

Livrable n°4 : Rapport de projet



Conception et Programmation Objet - Gestion de projet

Destinataires : M. Xavier **Gregut**, M. Neeraj **Singh**, M. Loic **Thierry**, Mme. Aurélie **Hurault**

Numéro de l'équipe : 4

Membres de l'équipe :

<u>Nom</u>	<u>Prénom</u>	<u>Rôle</u>
François	Yoann	Développeur - Responsable IHM
Pradier	Corentin	Développeur - Responsable BackEnd
Loppinet	Stéphane	Développeur - Responsable QA
Silvestre	Thomas	Développeur - Architecte logiciel
Ziuzin	Nikita	Développeur - UI/UX
Chaveroux	Pierre	Développeur - Chef de projet

Sommaire

Introduction	1
1. Fonctionnalités Bomber7	2
1.1. Fonctionnalités principales	2
1.2. Fonctionnalités secondaires :	3
1.3. Fonctionnalités qui font la différence	4
2. Architecture logicielle de l'Application	5
2.1. Diagramme UML général	5
2.2. Modèle MVC : UML des packages	6
2.2.1. Le modèle	6
2.2.1.1. Package entities	7
2.2.1.2. Package exceptions	8
2.2.1.3. Package map	8
2.2.1.4. Package square	9
2.2.1.5. Package texture	10
2.2.2. Les vues	11
2.2.2.1. L'interface MVCCComponent	11
2.2.2.2. Les écrans	11
2.2.2.3. Les composants UI	13
2.2.3. Les contrôleurs	15
3. Choix de Conception et Réalisation	16
3.1. Principaux choix de conception	16
3.2. Patrons de conceptions utilisés	16
3.2.1. MVC	16
3.2.2. Observateur	17
3.2.3. Singleton	17
3.2.4. Fabrique	17
3.2.5. Stratégie	17
3.3. Problèmes rencontrés et solutions apportées	18
3.3.1. Classe entité	18
3.3.2. Patron décorateur	18
3.3.3. Horloge & rafraîchissement	19
3.3.4. Test des classes utilisant indirectement libgdx & les assets	19
4. Méthodes Agiles	21
4.1. Description de l'organisation de l'équipe	21
4.2. Méthodes agiles mises en œuvre	21
4.3. Outils et pratiques utilisés	22

Introduction

Le projet Bomber7 est né de la volonté de notre équipe de concevoir un jeu vidéo multijoueur local, compétitif, s'inspirant librement du célèbre jeu rétro Bomberman, un jeu rétro apparu en 1983 sur MSX et ZX Spectrum. Nous avons choisi de revisiter ce concept en adaptant les arènes de jeu à l'univers de l'ENSEEIHT, notre école, afin d'offrir aux joueurs une expérience amusante et originale.

Dans Bomber7, jusqu'à quatre joueurs s'affrontent sur des cartes 2D différentes, enrichies de power-ups et de mécaniques de jeu ayant pour but de dynamiser les parties. L'objectif est de faire exploser les adversaires (monstres et joueurs humains/IA), et d'être le dernier en vie dans l'arène. L'expérience utilisateur est au cœur des enjeux de notre développement afin d'offrir des parties agréables et différentes les unes des autres.

L'interface utilisateur adoptée reste avant tout épurée, avec des pages simples : boutons, sliders, menus défilants... Un point d'honneur a été imposé, afin de proposer une interface intuitive accompagnée d'une ambiance sonore originale. L'utilisation des assistants IA (Dallee, Mistral AI Image Generation) a été strictement réduite à la génération de certains éléments visuels (logos, "sprites", etc, ...), l'aspect graphique sortant du cadre d'évaluation de ce projet.

Ce rapport a pour objectif de synthétiser le travail réalisé par l'ensemble du groupe en détaillant les choix conceptuels, techniques et organisationnels qui ont jalonné le développement de Bomber7. Il a aussi pour objectif de compléter la soutenance orale de projet du 20 juin 2025.

Dans un premier temps, nous présenterons les principales fonctionnalités effectivement développées dans notre jeu, en faisant un comparatif avec celles annoncées dans le cahier des fonctionnalités.

Dans un second temps, nous détaillerons l'architecture logicielle mise en place et la place des patrons de conception au sein de cette dernière, illustrée par des diagrammes UML.

Dans un troisième temps, nous détaillerons les choix techniques majeurs qu'il nous semble pertinent d'aborder et qui reflètent la qualité du travail effectué. Le code produit incluant une Javadoc et des commentaires, nous ne détaillerons que les aspects les plus significatifs de notre application.

Dans un troisième temps, nous présenterons les méthodes de gestion de projet adoptées, notamment l'utilisation de Github et des pratiques agiles pour organiser le travail en équipe. Puis, nous terminerons en partageant les difficultés rencontrées, les solutions apportées, ainsi que les perspectives d'évolution envisagées.

À travers ce document, nous souhaitons vous offrir une vision claire et structurée de notre démarche, tout en valorisant l'investissement de chaque membre de l'équipe. Nous espérons que ce rapport reflétera l'énergie et la persévérance investies dans ce projet.

1. Fonctionnalités Bomber7

L'objectif de cette première partie est de présenter un comparatif sur l'ensemble des fonctionnalités prévues de Bomber 7, précédemment mentionnées dans le rapport du Jalon "Sélection du sujet de projet", et les fonctionnalités réellement développées. Nous tâcherons d'en tirer un état d'avancement du développement par rapport au cahier des charges initial.

1.1. Fonctionnalités principales



- **Carte de jeu** : Grille 2D avec cases destructibles et indestructibles.
- **Direction artistique** : Les cartes de jeu sont représentatives des salles de l'ENSEEIHT.

Quatre cartes de jeu ont été développées et sont fonctionnelles. Les cartes représentent le CROUS, la cour centrale avec le Churros, le foyer étudiant et l'amphithéâtre de la halle au grain. Les blocs indestructibles régissent les limites de map et les assets de map, comme le Churros ou les tables du Foy (diminutif pour foyer étudiant). Les blocs destructibles, une fois détruits par le joueur, permettent de se déplacer sur la map.

- **Logique du personnage principal** : Déplacement dans les 4 directions 2D de l'espace (haut, bas, gauche, droite), pose de bombes, collisions avec les murs et obstacles

Les personnages de notre jeu peuvent se déplacer de haut en bas et de droite à gauche. Les déplacements en diagonale n'existent pas. Les personnages peuvent aussi poser des bombes pour détruire des joueurs et des blocs (destructibles). Ils ne peuvent également pas franchir les murs et assets incassables (tables du Foy, ...).

- **Logique des bombes** : Explosion après un délai donné avec une portée variable (en fonction des bonus), destruction des blocs destructibles uniquement.

La bombe par défaut en possession du joueur (sans bonus) est une bombe à retardement, qui explose au bout de 2,5 secondes après l'avoir posée. La bombe a un impact sur les blocs cassables et sur les joueurs (y compris le joueur qui la pose).

- **Logique des ennemis** : déplacement aléatoire des monstres, déplacement semi-intelligent des joueurs artificiels (algorithme basé sur l'aléatoire (Facile), le calcul de distances (Intermédiaire), avec adaptation du comportement (Difficile, avec possible implantation de Machine Learning), élimination par les explosions.

Seul un joueur IA avec un niveau d'intelligence faible a été développé. Les autres niveaux d'intelligence n'ont pas fait partie des objectifs des sprints effectués jusqu'à présent.

1.2. Fonctionnalités secondaires :



- **Bonus** : blocs permettant d'obtenir des pouvoirs

Les bonus suivants ont pu être implémentés au cours des quatre premiers sprints :

- Poser 1 bombe supplémentaire
- Obtention d'une vie supplémentaire
- Bombe à détonation manuelle (choix du timing d'explosion)
- Boost vitesse (Glisse sur Rollers) : permet de décupler la vitesse de déplacement (x2)

La logique des bonus et leurs vues ont pu être développées indépendamment. De premières idées de développement ont été émises pour les pouvoirs suivants, mais n'ont pas été la priorité de nos derniers sprints. Ainsi, le jeu ne possède pas les fonctionnalités de bonus suivantes :

- Gilet pare-balle (insensible aux bombes pendant 30 secondes),
- Plus de portée sur les bombes,
- Bonus avancées : ("Yoshi"), Malus aléatoire (commandes inversées, ...)

- **Interface Graphique Utilisateur** : menu principal : choix du nombre de joueurs, du nombre de robots, de la carte de jeu, menu pause : paramètres de jeu (quitter, retour, ...) animations des personnages : "sprites" (mouvement des jambes, bras, ...)

Les menus ont été les premières vues (et screens) qui ont été MERGE dans la main. Ainsi, les menus d'accueil, de sélection des personnages et des paramètres sont disponibles dans notre GUI. Les animations des personnages sont gérées par les méthodes draw() et render() de la vueMap, dont la fréquence du tic est instanciée par LibGDX.

- **Multijoueur local** : Jeu sur un seul clavier de un à quatre joueurs, bindings de touches personnalisables pour chaque joueur

Bomber7 est un jeu strictement en local, mais le mode multijoueur est la partie intéressante de l'expérience utilisateur. Bomber7 est à voir sous l'angle d'un jeu de borne d'arcade, où les touches nécessaires pour faire jouer les quatre joueurs sont sur le même clavier. Pour plus de confort, il est possible de connecter un clavier distinct à l'ordinateur pour chaque joueur, à l'image de ce qui a été fait lors de la démonstration technique.

- **Mode de jeu en équipe (2 contre 2)** : Les participants d'une même équipe choisissent une couleur. Ces derniers ne peuvent pas s'entre-tuer et chercheront à gagner ensemble.

Ce mode de jeu n'a pas été implémenté concrètement. Cependant, il est tout à fait possible que les joueurs fassent équipe en jeu en échangeant à l'oral pendant la partie.

1.3. Fonctionnalités qui font la différence

- **Cartes (et objets) avec pour thème l'ENSEEIHT :**

Toutes les différentes cartes qui étaient prévues à la création ont été créées. Il est donc possible de jouer sur les quatre cartes phare de notre école :

- Cour principale de l'N7 (Churros)
- Le Foyer Étudiant (Foy)
- Restaurant universitaire (Crous)
- Salle des diplômes (Halle aux grains)



Le mécanisme de déblocage des cartes dans l'ordre n'a pas été implémenté par manque de temps. Cependant, la charge de travail nécessaire à l'implémentation de cette fonctionnalité ne semble pas être particulièrement importante, inhérente au calcul des points du joueur en partie qui est déjà fonctionnel dans notre jeu.

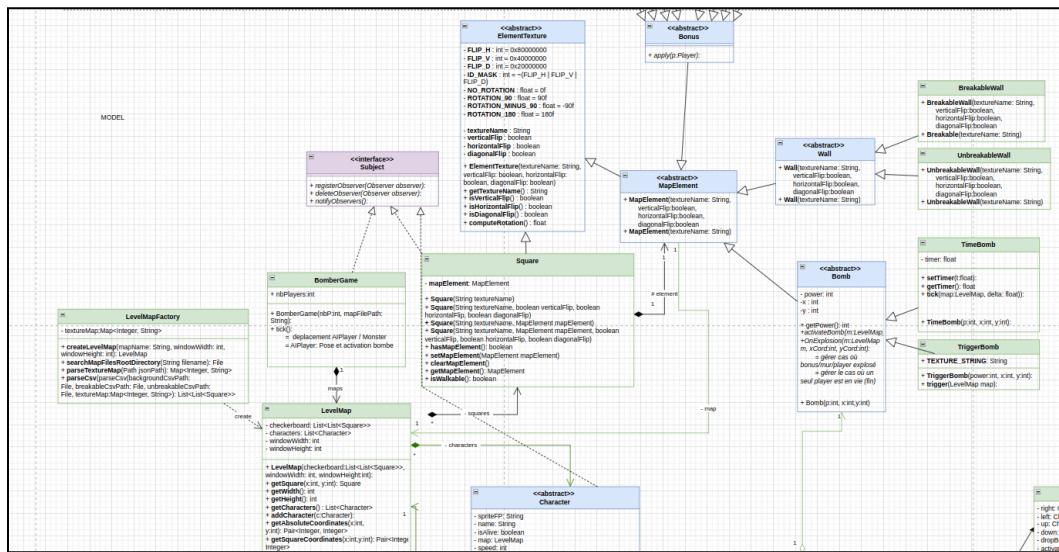
Ainsi, et plus généralement, il est considéré par l'équipe que le jeu a été développé à hauteur de 70% du cahier des charges original (fonctionnalités.pdf).

2. Architecture logicielle de l'Application

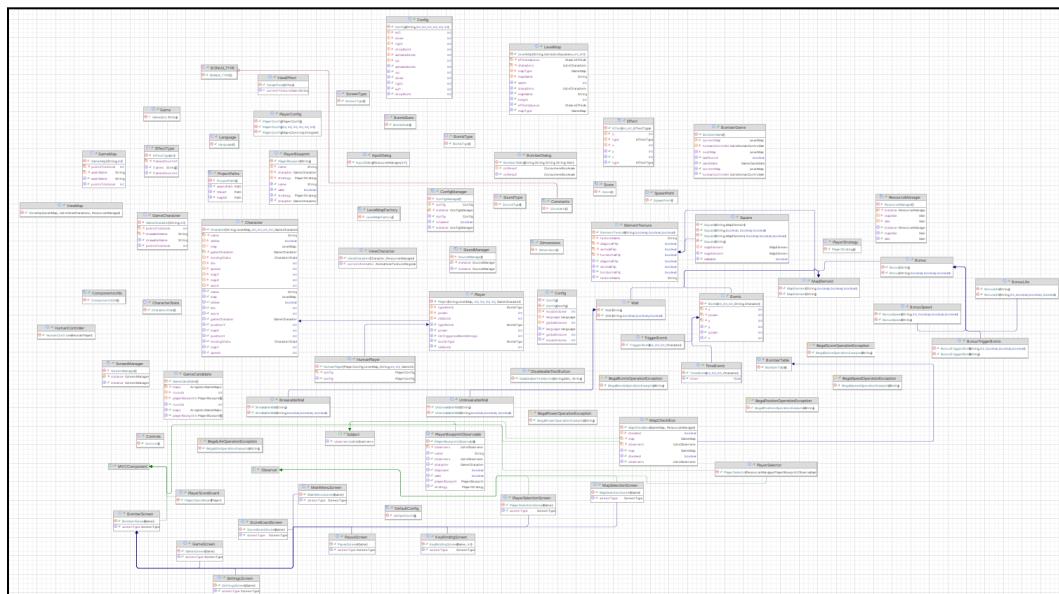
2.1. Diagramme UML général

Ce premier diagramme général a pour objectif de représenter à l'échelle notre architecture système pour ce projet. Nous avons travaillé tout au long du développement de l'application sur [draw.io](#) pour mettre à jour le diagramme UML qui a grandi avec les nouvelles fonctionnalités. Nous tâcherons de détailler dans cette partie les différents blocs et packages de notre application, en nous basant sur notre diagramme [draw.io](#) effectués à la main, ainsi que sur le diagramme UML généré par IntelliJ IDEA, reflétant les classes effectivement développée.

- Diagramme UML [draw.io](#) :



- Diagramme UML généré depuis IntelliJ IDEA :



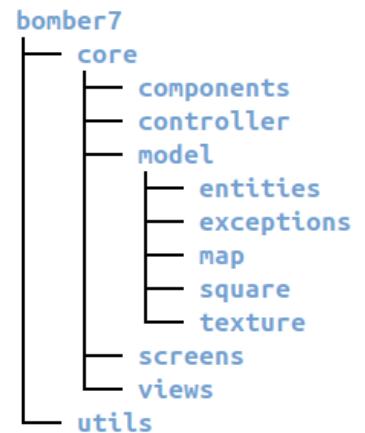
2.2. Modèle MVC : UML des packages

Nous tâcherons dans cette partie d'aborder en détail la structure des packages et ses interactions entre les classes de notre modèle, de nos vue et de nos contrôleurs. Cette partie du rapport est à considérer solidairement à la partie traitant des choix de conceptions, traitant des patrons de conceptions dont le résultat de l'utilisation est visible dans les diagrammes UML.

2.2.1. Le modèle

Le modèle est la partie centrale de notre système. En effet, c'est ce dernier qui est responsable de la logique de création, gestion et suppression de tous les composants de notre application. Le modèle se répartie en différents packages, les suivants :

- **components** : éléments graphiques personnalisés
- **controller** : configuration des touches du joueur
- **model** : la logique de notre application
- **entities** : les entités du jeu → Joueurs, IA, monstres, etc.
- **exceptions** : règles sur les setters (vie, vitesse, ...)
- **map** : création des cartes
- **square** : éléments relatifs aux cases de la map
- **texture** : manipulation des textures
- **screens** : écrans du jeu, visibles à l'utilisateur
- **views** : sous-mécanismes d'affichages



Cinq classes essentielles sont à la racine du projet, sous ./bomber/core :

BomberGame	Cette classe joue le rôle de main. Lorsque le jeu est lancé par l'utilisateur, le lanceur de libGDX Lwjgl3Launcher.java lance la classe BomberGame.java. Cette classe, héritant de la classe Game de LibGDX permet d'initialiser les gestionnaires de ressources, de lancer, pauser et quitter le jeu. C'est aussi dans cette classe que l'on peut retrouver la logique pure du jeu, ou l'on vérifie si la partie est gagnée par un des joueurs, les roulements de cartes entre les rounds ou encore l'initialisation des personnages.
ResourceManager	<u>Singleton</u> qui gère le chargement de l'ensemble des assets de notre application. Depuis cette classe, on récupère aussi bien les textures des éléments UI, des cases, des characters, que les bundle I18N contenant l'ensemble des éléments textuels de notre jeu.
ConfigManager	<u>Singleton</u> qui gère le chargement et la sauvegarde des paramètres du jeu. C'est grâce à lui qu'on peut augmenter le volume ou modifier les touches tout en les sauvegardant même si le jeu a été fermé. Cette sauvegarde se fait sous forme d'une classe serializable qu'on écrit dans un fichier .dat à chaque sauvegarde & qu'on charge au lancement de l'application.

ScreenManager	<u>Singleton</u> qui gère le passage d'un <u>écran</u> à un autre, en faisant appel à sa méthode showScreen(ScreenType). Il gère une pile d'écrans visités ce qui nous permet de facilement revenir aux écrans précédents, et il permet également de sauvegarder (ou pas) l'état d'un écran pour qu'en revenant dessus il reste dans le même état.
SoundManager	<u>Singleton</u> qui charge les sons et musiques du jeu, et qui permet de les jouer.

2.2.1.1. Package entities

Le package entities est le package qui contient toutes les classes inhérentes aux personnages. On détaillera dans cette partie le diagramme UML du package suivant :

Quatre classes sont en relations :

- Character.java :

Cette classe est la base de toutes les entités qui se déplacent sur notre jeu. Elle permet de centraliser la logique commune aux joueurs et monstres : leurs positions, les attributs (vie, vitesse, score, spawn)

- CharacterState.java :

Cette énumération comporte l'état du mouvement du joueur : il peut être en mouvement vers la droite, la gauche, le bas, le haut, statique ou mort.

- HumanPlayer :

Cette classe hérite de "Player" et permet créer un joueur en lui associant une configuration, une carte, un nom, des coordonnées, et un sprite.

- Player :

Le joueur est une spécialisation de la classe "Character" qui ajoute toutes les spécificités du comportement d'un joueur : état de sa vie, de ses bombes, de son déplacement, et ainsi de suite.



2.2.1.2. Package exceptions

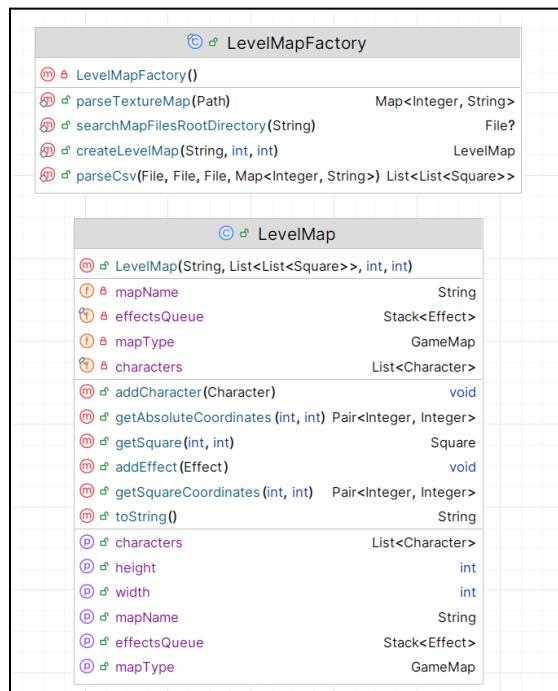
Le package exceptions est le package qui contient toutes les exceptions de notre programme. On y trouvera 6 exceptions :

- IllegalBombOperationException
- IllegalLifeOperationException
- IllegalPositionOperationException
- IllegalPowerOperationException
- IllegalScoreOperationException
- IllegalSpeedOperationException

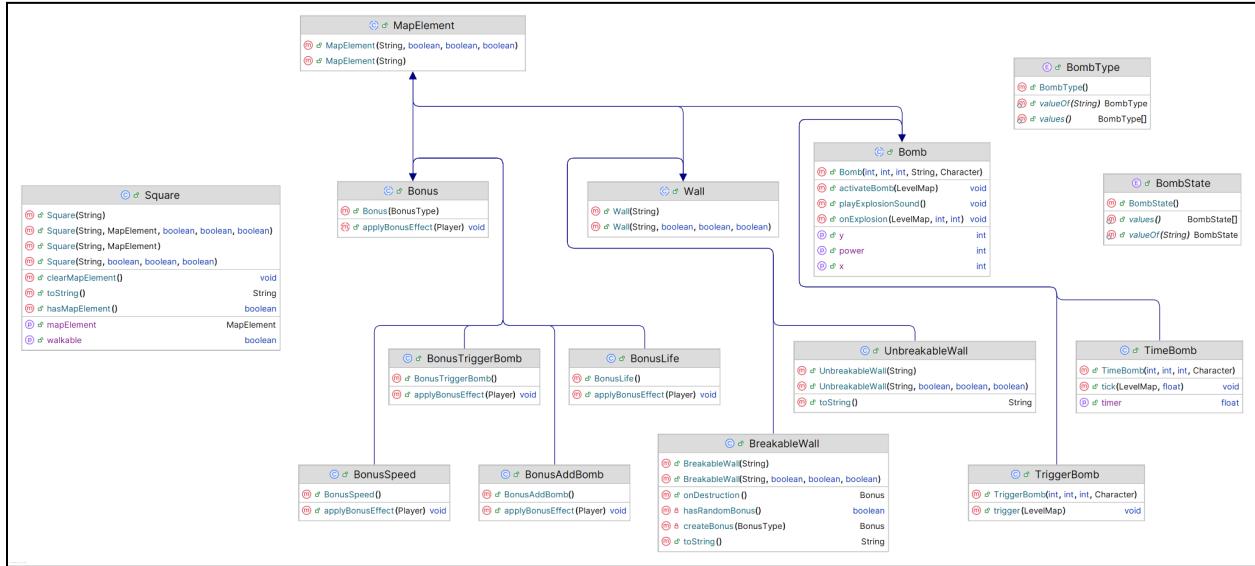
Toutes héritent de “RuntimeException” et traitent des cas de non-conformité de valeurs sur les éléments qu’elles traitent dans leur nom.

2.2.1.3. Package map

Map est le paquetage contenant le patron “factory” qui produit des “LevelMap” ainsi qu’une classe LevelMap. Une “LevelMap” est constituée d’un nom, d’un type, d’un échiquier, d’une liste de “characters”, d’effets, et d’une taille. Pour une explication détaillée, nous vous renvoyons vers la [partie détaillée du patron de conception “Factory”](#).



2.2.1.4. Package square



“Square” est un de nos plus gros paquetages. Il comprend notre unité de base : le “square”, ainsi que tous les “MapElements” qui peuvent s’ajouter à un “square”.

Ainsi, nous y retrouvons nos bombes, bonus, et murs.

Square	Notre unité de base. Compose une LevelMap. Il peut être “walkable” et contenir un MapElement.
MapElement	Classe abstraite, attribut de Square, caractérisé par une texture.
Wall	Un MapElement abstrait qui définit le comportement de base pour tous les murs du jeu.
UnbreakableWall	Un mur incassable par les bombes. Le visuel des cartes sera principalement constitué de ce type.
BreakableWall	Un mur que les joueurs pourront casser avec des bombes. Ce mur peut, avec une probabilité de 10%, faire apparaître un bonus lors de sa destruction.
Bomb	Une bombe est un MapElement. Elle appartient à un joueur, est d'un certain type (par défaut “TimeBomb”) avec une puissance donnée et un comportement défini. Une déflagration d'une bombe entame la détonation de toutes les bombes sur sa portée.
TimeBomb	Bombe par défaut, avec détonation au bout de 2.5 secondes.

TriggerBomb	Évolution de la bombe par défaut. Permet au joueur qui la pose de la déclencher manuellement sans contrainte de délai.
BombState	Énumération. Une bombe est soit placée (compte à rebours entamé), soit explosée. Classe dispensable mais mise en place pour de futures customisations des états de nos bombes.
BombType	Énumération de nos types de bombes ("Time-based" ou "Trigger")
Bonus	MapElement abstrait. Définit la signature d'une méthode pour appliquer un bonus.
BonusAddBomb	Bonus qui permet d'ajouter une bombe au joueur qui le ramasse.
BonusLife	Bonus qui permet d'ajouter au joueur qui le ramasse une vie.
BonusSpeed	Bonus qui permet d'ajouter au joueur qui le ramasse 1 point de vitesse.
BonusTriggerBomb	Bonus qui permet de modifier le type de bombe que le joueur possède pour l'intégralité de la manche.

2.2.1.5. Package texture

Comprend à date pour seul classe "ElementTexture" nous permettant d'effectuer des opérations sur nos textures afin de les orienter correctement (dans tous les axes) avant de les afficher dans la vue.

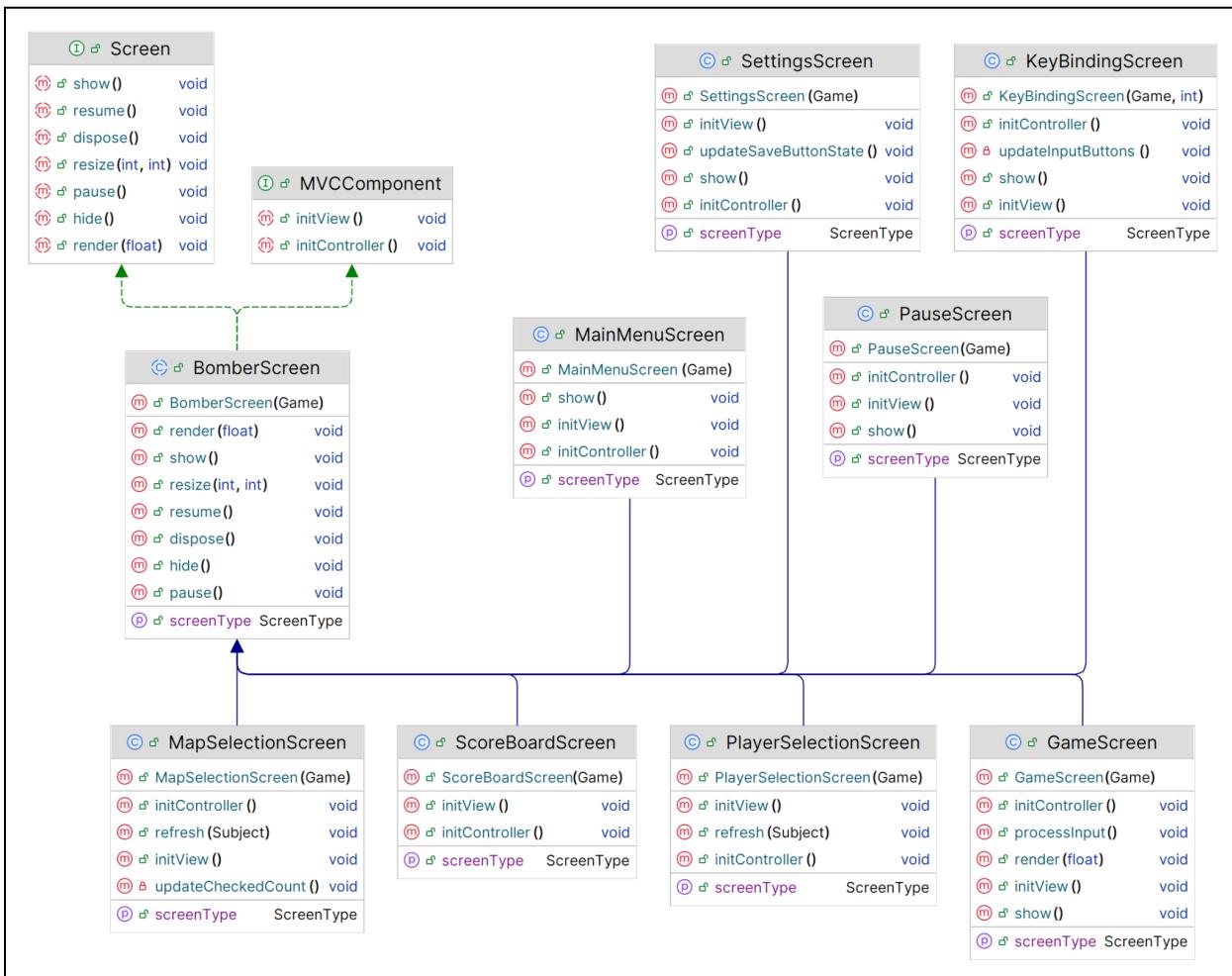
2.2.2. Les vues

Libgdx nous permet de gérer l'affichage de notre application d'une manière similaire à Swing ou JavaFX. Une multitude des composants UI est déjà présente dans le moteur de jeu, tels que les boutons, les labels, les screens, les dialogues. Nous les avons utilisés pour faire l'interface graphique de notre jeu, qui était complétée par quelques composants UI "fait maison", qui se basent sur les classes de Libgdx et implémentent des fonctionnalités supplémentaires spécifiques à nos besoins.

2.2.2.1. L'interface MVCCComponent

Pour pouvoir bien faire la distinction entre la vue et le contrôleur dans chacune de nos classes liées à l'affichage, nous avons décidé de créer une interface MVCCComponent que tous les composants d'affichage doivent implémenter. Celle-ci propose deux méthodes, initView() et initController(). Les classes qui implémentent cette interface doivent initialiser leur vue et leur contrôleur dans ces méthodes respectivement.

2.2.2.2. Les écrans

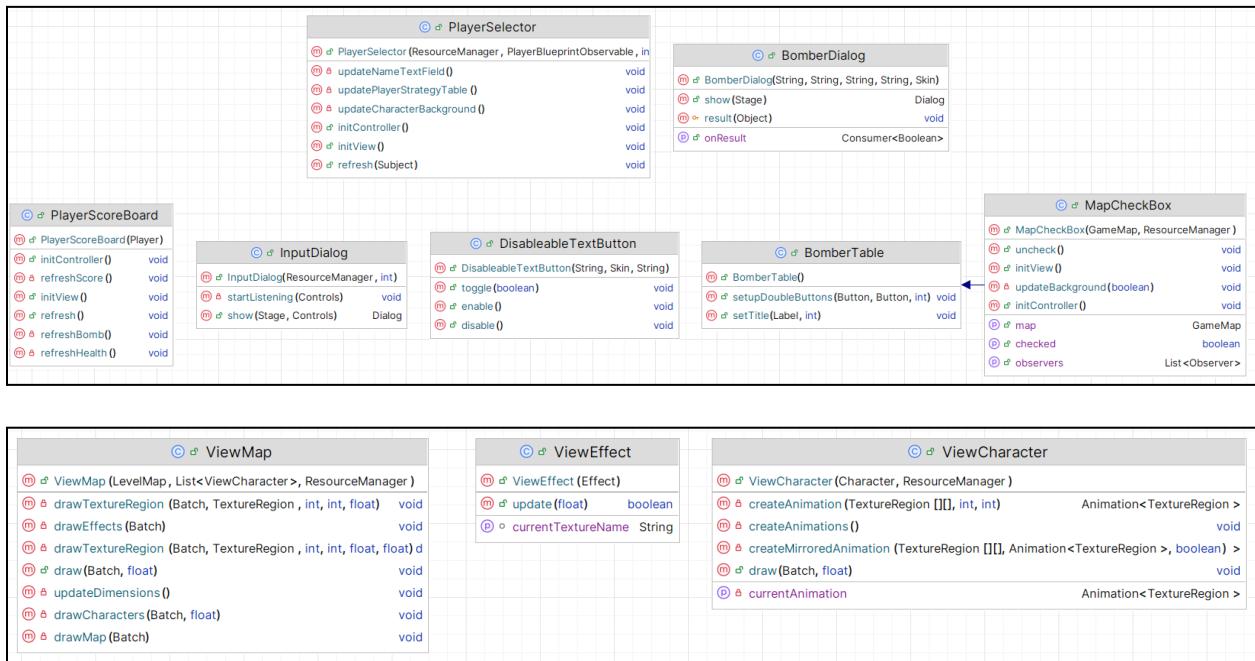


Les screens sont les différents écrans de notre jeu, tels que l'écran du menu principal, l'écran des réglages, etc... L'ensemble de ces écrans ont une classe mère abstraite BomberScreen, une implémentation de MVCCComponent, qui contient les éléments communs à tous les écrans (les ressources, le jeu). Cette classe mère hérite de la classe Screen de Libgdx. Les passages d'un écran à l'autre se font à l'aide du ScreenManager (cf. [modèle](#)).

Chaque écran a donc ses propres composants UI, sa propre logique de contrôleur et son propre "rôle" au sein de notre application.

MainMenuScreen	Menu principal du jeu. Dedans, on retrouve les options principales, telles que jouer, accéder aux paramètres ou quitter le jeu.
PlayerSelectionScreen	Écran de sélection des joueurs pour une partie, où l'utilisateur configure les noms, les skins et les stratégies des joueurs de la partie.
MapSelectionScreen	Écran de sélection des cartes et du nombre de rounds pour la partie. On affiche une liste de cartes sous forme de cases à cocher.
PauseScreen	Menu de pause du jeu permettant aux joueurs de reprendre la partie, accéder aux paramètres, ou retourner vers le menu principal.
SettingsScreen	Les paramètres du jeu tels que les niveaux de volume, la langue, et la configuration des touches.
KeyBindingScreen	Écran permettant aux joueurs de configurer leurs commandes. Elle implémente l'interface InputProcessor pour gérer les entrées utilisateur. On utilise le ConfigManager (cf modèle) pour accéder et modifier la configuration des contrôles du joueur.
GameScreen	Écran d'affichage d'une partie de Bomber7. Cette classe est importante car elle définit le taux de rafraîchissement de notre jeu. La méthode render de cet écran, appelée plusieurs fois par seconde, dit au modèle de se mettre à jour (vérifier si la partie est terminée, etc...) Nous nous retrouvons donc avec notre vue qui rafraîchit notre modèle . Cette entorse volontaire au concept MVC nous permet de se baser sur l'horloge interne de Libgdx pour rafraîchir notre modèle, chose qu'on était obligé de faire pour avoir un rendu fluide. Cf. Horloge & Rafraîchissement dans problèmes rencontrés

2.2.2.3. Les composants UI



Comme cité précédemment, nous avons dû créer nos propres composants UI pour pouvoir réaliser des fonctionnalités avancées.

Un exemple concret d'un composant UI sur-mesure sont les PlayerSelector. Ils gèrent l'ajout, la suppression et la configuration d'un joueur dans l'écran de configuration des joueurs. Ce composant permet de créer un squelette qui sera ensuite utilisé pour créer un vrai Character.



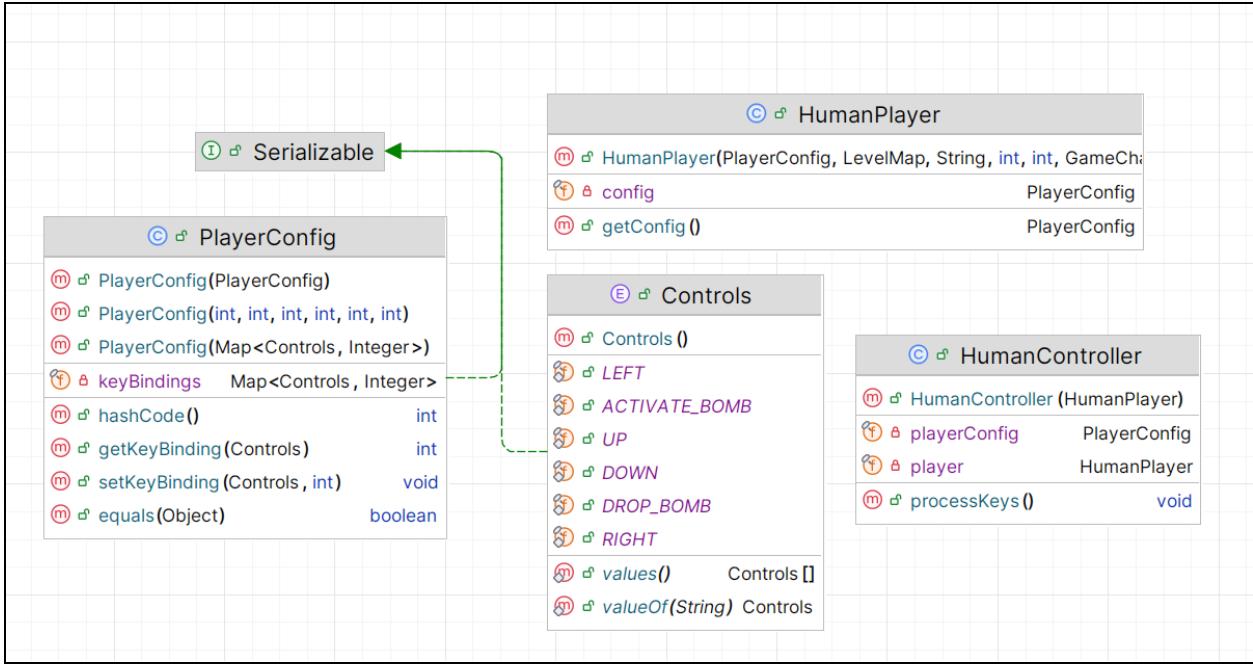
Voici une liste non exhaustive d'autres composants intéressants à mentionner :

InputDialog	Un dialogue qui permet de réassigner les contrôles d'un joueur spécifique dans le menu des paramètres du jeu. Elle capture une seule pression de touche et met à jour la configuration des contrôles en conséquence.
ViewMap	Vue responsable (ajoutée au GameScreen) de l'affichage de la carte du jeu, incluant les éléments de la carte (murs, bombes, etc), les personnages et les effets visuels. Elle utilise les objets Square pour dessiner chaque case de la carte en fonction de sa texture et gère les dimensions nécessaires pour centrer la carte.
ViewCharacter	Vue dédiée à l'affichage des personnages dans le jeu. Elle gère également les animations correspondantes (stand, die, moving-right, etc) en fonction de son état. Fait partie de ViewMap.
ViewEffect	Vue affichant spécialement les effets visuels animés sur la carte, comme les explosions. Elle met à jour l'animation en fonction du temps écoulé, et détermine la texture actuelle à afficher pour l'effet.

Les maquettes réalisées au préalable sur Figma sont fidèles aux différents écrans / composants UI disponibles dans notre jeu final. La "carte" est la partie qui diffère le plus de notre premier jet : c'est logique puisque nous ne connaissons pas encore au moment du rendu des fonctionnalités la manière dont elle serait représentée. En revanche, les autres écrans étant relativement standards pour un jeux vidéo d'arcade et/ou faisaient consensus dans l'équipe, la traduction en vue a essayé d'en respecter l'esthétique et l'esprit.

La comparaison faite avec l'ancienne disposition est possible en consultant notre [Figma](#).

2.2.3. Les contrôleurs



Hormis les contrôleurs liés aux screens initialisés dans initController() de chaque screen qui gèrent essentiellement la logique des différents boutons de l'UI, nous avons un autre contrôleur important pour la logique de jeu.

Le HumanController est un contrôleur qui gère les entrées utilisateur (appui sur les touches). A chaque coup d'horloge (update de BomberGame), la méthode processKeys() regarde quelles touches ont été appuyées et quelles sont les actions à réaliser. Pour savoir quelles touches il doit écouter, il se base sur l'instance de PlayerConfig qui contient la configuration des touches associés au HumanPlayer qu'il contrôle.

3. Choix de Conception et Réalisation

3.1. Principaux choix de conception

Plusieurs décisions de conception ont été choisies avant de coder, durant la phase de conceptualisation.

Tout d'abord, afin d'harmoniser la conceptualisation et d'avoir une logique de jeux cohérente pour tous les éléments, nous avons décidé de porter la responsabilité d'une action à l'entité responsable de cette action. Par exemple, lorsqu'une bombe explose, c'est sa responsabilité de mettre à jour la carte en supprimant les murs qu'elle détruit. Cela nous évite d'avoir une trop grande classe responsable d'arbitrer la partie. Cela permet aussi de réduire au maximum le nombre d'opérations, ce qui peut poser problème dans un jeu en temps réel. En contrepartie, cela nécessite par exemple qu'une bombe puisse faire des opérations sur la carte sur laquelle elle se trouve, pouvant soulever des questions de pouvant soulever des questions de dépendances croisées ou de respect du principe de responsabilité unique, notamment si la bombe commence à avoir trop de connaissances ou d'accès à d'autres éléments de la carte.

Ensuite, des questions se sont posées quant à l'utilisation de certaines fonctionnalités fournies par le moteur de jeux LibGDX. Il peut par exemple fournir des solutions déjà implémentées pour gérer les collisions ou pour générer automatiquement les cartes à partir des jsons. Dans une optique d'apprentissage et non de productivité, nous avons décidé de nous passer au maximum de ces solutions proposées par libGDX. Cependant, l'objectif n'étant pas de développer un moteur de jeux, nous avons tout de même utilisé certaines fonctionnalités, entre autre celles liées au rafraîchissement de l'écran et de l'horloge interne du jeu.

3.2. Patrons de conceptions utilisés

3.2.1. MVC

Comme vu en cours, ce patron a pour objectif de séparer clairement les responsabilités en trois parties distinctes permettant une meilleure maintenabilité de l'application :

- **Le modèle** contient les données et la logique de l'application.
- **La vue**, partie visible d'une interface graphique chargée d'afficher les données du model.
- **Les contrôleurs** permettent de traiter les actions de l'utilisateur afin de modifier le model puis la vue.

3.2.2. Observateur

Le patron observateur permet de gérer la communication entre les composants de manière réactive et découpée. Par exemple, dans la classe PlayerSelectionScreen, chaque joueur est représenté par un objet PlayerBlueprintObservable qui implémente le rôle de sujet. Ces objets notifient leurs observateurs lorsqu'un changement pertinent se produit, comme la validation ou la suppression du joueur. De même avec la classe MapSelectionScreen et MapCheckBOX.

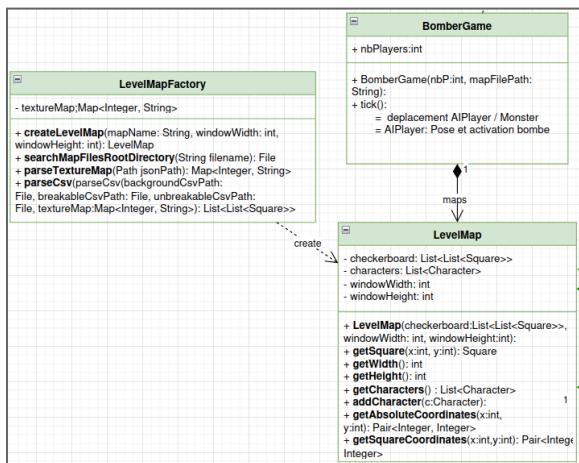
3.2.3. Singleton

Ce patron permet de garantir qu'il n'existe qu'une seule et unique instance d'une classe donnée au sein de l'ensemble de l'application, offrant un unique point d'accès à cette instance depuis n'importe quel endroit du code. Dans notre application, nous avons au total 4 singletons : ResourceManager, ConfigManager, SoundManager et ScreenManager (cf. [modèle](#)).

3.2.4. Fabrique

Le patron stratégie permet d'extraire un comportement présent dans une classe, mais qui peut s'exécuter de plusieurs façons, dans des classes séparées *stratégies*. Dans notre cas, cela permettait de répondre à la problématiques des joueurs et des monstres joués par l'ordinateur. Le patron nous permettait d'extraire les stratégies de déplacements qui diffèrent selon les difficultés de l'IA. Concernant seulement les joueurs IA, une stratégie pour déterminer quand poser une bombe est aussi pertinente car son implémentation diffère selon le niveau de difficulté, de la même façon que pour le déplacement.

3.2.5. Stratégie



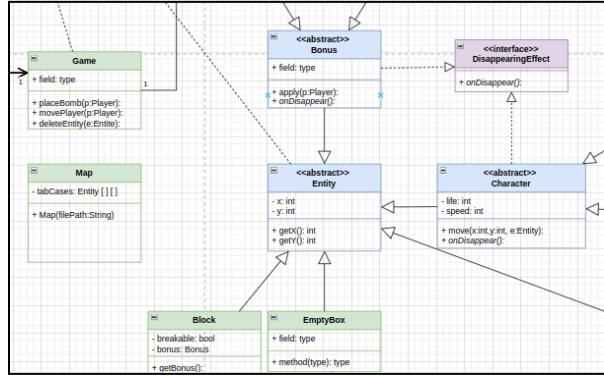
Séparer la logique de construction et l'utilisation de LevelMap, de même pour les tests.

Éviter un constructeur de 300 lignes (Factorisation).

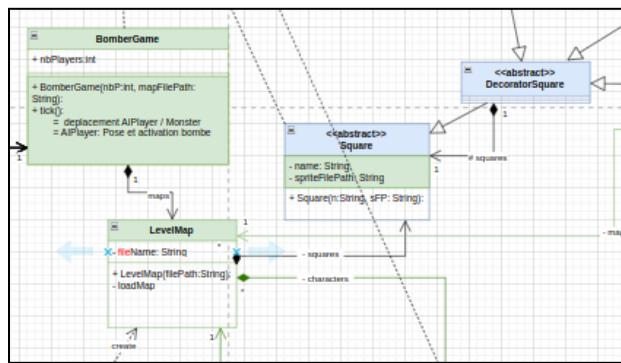
Avoir une meilleure évolutivité du code dans le cas où l'on souhaite créer des LevelMaps différentes.

3.3. Problèmes rencontrés et solutions apportées

3.3.1. Classe entité

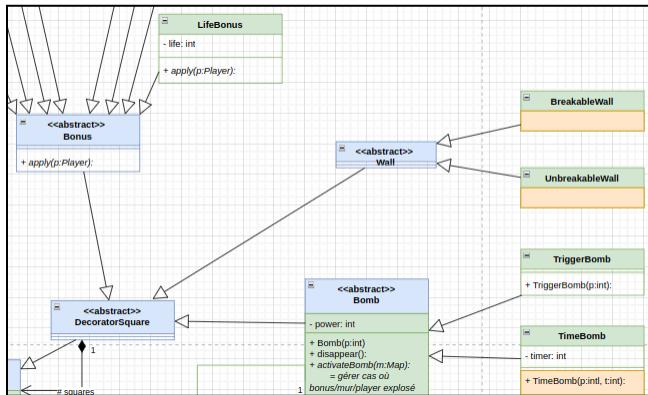


Dans la V1, soit la solution naïve, nous avions pensé à une classe Entity dont tous les acteurs héritaient pour pouvoir constituer un échiquier d'entités.

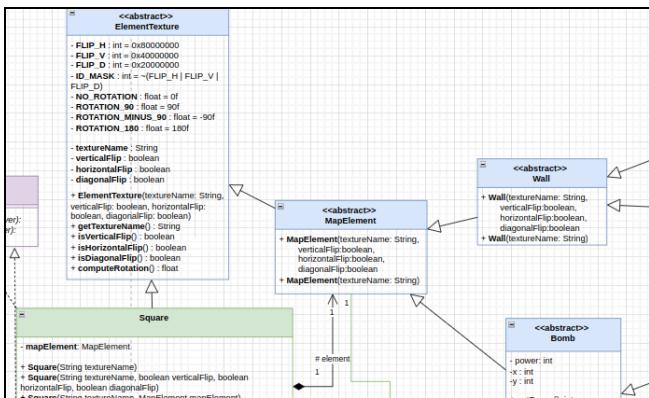


Le principal problème lié à cette approche est le manque de hiérarchie. Toutes les entités sont au même niveau sur l'échiquier. Ce qui veut dire qu'à chaque déplacement d'un personnage, il faut d'abord le localiser dans l'ensemble du tableau représentant l'échiquier. Ce qui est vraiment problématique dans la mesure où le personnage se déplacera quasi constamment. Depuis la V2, la map contient d'une part les cases du jeu et leur élément et d'autre les personnages avec leurs coordonnées respectives accessibles rapidement.

3.3.2. Patron décorateur



Dans la V2, l'idée d'avoir un décorateur a émergé pour gérer les éléments constituant une case.



Cette vision était erronée, car dans le cas du jeu Bomberman, il n'y a qu'un seul élément possible sur une case à la fois, d'où la modification en une classe abstraite MapElement dont héritent les murs, les bonus et les bombes.

3.3.3. Horloge & rafraîchissement

Pour rafraîchir le jeu à chaque coup d'horloge, l'utilisation de Libgdx nous a contraint de faire un choix entre:

- Utiliser, dans notre modèle, la méthode de Libgdx renvoyant le delta entre deux coups d'horloge (tick en anglais): `Gdx.graphics.getDeltaTime()`.
- Rafraîchir le jeu via les méthodes `render()` des vues, appelées à chaque coup d'horloge.

La première approche présentait une contrainte majeure, à savoir que cette méthode dépend de l'instance Libgdx avec le GUI créé. Or, dans les tests unitaires JUNIT, nous ne pouvons pas lancer d'instance Libgdx, ce qui empêche leur exécution. Pour contourner ce problème, une solution existe en théorie, à savoir l'utilisation d'un Headless Libgdx, c'est-à-dire, émuler le système sans avoir besoin d'un GUI. En pratique, impossible de faire marcher le Headless Libgdx, plusieurs heures passées à débugger. C'est donc la deuxième solution que nous avons préférée malgré l'entrave au principe hiérarchique du MVC (le Modèle notifie les Vues).

3.3.4. Test des classes utilisant indirectement libgdx & les assets

Un autre problème lors de la création des tests était de tester les classes du modèle qui utilisent les assets du jeu dans leur logique. C'est par exemple le cas de la bombe qui, lorsqu'elle explose, fait appel au SoundManager (cf. [modèle](#)) pour jouer un son (en supposant qu'il l'aura préalablement chargé depuis un fichier). Pour contourner ce problème et quand même pouvoir tester la logique d'une classe, nous avons utilisé le framework [Mockito](#). Dans notre cas, il nous a permis de faire en sorte que rien ne se passe lorsque la méthode responsable de jouer le son était appelée.

```

// Instantiate a TimeBomb at position (2, 3) with a timer of 5 sec
timeBomb = Mockito.spy(new TimeBomb( p: 2, x: 2, y: 3, super.testCharacter));
Mockito.doNothing().when(timeBomb).playExplosionSound();

```

3.3.5. Vitesse & Collisions



Il aurait été facile de faire déplacer le joueur d'une case en case mais cela n'aurait pas été très naturel. Nous avons donc décidé d'avoir la possibilité de se mouvoir de quelques pixels. Ainsi, il faut vérifier la case sur laquelle nous nous trouvons (mapX , mapY), à chaque déplacement aussi infime soit-il, afin de gérer les événements (ramasser un bonus, explosion d'une bombe).



Les différentes positions du personnage ne prennent pas exactement la même place. Le personnage est plus large s'il est de face que de côté.



Malgré que l'on sache sur quelle case nous nous trouvions, un autre problème de collision subsisté. Dans l'image ci-contre, nous sommes bien considéré dans la case en bas à gauche, mais notre tête dépasse et est en collision avec le mur. Ce comportement était le même avec les autres côtés. On calcule les quatre coins de la hitbox du personnage (intérieur du cadre rouge sur la photo précédente), représentant les extrémités du corps du personnage. Pour chacun de ces coins, on vérifie que la case correspondante est bien sur l'échiquier et surtout que ce n'est pas un mur.

4. Méthodes Agiles

4.1. Description de l'organisation de l'équipe

Nous avons adopté la méthodologie agile en organisant notre travail sous forme de sprints, c'est-à-dire des cycles courts et réguliers permettant de planifier, réaliser et livrer progressivement les différentes fonctionnalités attendues.

Chaque sprint débutait par une réunion en présentiel de planification sur le prochain sprint à venir, et aussi une revue de l'ancien sprint pour intégrer éventuellement des correctifs. On a profité de ces sprints pour faire de la mise à niveau entre nous, simplement pour avoir une connaissance globale sur l'architecture logicielle et ses spécificités développées par chacun.

Pour garantir le bon déroulement de cette organisation, nous avions un Scrum Master au sein de l'équipe. Pierre avait ce rôle d'animer les réunions clés (planification, revues et rétrospectives), de veiller au respect du cadre Scrum, et également de faciliter la collaboration entre les membres.

4.2. Méthodes agiles mises en œuvre

En gestion de projet, nous avons la chance de compter dans notre équipe trois membres faisant de la GDP agile en entreprise. Le MOOC Gestion de Projet encadré par le docteur Rémi Bachelet et son équipe, proposée en cours de CAM (*Carriers and Management*), a aussi été utile pour remédier à l'absence des cours de gestion de projet, faute d'encadrant.

Dans le premier rapport Jalon “Fonctionnalités du sujet”, nous avons mobilisé plusieurs concepts abordés dans le cours de CAM tels que la matrice SWOT, la matrice RACI ou encore le diagramme de GANTT. Nous allons à présent décrire brièvement l'apport de chacune de ces méthodes ainsi que leur facteur clé ou non dans notre gestion de projet.

Concrètement, la matrice SWOT identifie nos forces, faiblesses, opportunités et menaces. Elle offre une vision claire de nos contraintes et de nos atouts. Cet outil pas nécessairement essentiel à tout de même permis de situer le cadre de notre travail.

De son côté, la matrice RACI clarifie la répartition des rôles et responsabilités au sein du projet, en définissant précisément qui est responsable, qui approuve, qui est consulté et qui doit être informé pour chaque tâche. Honnêtement, nous avons naturellement respecté cette matrice, la composition de notre groupe constitué d'étudiants en BUT Réseaux & Télécommunications et en Informatique, rendait naturelle cette chaîne de vérification. Nos compétences et formations respectives, ainsi que la répartition des tâches selon les préférences de chacun, ont rendu l'application de la matrice cohérente.

4.3. Outils et pratiques utilisés

- **GitHub Projects**

Un de nos outils directeurs en gestion de projet était la section “Projects” de GitHub. Il s’agit d’un outil puissant pour organiser, planifier et suivre l’avancement des tâches et du développement. Nous avons mis en place différents tableaux de Kanban afin d’assurer un suivi efficace des tâches au sein du groupe. Les sprints étaient organisés dans ces tableaux à travers plusieurs catégories : “BackLog”, “In Progress”, “Ready” et “In Review”.

Chaque tâche (issue) est liée à une branche de développement spécifique. Nous y ajoutons également les membres de l’équipe concernés ainsi que des détails spécifiques à la tâche en cours. L’objectif était de favoriser au maximum la transparence et l’efficacité globale du groupe.

De plus, la catégorie “Roadmap” dans “Projects” a permis également d’avoir un aspect visuel sur la temporalité de nos actions, permettant ainsi de mieux anticiper nos échéances.

La puissance de Github réside dans sa capacité à regrouper la gestion de code et gestion de projet au sein d’un même espace. Cela répond à un besoin fort de praticité et de productivité pour notre équipe. On préfère centraliser notre gestion de projet dans une seule application, et tirer pleinement parti de toutes ses fonctionnalités afin d’optimiser au maximum notre temps de travail.

Beaucoup d’entre nous étaient sceptiques vis-à-vis de l’utilisation de ces outils de gestion de projet. Avec le recul, on réalise à quel point ces outils sont pertinents et importants dans la phase de développement. D’autant plus sur ce type de projet réunissant 6 personnes sur une durée relativement longue.

- **Pipeline**

La mise en place d’un pipeline CI/CD (Intégration Continue/Déploiement Continu) a constitué un atout non négligeable. Il nous a permis d’éviter les merges involontaires/erronées. Il était constitué de plusieurs états:

- Vérification des failles liées à des dépendances
- Lancement des tests Java et vérification du checkstyle
- Publication de la javadoc [en ligne](#) si modification sur main :

De plus, la branche main était protégée (impossible de faire des modifications directement sans Pull Request). Enfin, en cas de Pull Request, nous générions automatiquement une revue du code par une IA qui met en lumière les axes d’amélioration et les PullRequests vers le main doivent être obligatoirement contrôlées par au moins deux développeurs humains.

Conclusion

Finalement, ce projet a permis à notre équipe de relever de nombreux défis, tant sur le plan technique qu'organisationnel. Chaque membre de l'équipe a pu exprimer ses compétences, sa créativité et sa capacité à collaborer. En partant d'un projet long sans contraintes particulières, nous avons apprécié la liberté qui nous était offerte, ainsi que tous les créneaux horaires qui étaient alloués pour celui-ci sur les dernières semaines.

Ainsi, en revisitant l'univers de Bomberman à travers le prisme de l'ENSEEIHT, nous avons su garder une excellente dynamique de travail tout au long de la phase de développement du jeu. Le fait de voir des résultats concrets à chaque sprint entre les animations et le visuel a vraiment renforcé notre motivation à poursuivre le projet et le mener à terme. De plus, l'objectif étant de monter en compétences, nous avons ainsi fait preuve de rigueur dans nos choix conceptuels et architecturaux. Les choix techniques que nous avons opérés témoignent de notre volonté de produire un jeu robuste, évolutif et bien documenté. Le fait de travailler en groupe à six a offert la possibilité d'avoir différents points de vue, et à ne pas nous focaliser excessivement sur un seul aspect.

Au-delà des aspects purement techniques, Bomber7 a été l'occasion de renforcer la cohésion du groupe, mettre en pratique les méthodes agiles et s'entraider. En effet, certains membres de l'équipe possédaient déjà des compétences spécifiques sur certains sujets (design pattern) et ont pu en faire bénéficier au reste de l'équipe. Quant aux difficultés rencontrées, loin de constituer des obstacles importants, ont été autant d'opportunités d'apprendre, de s'entraider pour qu'à l'avenir, on puisse avoir une réelle expertise sur le sujet.

En conclusion, nous sommes fiers du résultat obtenu, nous avons abouti à un projet solide, en nous imposant de respecter rigoureusement les principes de gestion de projet, ainsi que la modélisation UML, ce qui a directement contribué à la réussite de Bomber7. A présent que le développement initial du projet touche à sa fin, les perspectives d'évolution sont nombreuses : enrichissement des mécaniques de jeu, ajout de nouveaux modes et d'IA, etc. Il nous appartient de réfléchir à la trajectoire que prendra Bomber7.

Résumé des points clés :

- Projet : Jeu multijoueur local inspiré de Bomberman avec des cartes et éléments personnalisées de l'ENSEEIHT
- Fonctionnalités : Déplacement, bombes, bonus, IA Simple (branche à part) menus animés, ambiance sonore, multijoueur local
- Architecture : MVC et design patterns (Singleton, Fabrique, etc), modularité
- Méthodes : Gestion agile (sprints, Github Projects, CI/CD)
- Difficultés : Tests LibGDX, collisions, rafraîchissement via le tick()
- Perspectives : IA avancée, nouveaux modes de jeu, davantage de contenu

Réflexions finales et perspectives d'amélioration

L'ensemble de l'équipe est d'accord sur le point que ce projet fut une expérience enrichissante. On a pu s'adapter aux défis inhérents à la réalisation d'un jeu en temps réel et la partie visuel de celui-ci. Néanmoins, plusieurs pistes d'amélioration subsistent : l'implémentation de personnages IA, le développement de nouveaux modes de jeu (coopératif, compétitif) ainsi que l'enrichissement des mécaniques de gameplay par de nouveaux événements dynamiques. Ces axes constituent une excellente base pour un développement itératif si le projet venait à être poursuivi en dehors des cours.

Avec notre architecture bien structurée actuellement en place, ces évolutions pourraient être intégrées de manière fluide sans forcément nécessiter de refonte majeure du code existant. On pourrait même étendre le jeu à plusieurs plateformes grâce au moteur LibGDX, ouvrant éventuellement la voie à une version jouable sur mobile. Une version multijoueur en ligne pourrait également être envisagée.

Enfin Bomber7 constitue une excellente base de travail pour étendre le jeu dans une certaine mesure, voire en faire un projet open source.