	Jeu . . . . .	9
3.1.3	Serveur . . . . .	10
3.2	Modélisation . . . . .	10
3.2.1	Général . . . . .	10
3.2.2	Menus . . . . .	11
3.2.3	Editeur de carte . . . . .	14
3.2.4	Jeu . . . . .	15
3.2.5	Réseau . . . . .	29
3.3	Différences entre Android et iOS . . . . .	36
<b>4</b>	<b>Développement</b>	<b>37</b>
4.1	Mobile . . . . .	37
4.1.1	Menus . . . . .	37
4.1.2	Editeur de carte . . . . .	40
4.1.3	Jeu . . . . .	43
4.2	Serveur . . . . .	50
4.2.1	Servlet . . . . .	50
4.2.2	BDD . . . . .	51

<b>5</b>	<b>Manuel d'utilisation</b>	<b>53</b>
5.1	Menus . . . . .	53
5.1.1	Premier lancement . . . . .	53
5.1.2	Menu . . . . .	53
5.1.3	Jeu local <b>1</b> . . . . .	54
5.1.4	Jeu multi <b>2</b> . . . . .	55
5.1.5	Editeur de map <b>3</b> . . . . .	56
5.1.6	Options <b>4</b> . . . . .	57
5.2	Jeu . . . . .	58
5.3	Editeur . . . . .	58
<b>6</b>	<b>Discussion</b>	<b>64</b>
6.1	Problèmes . . . . .	64
6.1.1	Android . . . . .	64
6.1.2	iOS . . . . .	64
6.2	Améliorations . . . . .	65
6.2.1	Jeu . . . . .	65
6.2.2	Serveur . . . . .	65
<b>7</b>	<b>Conclusion</b>	<b>66</b>
	description=Le langage Java est un langage de programmation informatique orienté objet créé par James Gosling et Patrick Naughton	

# Chapitre 1

## Introduction

Nous sommes amenés, dans le cadre de notre première année en master informatique à la faculté des Sciences de Montpellier, à travailler sur l'élaboration complète d'un projet, de son analyse à la conception puis à sa programmation. Tout au long de ce deuxième semestre, le projet nous permet de comprendre quelles sont les phases de développement d'un projet et comment celui-ci doit être conduit. Nous pouvons par ailleurs mesurer notre capacité à réagir face à des problèmes en nous impliquant dans ce projet. Le projet que nous avons choisi est de réaliser sur deux systèmes d'exploitations de téléphone différents un jeu de type Bomberman.

Ce projet nous permet de mettre en application l'enseignement que nous avons acquis tout au long de ce semestre. Ce dernier étant spécialisé pour chaque étudiant, il nous a permis de travailler en collaboration avec des personnes aux capacités différentes et d'ainsi mettre nos connaissances en commun. Mais il permet aussi de se faire une idée du travail demandé dans le monde des entreprises et d'ainsi nous préparer à notre stage que nous devrons réaliser l'an prochain.

Afin de comprendre la démarche que nous avons utilisée pour mener ce projet à son terme, notre rapport se compose de trois grandes parties :

Tout d'abord, dans une première partie nous présentons le cadre général de notre projet, c'est-à-dire le groupe de projet, les outils de développement ainsi que la manière dont nous nous sommes organisés pour mettre à bien le développement de ce dernier. Ensuite dans une seconde partie, nous présenterons le travail d'analyse que nous avons effectué pour pouvoir ensuite, dans une troisième partie, expliquer le développement du projet.

## Chapitre 2

# Présentation

En tant que première année de master, nous avons dû développer un projet tout au long de ce semestre. Chaque année une liste de projets est présentée aux étudiants. Tous les étudiants doivent former des groupes pour développer l'un des projets choisis. Puis ces derniers sont associés à un tuteur. Etant donné que nous voulions développer notre propre projet, nous avons formé un groupe de quatre personnes : Olivier BONVILA, Kilian COUSEIN, Ludovic PITIOT et Benjamin TARDIEU. Ensuite, pour pouvoir valider notre sujet, nous avons dû trouver un tuteur voulant bien s'occuper de notre tutelle. Mr Laurent Deruelle a gentiment accepté de s'occuper de notre groupe, mais pour que ce dernier soit entièrement validé, des modifications du sujet ont été nécessaires.

### 2.1 Projet

Etant donné que nous voulions principalement créer une application iPhone, nous avons demandé à développer notre propre projet. Un jeu de type bomberman. Mais ce dernier a dû subir des modifications pour être approuvé. Nous avons donc dû développer le jeu sous iPhone et sous Android. Le but étant de comparer la différence de développement entre les deux types de téléphones et de développer des fonctionnalités en rapport avec les parcours d'enseignement que nous avons choisis ce semestre.

Un jeu de type Bomberman est un jeu d'action dont le but est simple. Le joueur incarne un poseur de bombes et ce dernier doit faire exploser ses ennemis pour pouvoir gagner la partie. A ceci s'ajoute toute une liste de bonus et malus permettant de complexifier le jeu et de le rendre plus amusant.

Pour pouvoir rapprocher le développement de cette application à notre parcours d'enseignement. Nous avons choisi de développer un mode solitaire qui permettra de jouer contre une ou plusieurs intelligences artificielles qui est en rapport avec le cursus Ingénierie de l'Intelligence Artificielle (I2A) qui nous est enseigné. Ensuite pour ce qui est du parcours Combinatoire, Algorithmique, Sécurité et Administration Réseau (CASAR), nous avons décidé d'implémenter un mode multijoueur qui permettra à plusieurs joueurs

connectés en Wi-Fi de jouer en réseau grâce à un serveur qui combinera un serveur d'applications et un serveur web. Puis pour ce qui est du parcours Données, Interaction et Web (DIWEB), la partie serveur permettra de palier à l'enseignement de ce dernier parcours.

## 2.2 Plate-formes de développement

### 2.2.1 Android

Android est un système d'exploitation open source fournis par Google et développé par la startup Android. Il est majoritairement utilisé sur smartphones, PDA et autres terminaux mobiles mais aussi sur tablettes graphiques et même sur certains téléviseurs.

Disponible via une Licence Apache version 2, Android est fondé sur un noyau Linux, il comporte une interface spécifique, développée en Java, les programmes sont exécutés via un interpréteur JIT, toutefois il est possible de passer outre cette interface en programmant ses applications en ANSI ou C++, mais le travail de portabilité en sera plus important.

Il nous est aussi permis d'utiliser du XML principalement pour les interfaces graphiques ou encore pour la portabilité des données entre chaque type d'architecture.

Sa première version date du 5 novembre 2007, actuellement la dernière version disponible est la 3.1 Honeycomb (Rayon de miel) mais nous utiliserons ici par soucis de compatibilité la version 2.0 dans le but de toucher un maximum de périphériques.

Afin de pouvoir développer nos propres applications, Android met à disposition un SDK ainsi qu'un greffon pour Eclipse (ADT) afin de simplifier l'utilisation de celui-ci.

Le SDK d'Android comporte les outils de base pour développer une application en Java, le développement en ANSI et C++ nécessitant d'utiliser le NDK. Il se compose des bibliothèques principales d'Android, de divers exemples d'applications et surtout d'un émulateur de terminal Android qui permet de pouvoir tester directement ses applications directement sur Linux, Mac ou Windows sans nécessairement avoir besoin d'un terminal mis à part pour OpenGL-ES 2.0 qui n'est pas encore pris en charge par celui-ci, sinon le SDK permet très bien de tester ses applications en temps réel sur tout périphériques tournant sur Android que ce soit en liaison directe avec un ordinateur ou en récupérant l'archive de l'application au format APK.

Le format APK est une variante du format JAR qui est utilisée pour la distribution et l'installation de composants regroupés sur le système d'exploitation Android.

### 2.2.2 iOS

Apple est une entreprise multinationale américaine qui a été créée le 1<sup>er</sup> avril 1976 à Cupertino (Californie) par Steve Jobs et Steve Wozniak. Elle a pour but de concevoir et vendre des produits électroniques grand public, des ordinateurs personnels et des logiciels informatiques. Parmi les produits les plus répandus de l'entreprise se trouvent les ordinateurs Macintosh (1984), l'iPod (2001), l'iPhone (2007) et l'iPad (2010). Apple a développé deux systèmes d'exploitations (Mac OS X et iOS) permettant de contrôler ses différents produits, notamment les ordinateurs Macintosh, l'iPod et l'iPod touch.

iOS, connu sous le nom de iPhone OS avant Juin 2010, est le système d'exploitation mobile d'Apple. Ce dernier a été développé originellement pour l'iPhone, puis a été étendu pour l'iPod touch, iPad et Apple TV (2007). C'est dérivé de Mac OS X dont il partage les fondations. Comme celui-ci, iOS comporte quatre couches d'abstraction : une couche « Core OS », une couche « Core Services », une couche « Media » et pour finir une couche « Cocoa ». Pour développer une application Cocoa, il est imposé d'utiliser l'Objective-C comme langage de programmation.

L'Objective-C est le langage de programmation orientés objet. Il a été inventé au début des années 1980 par Brad Cox, créateur de la société Stepstone. Son objectif était de combiner la richesse du langage Smalltalk et la rapidité du ANSI. C'est une extension du ANSI qui ne permet pas l'héritage multiple contrairement au C++. Aujourd'hui, il est principalement utilisé pour développer des applications sur Mac OS X et iPhone. Une nouvelle version Objective-C 2.0 a été indrotuite avec Mac OS X 10.5 (Léopard) en octobre 2007.

Le SDK d'iOS fournit les principaux outils nécessaires pour réaliser une application sur iPhone. Il est composé notamment de Xcode, Interface Builder, iPhone Simulator et Instruments. Xcode est l'outil de développement Apple, il permet la création de projets iPhone, l'édition du code source Objective-C, la compilation et le débogage des applications. Interface Builder quant à lui permet de construire des interfaces graphiques. L'iPhone Simulator est un logiciel simulant le comportement d'un iPhone, ce qui permet de pouvoir tester les applications directement sur l'ordinateur. Grâce à Instruments, il est possible analyser un programme pour surveiller l'état de la mémoire, l'utilisation du réseau, du CPU, etc. Pour utiliser tous ces outils, il est obligatoire de posséder un ordinateur Macintosh sous Mac OS X et de s'enregistrer sur iPhone Dev Center pour pouvoir télécharger et installer le SDK d'iOS.

## 2.3 Organisation

Afin de garder notre projet cohérent et par sécurité, nous avons choisi de le stocker sur les serveurs de *Google Code* qui mettent gratuitement à disposition des gestionnaires de versions (Subversion (SVN)) distribués sous Licence Apache et BSD.

Les gestionnaires de versions comme leur nom l'indique, permettent d'avoir à porté de main toutes les versions qu'il y a eu d'un fichier depuis sa création, cela permet donc de pouvoir revenir en arrière si une erreur a été commise. De plus grâce à cela, notre projet reste cohérent dans le sens où pour pouvoir être mis à jour, il faut à tout prit avoir modifié un fichier à partir de la dernière version de celui-ci.

Dernier avantage d'avoir utilisé un gestionnaire de version et qu'il est hébergé sur le net et donc chaque membre de l'équipe peut y accéder où qu'il soit.

De plus *Google Code* met aussi à disposition de ses utilisateurs des wikis. Un wiki est un site Web dont les pages sont modifiables par les développeurs afin de permettre l'écriture et l'illustration collaboratives des documents numériques qu'il contient.

Malgré de nombreuses réunions quotidiennes afin d'organiser au mieux le développement de cette application dont nous ignorons tout à nos début. Nous avons utilisé le wiki



qui nous a été fourni afin de partager au mieux les découvertes que nous faisons au fur et à mesure ainsi que les articles qui nous semblaient intéressants.

En ce qui concerne la répartition du travail, celle-ci s'est faite naturellement car seulement la moitié du groupe possédait de quoi développer sous iOS, de là nous avons donc créé deux équipes une sous Android et l'autre sous iOS.

DIAGRAMME DE GANT

# Chapitre 3

## Analyse

– Introduction

### 3.1 Cahier des charges

#### 3.1.1 Menus

Les menus se doivent d’être clairs et de rendre l’utilisation de l’application aisée. Il s’agit d’un jeu ne demandant aucune compétence particulière. Il va donc toucher un public large et doit pouvoir convenir à tout utilisateur. Cela passe d’abord par une navigation intuitive dans les menus.

Nous avons pour ce faire établi un diagramme d’activité reflétant les différents parcours possibles par un utilisateur lors de sa navigation dans l’application (voir section modélisation p13).

#### 3.1.2 Jeu

Le jeu est la partie la plus importante du projet. Il se décompose en trois parties : le modèle, la vue et le contrôleur. Le modèle est composé du moteur physique, du moteur de rendu ainsi que de la hiérarchie de classe permettant de représenter l’ensemble des objets du jeu. La vue quant à elle est composée d’objets graphiques simples (Bouttons, images, ... ) et d’une partie représentant le jeu. Elle se doit d’être ergonomique et de permettre à l’utilisateur de pouvoir jouer très simplement. Le contrôleur permettra de faire le lien entre les actions de l’utilisateur sur le modèle. Cette décomposition permettra dans le futur de pouvoir modifier facilement le modèle et/ou la vue.

L’application se doit de pouvoir changer de langue, avec comme langues initiales le français et l’anglais. Elle doit permettre à l’utilisateur de jouer à des parties solitaires ou multijoueur. Ce dernier possèdera un compte hors ligne et en ligne. Le premier permettra de personnaliser son profil comme par exemple pour modifier la couleur du joueur ou encore changer son pseudo... Il servira aussi à enregistrer les informations et les préférences de connexion sur une base de données locale, mais également les scores des joueurs (nombre

de parties gagnées ou perdus). Le jeu devra permettre à l'utilisateur de pouvoir créer différents comptes hors lignes en cas de partage de téléphone avec un ami ou un membre de famille, pour pouvoir garder en mémoire ses scores et ses préférences. Le compte en ligne quand à lui servira seulement à établir une connexion avec le serveur distant pour pouvoir jouer en multijoueur. Un menu d'aide doit apparaître pour pouvoir aider le joueur à comprendre le but du jeu et comment jouer. Ce dernier doit être simple et très explicite étant donné la large tranche d'âge d'utilisateur que vise cette application. Ensuite un éditeur de carte permettra aux utilisateurs de créer un large choix de cartes, grâce à une multitude de différents objets qui composeront les cartes. Ces dernières pourront être seulement utilisées en mode solitaire. Pour les parties solitaires une intelligence artificielle avec trois niveaux de difficulté devra permettre à un joueur débutant, intermédiaire ou confirmé de jouer comme bon lui semble pour pouvoir améliorer sa manière de jouer.

### **3.1.3 Serveur**

Le serveur représente la partie réseau de notre projet. Il doit pouvoir rendre fonctionnel le jeu entre plusieurs téléphones (qu'ils soient de type iOS ou Android). Autrement dit il servira d'hébergeur pour les parties et il se chargera de faire l'interagir des joueurs entre eux, via leur mobile. Nous parlons donc ici des parties multijoueur.

Il devra être capable d'enregistrer des inscriptions de nouveaux joueurs, avec vérification qu'il n'y ait pas de doublons. Ces derniers seront inscrits dans la base de données du serveur. Les joueurs devraient ainsi pouvoir se connecter en utilisant le couple username/mot de passe, préalablement choisi. Suite à cela les utilisateurs seront à même de lister les parties en cours, ils pourront choisir de créer des parties ou de les rejoindre.

Voilà concernant les fonctionnalités qui ont été demandées pour la partie serveur.

## **3.2 Modélisation**

### **3.2.1 Général**

Le fait de développer des applications iOS et Android, nous a imposé le choix du langage. En effet pour développer des applications iPhone, il faut utiliser le langage Objective-C, puis pour les applications Android c'est le langage Java qui est utilisé. C'est deux langages ont une syntaxe complètement différente mais sont quand même très proche car ce sont des langages orientés objets. Grâce à cela nous avons pu mettre au point une modélisation générale de l'application que nous avons ensuite adaptée à chaque langage.

Nous avons décidé de développer en anglais car premièrement c'est la langue la plus utilisée dans le monde de la programmation et deuxièmement car la syntaxe des langages est toujours en anglais. Cela permettra à n'importe quel utilisateur de n'importe quelle nationalité de comprendre le code de l'application.

La documentation des deux applications est également en anglais. Cette documentation permettra à tout utilisateur de comprendre le fonctionnement de l'application. Nous avons créé une documentation pour l'application iPhone et pour l'application Android. Chacune de ces documentations est au format HTML et sera donc directement visible grâce à un navigateur Web.

Pour ce qui est de la modélisation globale du projet, nous avons choisi de développer les deux applications selon le modèle Modèle Vue Contrôleur (MVC). Ce dernier est une architecture et une méthode de conception qui organise l'Interface Homme Machine (IHM) d'une application logicielle.

Le modèle représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Il décrit et contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité.

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc).

Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer.

Grâce à cette méthode de conception le code est décomposé en trois parties bien distinctes qui permettent la maintenance et l'amélioration du projet. Cela permet aussi de modifier chacune des trois parties sans avoir à modifier les deux autres. Cela permet par exemple de changer de vue et avoir des interfaces graphiques différentes.

### 3.2.2 Menus

Au démarrage de l'application vous arrivez sur un menu d'accueil. Depuis celui-ci vous pourrez accéder à l'aide, à la liste des comptes locaux ou à la création d'un nouveau compte local. Les menus de l'application ont été réalisés pour que l'utilisateur puisse les utiliser de manière intuitive. Ils se divisent en quatre grandes sections.

Vous avez tout d'abord la section de création de parties locales. Vous aurez accès à une liste de cartes ainsi qu'au réglage de difficulté des bots, leur nombre et le temps de jeu. Le type de partie sera une fonctionnalité à venir. Vous n'aurez plus qu'à créer la partie configurée.

Dans la même catégorie se trouve la section des parties multijoueurs. En accédant à celle-ci vous allez pouvoir vous connecter à votre compte multijoueur, ou le créer s'il n'est pas déjà fait. Vous accèderez ensuite à la liste des parties multijoueurs, que vous pourrez rejoindre, ou choisir de créer la votre. Dans le menu création le principe est proche des parties locales.

Suite à ces deux sections vient ensuite l'éditeur de cartes. C'est depuis ce menu que vous créerez une nouvelle map de jeu local ou éditez l'une d'entre elles. Choisissez votre nom de carte et l'éditeur s'ouvrira ensuite à vous. Il vous sera possible, à la fin, d'enregistrer votre carte si vous désirez la conserver et l'utiliser comme carte de jeu.

Et enfin vient le menu des options. Depuis ce dernier vous pourrez gérer vos préférences systèmes telles que le volume ou la langue de l'application(anglais, français). Une sous-section de gestionnaire de profil est aussi présente. Une édition de vos comptes locaux, multijoueurs ou même vos paramètres de jeu comme la position du menu, sont modifiable depuis ce menu à onglets.

### **Base de données**

Une base de données locale a elle aussi été conçue. Cette dernière a pour but de stocker plusieurs types de données.

En effet dès lors qu'un compte local est créé sur le téléphone dans la table PlayerAccount, il est possible de conserver ses préférences de joueur tel que la couleur du joueur, le pseudonyme ou même ses paramètres de connexion multijoueur. Vous pourrez créer autant de comptes locaux que vous désirez, et il sera possible possible d'éditer ou choisir son compte.

L'application est par ailleurs en mesure de conserver les valeurs sonores, la langue et même le dernier utilisateur de l'application, grâce à un son id qui est clé étrangère dans la table System(attribut lastUser).

De plus l'application sera délivrée avec quelques cartes officielles, mais l'utilisateur aura libre droit de créer ses propres cartes de jeu via un éditeur. Elles seront alors stockées dans la table Map avec toujours une clé étrangère vers l'id de son créateur(owner).

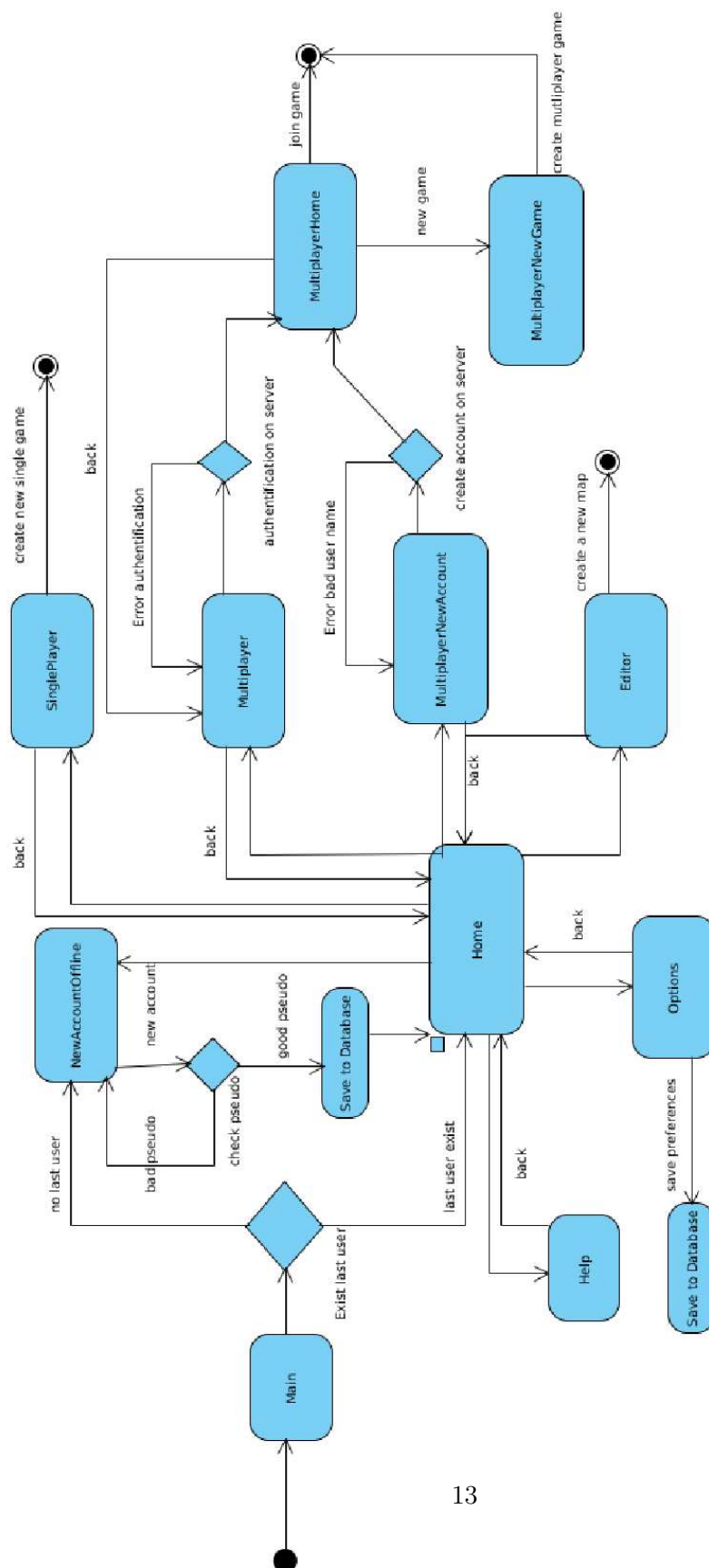


FIGURE 3.1 – Diagramme d'activité

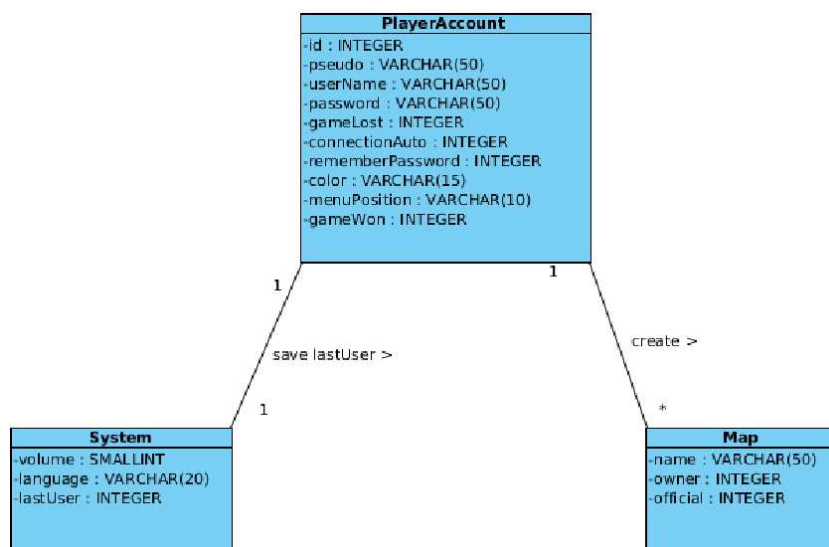


FIGURE 3.2 – Diagramme de classe Base de données

## Scénarios

### 3.2.3 Editeur de carte

L'éditeur de carte est une fonctionnalité qui va permettre à un utilisateur de créer facilement ses propres cartes pour ensuite y jouer dessus contre l'intelligence artificielle. Après avoir réfléchi sur toutes les fonctionnalités que l'éditeur de carte devait remplir, nous avons retenu celles-ci : permettre à l'utilisateur de créer une nouvelle carte, mais aussi de charger une ancienne carte précédemment créée. Ensuite lui donner la possibilité de modifier le sol de la carte et aussi ajouter ou supprimer des blocs de la carte et enfin la dernière fonctionnalité que l'éditeur de carte implémente c'est de pouvoir placer les différents points de départ des joueurs sur la carte.

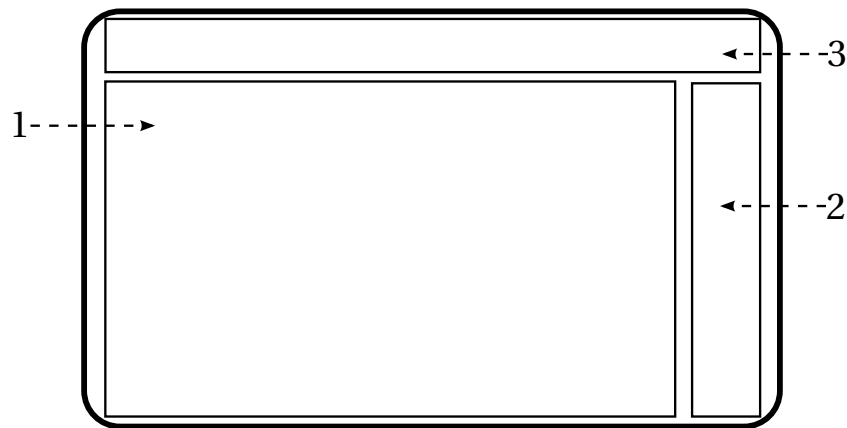
Pour réaliser cette partie de l'application, nous avons utilisé le modèle de conception MVC pour diviser le code de l'éditeur de carte. Grâce à cette décomposition, le code est plus lisible et plus facile à réutiliser.

## Modèle

La partie modèle va contenir toute les données de l'éditeur de carte. Les cartes sont les principales données qu'il va devoir manipuler. Pour cela nous avons décidé de la représenter sous la forme de deux matrices, la première représentant les objets du premier niveau (le sol) et la deuxième la matrice du second niveau (les blocs, les points de départ des joueurs, etc).

## Vue

Ensuite, la vue représentera l'interface graphique de notre éditeur de carte. La principale difficulté pour réaliser l'interface graphique était de devoir rentrer toutes les informations nécessaires pour l'éditeur de carte dans un écran de type smartphone<sup>1</sup>. Après plusieurs prototypes d'interface, nous avons décidé de séparer l'interface en trois parties. Tout d'abord la plus grande partie, l'affichage de la carte, qui comment étant la principale information à afficher, nous avons essayé de maximiser sa taille. Ensuite un menu à droite permettant au joueur de changer d'outil. Et la dernière partie affiche les différents éléments permettant de contrôler l'éditeur de carte. L'utilisateur aura juste à choisir l'outil qu'il veut placer sur la carte grâce au menu de droite et ensuite lui suffira d'appuyer sur la carte pour placer un bloc dessus.



## Contrôleur

Pour finir le contrôleur aura pour but de faire la liaison entre les données du modèle et de la vue. Chaque vue possède son contrôleur, et il y a un contrôleur global possédant les contrôleurs de chaque vue.

### 3.2.4 Jeu

#### Intelligence artificielle

Comme nous avons vu dans le cahier des charges, nous avons mis en place une intelligence artificielle permettant à un joueur de jouer en solitaire.

Tout d'abord nous avons dû réfléchir à toutes les actions que les bots pourraient effectuer, lors d'une partie. Premièrement l'action la plus importante que l'intelligence artificielle doit savoir faire c'est de pouvoir se déplacer librement sur la carte en fonction des différents éléments de la carte et des actions effectuées par les autres joueurs. Elles

---

1. Traduit littéralement comme « téléphone intelligent » en français, c'est un terme utilisé pour désigner les téléphones évolués, qui possèdent des fonctions similaires à celles des assistants personnels. Certains peuvent lire des vidéos, des MP3 et se voir ajouter des programmes spécifiques.



sont notamment capable de suivre le joueur le plus proche pour l'attaquer, fuir en cas de danger. Ensuite les bots sont capables de détruire des murs pour pouvoir atteindre un point de la carte et de poser une bombe pour essayer de tuer un ennemi.

Comme nous venons de le voir, la principale action que l'intelligence artificielle doit être capable effectuer, c'est de pouvoir se déplacer dans son environnement. C'est pour cela que nous avons regardé comment déplacer un objet dans un graphe, et nous avons trouvé plusieurs algorithmes étant capable de rechercher un chemin dans un graphe, mais parmi tous ces algorithmes, nous en avons retenu deux en particulier.

## Pathfinding

### A\*

Tout d'abord l'algorithme de recherche A\* a pour but de rechercher un chemin dans un graphe entre un nœud initial et un nœud final tous deux préalablement définis. A\* permet de trouver l'un des meilleurs (mais pas forcément le meilleur) chemins existant entre un point A et un point B (il retourne le premier chemin trouvé). La force de cet algorithme est le temps de calcul et l'exactitude des résultats, contrairement à Dijkstra qui est le deuxième algorithme que nous avons retenu qui lui fournit toujours le meilleur résultat (le plus court chemin entre deux points) mais dans un temps d'exécution beaucoup plus long que l'algorithme de A\*. Et comme il peut y avoir jusqu'à trois bots<sup>2</sup> et que l'intelligence artificielle doit régulièrement recalculer son chemin en fonction des actions effectuées par les autres joueurs, nous avons donc choisi l'algorithme de A\*.

Maintenant que l'on sait quel algorithme utiliser pour rechercher un chemin dans un graphe, nous allons voir en détail comment marche l'algorithme de A\*.

Pour comprendre comment l'algorithme marche, nous allons nous aider d'un dessin représentant une carte avec un point A (départ) affiché en vert, un point B (arrivée) en rouge, et où les cases en bleu représentent les murs.

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							

---

2. Bot (diminutif de robot) désigne un personnage contrôlé par l'ordinateur.

La première chose que l'on peut observer, c'est que la carte est divisée en cases. Chaque case de la matrice représente un node qui peut être soit traversable, soit non traversable. Dans l'application, il n'y a que les murs ou les bombes que l'on ne peut pas traverser sinon tous les autres objets ou joueurs sont traversables. Le but de l'algorithme est donc de trouver un chemin entre A et B en évitant les murs.

Durant le déroulement de l'algorithme, nous avons utilisé deux listes qui contiennent des cases de la carte. Il y a une liste dite « listeOuverte » et l'autre « listeFermée ». La listeOuverte contient une liste de cases qui pourraient éventuellement faire partie du chemin, mais pas forcément, pour le moment elle sera vide. Plus précisément c'est une liste de cases que nous devons vérifier. Ensuite, au niveau de la listeFermée, elle contient toutes les cases que nous aurons déjà vérifiées, au début de l'exécution, elle contient que le point de départ (B3).

Commençons les explications du déroulement de l'algorithme. Tout d'abord, il faut savoir qu'un joueur peut se déplacer dans toutes les directions, donc nous allons ajouter toutes les cases adjacentes à la listeOuverte qui sont traversables, il y en a huit (A2, B2, C2, A3, C3, A4, B4, C4).

Ce qui nous donne :

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							

Les carrés avec un contour rouge sont les carrés présents dans la listeOuverte et les carrés qui ont une couleur un peu plus foncée que les autres sont ceux qui se trouvent dans la listeFermée.

Maintenant pour choisir la case par laquelle on doit passer, nous devons rajouter trois données « F », « G » et « H » :

**G** : c'est le coût de mouvement pour aller de la case A à une case donnée sur la grille, en suivant le chemin généré jusqu'à cette dernière.

**H** : c'est l'heuristique, c'est à dire le coût estimé pour aller du point courant à l'arrivée. Comme nous ne connaissons pas vraiment la distance qu'il nous reste à parcourir, car toutes sortes d'obstacle peuvent se trouver sur notre chemin (objet non traversable). Donc nous allons devoir l'approximer grâce à une fonction, pour la calculer nous avons choisi d'utiliser l'heuristique de Manhattan, qui consiste à compter le

nombre de bloc (à vol d'oiseau et sans prendre les diagonales) qui lui reste à parcourir.

**F** : c'est  $G + H$

Chaque case de la listeOuverte ou de la listeFermée vont devoir posséder toutes ces données, plus les coordonnées de leur père, c'est à dire les coordonnées de la case qui vient de les ajouter dans la la listeOuverte. Pour calculer G, nous allons assigner un coût de 10 pour chaque déplacement horizontal ou vertical, et un coût de 14 pour un mouvement en diagonale. Nous utilisons ces données car la distance nécessaire pour se déplacer est la racine carrée de 2, ou approximativement 1.41 fois le coût d'un déplacement vertical ou horizontal. Nous utiliserons donc 10 et 14 pour des raisons de simplification. Par conséquent, nous allons multiplier par 10 le coût H pour qu'il soit cohérent par rapport à G.

Donc maintenant, nous devons avoir cette matrice :

	A	B	C	D	E	F	G
1							
2	74 B3	60 B3	54 B3				
3	14 60	10 50	14 40				
4	60 B3	F Père	40 B3				
5	10 50	G H	10 30				
6	74 B3	60 B3	54 B3				
7	14 60	10 50	14 40				
8							

Après avoir ajouté toutes les cases adjacentes à la case courant, il suffit de prendre la case qui a le plus petit coût F et ensuite de la rajouter dans la listeFermée et de la supprimer de la listeOuverte. Nous obtenons donc :

	A	B	C	D	E	F	G
1							
2	74 B3	60 B3	54 B3				
3	14 60	10 50	14 40				
4	60 B3	F Père	40 B3				
5	10 50	G H	10 30				
6	74 B3	60 B3	54 B3				
7	14 60	10 50	14 40				
8							

Ensuite on regarde toutes les cases adjacentes à la dernière case ajoutée dans la listeFermée. Si elles se trouvent déjà dans la listeOuverte, on vérifie que leurs coût soient inférieur au coût de la case correspondante déjà dans la listeOuverte, si oui alors on la remplace sinon on ne fait rien.

Pour finir on répète cette opération jusqu'on arrive à la case d'arrivée, nous obtenons ça :

	A	B	C	D	E	F	G
1	94 B2	80 B2	74 C2				
	24 70	20 60	24 50				
2	74 B3	60 B3	54 B3				
	14 60	10 50	14 40				
3	60 B3	F Père	40 B3		82 F4	F4	82 F4
	10 50	G H	10 30		72 10	68 0	72 10
4	74 B3	60 B3	54 B3		74 E5	68 E5	88 F4
	14 60	10 50	14 40		54 20	58 10	68 20
5	94 B4	80 B4	74 C4	74 C5	74 D5	74 E5	102 F4
	24 70	20 60	24 50	34 40	44 30	54 20	72 30

Pour finir, il nous suffit juste de récupérer la case d'arrivée et de regarder son père, puis de répéter cette opération avec la case obtenu jusqu'à arriver à la case de départ. Grâce à ça nous obtenons cette dernière étape de l'algorithme :

	A	B	C	D	E	F	G
1	94 B2	80 B2	74 C2				
	24 70	20 60	24 50				
2	74 B3	60 B3	54 B3				
	14 60	10 50	14 40				
3	60 B3	F Père	40 B3		82 F4	F4	82 F4
	10 50	G H	10 30		72 10	68 0	72 10
4	74 B3	60 B3	54 B3		74 E5	68 E5	88 F4
	14 60	10 50	14 40		54 20	58 10	68 20
5	94 B4	80 B4	74 C4	74 C5	74 D5	74 E5	102 F4
	24 70	20 60	24 50	34 40	44 30	54 20	72 30

Comme on peut le voir sur le schéma précédent, le chemin qu'à trouvé l'algorithme de A\* est le suivant :  $B3 \rightarrow C4 \rightarrow C5 \rightarrow D5 \rightarrow E5 \rightarrow F4 \rightarrow F3$ . L'algorithme aurait pu d'autre chemin équivalent à celui-ci mais le principe de cette algorithme c'est de renvoyer le premier chemin qu'il trouve.

### Parcours en largeur

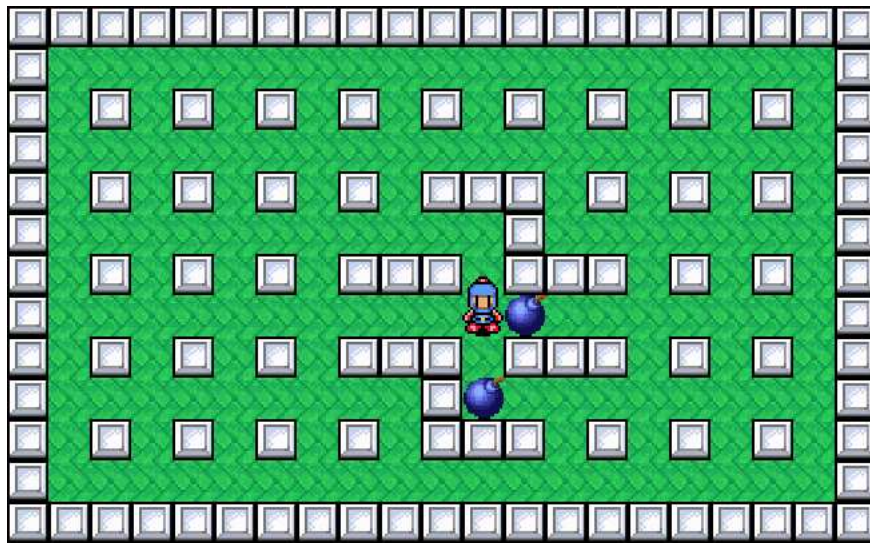
Contrairement à l'algorithme A\*, le parcours en largeur se contente d'un nœud initial

mais ne nécessite pas un nœud final. En effet ici nous allons partir d'un point et chercher le premier point correspondant à nos attentes, pour cela nous effectuerons donc un parcours en largeur.

Le parcours en largeur consiste à partir d'un sommet S, lister d'abord tous les voisins de S pour ensuite les explorer un par un. Ici nous allons donc regarder toutes les cases autour de nous puis regarder tous leurs voisins et cela ainsi de suite jusqu'à trouver un point correspondant à nos attentes.

Le contexte est le suivant, un bot découvre qu'il est sur la trajectoire d'explosion d'une bombe est va donc fuir vers la case sûre la plus proche or il n'a aucune idée d'où elle se trouve.

L'image suivante représentera la situation initiale :



Comme dit précédemment il n'y a que les murs ou les bombes que l'on ne peut pas traverser sinon tous les autres objets ou joueurs sont traversables. Nous considérons que les bombes ont ici un champs d'explosion en nombre de cases de 5.

Représentons la carte ci-dessus d'une façon plus parlante en remplaçant les divers objets mis à par le joueur par des numéros leur correspondant, à savoir :

1. Les zones dites dangereuses donc ici le champs d'explosion de la bombe.
2. Toutes les cases infranchissables telles que les murs ou encore les bombes

Ce qui nous donne :

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	0	2	2
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
2	0	2	0	2	0	2	0	2	0	2	0	2	2	2	0	2	0	2	2
2	0	0	0	0	0	0	0	0	0	0	0	0	1	2	0	0	0	0	2
2	0	2	0	2	0	2	0	2	0	2	2	2	1	2	2	2	0	2	2
2	0	0	0	0	0	0	0	0	0	1	1	1	1	2	1	1	1	1	2
2	0	2	0	2	0	2	0	2	0	2	2	2	1	2	2	2	0	2	2
2	0	0	0	0	0	0	0	0	0	0	0	2	2	1	1	1	1	0	2
2	0	2	0	2	0	2	0	2	0	2	0	2	2	2	0	2	0	2	2
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

Mettons nous à present à la place du bot.

Nous allons donner un poid aux cases que nous allons parcourir correspondant à la distance par rapport à la case initiale, ainsi qu'une direction qui correspondra à la direction initiale que le bot devra emprunter pour utiliser ce chemin, c'est à dire par exemple que tout chemin découvert dont l'origine est une case à droite de la notre aura comme direction droite.

A partir d'une case donnée, nous ne regarderons que les voisines ayant un poids de 0 car si leur poids est différent cela voudra dire que nous les avons déjà vu precedemment et bien évidemment, nous ignorerons les murs ainsi que les bombes.

Nous avons changé la valeur des cases intraversables pour ne pas confondre avec le poids des cases visitées.

Appliquons l'algorithme de parcours en largeur aux cases voisines de la notre.

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	X	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	X	X	1 ↑	X	X	X	0	X	X
X	0	0	0	0	0	0	0	0	0	0	0	1 ←	1	X	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	X	X	1 ↓	X	X	X	0	X	X
X	0	0	0	0	0	0	0	0	0	0	0	X	X	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Toutes les cases découvertes étant considérées comme dangereuses (voir le schéma précédent) nous continuons à appliquer l'algorithme jusqu'à arriver à notre but.

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	2 ↑	X	0	0	0	0	0	X
X	0	X	0	X	0	2	0	X	0	X	X	X	1 ↑	X	X	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	2 ←	1 ←	1	X	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	X	X	1 ↓	X	X	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	X	X	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	3 ↑	2 ↑	X	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	X	X	1 ↑	X	X	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	3 ←	2 ←	1 ←	1	X	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	X	X	1 ↓	X	X	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	X	X	0	0	0	0	0	0	X
X	0	X	0	X	0	X	0	X	0	X	0	X	X	X	0	X	0	X	0	X
X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	X
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Ici nous avons découvert une case non dangereuse en (13,6)! Nous récupérons donc la direction enregistrée dans cette case et bougeons en fonction de celle-ci. Si plusieurs cases avaient été découvertes le choix aurait été arbitraire car elles auraient toutes été à la même distance.

– Diagramme classe

Comme vue dans le cahier des charges, l'application est divisée en trois grandes parties : le modèle, la vue et le contrôleur.

## Modèle

Le modèle constitue la *base* de notre projet. C'est sur celui-ci que nous avons construit le reste de l'application. Nous avons modélisé celui-ci de manière à ce qu'il soit clair et simple. Ce dernier se décompose en trois sous-partie : la partie hiérarchie des objets, la

partie moteur puis la partie éditeur de carte.

## Hierarchie des objets

Pour concevoir la hierarchie des objets, il a fallu tout d'abord distinguer la totalité des objets qui seraient disponible dans le jeu ainsi que leurs différences et leur point communs. Nous avons donc distingué quatre grands types d'objets : les objets destructibles, les objets indestructibles, les objet animés ainsi que les objets inanimés. En sachant que les objets pouvaient être destructibles-animés, destructible-inanimés, indestructible-animés ou indestructible-inanimés. Nous avons choisi que tout objet serait considéré comme un objet animé pour éviter des soucis de modélisation. Ainsi les objets posséderont tous une sequence d'animation d'images qui sera dessiné a l'écran. Cette derniere sera répété dans le cas d'une animation qui se répété et qui comporte plusieurs images, sinon si elle possède simplement qu'une seule image, seule cette image sera dessiné a l'écran. Nous avons ensuite distingué une multitude de point communs entre les différents objets dont voici le nom de l'attribut et leur utilité :

***position*** indique la position en pixel de l'objet sur l'écran

***nom*** indique le nom de l'objet

***hit*** est à 1 si l'objet peut être touché par une explosion (0, sinon)

***level*** est à 1 si l'objet est animés (0,sinon)

***fireWall*** est à 1 si l'objet ne laisse pas passer les flammes des explosions (0,sinon)

***damages*** indique si l'objet peut infliger des damages

***idle*** est une image qui représente l'objet en général

***animates*** est une table de hachage contenant l'ensemble des images composant la sequence d'animation de l'objet (vide si inanimés)

***destroy*** est une table de hachage contenant l'ensemble des images composant la sequence d'animation de destruction de l'objet (vide si indestructible)

***currentFrame*** permet de connaitre le numéro de l'image courante de la sequence d'animation en cours d'affichage

Nous avons donc décider de concevoir une classe "Object" pour modéliser toutes ces propriétés que les objets ont en commun. Cette classe sera abstraite car tout objet est destructible ou indestructible et cela sera décrits dans des classes plus spécialisés.

Ensuite nous avons pensé a créer deux autres classes : "Destructible" et "Undestructible" pour les objets destructible et indestructible. Ces deux classes héritent de "Object" car elles sont des objets. Ce qui différencie ces deux classes est le champ *life* qui permet de savoir combien de fois l'objet doit être touché par une bombe avant d'être détruit.

Nous avons ensuite distingué deux autres types d'objets encore plus spécifique : Les joueurs et les bombes. Ces derniers sont des objets destructibles puisqu'il ne dure pas toute la partie selon le mode de jeu.

Les bombes possèdent deuxnouveaux attributs qui les différentie des autres objets :



**type** permet de connaître le type de bombe

**owner** permet de connaître le joueur qui a posé la bombe

Quand à la classe joueur celle-ci possède encore d'autres attributs :

**color** permet d'afficher à l'écran le joueur avec sa bonne couleur

**bombsTypes** permet de connaître le type de bombe qu'il peut poser

**powerExplosion** permet de connaître la portée d'explosion de ses bombes

**timeExplosion** permet de connaître le temps d'explosion des bombes

**speed** permet de connaître la vitesse du joueur

**shield** permet de connaître la valeur du bouclier

**bombNumbers** permet de connaître le nombre maximum de bombes que le joueur peut poser

**isTouched** permet de savoir si le joueur viens d'être touché

**isKilled** permet de savoir si le joueur est mort

**isInvincible** permet de savoir si le joueur est invincible

Cette hiérarchie de classe nous permettra donc de modéliser l'ensemble des objets que le jeu pourra afficher.

## Moteur

Pour ce qui est du moteur. Celui-ci est représenté par une classe "Engine". Cette classe va contenir l'ensemble des méthodes qui vont permettre de d'établir les collisions. C'est aussi cette classe qui s'occupe de mettre à jour les bombes ainsi que l'IA du jeu. Une instance d'un moteur est associé à une partie dont le nom de classe est "Game". Cette classe "Game" est abstraite et représente une partie avec toutes les options qu'elle contient. C'est à dire qu'elle possède des attributs permettant de décrire le type de partie, un tableau avec chaque joueur de la partie, ainsi que le carte du jeu. La classe possède toutes les méthodes d'initialisation, de mise à jour, de dessins et de fin de la partie. Enfin pour différencier les différents types de parties nous avons utiliser le pattern<sup>3</sup> décorateur. Ainsi il y aura deux grands types de parties : les parties solitaires et les parties multijoueurs. Puis ces parties sont décorés par le type de partie : survivor<sup>4</sup> ou death match<sup>5</sup>.

Ensuite pour ce qui est des cartes, une classe abstraite nommé "Map" contient le nom et la taille de la carte ainsi qu'un tableau contenant la position initial de chaque joueur sur la carte. Puis une classe "GameMap" qui hérite de map permet de dessiner la carte à l'écran, elle se compose d'une image représentant le sol puis d'un tableau d'objets animés qui permet de représenter l'ensemble du reste des objets. Ensuite nous utilisons un autre

---

3. Un pattern est un modèle de conception en informatique

4. Le mode survivor est un mode où chaque joueur à un nombre de vie limité et dont le gagnant est celui qui sera le dernier à être en vie

5. Le mode death match est un mode où il y a un temps limité et dont le joueur gagnant est celui qui a tué le plus d'ennemis

type de carte qui va permettre de faciliter les collisions, que ce soit pour un joueur humain ou pour une intelligence artificielle. Cette classe est appelé "CollisionMap" et contient une matrice d'objet "CollisionCase" ainsi qu'un tableau qui contient l'ensemble des bombes posés sur la carte. Les "CollisionCase" sont composé de deux tableaux. Chaque valeur d'un des tableau est associé à une valeur de l'autre tableau. Un tableau *types* contient les différents types de danger ou d'objet qu'il existe sur cette case et un tableau *counters* permettra en fonction de chaque type de compter le nombre de fois que ce danger ou objet est présent sur cette case. Par exemple on peut très bien avoir une case qui est dans le champ d'explosion de quatres bombes et qui contient un bloc de type destructible. Ainsi le tableau *type* contiendra deux champs , un pour le type zone dangereuse et un autre pour le type bloc destructible puis le tableau *counters* contiendra donc dans sa case associé à la case zone dangereuse une valeur égale à quatre et et une valeur de un pour l'autre.

## **Vue**

L'interface graphique du jeu est décomposé en trois partie. Une partie qui représente le menu d'information de la partie, une autre qui représente le menu des actions possibles du joueurs et une dernière qui représente la partie en cours. Le menu d'information se situe en haut de l'écran. Il permet d'afficher le score des joueurs, le temps lors d'une partie death match ainsi que les bonus du joueurs (le nombre de bombes qu'il peut poser, la portée de l'explosion des bombes, la vitesse du joueur et les bonus de vie) et de mettre le jeu en pause. Le menu d'action se situe à droite de l'écran. Il permet à l'utilisateur de poser les bombes grâce à un bouton et aussi de changer de type de bombes grâce à une liste déroulante. Quand à la partie, elle est afficher au centre de l'écran et prend le maximum de place possible pour que le jeu soit le plus visible possible. Elle permet d'afficher la carte ainsi que les joueurs et les bombes. Mais elle sert aussi à écouter les mouvements du doigt de l'utilisateur pour modifier les coordonnées du joueur dans le modèle pour pouvoir ensuite rafraichir l'écran et voir le joueur se déplacer.

## **Controlleur**

Le controlleur va permettre de faire la liaison entre la vue et le modèle. La majorité de ces méthodes sont appelé lorsque l'utilisateur interagit avec la vue. Ces méthodes vont par la suite modifier le modèle qui va permettre de mettre à jour la vue. Le controlleur est divisé en quatre sous-controlleur. Chacun des trois premiers est respectivement associé à l'une des trois vues cité ci-dessus. Puis le quatrième est plus général, il permet de faire la liaison entre les trois autres controlleurs. Car en effet chacune des actions effectués sur l'une des vue peut modifier l'une des deux autres.

## **GamePlay**

Nous avons choisi d'établir un gameplay immersif. C'est à dire que notre interface graphique sera utilisée de manière souple et intuitive. Notre gameplay est divisé en trois

parties : les déplacements, la pose des bombes et la gestion des différents types de bombes.

Pour ce qui est des déplacements du joueurs. Nous avons décidé que l'utilisateur utilisera toute la surface de l'écran pour se déplacer. Ainsi si il veut se déplacer vers la droite, il fait glisser son doigt de la gauche vers la droite, puis tant qu'il restera appuyer sur l'écran, le joueur continuera de se déplacer. Cette manière de se déplacer est précise et très intuitive contrairement à l'utilisation d'un joystick virtuel ou de l'accéléromètre.

Pour la pose des bombes un simple bouton est mis en évidence en bas à droite de l'écran. Celui-ci est assez gros pour que l'utilisateur n'est pas à appuyer sur une zone trop précise en cas de manipulation rapide.

Puis pour le choix des différentes bombes, une simple liste déroulante est mise en évidence à droite du jeu, pour pouvoir changer de bombe rapidement.

## Tile Mapping

La gestion des images a été une partie très importante de l'analyse. Car le développement mobile impose plusieurs contraintes, notamment la gestion de la mémoire et la vitesse de calcul. Nous nous sommes donc inspiré des premiers jeux consoles comme Super Mario Bros ou Zelda qui ont été développés sur les premières consoles comme la NES<sup>6</sup> ou la GameBoy<sup>7</sup>. Nous avons donc conçu le moteur du jeu selon le principe du Tile Mapping<sup>8</sup> pour minimiser l'utilisation des ressources des téléphones.

Le principe du Tile Mapping est d'utiliser des petites images que nous appellerons *tiles*. Ces tiles sont contenues dans une image appelée *sprite* (cf ci-dessous). Ensuite une matrice de nombre entier est utilisée pour représenter l'environnement du jeu. Chaque nombre entier représente un tile. Puis grâce à cette association, la carte sera dessinée à l'écran selon la matrice. Voici un exemple :



Nous avons donc repris ce principe et nous l'avons amélioré. En effet grâce à la programmation objet, les entiers sont représentés directement par des objets contenant le tile qui le représente. Ainsi, nous stockons dans une matrice tous les objets inanimés (les objets dont le tile restera le même tout le long de la partie) et nous parcourons cette matrice pour dessiner chaque objet à sa position pour obtenir une nouvelle image au format png. Ainsi à chaque rafraîchissement de l'écran, seule la nouvelle image est

6. La Nintendo Entertainment System, ou NES, est une console de jeux vidéo 8 bits

7. La Game Boy est une console de jeux vidéo portable à la puissance comparable à celle de la NES

8. Le Tile Mapping est une technique de modélisation graphique

dessiné et non pas chaque tile. Cette méthode permet d'éviter un parcours intempestif de la matrice. Ensuite pour le reste des objets (les objets animés composés d'une sequence de tiles) nous avons décidé de les stocker dans une table de hachage dont la clé est la position de l'objet (pour y accéder plus rapidement). Ainsi à chaque rafraichissement de l'écran on va dessiner l'image des objets inanimés puis parcourir entierement la table de hachage et dessiner le tile courant de la sequence d'animation de chaque objet.

### **La gestion des images et du son**

Chaque image, tile et chaque son dont est composé le jeu sont chargé au lancement de l'application. Tous les tiles sont stocké dans des images appelés sprites. Un sprite est donc un ensemble d'image. L'ensemble des tiles sont stockés dans différents sprites. Il y a un sprite pour les bombes, un pour les joueurs et un autre pour le reste des objets. Chaque sprite contient l'ensemble des images des objets qu'il représente. Chaque sprite est associé à un fichier XML<sup>9</sup>. Le fichier XML peut être représenté sous forme d'un arbre. Les fichiers XML ressemble tous plus au moins a ce genre d'arborescence :

---

9. Extensible Markup Language, (langage de balisage extensible) est un langage informatique de balisage générique

<objects>

```
<object_type name="object_name1" hit="0" level="0" fireWall="0" life="0" damages="0">
  <animation name="idle" canLoop="false" sound="sound_path">
    <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="0" />
  </animation>
  <animation name="animate" canLoop="true" sound="sound_path">
    <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="1" />
    <framerect top="0" left="60" bottom="30" right="90" delayNextFrame="1" />
    <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="1" />
    <framerect top="0" left="0" bottom="30" right="30" delayNextFrame="1" />
  </animation>
  <animation name="destroy" canLoop="false" sound="sound_path">
    <framerect top="0" left="90" bottom="30" right="120" delayNextFrame="1" />
  </animation>
</object_type>
```

```
<object_type name="object_name2" hit="1" level="1" fireWall="1" life="1" damages="0">
  <animation name="idle" canLoop="false" sound="sound_path">
    <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="0" />
  </animation>
  <animation name="animate" canLoop="true" sound="sound_path">
    <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="1" />
    <framerect top="30" left="60" bottom="60" right="90" delayNextFrame="1" />
    <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="1" />
    <framerect top="30" left="0" bottom="60" right="30" delayNextFrame="1" />
  </animation>
  <animation name="destroy" canLoop="false" sound="sound_path">
    <framerect top="30" left="90" bottom="60" right="120" delayNextFrame="1" />
  </animation>
</object_type>
```

</objects>

Chaque fichier XML commence par une balise racine permettant de lister les objets qu'elle contient (<objects>). Ensuite chaque objet est décrit par une balise qui contient le type, le nom de l'objet ainsi que l'ensemble de ses propriétés. La propriété *hit* est à 1 si l'objet peut être touché par une explosion (0, sinon), le champ *level* est à 1 si l'objet est animés (0, sinon), le champ *fireWall* est à 1 si l'objet ne laisse pas passer les flammes des explosions, le champ *life* indique le nombre de fois que l'objet doit être touché par des flammes pour qu'il soit entièrement détruit, et enfin le champ *damages* indique si l'objet peut infliger des dommages. Par exemple <destructible name="herb" hit="1" level="1" fireWall="1" life="1" damages="0"> décrit un objet de type *destructible*, qui peut-être touché par des flammes, qui est animé, ne laisse pas passer les flammes des bombes, possède une vie, n'inflige pas de dommage et dont le nom est *herb*. Ensuite chaque objet possède au plus trois balises animations (sauf pour les joueurs). Tout objet possède au moins la première balise : <animation name="idle" canLoop="false" sound="sound\_path">

qui est celle dont le nom est *idle*. Elle représente l'image standard de l'objet. Par exemple pour un objet bombe, ce sera l'image qui sera affiché dans la liste des bombes pour pouvoir sélectionner ses bombes. Le champs *canLoop* permet de savoir si l'image doit être répété en boucle et le champ *sound* contient le chemin d'accès au fichier de son de l'animation si elle en possède un. Il y a ensuite la même balise mais dont le nom est *animate*, celle-ci contiendra toutes les sequences d'images d'un objet qui est animés. Puis la dernière est celle dont le nom est *destroy*. Cette dernière contiendra l'ensemble des images composant la sequences d'animation de destruction de l'objet. Enfin chaque balise de type animation contient des balises de type *framerect*. Ces balises permettent de donner la position de chaque image de la séquence d'animation dans le sprite ainsi que le delay de rafraichissement entre chaque image. Par exemple `<framerect top="60" left="30" bottom="90" right="60" delayNextFrame="1" />` représente une image dont le bord du haut est situé à 30pixels, le bord du bas à 90pixels, le bord de gauche à 30pixels et le bord de droite à 60pixels en partant du coin en haut à gauche du sprite. Puis pour le champ *delayNextFrame* celui-ci informe que l'image suivante sera déclenché après un delay de 1s.

Grâce à cette modélisation, l'application va parcourir au démarrage l'ensemble des fichiers XML et va créer une table de hachage pour chaque ensemble d'objet du fichier. Lors de ce parcours, l'application va instancier chaque objet avec toutes propriétés que lui indique le document XML, ainsi que ses sequences d'animation. Ensuite lorsqu'un objet devra être utilisé dans le jeu, il suffira d'utiliser une copie de l'objet déjà chargé en mémoire pour éviter de devoir reparcourir le fichier.

### 3.2.5 Réseau

#### Serveur

Il a été fixé dans le cahier des charges que notre serveur devrait pouvoir effectuer plusieurs tâches particulières séparées. Nous avons donc décidé de les compartimenter en classes.

Notre serveur est crée sur une base de servlet. Ce fût ici aussi un point nouveau pour nous, réitérant les phases d'analyse, de découverte, de test et de mise en place. Le fonctionnement est basé sur les échanges de requêtes type HTTP, où à chaque demande correspond une réponse.

Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP. Une servlet s'exécute dynamiquement sur le serveur web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données.

Les six éléments situés sur la partie haute du schéma ci-dessous(respectivement ServletInscription, ServletConnection, ServletGamesList, ServletCreateGame, ServletConnectionGame et ServletManageGame), représente les différentes tâches qu'un utilisateur puisse demander au serveur. Elles sont reliées à une classe nommée ContextListener, qui leur permettra d'accéder aux mêmes données sans qu'il y ait de conflits. La partie basse représente les objets qui seront utilisés pour les parties en multijoueurs. Bien évidemment ces objets sont très proches de ceux utilisés dans les parties locales(Schéma

3.3).

Comme il a été dit précédemment, notre serveur est accessible via des requêtes HTTP contactant des servlets. Ces servlets sont stockées dans un serveur d'application nommé Apache Tomcat. Il s'agit d'un conteneur libre de servlets Java 2 Enterprise Edition, mais il fait aussi office de serveur Web.

Le scénarios le plus probable serait le suivant. Un utilisateur désire jouer en ligne contre de vrais joueurs. Il va alors passer par l'inscription et créer son compte sur le serveur(Inscription). Une fois cette étape obligatoire faite, il pourra choisir entre rejoindre une partie en ligne en cours(ConnectionGame), ou en créer une nouvelle(CreateGame). Dès lors qu'il aura accès à une partie en ligne, un contact régulier avec le serveur sera obligatoire afin de réaliser les interactions entre les différents joueurs(ManageGame). Tout ceci devra se réaliser bien sûr dans une durée infime afin de ne pas pénaliser les joueurs.

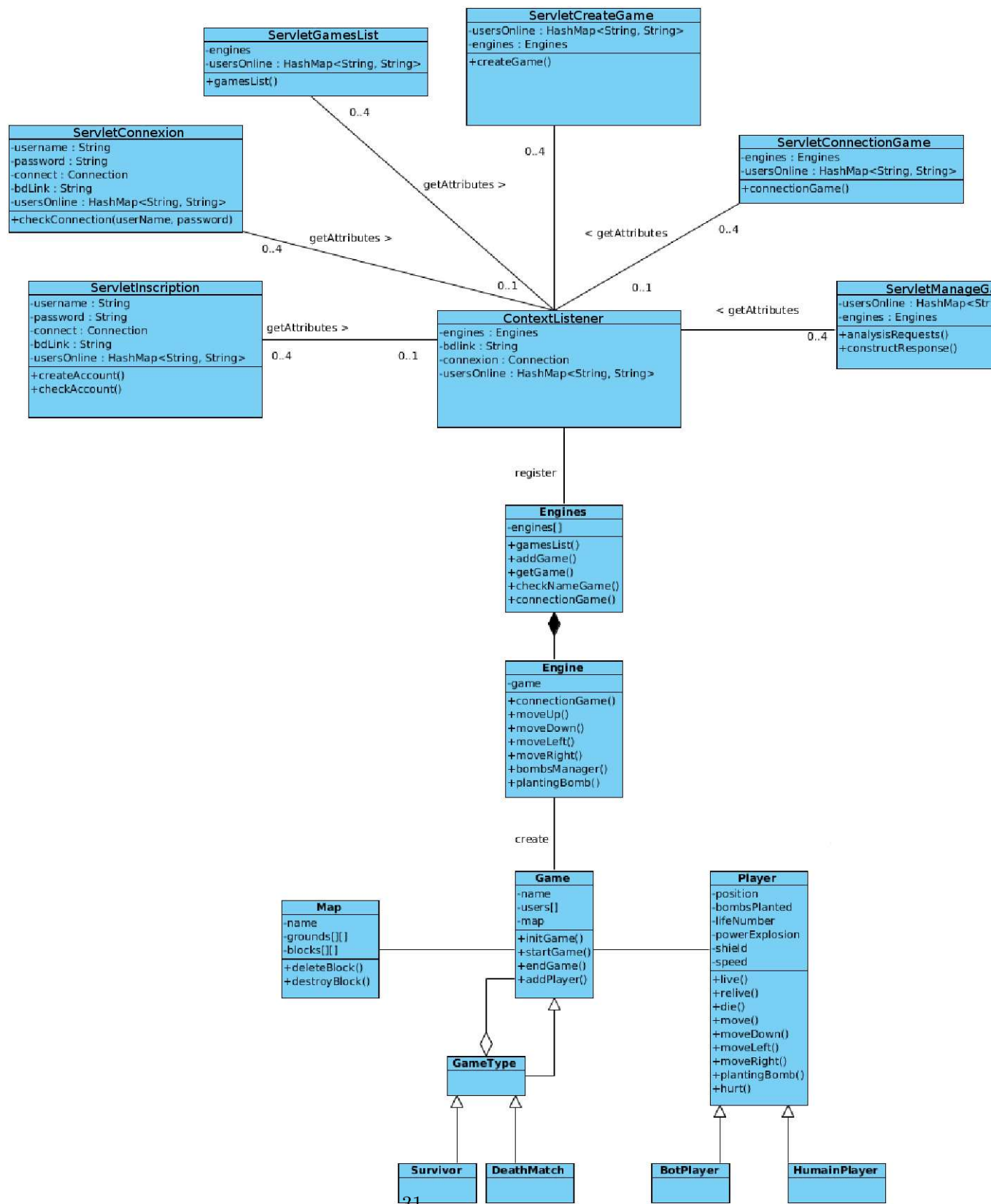


FIGURE 3.3 – Serveur



## JSON

Soucieux des performances et de la rapidité des échanges entre applications et serveur, nous avons mis en place un protocole de communication client/serveur où les messages transitant sont des flux JSON<sup>10</sup>. Contrairement au XML qui peut représenter des données orientées document, JSON se focalise sur la description d'objets. Un autre avantage reconnu de JSON par rapport à XML est qu'il est nettement moins verbeux que ce dernier. Quoi qu'il en soit JSON reconnaît la philosophie des services web exposant une interface d'échange : il s'agit d'envoyer et de recevoir des informations dans un format facilement manipulable par le protocole de transport HTTP.

Voilà pourquoi le JSON semblait être un format de données d'échanges optimal pour véhiculer le plus d'informations avec une taille moindre. Il est aussi en adéquation avec notre politique d'utilisation web pour un serveur. De plus étant beaucoup utilisé, nos deux langages mettent à disposition des outils de sérialisation de leurs objets en JSON.

Ci-dessous un exemple concret de notre protocole de communication JSON entre serveur et application cliente.

ServletInscription

Player => Serveur

```
{"username","password"}}
```

Serveur => Player

```
{"OK"} ou {"BU"}
```

ServletConnexion

Player => Serveur

```
{"username","password"}}
```

Serveur => Player

```
{"OK"} ou {"BU"}
```

ServletGameList

Player => Serveur

```
{"userKey"}
```

Serveur => Player

```
{[{"class":"Game","map":"mapName","name":"gameName",  
  "playerNumberConnected":nbConnected,"type":"gameType"},{..},{..}]}
```

ServletCreateGame:

Player => Server:

```
{"userKey": <userKey>,
```

---

10. JavaScript Object Notation : format de données textuel issu du JavaScript (ECMAScript pour plus exact) où il était employé comme une syntaxe pour décrire les valeurs des instances d'objets

```
"game": {"name":<name>, "type":<type>, "map":<map>, "ennemiesNumber": <ennemiesNumber>}}
```

```
Server => Player:  
{ "OK" } ou { "errorType" }
```

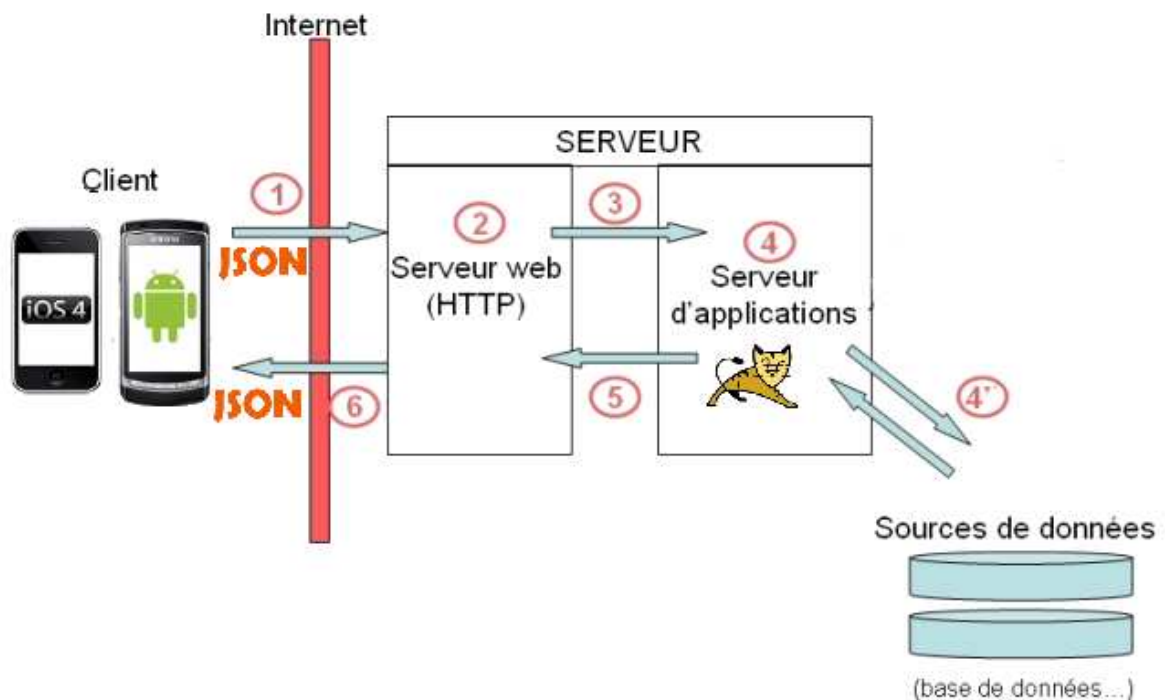
```
ServletConnectionGame:  
Player => Server:  
{ ["userKey", "gameName"] }
```

```
Server => Player:  
{ [<1/2/3/4>, "play<true/false>", "map", "time<mm:ss>"] }  
ou  
{ "errorType" }
```

```
ServletManageGame:  
Player => Server:  
{ "userKey", "gameName", "action" }
```

```
Server => Players: (Player, bombs, blocs, score, time)  
{ [  
  [ ["x", "y", "direction", "dead <true/false>"], [...] ],  
  [ ["x", "y", "type", "explode <true/false>"], [...] ],  
  [ ["position": { "x", "y" }, "bonus": <bonus> ], [...] ],  
  [1,2,3,4],  
  "time <mm:ss>"] }  
ou  
{ "errorType" }
```

## Schéma de fonctionnement



1. Le client émet une requête pour demander une ressource au serveur. Par exemple la création de son compte multijoueur, qui pourrait se situer <http://Bomberklob.com/inscription>
2. Côté serveur, c'est le serveur web qui traite les requêtes HTTP entrantes. Il traite donc toutes les requêtes, qu'elles demandent une ressource statique ou dynamique. Seulement, un serveur HTTP ne sait répondre qu'aux requêtes visant des ressources statiques.
3. Ainsi, si le serveur HTTP s'aperçoit que la requête reçue est destinée au serveur d'applications, il la lui transmet. Les deux serveurs sont reliés par un canal, nommé connecteur.
4. Le serveur d'applications (dans notre cas Tomcat) reçoit la requête à son tour. Lui est en mesure de la traiter. Il exécute donc la servlet correspondante à la requête, en fonction de l'URL, en récupérant les valeurs dans le flux JSON entrant. Cette opération est effectuée à partir de la configuration du serveur, grâce un fichier web.xml faisant le mapping entre URL et servlet associée.  
La servlet est donc invoquée, et le serveur lui fournit notamment deux objets Java exploitables : un représentant la requête, l'autre représentant la réponse. La servlet exécute sa fonction et génère la réponse à la demande, sous forme de flux JSON. Cela peut passer par la consultation de sources de données, comme des bases de données (4' sur le schéma).

### **Base de données**

Afin de pouvoir conserver les utilisateurs en ligne ainsi que leurs infos personnels et permettre une authentification, nous avons dû établir une base de données sur le serveur. Cette dernière a été pensée comme demandé pour l'enregistrement de comptes. Une unique table nommée Users remplit donc cette fonction. Le serveur devra pouvoir y accéder en écriture (inscription) comme en lecture (connexion). Elle ne comportera que deux champs, userName et password. Dès lors que l'utilisateur désirera créer un compte multijoueur, il renseignera dans l'application son userName souhaité ainsi que son mot de passe. Ce couple sera alors envoyé au serveur qui vérifiera dans cette base de données, que le userName (unique) n'est pas déjà utilisé. Auquel cas un nouveau n-uplet sera inséré et permettra l'authentification de l'utilisateur par la suite. Les mots de passe seront bien évidemment cryptés pour des raisons de sécurité.

### 3.3 Différences entre Android et iOS

Lors du développement sous en Android et iOS, nous avons été confronté à divers problèmes. Notamment des problèmes liés aux différences entre ces deux systèmes d'exploitations. Dans un premier temps, le langage utilisé n'est pas le même, le premier utilise le langage Java alors que l'autre utilise l'Objective-C. Ces deux langages étant des langages orientés objets, la difficulté réside surtout dans la différence des syntaxes. Ensuite, lors de l'implémentation de notre diagramme de classe notre premier soucis a été les différences de modélisation des vues et des controlleurs. En effet sous Android, les écouteurs d'une vue sont directement situés dans le controlleur de cette dernière alors que sous iOS les écouteurs peuvent être implémentés soit dans la vue soit dans son controlleur. Pour respecter au maximum le modèle MVC, nous avons choisi de placer les écouteurs dans la vue sous iOS (ce qui est recommandé par Apple).

## Chapitre 4

# Développement

### 4.1 Mobile

#### 4.1.1 Menus

##### API et widget

##### Android

Afin de créer les différents menus de notre application, Android met à la disposition des développeurs une API<sup>1</sup> très bien fournie. Parmi celle-ci le package Widget nous a été très utile.

Grâce à ce dernier de nombreux objets ont été utilisés afin de mettre en oeuvre et rendre pleinement fonctionnel nos menus. Parmi les plus utilisés il y a eu bien sûr Button, TextView, CheckBox et EditText pour les plus explicites. Les objets comme Spinner, SeekBar et Gallery étant respectivement utilisés pour les menus déroulants, les barres de progression et les galeries d'images pour la sélection de cartes de jeu.

Les composants graphiques sont créés ici au travers du fichier déclaratif XML via une syntaxe bien particulière. Cette méthode est vraisemblablement préférable, du moins lorsque l'interface graphique est figée, connue à l'avance. Exemple :

```
<Spinner android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/accounts"
    android:layout_gravity="center_horizontal"
    android:prompt="@string/ChooseUserAccount"></Spinner>
```

Pour récupérer la référence d'un widget créé depuis le fichier xml de layout, il convient d'utiliser la méthode findViewById de la classe Activity dans nos classes Java.

Exemple :

---

1. API : Application Programming Interface, c'est un ensemble de classes mis à disposition par une bibliothèque logicielle.

```
Spinner sp = (Spinner)findViewById(R.id.accounts);
```

On peut remarquer que cette méthode accepte en paramètre un int et non un String comme on pourrait s'y attendre. En effet, l'id est exprimé sous forme de constante int, on ne passe pas à la méthode la chaîne de caractères proprement dite. Grâce à cela, la méthode est sûre et on évite ainsi les erreurs à l'exécution qu'on pourrait avoir si on spécifiait un id ne correspondant à aucun widget.

Concernant leur positionnement, un système de layout est utilisé. Les layouts sont des ViewGroup responsables du dimensionnement et du positionnement des widgets à l'écran. Il en existe plusieurs, chacun adoptant une stratégie bien spécifique.

En ce qui nous concerne nous avons principalement utilisé les *ListView*, *LinearLayout*, *TableLayout* et enfin les *RelativeLayout*. Ce dernier nous a été très utile. En effet les widgets contenus dans un RelativeLayout peuvent déclarer leur position relativement par rapport à leur parent ou par rapport aux autres widgets. De ce fait nos menus et autres interfaces graphiques conservent leur proportion et leur agencement originel.

Les listeners présents dans les classes java permettent à leur tour d'écouter les événements utilisateurs ou system et réagir en fonction comme l'accès au menu suivant ou le lancement d'une partie.

Les activités sont des composants centraux des applications. Ce sont également les composants qui portent les éléments visuels de l'interface utilisateur agencés sur l'écran. La navigation entre les écrans se fait dans notre cas de façon explicite. L'ordre de changement d'activité est véhiculé par un objet de type Intent (intention en anglais). Les activités s'enchaînent les unes aux autres par invocation directe. C'est-à-dire qu'une activité donnée déclenche l'affichage d'une autre activité en appelant la méthode startActivity avec un Intent mentionnant clairement le nom de l'activité.

Voici un exemple représentant la cohabitation listener Activity :

```
private Button create;
this.create = (Button)findViewById(R.id.SinglePlayerGame);
this.create.setOnClickListener(this);

public void onClick(View view) {
    Intent intent = null;
    if( view == create ){
        intent = new Intent(SinglePlayer.this, SinglePlayerLayout.class);
        startActivity(intent);
        this.finish();
    }
}
```

## iOS

Comme Android, nous avons utilisé les widgets<sup>2</sup> pour réaliser notre interface gra-

---

2. Widget est un élément de base d'une interface graphique avec lequel un utilisateur peut interagir (par exemple : une fenêtre, une zone de texte, un bouton, etc).

phique. iOS fournit lui aussi une API couplée à une documentation très complète facilitant la réalisation d'une interface utilisateur ergonomique. Parmi les widgets les plus utiles, nous retrouvons les `UIView`, `UITableView`, `UITableViewCell`, `UILabel`, `UITextField`, `UIButton` qui permettent respectivement de créer une vue, un tableau, une cellule du tableau, des labels, des champs de texte et des boutons.

Pour manipuler ces différents composants graphiques, il est possible et beaucoup plus pratique d'utiliser Interface Builder pour réaliser des menus. Interface Builder permet de gagner du temps lors de la réalisation d'une interface graphique standard (de type menu). Cette réalisation est possible via un outil visuel et non des lignes de code, Interface Builder va créer des fichiers `xib`<sup>3</sup> qui vont générer à la compilation des fichiers `nib`<sup>4</sup>. Ce dernier stocke les informations relatives à l'interface sous forme d'un fichier `xml`, mais nous n'avons pas besoin de savoir réellement comment ça fonctionne car il est inutile de modifier directement le fichier `xml`.

Ensuite pour contrôler les différents éléments de l'interface préalablement créés avec Interface Builder, il suffit de les lier au code de la vue via ce dernier. Grâce à ces liaisons, il nous sera possible de les modifier ou les contrôler avec du code et donc de pouvoir les accorder avec les données du modèle.

La gestion des écouteurs se fait grâce au même procédé qui est utilisé pour lier les éléments de l'interface avec la vue sauf que ça créera une méthode au lieu de créer un champ. Voici un exemple d'écouteur :

```
- (void)playerAction:(id)sender {
    if (sender == playerWhite) {
        self.editorInformation.colorPlayer = @"white";
    }
    else if (sender == playerBlue) {
        self.editorInformation.colorPlayer = @"blue";
    }
    else if (sender == playerRed) {
        self.editorInformation.colorPlayer = @"red";
    }
    else if (sender == playerBlack) {
        self.editorInformation.colorPlayer = @"black";
    }
}

[editorInformation changeTool:@"player"];
```

Pour finir, l'un des principaux points de la gestion de l'interface graphique repose sur le modèle MVC et donc chaque vue doit être associée à un contrôleur.

---

3. `xib` est l'acronyme de `Xml Interface Builder`.

4. `nib` est l'acronyme de `NeXT Interface Builder`.



Première utilisation

Création utilisateur

Gestion utilisateur

Gestion des préférences système

Création de carte (charger)

Création partie solo (tout)

Création partie multi (officielle)

#### 4.1.2 Editeur de carte

##### Le modèle

La carte est la principale information que l'éditeur de carte doit gérer. Pour représenter une carte, nous avons choisi de la représenter sous la forme de deux matrices ayant les mêmes dimensions que la carte. La première matrice contient des *Objects* représentant le sol et la deuxième contient elle aussi des *Objects* mais ces derniers représentent des blocs et non le sol. Pour gérer les deux matrices nous avons utilisé des *NSMutableArray* qui sont des tableaux Objective-C que l'on peut modifier, ils contiennent eux aussi des *NSMutableArray* car il n'existe pas des constructeurs de matrice comme en JAVA. Ensuite l'autre information que la carte doit posséder c'est la position de départ des joueurs. Nous avons utilisé un *NSMutableDictionary* permettant de faire la liaison entre la couleur du joueur et son point de départ.

Lorsque l'utilisateur a terminé de créer sa carte, il a besoin de la sauvegarder. Pour cela nous avons donc utilisé la sérialisation des objets et lors de l'enregistrement de la carte ça génère un fichier « .klob » qui contiendra les informations nécessaires pour la reconstruction de la carte lors de son chargement qui la reconstruira en la désérialisant. Pour utiliser la sérialisation en Objective-C, nous avons juste besoin implémenter le protocole *NSCoding* qui comporte deux méthodes à définir : - *(id)initWithCoder : (NSCoder \*)aDecoder* permettant de désérialiser et - *(void)encodeWithCoder : (NSCoder \*)aCoder* s'occupant de la sérialisation.

##### La vue

Pour faciliter le développement mais aussi sa réutilisabilité, nous avons découpé la vue de l'éditeur de carte en trois zones. Chaque une de ces vues est associée à un contrôleur qui permettra de faire la liaison avec le modèle. Pour concevoir les vues nous avons utilisé la classe *UIView* qui est prévue pour réaliser des vues pour une interface graphique sur iPhone.