



ECOLE
POLYTECHNIQUE
DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES

RAPPORT DE PROJET

Bombberman INFO-H-200

Auteurs :
Nicolas ENGLEBERT
Cédric HANNOTIER
Enes ULUSOY

Année 2014 - 2015

Abstract

“Bomberman”, par Nicolas Englebert, Cédric Hannotier et Enes Ulusoy
Université Libre de Bruxelles, 2014 - 2015

Le but de ce projet est de développer un prototype du jeu Bomberman en JAVA. Si JAVA a été choisi pour ce projet, c’est pour son côté orienté objet. La programmation orientée objet n’est qu’un paradigme de programmation comme la programmation procédurale en est un autre. Ce paradigme - né au début des années 60 par Dahl et Nygaard - est basé sur l’interaction d’objets représentant une idée, un concept, ... possédant sa propre structure lui permettant de communiquer avec d’autres. Cette interaction entre objets permet la conception de programmes évolués. La modélisation de ces interactions conduit à l’utilisation d’un design pattern. Alors qu’initialement le modèle MVC avait été choisi, la présente version de ce projet en diffère quelque peu. Pour diverses difficultés techniques - du à un manque d’expérience - la vue s’est vue fusionnée avec la vue ; la vue-contrôleur vérifie les entrées de l’ utilisateur ainsi que l’interface graphique tandis que le modèle est le cœur du projet. Ce rapport détaille l’approche du problème, la répartition du travail, la structure ainsi que la modélisation des interactions de nos objets.

Mots clefs : bomberman, JAVA, programmation orientée-objet, design pattern, modèle MVC

Contents

1	Problématique et approche	1
1.1	Problématique	1
1.2	Répartition du travail	1
2	Fonctionnalités	2
2.1	Initialisation du jeu	2
2.2	Déroulement de la partie	2
2.3	Bonus et améliorations	2
3	Designs Patterns	3
3.1	Model & ViewController	3
3.2	Implémentation du MVC à notre projet	3
3.2.1	ViewController	3
3.2.2	Model	4
4	Diagrammes UML	6
4.1	Diagramme des classes	6
4.2	Diagramme des séquences	6

1 Problématique et approche

Bien que chacun des membres de notre groupe avait une certaine connaissance de l'informatique (notamment grâce au cours de première), aucun d'entre nous n'avait la moindre connaissance en programmation orientée objet.

1.1 Problématique

N'ayant aucune connaissance dans ce paradigme qu'est la programmation orientée objet, notre première problématique a été d'acquérir les bases théoriques nécessaires.

Après avoir acquis celles-ci, nous avons une bonne compréhension théorique des concepts mais aucune expérience au niveau pratique. Afin de ne pas se lancer dans l'inconnu et pour exploiter au maximum nos connaissances théoriques, nous avons commencé, avant même d'écrire une seule ligne de code, par réaliser le diagramme des classes que nous avons présenté dans notre pré-rapport.

Ce travail préliminaire nous a été d'une grande utilité car nous avons pu développer le jeu de façon rapide et structuré. A l'exception de quelques bonus rajoutés et l'une ou l'autre légère modification, notre diagramme de classe final est semblable à 85% à l'initial.

Le fait que la structure ait été pensée avant toute chose nous a permis de ne rencontrer qu'un faible nombre de bugs lors du développement. Cependant, nous avons été confrontés à une autre problématique importante : notre jeu était extrêmement gourmand en ressources. Le problème était lié à la fréquence de rafraîchissement de notre interface graphique. Pour résoudre ce problème, nous avons optimisé l'affichage par trois changements :

1. La grille complète n'est chargée qu'une seule fois, lors de l'initialisation du jeu.
2. L'affichage n'est plus actualisé de façon continue.
3. Plutôt que de rafraîchir l'intégralité de la fenêtre, seuls les modifications du plateau de jeu sont *repaint*.

Les deux points présentés ici étaient de loin les plus grosses problématiques rencontrées. Bien sûr nous avons connu plusieurs problèmes isolés, mais ceux-ci étaient plus des "petits bugs" qu'une véritable problématique.

1.2 Répartition du travail

Au niveau de la répartition du travail, nous n'avons pas séquencé le projet en parties distinctes mais nous avons travaillé parallèlement sur chacune des parties du projet (la collaboration nous étant grandement facilitée par l'utilisation de *git*). Par exemple, au niveau de la gestion de la bombe, nous avons procédé de la façon suivante :

Nico s'est chargé du dépôt de la bombe et de l'auto-destruction de cette dernière.

Enes s'est chargé de la destruction des blocs causés par l'explosion de la bombe ainsi que la génération aléatoire des bonus.

Cédric s'est chargé de l'auto-déclenchement d'une bombe lorsque celle-ci se fait elle-même exploser par une autre bombe.

Cette approche nous a permis de chacun avoir une connaissance de l'intégralité du code et d'ainsi savoir nous aider de façon rapide lors de l'apparition d'un bug.

2 Fonctionnalités

2.1 Initialisation du jeu

La première fonctionnalité de notre application est la gestion d'un mode *multijoueur*. L'initialisation commence par une interaction avec les différents joueurs via une interface graphique (composée de *boutons* et de *champs de texte*) afin de recueillir leur nombre ainsi que leurs noms. Un tutoriel est également disponible, afin d'informer les joueurs des règles du jeu ainsi que les commandes.

Le choix des joueurs étant fait, le plateau de jeu s'initialise. La disposition des personnages se fait sur les coins de celui-ci, la position des blocs incassables est fixe mais celle des blocs cassables se fait de façon aléatoire.

2.2 Déroulement de la partie

Comme tout *Bombberman*, chaque joueur est capable de contrôler son personnage à l'aide du clavier. Les touches directionnelles et de dépôt de bombe pour chaque joueur seront spécifiées préalablement dans le tutoriel.

Le programme est capable de gérer la collision entre les joueurs et les blocs, mais également entre les différents personnages sans oublier les bombes.

Lorsqu'une bombe explose, une *croix de flamme* influe sur tous les éléments se trouvant sur son chemin, sauf s'il s'agit d'un bloc incassable. En outre, les joueurs touchés perdront une vie et les autres blocs seront détruits. Le rayon d'action de la flamme est initialement d'une case, mais peut augmenter si le joueur gagne le bonus adéquat.

Lorsqu'un joueur gagne la partie, une boîte de dialogue apparaît et l'informe de sa victoire pour ensuite rediriger les joueurs vers l'écran d'accueil.

2.3 Bonus et améliorations

Une brique cassable ne se contente pas toujours d'être détruite lors d'une explosion. En effet, elle peut entraîner l'apparition de divers bonus choisis aléatoirement dont la fréquence d'apparition dépend de l'avantage qu'il donne. Ces derniers sont classés de la sorte :

- **Augmentation du nombre de bombes** : le joueur se verra attribuer un nombre plus grand de bombes à déposer simultanément.
- **Augmentation du nombre de vies** : le joueur pourra récupérer une vie à l'acquisition de ce bonus.
- **Portée de la bombe** : augmente la portée destructives de la bombe.
- **La bombe atomique** : tous les autres joueurs perdent une vie.
- **Le téléporteur** : le joueur est téléporté de façon aléatoire sur une case vide.

Afin de donner un peu plus de vie au jeu, la musique de fond *leekspin* est jouée. L'explosion d'une bombe produit le son d'une explosion et le bonus *bombe atomique* suspend la musique de fond et joue un son des plus lugubres qu'il puisse exister.

3 Designs Patterns

Alors qu'initialement nous comptions adopter le modèle MVC, nous avons connu quelques soucis lors de sa mise en place. Du à notre manque d'expérience dans le monde de la programmation orientée objet, nous n'avons pas réussi à scinder de façon complète le contrôleur de la vue. Ne ne sommes pas parvenu à dissocier la gestion des entrées utilisateurs de l'actualisation de l'affichage produit par ces mêmes actions. Étant quelque peu en manque de connaissance et surtout en manque de temps, nous avons préféré trouver une solution alternative, à savoir le regroupement de ces deux entités.

3.1 Model & ViewController

Le design pattern que nous avons choisi pour notre projet est le modèle-vue-contrôleur, MVC en abrégé. L'intérêt de l'utilisation de ce patron est de séparer notre programme en trois parties différentes, ayant chacune un objectif particulier :

1. Un modèle
2. Une vue-contrôleur

Le *modèle* se charge de la gestion des données : les objets qui le constituent sont chargés de leurs traitements. Après avoir mis à jour les données, le modèle informera la vue-contrôleur de se mettre à jour. La *vue-contrôleur* est chargée d'un double rôle. Son premier consiste en la mise à jour de l'interface graphique sur demande du modèle. Le deuxième rôle qu'elle tient est de vérifier la validité des données entrées par l'utilisateur et des les transmettre au modèle si celles-ci s'avèrent valides. La seule différence entre ce modèle et le modèle MVC est que, pour des raisons purement techniques, nous n'avons pas été en mesure de séparer la vue du contrôleur en deux classes distinctes.

La motivation à utiliser ce design pattern découle de la séparation claire entre l'interface graphique gérée par la vue et le traitement des données par le modèle. Cette séparation rend le développement de l'interface graphique plus simple que si celle-ci devait être "éparpillée" dans les différentes parties du code. La séparation des tâches est un avantage de poids pour ce projet réalisé en équipe : la répartition des travaux ainsi que la maintenance peut se faire de façon naturelle. De plus, "découper" un code en plusieurs parties diminue grandement la complexité générale au niveau de la conception.

3.2 Implémentation du MVC à notre projet

3.2.1 ViewController

Notre *ViewController* se compose de deux classes :

1. `GameWindow`
2. `GamePanel`

La classe `GameWindow` hérite de la classe `GameJFrame`. C'est elle qui se charge d'afficher une fenêtre à une taille pré configurée. Le contenu de cette fenêtre est géré par la classe `GamePanel` qui hérite de la classe `JPanel`. Celle-ci récupère les modifications effectuées par le modèle après que celui-ci lui ait informé de se mettre à jour pour ensuite les afficher mais cette classe se charge également de récupérer les entrées utilisateurs pour les transmettre au modèle.

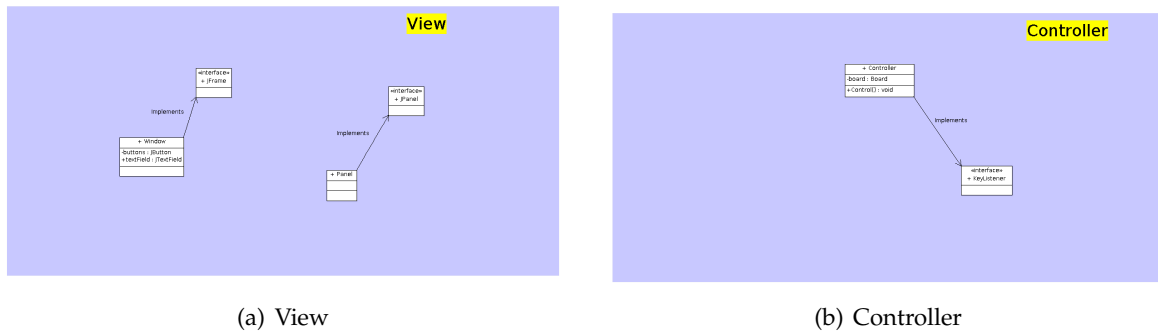


Figure 1: Diagramme des classes : View/Controller **A CHANGER**

3.2.2 Model

Notre *model* se compose de deux interfaces :

1. IPlayer
2. Explosion

et de dix classes dont deux classes mère (●) où la première est abstraite. Les implémentations d'interfaces sont représentées par \rightarrow , les classes filles par une sous-liste (-) et l'implémentation d'interface entre une classe concrète et une interface par \rightarrow^1 :



La classe avec laquelle notre contrôleur communique est la classe Board. La classe board crée une matrice remplie d'éléments de type Element : joueurs, différents blocs, bombes, etc. Lorsque l'utilisateur demande d'effectuer un déplacement, celui-ci est renseigné à cette classe via le contrôleur pour finalement modifier les attributs de positions du joueur. La matrice remise à jour, un message sera envoyé à la vue afin de remettre à jour l'interface graphique.

Les différents composants de notre classe Board sont tous de types Element. La classe Element possède cinq classes filles : Player, Bomb, Block, Bedrock, Bonus. Le nom des classes est assez explicite si ce n'est pour les deux classes chargées de la gestion des blocs :

1. Block : une case pouvant être détruite suite à l'explosion d'une bombe.
2. Bedrock : une case ne pouvant être détruite (ainsi qu'une petite référence à *Minecraft*).

Afin de gérer les explosions de façon la plus efficace que possible, la méthode `explode()` a été définie au sein de la classe Element. Or, cette méthode n'a pas le même comportement pour un objet de la classe Bedrock que pour un de la classe Bomb mais a le même pour un objet de la classe Bonus. Afin que cette méthode puisse être utilisée de façon polymorphe et

¹Pour plus de clarté, consultez la Figure 2

pour éviter les doublons de code, l'interface `Explosion` implémentée à `Element` permet de définir un comportement adapté à l'objet concerné.

Cependant, la classe `Player` doit posséder des attributs et des méthodes que les autres classes héritant de `Element` ne possèdent pas. Afin de conserver le polymorphisme, l'interface `IPlayer` a été implémentée à notre classe `Player`.

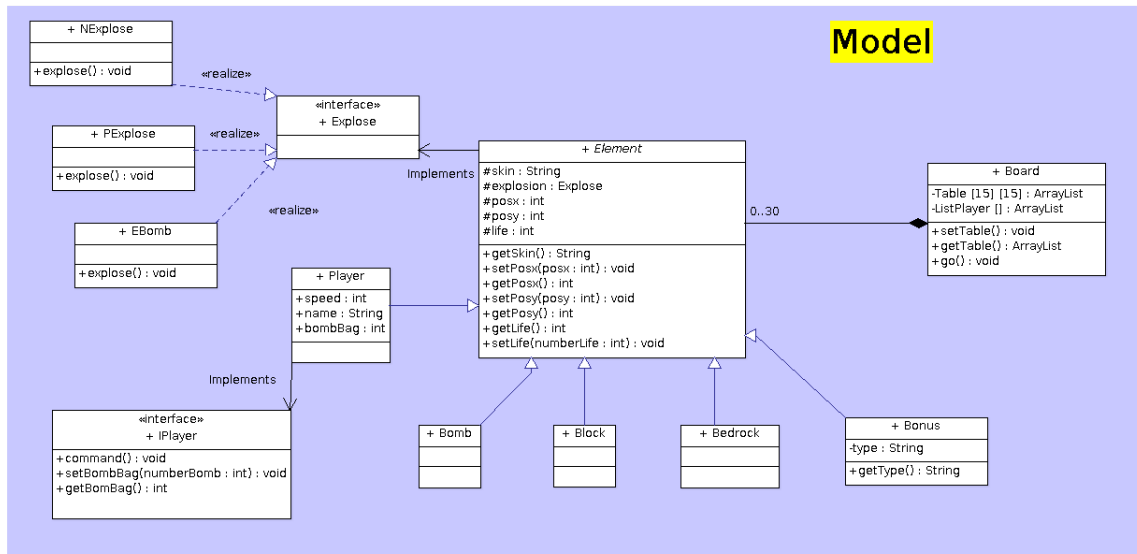


Figure 2: Diagramme les classes : Model

4 Diagrammes UML

4.1 Diagramme des classes

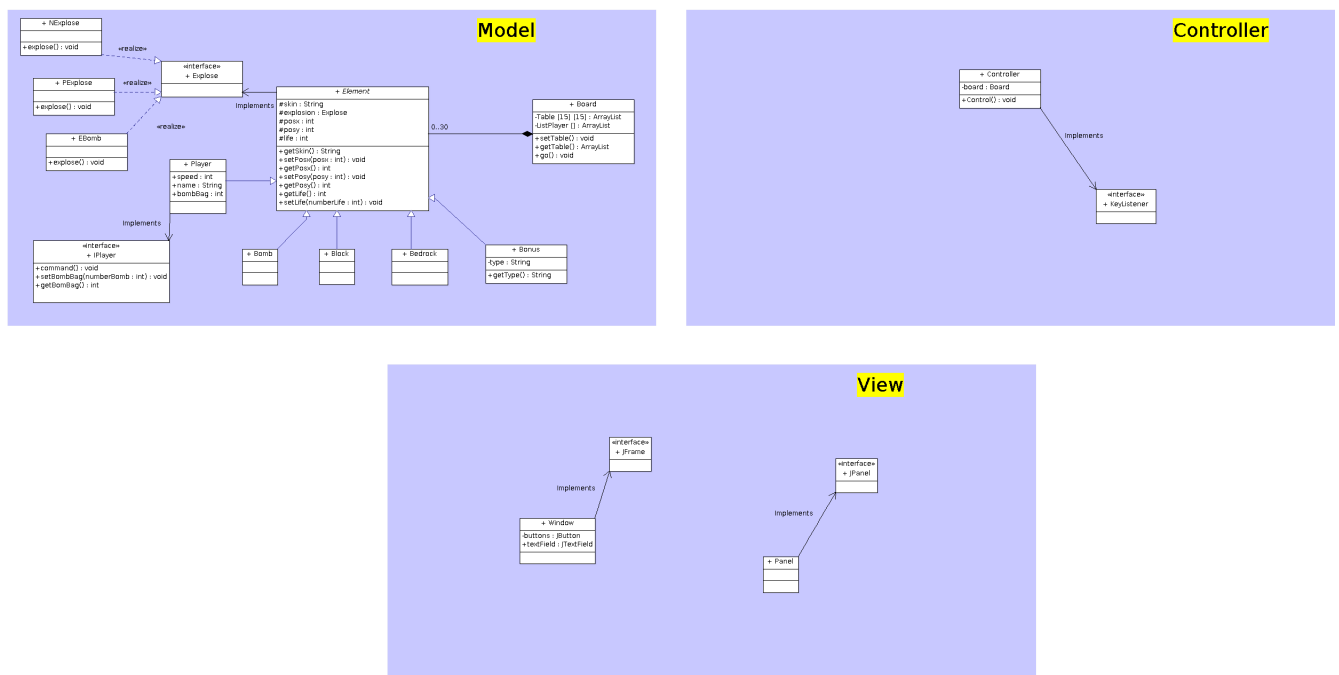


Figure 3: Diagramme des séquences

4.2 Diagramme des séquences

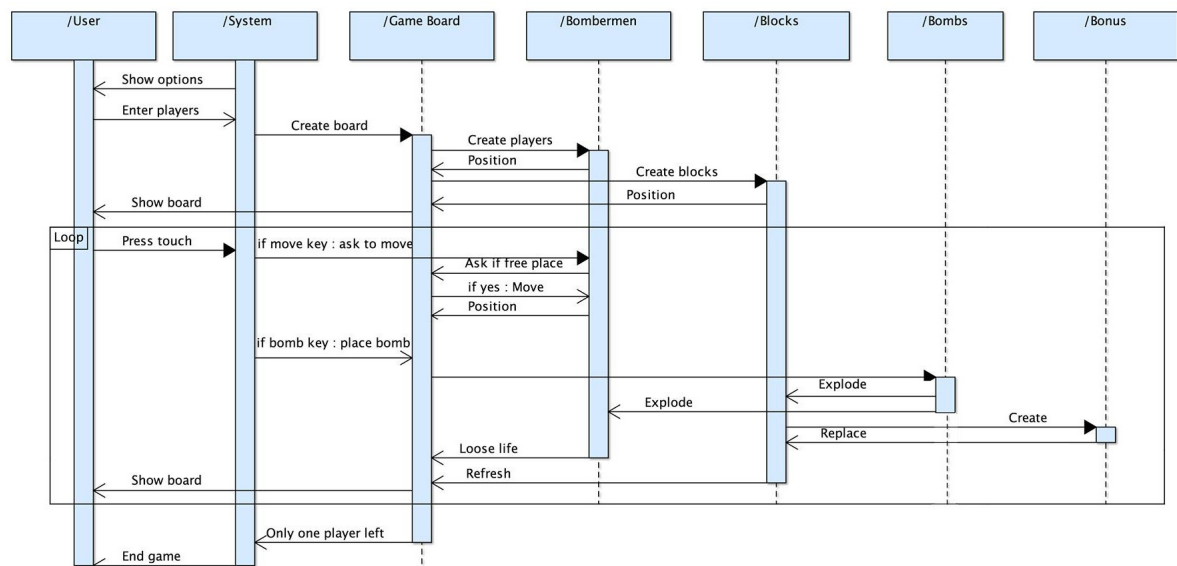


Figure 4: Diagramme des séquences