ECOLE
POLYTECHNIQUE
DE BRUXELLES

UNIVERSITÉ LIBRE DE BRUXELLES

RAPPORT DE PROJET

Bomberman
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 Architecture

Avant même de commencer le développement du *Bomberman*, il a fallu s'intéresser sur comment les objets issus de nos différentes classes allaient interagir entre eux. Même si cette étape n'est pas fondamentalement indispensable, veiller à adopter un *design pattern* structuré s'avère l'être si l'on désire s'affranchir au maximum de toute redondance, lourdeur dans la gestion des classes ...

1.1 Model-View-Controller

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
3. Un contrôleur

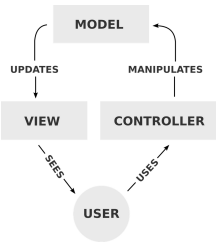


Figure 1 – MVC

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 de se mettre à jour. La *vue* concerne l'interface graphique de notre programme, chargé d'interagir avec l'utilisateur. Le *contrôleur* reçoit les actions de l'utilisateur entrées via la vue. Son but est de vérifier la validité de ces données pour ensuite prévenir le modèle d'en prendre compte.

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.

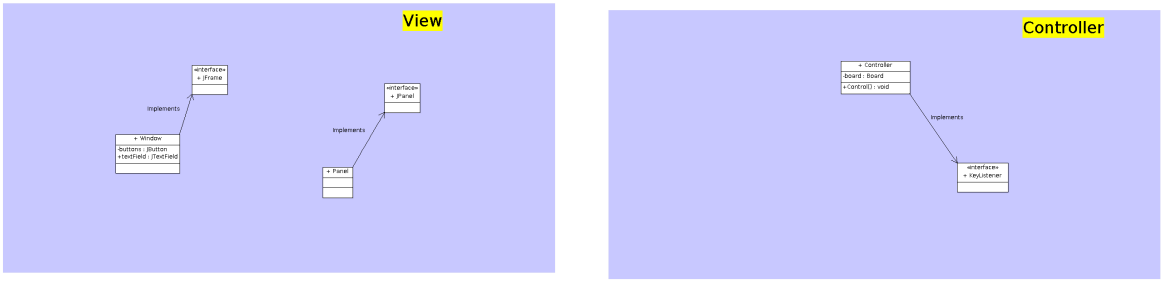
1.2 Implémentation du MVC à notre projet

1.2.1 View

Notre *view* se compose de deux classes :

1. `Window`
2. `Panel`

La classe `Window` hérite de la classe `JFrame`. 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 `Panel` 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.



(a) View (b) Controller

Figure 2 – Diagramme des classes : View/Controller

1.2.2 Controller

Notre *controller* ne se compose que d'une unique classe pour l'instant :

1. `Controller`

S'il n'y a qu'une seule classe, c'est parce que le nombre d'actions susceptibles d'être effectuées par l'utilisateur est assez limité. En effet, la classe `Controller` se contente de récupérer les touches rentrées par l'utilisateur (par exemple, les touches directionnelles) et de les transmettre au modèle si cette dernière déclenche bien une action. Cette figure est encore incomplète dans la mesure où il va falloir ajouter les différents *ActionListener*. Ceux-ci concernent les différents boutons qui seront présents dans l'interface graphique avant le lancement du jeu. "L'esthétique" du projet n'ayant pas encore été approfondie, le groupe a préféré garder un modèle purifié, qui sera plus détaillé pour le rapport final.

1.2.3 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 →, les classes filles par une sous-liste (-) et l'implémentation d'interface entre une classe concrète et une interface par ↠[1] :

- `Element`
  - → `Explosion`
    - ↠ `NExplode`
    - ↠ `PExplode`
    - ↠ `EBomb`
  - `Player` → `IPlayer`
  - `Bomb`
  - `Block`
  - `Bedrock`
  - `Bonus`
- `Board`

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

[1] Pour plus de clarté, consultez la Figure 3

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 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`.

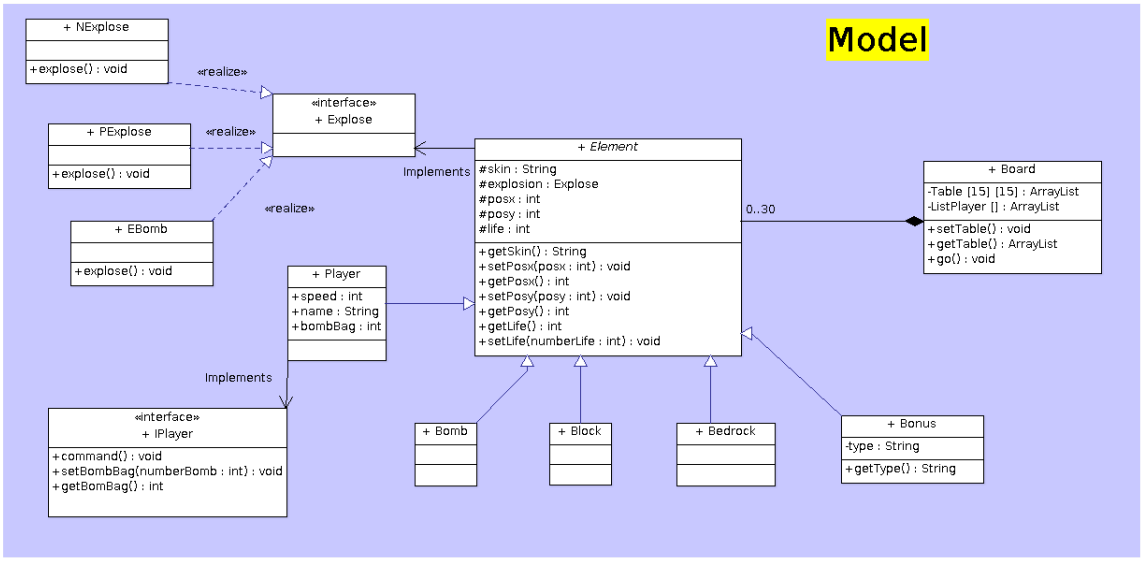


Figure 3 – Diagramme les classes : Model

2 Fonctionnalités

2.1 Initialisation du jeu

Premièrement, puisque la gestion des IA n'est pas imposée, l'utilisateur doit pouvoir jouer en *multijoueur*. C'est pour cela que le programme débute avec une interaction avec l'utilisateur, à l'aide d'une interface graphique composée de *boutons* et de *champs de texte*, afin de recueillir le nombre de joueurs désirés ainsi que leur nom.
Ensuite, la fenêtre affiche le plateau de jeu sur lequel les différents éléments ont été initialisés,

en tenant compte des choix de l'utilisateur. La disposition des personnages se fait sur les coins du plateau et la position des blocs incassables est fixe, alors que celle des blocs cassables est choisie aléatoirement.

2.2 Déroulement de la partie

Comme tout *Bomberman*, chaque joueur est capable de contrôler son personnage à l'aide des touches du clavier. Les touches directionnelles et de dépôt de bombe pour chaque joueur seront spécifiées à l'avance.
Le programme est capable de gérer la collision entre les joueurs et les blocs, mais pas avec les autres personnages et les bombes. Le personnage se contentera donc de traverser ces éléments sans interaction.
Lorsqu'une bombe explose, une *croix de flamme* influe sur tous les éléments se trouvant sur son chemin, sauf la brique non cassable. En outre, les joueurs touchés perdront une vie et les briques cassables seront détruites. Le rayon d'action de la flamme est initialement de 1 case, mais peut augmenter à l'aide des différents bonus disponibles durant la partie.

2.3 Les bonus

Une brique cassable ne se contente pas d'être détruite lors d'une explosion. En effet, elle peut entraîner l'apparition de divers bonus choisi aléatoirement parmi ceux implémentés dans le code. Ces derniers sont classés de la sorte :

- **Augmentation de vitesse :** le joueur pourra se déplacer plus rapidement sur le plateau de jeu.
- **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.
- **Le "freeze" :** le joueur qui obtient ce bonus "gèle" tous les autres joueurs qui ne pourront effectuer aucune action durant un temps déterminé.

Cependant, un bonus n'apparaît pas tout le temps. En effet, le principe repose sur le fait d'effectuer une opération aléatoire à l'aide de la méthode *random* de la class *math* et si le résultat est un nombre supérieur à 0.5, un bonus apparaîtra. Dans le cas contraire, le bloc se contente d'exploser.

3 Diagrammes UML

3.1 Diagramme des classes

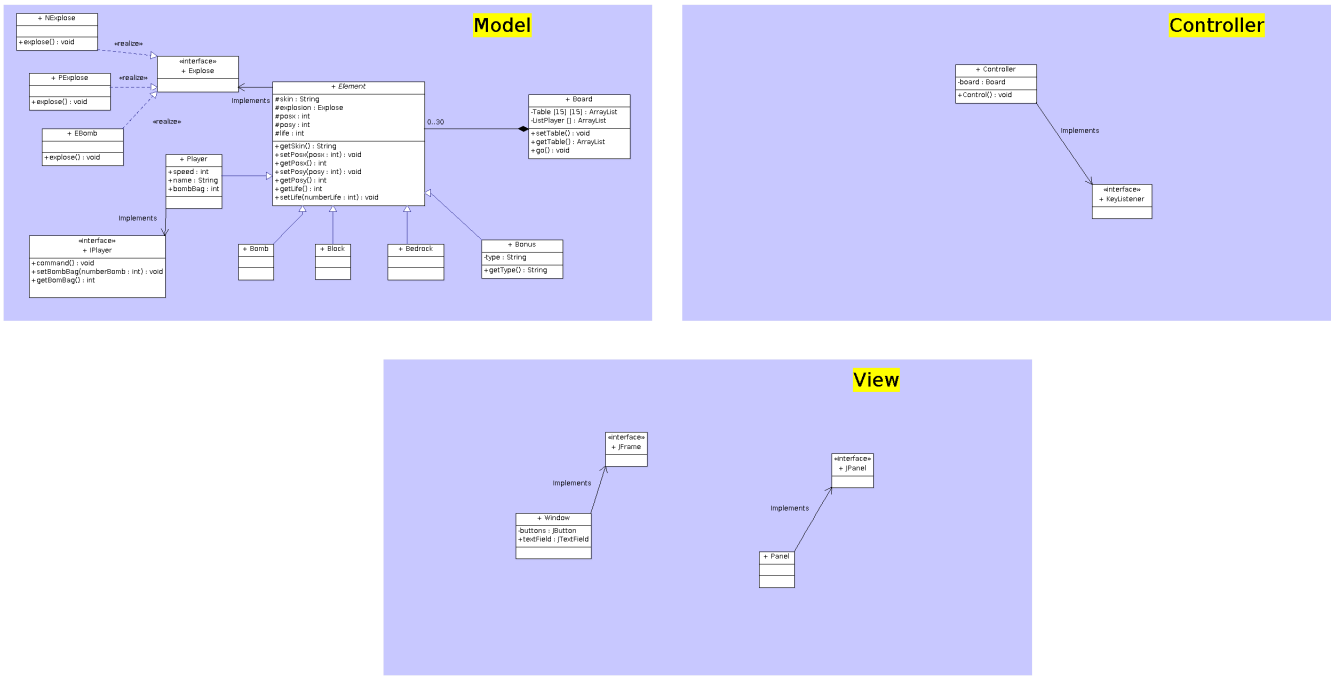


Figure 4 – Diagramme des séquences

3.2 Diagramme des séquences

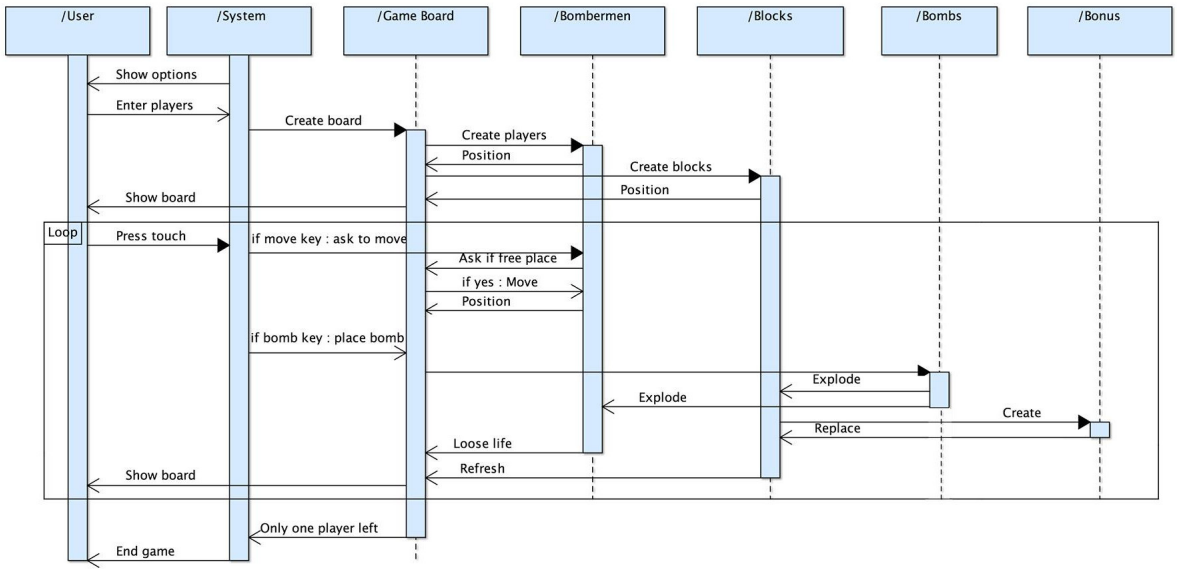


Figure 5 – Diagramme des séquences