TOULOUSE INP N7

Conception et Programmation Objet

Projet long : conception d'un jeu vidéo de survie en Java

I. Rapport du projet

Cédric ABDELBAKI
Thomas GRUNER
Yanis KOUIDRI
Jérémy THERET
Eric YU
1APPSN - SEMESTRE 6

Table des matières

I. Introduction

Ce rapport a pour but de vous présenter le travail accompli dans le cadre de la réalisation du projet long de la matière Conception et Programmation Objet. Il vous présentera les aspects techniques du travail accompli ainsi qu'une présentation de notre organisation en matière de gestion de projet.

Notre groupe se compose des quatre apprentis suivants : Cédric ABDELBAKI, Thomas GRUNER, Yanis KOUIDRI, Jérémy THÉRET et Eric YU. Nous avons fait le choix de développer un jeu vidéo de survie en vue de côté en deux dimensions. Dans notre jeu, vous aurez la possibilité d'explorer un monde généré aléatoirement et de récolter des "blocs" qui seront stockés dans votre inventaire.

Nous avons choisi ce sujet que nous trouvons passionnant pour sa complexité et son potentiel pédagogique en matière de conception de solutions de modèle, de vue et de contrôle. Pour le mettre en œuvre, nous avons utilisé le langage Java et la librairie graphique Java Swing. Aucun moteur de jeu n'a été utilisé pour la conception de notre jeu et nous avons donc dû en développer toutes les fonctionnalités.

Nous remercions nos enseignants, Xavier CRÉGUT, Aurélie HURAULT, Marc PANTEL et Gilles FRANCOIS pour les enseignements dispensés et les conseils apportés tout au long de la réalisation de ce projet.

Nous espérons que de ce rapport vous apportera toutes les réponses aux questions que vous pourriez vous poser et nous tenons à votre disposition aux adresses électroniques suivantes pour tout complément :

cedric.abdelbaki@etu.inp-n7.fr
thomas.gruner@etu.inp-n7.fr
yanis.kouidri@etu.inp-n7.fr
jeremy.theret@etu.inp-n7.fr
eric.yu@etu.inp-n7.fr

Nous vous souhaitons une agréable lecture.

II. Principales fonctionnalités

Les tableaux ci-dessous présentent les EPICs, les user stories et technical stories de notre projet :

EPICS	US / TS	
Interactions avec le monde	**US** : Casser bloc	Terminé **SPRINT 4**
	US Poser bloc	Pas commencé
Personnage joueur	**TS** : Créer le personnage au lancement du jeu	Terminé **SPRINT 1**
	US : Afficher le personnage au lancement du jeu	Terminé **SPRINT 1**
	US : Déplacer le personnage sur l'axe x	Terminé **SPRINT 2**
	US : Déplacer le personnage sur l'axe y	Terminé **SPRINT 2**
	US : Faire apparaître le personnage au niveau du sol	Terminé **SPRINT 3**
	US : Centrer la caméra sur le joueur lors de son déplacement	Terminé **SPRINT 4**
	US : Choisir le mode de contrôle de la caméra	Terminé **SPRINT 4**

II. Principales fonctionnalités

Personnages non joueurs	**TS** : Créer les personnages non joueurs	Pas commencé
	US : Voir les personnages non joueurs se déplacer	Pas commencé
	TS : Implémenter le comportement des personnages non joueurs	Pas commencé
Interface du jeu	**US** : Créer la fenêtre de l'application	Terminé **SPRINT 2**
	US : Créer la scène du jeu	Terminé **SPRINT 2**
	US : Améliorer graphiquement la fenêtre du jeu	Terminé **SPRINT 3**
	US : Voir les crédits du jeu	Terminé **SPRINT 4**
	US : Pouvoir paramétrer les commandes du jeu	Terminé **SPRINT 4**

II. Principales fonctionnalités

Carte	**TS** : Représenter la carte	Terminé **SPRINT 1**
	TS : Représenter la tuile	Terminé **SPRINT 1**
	TS : Définir le type de carte	Terminé **SPRINT 2**
	US : Générer procéduralement une carte	Terminé **SPRINT 2**
	US : Afficher la carte	Terminé **SPRINT 3**
	TS : Optimiser le rendu de la carte	Terminé **SPRINT 4**
Physique du monde	**TS** : Gérer la boucle de jeu	Terminé **SPRINT 1**
	TS : Gérer les collisions avec le monde	Terminé **SPRINT 4**
	TS : Gérer la gravité	Terminé **SPRINT 4**

II. Principales fonctionnalités

Gestion de l'inventaire	**TS** : Modéliser l'inventaire	Terminé **SPRINT 2**
	US : Ouvrir l'inventaire	Terminé **SPRINT 3**
	US : Afficher la barre d'inventaire	Terminé **SPRINT 3**
	TS : Séparer le modèle de la vue	Terminé **SPRINT 3**
	US : Récupérer des ressources	Terminé **SPRINT 4**
	US : Interagir avec la barre d'inventaire	En cours
	US : Interagir avec le menu d'inventaire	En cours
	US : Pouvoir empiler des Items dans l'inventaire	Pas commencé

III. Découpage et choix de conception

1) Présentation du découpage

Afin d'organiser au mieux notre projet nous avons décidé de découper notre application en trois paquetages :

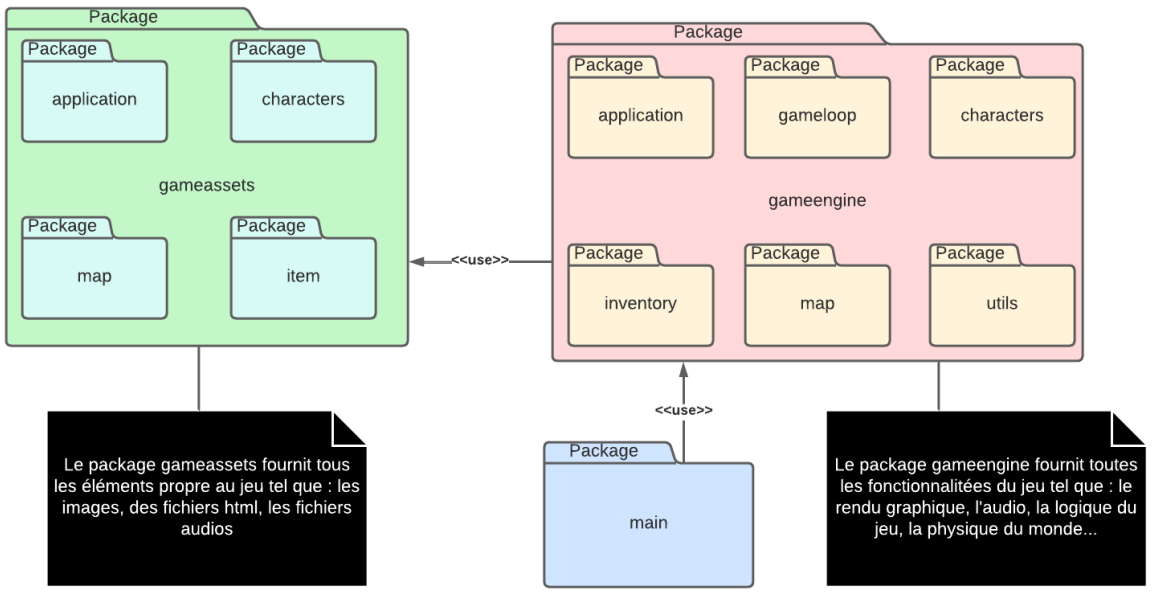
- **gameassets**
- **gameengine**
- **main**

Ces trois paquetages se trouvent dans le dossier **src** (source) qui contient tout le code source de notre jeu.

Le paquetage **gameengine** correspond aux fichiers d'extension *.java* (hormis le main), c'est là que nous définirons toutes les mécaniques de notre jeu. Le paquetage **gameassets** regroupe toutes les ressources nécessaires à notre application telles que les images, les sons, les textes…
Enfin, nous avons le paquetage **main** contenant la classe principale de notre programme, chargée d'instancier chaque objet et d'initier le lancement du programme.

Le découpage au sein des différents paquetages présents dans **gameengine** respecte le modèle MVC mais chaque composant dispose de ses propres paquetages **model**, **view** et **controller**. De cette façon, chaque composant opère comme un module indépendant pouvant être importé et utilisé quel que soit le contexte (utilisation dans d'autres projets).

2) Diagramme de paquetage



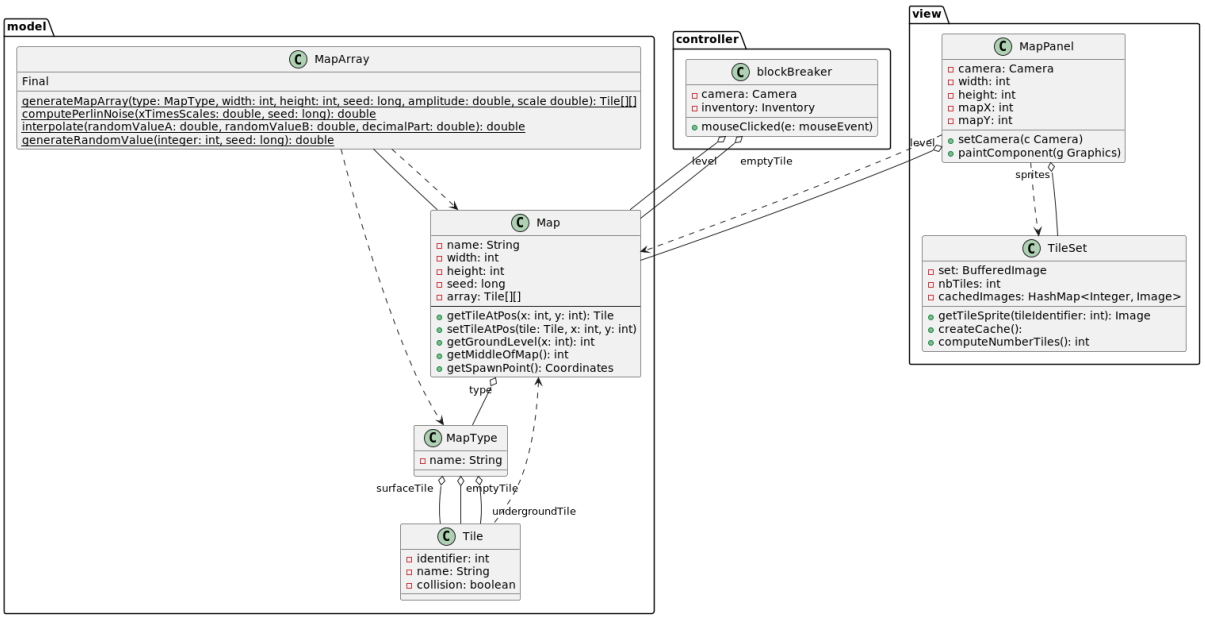
IV. Architecture de l'application

1) Composant Map

A) Présentation

Le composant **Map** intègre les classes nécessaires à la représentation de la carte, des tuiles, à la génération procédurale du terrain et à l'affichage de la carte. Elle dispose de ses packages **model**, **view** et **controller**. Le package **model** contient les classes **Map**, **MapArray**, **MapType** et **Tile**, le package **view**, les classes **MapPanel** et **Tileset** et le package **controller**, la classe **BlockBreaker** utilisée pour gérer la casse des blocs.

B) Diagramme de classes



C) Fonctionnement

a) Model

Lors du lancement de l'application, un nouvel objet de classe Map est instancié. L'un des attributs de cet objet est un tableau de tuiles représentant la position de chaque bloc sur la carte. Ce tableau est obtenu en faisant appel dans le constructeur à la méthode statique **generateMapArray()** de la classe **MapArray**. Cette méthode utilise une implémentation en 1D de l'**algorithme de Perlin** permettant de générer des "sinusoïdes" aléatoires en jouant sur des paramètres d'amplitude et d'échelle. Cette "sinusoïde" représente en réalité le niveau du sol : ce qui se trouve au-dessus sera rempli par des tuiles vides.

Ce qui se trouve en dessous sera rempli par des tuiles souterraines. Pour choisir les tuiles à utiliser pour la génération d'une carte donnée, on instancie un objet de type **MapType** contenant des attributs représentant ces tuiles de vide, de surface et souterraines. Cette représentation permet ainsi d'envisager différents types de cartes utilisant différents blocs pour représenter différentes planètes à partir de tuiles toutes instanciées au lancement du programme.

b) View

La classe **MapPanel** hérite de **JPanel** et prend en paramètre un objet de type **Map** et un tileset de type **Tileset**. La classe **Tileset** permet d'instancier un objet représentant les images des tuiles du jeu et fournit une méthode permettant d'extraire l'image d'une tuile précise en passant son identifiant en paramètres. Un système de cache est utilisé pour éviter de créer de nouveaux objets de type **Image** à chaque fois qu'une tuile est redessinée (ce qui provoque des ralentissements).

c) Controller

La classe **blockBreaker** implémente les méthodes de **MouseAdapter** pour permettre la destruction des blocs.

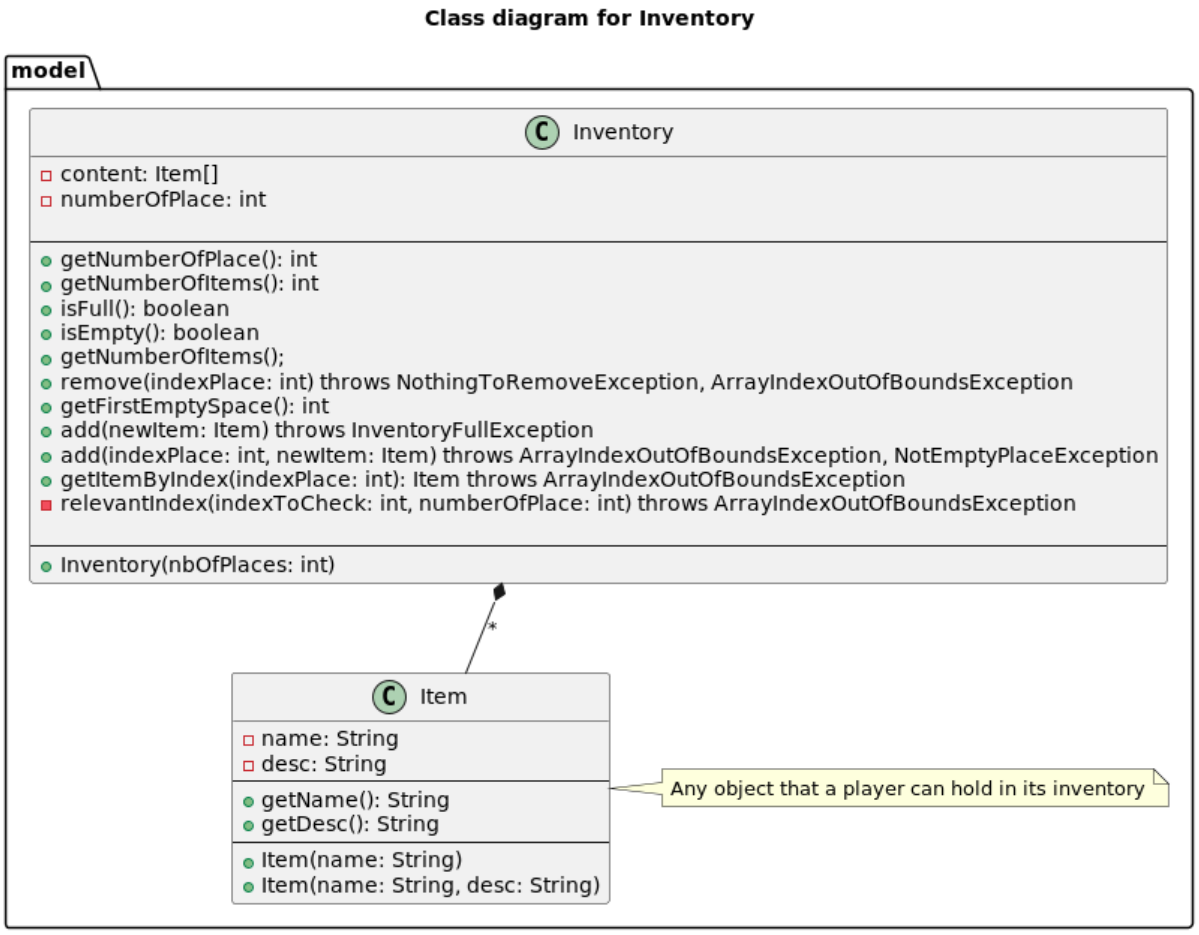
2) Composant Inventory

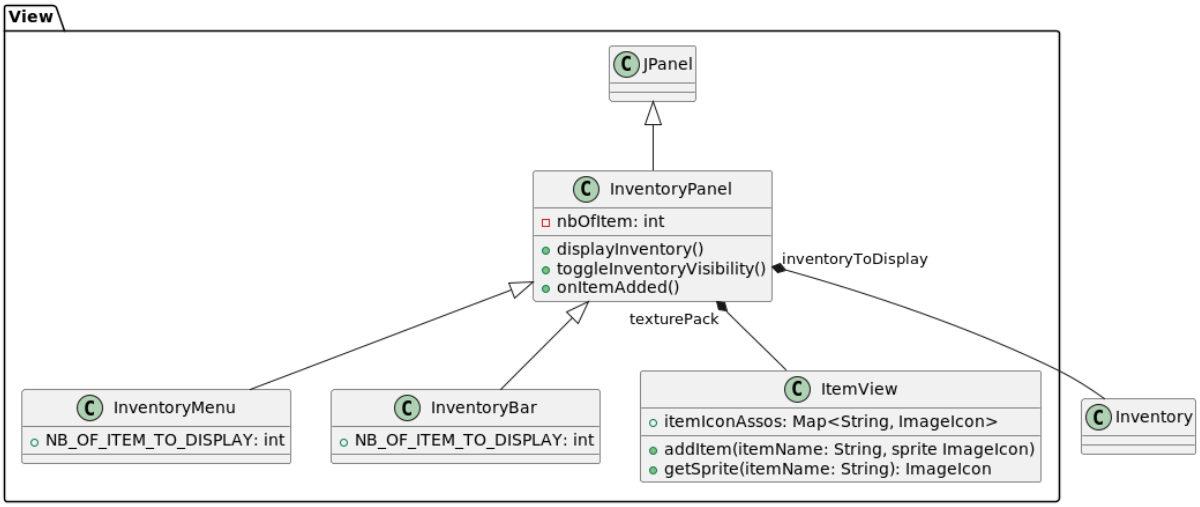
A) Présentation

L'inventaire représente, dans notre jeu, tous les Items que le personnage possède. Un Item est un objet que nous avons créé et que le joueur pourra posséder et utiliser. Naturellement, quand une nouvelle partie est lancée, le joueur commence avec un inventaire vide puis, au fil de son aventure, il se remplit avec les diverses ressources que le joueur pourra collecter.

Dans le jeu l'inventaire à une taille fixe que nous avons défini à 40 emplacements. Dans la pratique, quand le joueur casse un bloc sur la carte, un nouvel élément (Item) est ajouté à son inventaire. Pour le moment, les Items ne s'empilent pas, cela veut dire qu'une place d'inventaire correspond à un et un seul Item.

B) Diagrammes de classes





C) Fonctionnement

a) Model

Comme nous connaissons, au moment de l'instanciation de l'inventaire, sa taille et que cette dernière est fixe, nous avons décidé de stocker son contenu dans un tableau d'Items.
Item est un objet qui contient un nom et une éventuelle description, sa représentation visuelle sera définie dans le paquetage vue.

Bien que l'inventaire possède 40 places dans notre jeu, le nombre de places est choisi à l'instanciation de l'objet. Cela rend la classe plus générique et permet de la réutiliser. Par exemple, si nous souhaitons ajouter un système de coffres de stockage dans notre jeu, la classe inventaire pourra être réutilisée (les coffres auraient aussi un inventaire, potentiellement plus grand que celui du joueur).

L'inventaire implémente une méthode d'ajout d'Items avec ou sans indice. Sans indice, l'Item s'ajoute dans la première case libre de l'inventaire.
Nous avons choisi de lier les Items à l'inventaire (relation de composition). Ainsi, on peut ajouter plusieurs fois le même Item dans l'inventaire, il sera dupliqué.

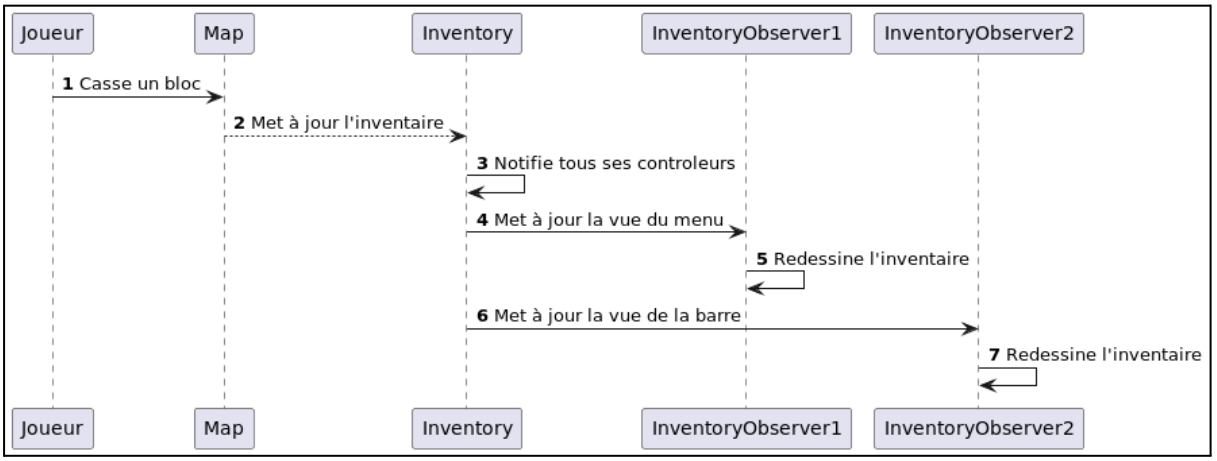
b) View

En ce qui concerne la vue, nous avons d'abord créé une classe ItemView qui fait le lien entre un nom d'Item et sa représentation graphique. Cela permet de séparer le modèle, la vue et le contrôleur. Pour afficher l'inventaire, nous avons créé une classe générique **InventoryPanel** qui hérite de **JPanel** ainsi que deux classes qui héritent de **InvenroyPanel** : **InventoryMenu** et **InventoryBar**.

Pour afficher l'inventaire, il faut une instance de la classe **Inventory** et une instance de la classe ItemView. Ainsi, pour un même inventaire, il est possible de choisir quel ensemble de textures nous souhaitons appliquer aux Items.

c) Controller

Afin que la vue de l'inventaire se mette à jour lorsque l'inventaire est modifié, nous avons créé une interface **InventoryObserver** qui définit la signature de la méthode lorsqu'un Item est ajouté à l'inventaire. Nous avons ensuite ajouté un ensemble d'**InventoryObserver** à la classe **Inventory**. Ainsi, quand nous ajoutons un Item à l'inventaire, chaque observateur de l'inventaire est notifié et les différentes vues de cet inventaire sont redessinées.



3) Composant Application

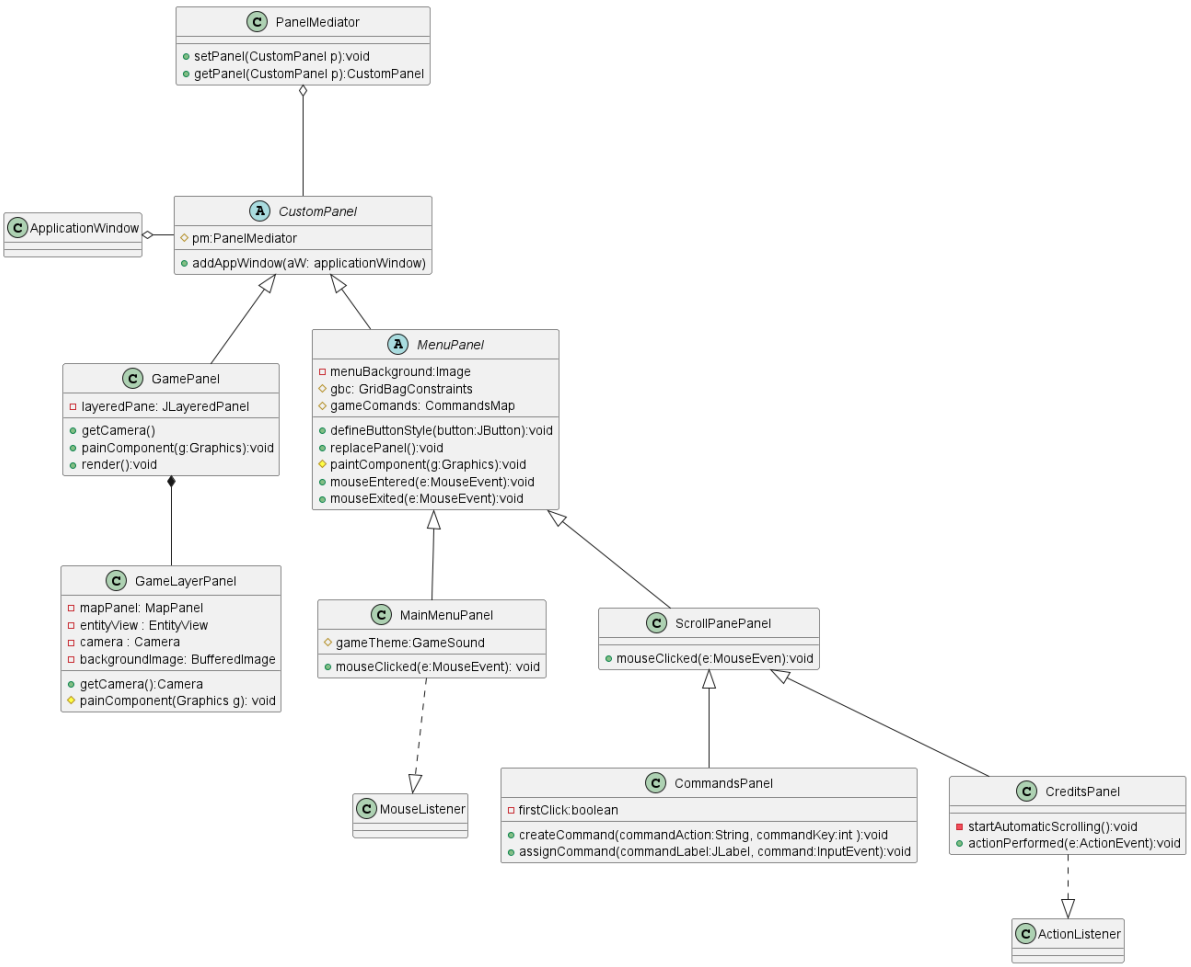
A) Présentation

Les classes ****Panel*** sont des classes héritant de la classe JPanel et s'occupent d'ordonner et de superposer les différents éléments d'affichage (vue du personnage, de l'inventaire…) afin de permettre au joueur d'interagir avec le jeu.

Le point d'entrée à l'application est la classe ***ApplicationWindow*** qui met en place le patron de conception du **Singleton** qui limite la création d'une fenêtre d'application à une unique instance. C'est à l'intérieur de cette fenêtre (*JFrame*) que les différents panels de l'application se succèderont, le recensement de tous les panels se fait dans un *Médiateur* appelé ***PanelMediator***.

Un ****Panel*** représente une page de l'application (menu, crédits, écran titre, jeu…) avec les éléments qu'il contient (boutons, personnages et environnement, HUD…). Chaque nouveau Panel hérite d'une classe abstraite ***CustomPanel*** qui recense chaque panel dans le ***PanelMediator***.

B) Diagramme de classes



C) Fonctionnement

Lors du lancement de l'application, **l'ApplicationWindow** affiche en premier lieu le titre principal **MainMenuPanel**.
Afin de découpler l'association avec les différents types de **CustomPanel**, le **PanelMediator** fait office de médiateur et permet d'implémenter la logique de changement de panels, nous avons préféré ce patron plutôt que celui de l'observateur car l'état du panel affiché ne devrait hiérarchiquement pas être celui qui notifie l'**ApplicationWindow** de changer de page. Il était plus propice de définir la logique du changement de page affiché à une classe à part entière.

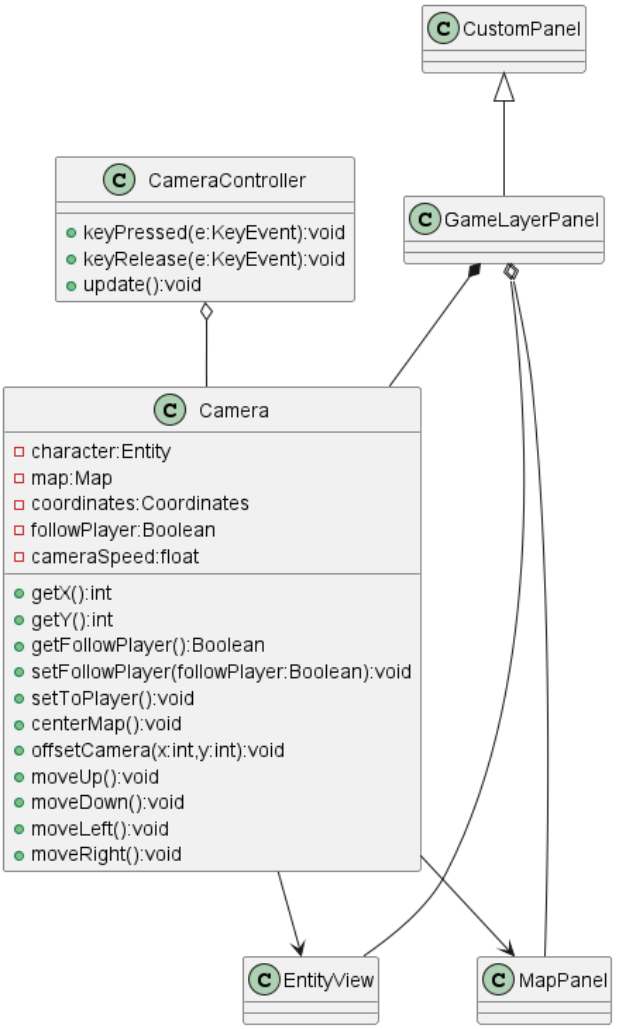
Une des pages les plus importantes est le GamePanel qui regroupe tous les éléments de vue des composantes du jeu en un **GameLayerPanel**. Le GamePanel s'occupe d'afficher au premier plan le HUD (composé uniquement de l'inventaire pour l'instant) et en second plan le panel de jeu superposé. Ce dernier superpose en effet les différentes vues obtenues à partir des composantes, c'est-à-dire, l'image de fond comme backgroundImage, la map puis la vue entité.
Ce qui est affiché alors à l'écran sont les éléments délimités par la position de la caméra (c.f 4) Camera).

4) Composant Camera

A) Présentation

Afin de pouvoir se déplacer dans l'environnement de jeu, il était nécessaire de concevoir une logique pour l'affichage qui dépendrait de la position du personnage sur cette grille de jeu. Ainsi, seuls les éléments autour du personnage et dans le périmètre de la caméra ont besoin d'être affichés au joueur.
Par défaut, la caméra est associée au personnage et se déplace selon la position du joueur, mais il est également possible de la dissocier et de mouvoir la caméra elle-même à l'aide des flèches directionnelles.

B) Diagramme de classes



C) Fonctionnement

Afin de pouvoir suivre le joueur sur la map, la Caméra a besoin d'être instanciée par leur aide.
Un booléen ***followPlayer*** indique si la caméra doit suivre ou non le personnage principal. Le comportement par défaut est que la caméra suit le personnage et est instanciée sur celui-ci. À chaque itération de boucle de jeu, la caméra se centre automatiquement sur le personnage (à condition que celui-ci soit dans le périmètre de la carte, auquel cas la caméra ne pourra plus se déplacer).

Dans le cas où il ne le suit pas, la ***CameraController*** mettra à jour les coordonnées selon les déplacements de la caméra elle-même (à qui on aura attribué des touches de contrôles).
Il aurait été possible d'implémenter un Observateur afin de notifier le ***CameraController*** des déplacements du joueur et de se désabonner de celui-ci lorsqu'on décide de ne plus suivre le personnage. Néanmoins, il a été jugé comme trop lourd à implémenter pour qu'il soit uniquement utilisé par la caméra. Il sera fortement considéré lorsque davantage de composants nécessiteront d'être avertis par l'état du joueur.

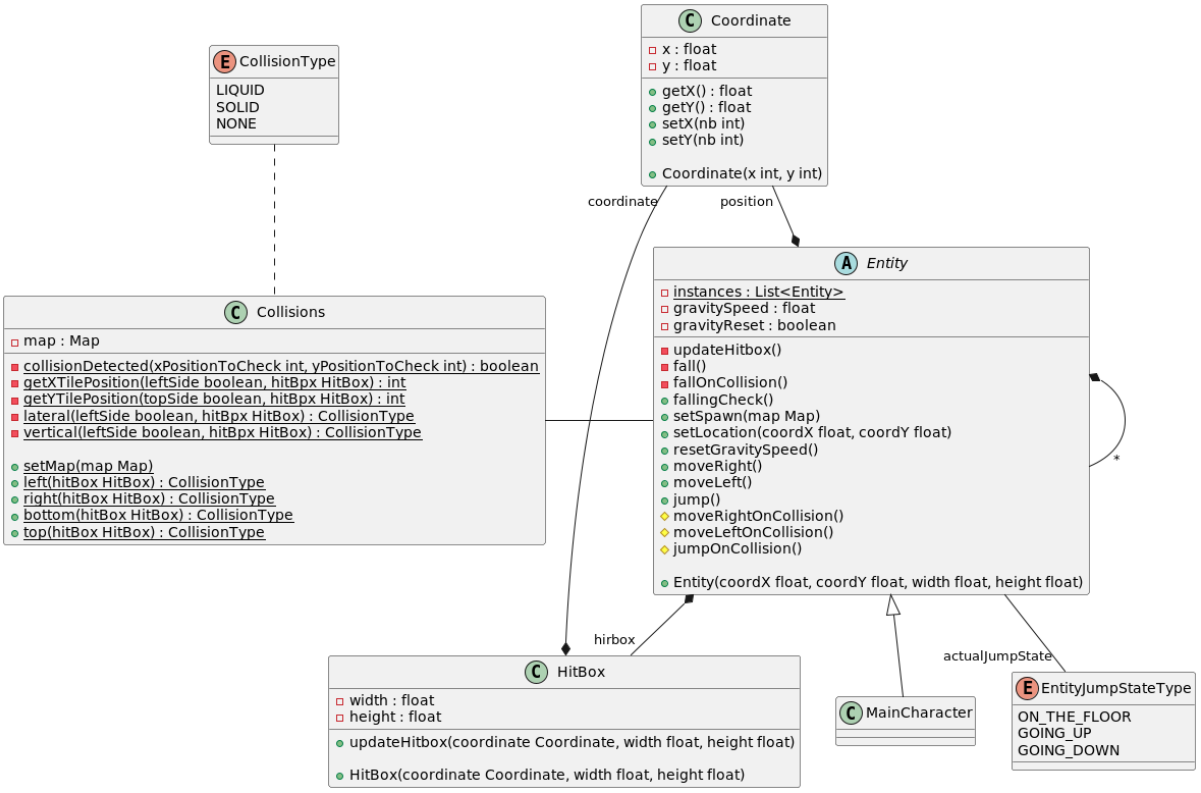
La position de la caméra est utilisée dans la méthode *paintComponent(Graphics g)* de la classe ***GameLayerPanel***. À chaque fois que l'environnement de jeu est peint, le ***GameLayerPanel*** utilise les coordonnées de la caméra comme offset et translate la map par cet offset pour afficher correctement les éléments dans le périmètre de la carte.

5) Composant Character

A) Présentation

Le composant character permet d'avoir les bases pour créer des entités (personnages joueurs et non joueurs). Pour se faire, chaque nouvelle entité héritera de la classe **Entity**. Les différentes méthodes dans la classe **Entity** permettent pour l'instant de gérer les déplacements et les collisions. Cela permet d'avoir une sorte de "kit" pour pouvoir gérer toutes les nouvelles entités possibles.

B) Diagramme de classes



C) Fonctionnement

a) Model

MainCharacter représente le personnage principal. **Maincharacter** est un singleton. **Maincharacter** hérite de la classe **Entity**. Les déplacements (**jump**, **moveRight**, **moveLeft**, **fall**) déplacent l'**Entity** et les méthodes **OnCollision** collent le personnage au bord du bloc car une collision a été détectée. Pour savoir si une collision a été détectée, on utilise les méthodes de la classe **Collisions**. La **HitBox** représente la place que prend le personnage dans son environnement (en taille de blocs). On peut donc avoir des hitbox de différentes tailles (par exemple une vache de 2 blocs de large et 0.5 blocs de haut).

b) View

EntityView hérite de la classe **JPanel,** contient le personnage et affiche l'image du personnage.

c) Controller

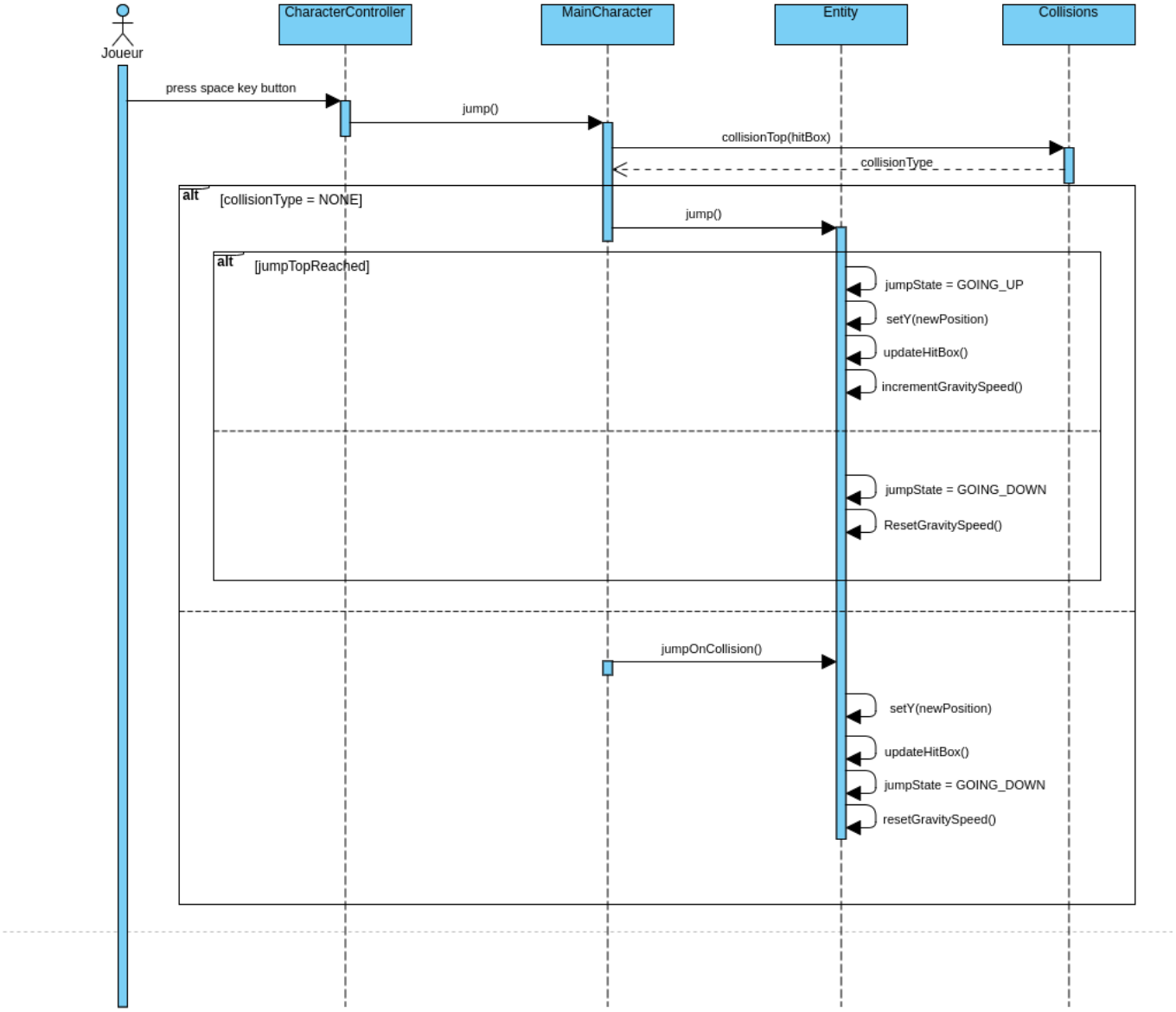
CharacterController réalise un **KeyListener.** Selon la touche utilisée par l'utilisateur, le personnage se déplace dans la direction indiquée. La méthode **update** met à jour le modèle selon les actions de l'utilisateur.

V. Dynamique du système

Afin d'illustrer quelques-unes des dynamiques que réalise le système, des diagrammes de séquences et des scénarios ont été réalisés.

1) Le joueur presse la touche "space" afin d'effectuer un saut

A) Diagramme de séquence



B) Scénario 1 : Aucune collision n'est détectée lors de la phase du saut

1) Le joueur appuie sur la touche entrée de son clavier.
2) La touche est enfoncée, la méthode jump est appelée dans le contrôleur.
3) Un appel de la méthode **collisionTop** est réalisé, permettant une simulation du déplacement du joueur sur l'axe Y, et vérifiant si la tuile du dessus est une tuile provoquant une collision ou non.
4) Dans ce scénario, aucune collision n'est détectée, la méthode jump de la classe mère est appelée.
5) Celle-ci vérifie si la courbe haute du saut a été atteinte.
    a) Elle n'est pas atteinte, l'état du personnage, ses nouvelles coordonnées, celle de sa hitBox sont mis à jour. La valeur de la vitesse de gravité est incrémentée (pour simuler cet effet de prise de vitesse).
    b) Elle est atteinte, l'état du personnage est mis à jour afin d'indiquer qu'il doit redescendre, et une remise à niveau de la valeur de la vitesse de gravité est effectuée.

C) Scénario 2 : Une collision est détectée lors de la phase du saut

1) Le joueur appuie sur la touche entrée de son clavier.
2) La touche est enfoncée, la méthode **jump** est appelée dans le contrôleur.
3) Un appel de la méthode **collisionTop** est réalisé, permettant une simulation du déplacement du joueur sur l'axe Y, et vérifiant si la tuile du dessus est une tuile provoquant une collision ou non.
4) Dans ce scénario, une collision est détectée, la méthode **jumpOnCollision** de la classe mère est appelée.
5) Celle-ci va calculer la distance entre le bloc de collision et la position du personnage actuel, mettre à jour sa position avec cette dernière valeur, changer l'état du personnage pour signifier qu'il faut retomber, et remettre à jour la valeur de la vitesse de gravité.

VI. Mise en oeuvre des méthodes agiles

Pour bien gérer le projet, nous avons mis en pratique un grand nombre de principes provenant de la méthodologie **Agile** enseignée. Nous avons donné la responsabilité de **Scrum Master** à Thomas GRUNER ayant la plus grande expérience avec la méthodologie. Le rôle de **Product Owner** a lui été donné à Cédric ABDELBAKI, c'est lui qui possédait la vision du produit et émettait les critiques d'un point de vue client.

Interactions avec le monde	1000 VM
Personnage	2000 VM
NPC	500 VM
Interface de jeu	2500 VM
Map	1500 VM
Physique du monde	1500 VM
Gestion de l'inventaire	1000 VM

Tableau des Epics et valeurs métiers attribuées

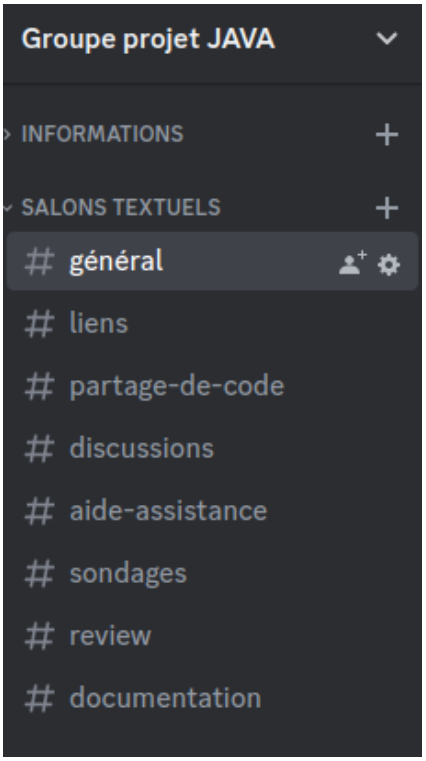
Tout d'abord, les différentes parties de l'application ont été scindées en **Epics** auxquelles ont été attribuées des **valeurs métiers** afin de prioriser au mieux les fonctionnalités à implémenter. Les **Epics** les plus prioritaires ont ensuite été découpées en **User Stories** suffisamment petites pour être implémentées au cours d'un sprint.

Afin de pouvoir être à jour au niveau de l'état de l'avancement du projet, des **dailys** étaient mises en place où chacun décrivait ses problèmes ou ses réussites. Cela s'est fait naturellement sans avoir eu besoin d'une structure rigide, les daily étant faites entre les cours.

Des **sprint planning** et des **sprint review** étaient organisés pour pouvoir discuter de la mise en œuvre, de la structure du sprint et des améliorations à adopter pour faciliter le sprint suivant.

1) Outils utilisés

A) Communication au sein du groupe : Discord



Nous avons choisi d'utiliser **Discord** pour la communication générale, afin de demander une **Pull Request**, donner des conseils ou présenter les dernières avancées à l'équipe.

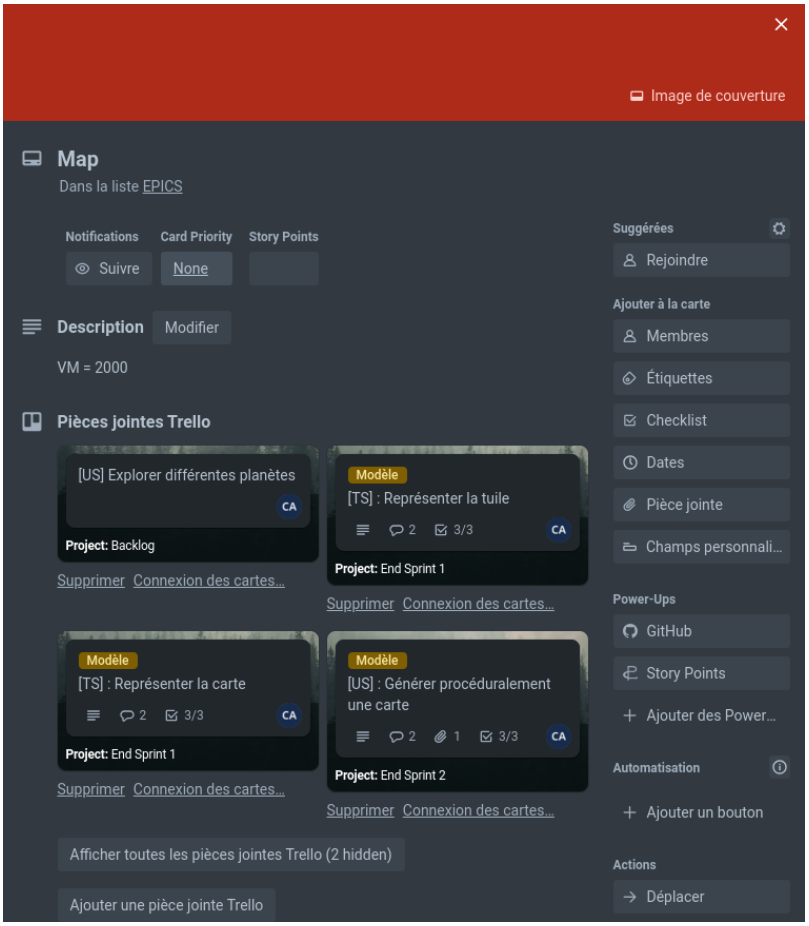
Lorsqu'un des membres de l'équipe rencontrait un problème, il pouvait rapidement envoyer un message et une session de peer coding pouvait être démarrée.

Des ressources en lien avec la programmation du jeu peuvent également être partagées : tutoriels, documentations…

C'est également par le biais de cette application que nous avons planifié notre oral ainsi que la rédaction de la présentation, du manuel utilisateur et du rapport.

B) Tableau de bord Kanban : Trello

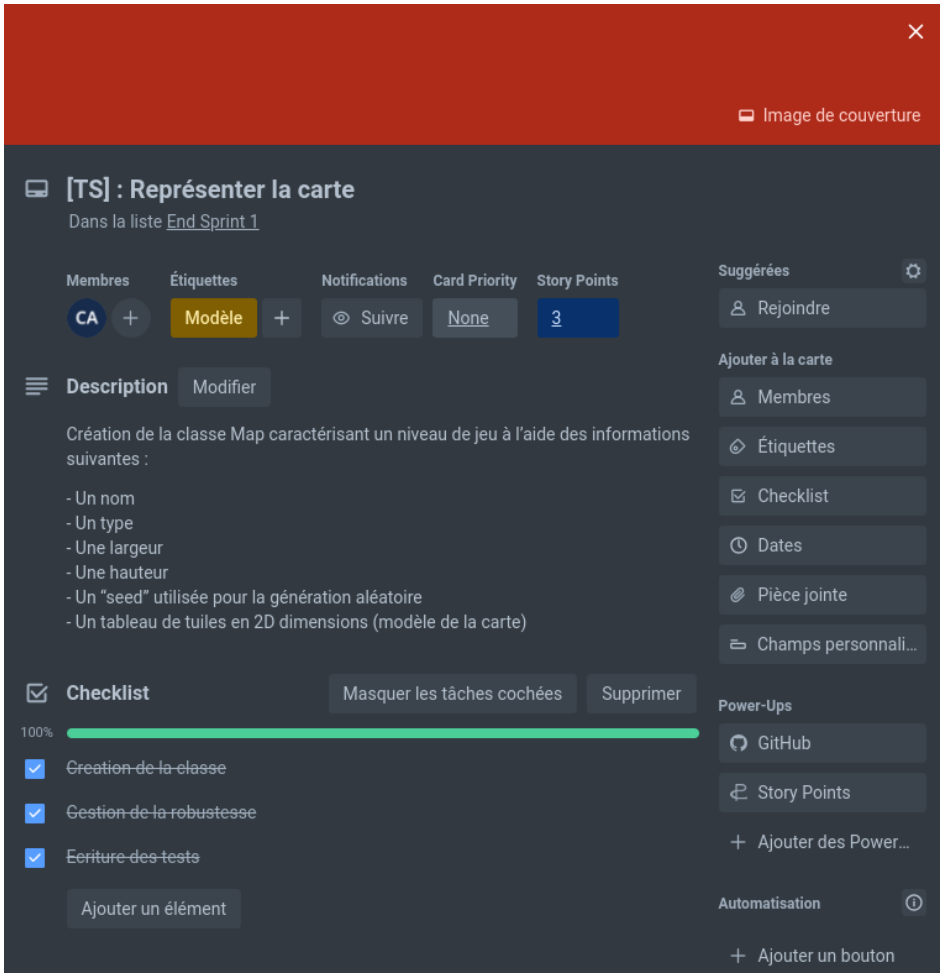
Trello a été choisi comme tableau de bord Kanban. Toute la hiérarchie **EPICs**, **User Stories** et leur avancement étape par étape sont présentés et actualisés au fil des sprints. Dans un premier temps, nous avons fait un découpage et listé les différentes **EPICs** qui allaient être réalisées.



Exemple d'une carte EPIC sur l'application Trello.

Les **EPICs** sont définies de la façon suivante :

- La **Valeur Métier** associée à l'EPIC en description
- Un **code couleur** différent pour chaque EPIC
- Les **différentes US et TS associées** à l'EPIC en **pièce jointe** pour faciliter leur accès.



Exemple d'une carte Technical Story sur Trello

Les **US/TS** sont présentés de la manière suivante :

- (**Pour les US**) La description suivante : *"En tant que <persona>, je souhaite <fonctionnalité> afin de <bénéfice>." ;*
- Le **code couleur** qui l'associe à une EPIC ;
- Le(s) **membre**(s) travaillant sur son implémentation ;
- Une **étiquette** qui permet de savoir si cela relève de la **Vue**, du **Modèle** ou du **Contrôleur** ;
- Les **story points** associés ;
- Une **description** rapide ;
- Une **checklist** facultative qui permet de découper une US (si trop grosse) en plusieurs **tâches** succinctes.

Partie	Description
To Do	Les **US** qui doivent être réalisées pour le sprint actuel.
In Progress	Les **US** qui sont en train d'être réalisées par un membre de l'équipe
Refactor	Lorsque le code doit être amélioré pour des soucis de lisibilité, de performances, ou de structure.
Testing	Une fois qu'une tâche est terminée, elle est déplacée dans cette colonne pour être testée avant d'être déployée ou considérée comme terminée.
Code Review	Le code est fonctionnel et doit être revu par un membre de l'équipe pour avoir une opinion supplémentaire. On s'assure de la qualité du code, on identifie les erreurs et bonnes pratiques à mettre en place.
Blocked	L'**US** ne peut pas être poursuivie pour des raisons diverses (notamment dépendances par rapport aux autres **US** non réalisées).
Done	L'**US** est terminée et a été merge sur la branche de développement.

De grandes parties ont été définies pour suivre la progression des **User Stories**.
Au début de chaque **sprint**, un **objectif** était choisi et les **US** pertinentes étaient alors proposées puis envoyées dans le **To Do** pour le sprint.

C) Github

Nous avons choisi **Github** comme système de contrôle de **version** (VCS) et avons mis en place tout une structure pour le processus de développement de l'application.

Description du processus de développement et d'intégration :

Nous avons dans un premier temps créé une **branche dev** qui sera utilisée comme branche principale. La **branche main**, quant à elle, sera mise à jour quand la **branche dev** aura suffisamment de contenu pour proposer une version 1 du jeu. Nous avons décidé que la **branche dev** devait être fonctionnelle, mais qu'il est possible de lui ajouter des petites améliorations afin de la faire évoluer pas à pas. Pour éviter d'ajouter du code incorrect à la **branche dev**, nous avons choisi de la protéger, c'est-à-dire que nous ne pouvons pas réaliser de commits sur cette branche mais uniquement à travers des **Pull Requests** avec l'aval d'au moins un autre membre de l'équipe.

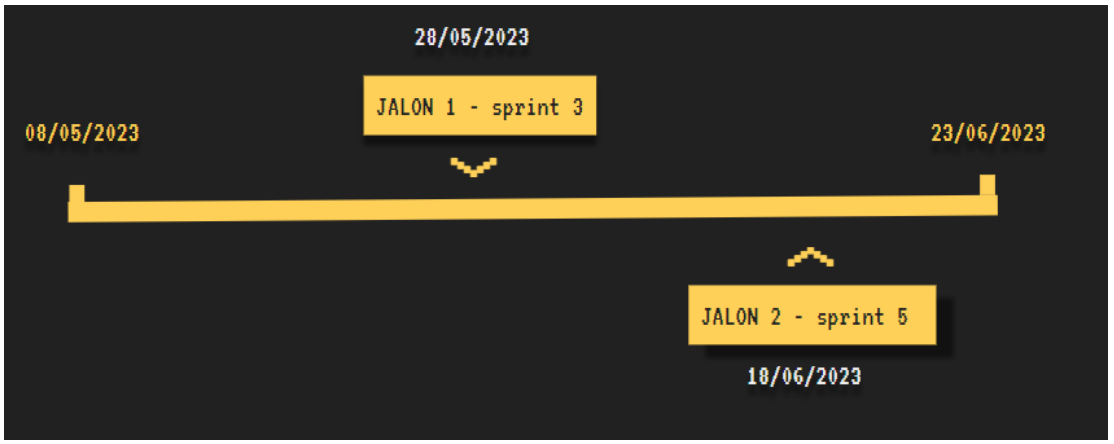
Afin d'ajouter une nouvelle fonctionnalité à notre projet ou bien de corriger un bug que nous avons repéré, nous créons une nouvelle **branche** à partir de la **branche dev** que l'on nomme "Feature/XXX" ou bien "Correction/YYY", nous faisons les ajouts et corrections nécessaires. Une fois que tout est fonctionnel, nous faisons une **Pull Request** vers la **branche dev**, un ou plusieurs autres membres du groupe se chargent de relire nos modifications et de valider ou non notre **Pull Request**. Si la **Pull Request** est approuvée, la **branche** est fusionnée (**merge**) puis supprimée. Sinon, l'auteur de la branche doit la corriger selon les remarques faites par le relecteur jusqu'à approbation. Une **branche** Feature/XXX correspond en général à une **User Story**.

Les outils proposés par **GitHub** permettent de suivre et d'effectuer les validations de **Pull Request** directement dans un navigateur. Une fois validé, **GitHub** propose de faire la fusion et de supprimer la branche, tout ceci est plus convivial que d'effectuer ces actions en ligne de commandes.

Lien vers notre projet : https://github.com/Yanis-Kouidri/pixel-adventurer/tree/dev

2) Avancement du projet par sprint

Nous avons lors de la mise en place du **framework** décidé d'un planning prévisionnel pour structurer les projets et de définir des objectifs à atteindre. En se basant sur ces **jalons**, nous avons pu évaluer si le projet se déroulait de manière attendue et sans retard.



Dans un premier temps, nous avons posé des jalons : un de mi-projet au **sprint 3** et un de fin de projet avant la soutenance au **sprint 5**.
Au début du projet, des objectifs ont été fixés :

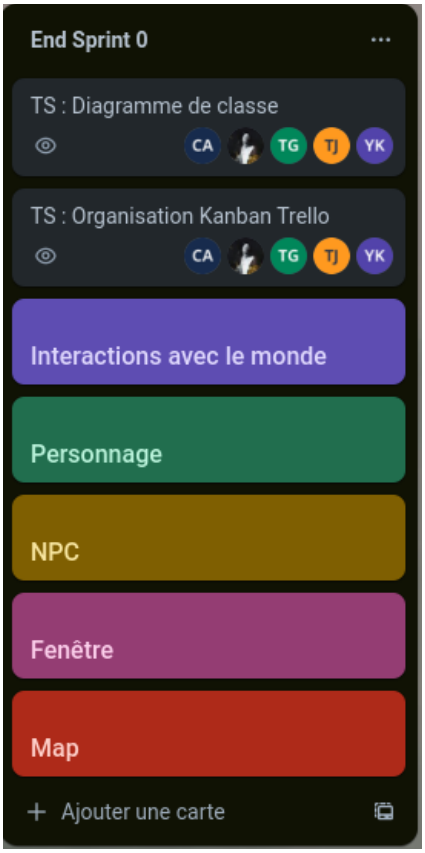
- Pour le **jalon 1** :
  - Une map générée aléatoirement ;
  - Un personnage se déplaçant sur la map ;
  - Des blocs qui peuvent être cassés et posés.
- Pour le **jalon 2** :
  - La possibilité de générer différentes planètes ;
  - Un système de sauvegarde de parties ;
  - Des personnages non-joueurs qui peuplent le monde.

Néanmoins, nous avons rapidement vu qu'atteindre ces objectifs n'allait pas être réaliste dans le temps imposé si nous voulions créer un moteur de jeu réutilisable. À partir du **sprint 1**, les objectifs ont donc été réévalués :

- Pour le **jalon 1** :
  - Une map générée aléatoirement ;
  - Un personnage qui se déplace sans collision.
- Pour le **jalon 2** :
  - L'implémentation de la collision sur le personnage ;
  - Des blocs qui peuvent être cassés et posés ;
  - Une gestion de la caméra ;
  - Une amélioration visuelle de l'interface.

C'est également à ce moment-là que nous avons remarqué que certaines **User Stories** ont été découpées de manière beaucoup trop grossières pour pouvoir être embarquées de cette manière. Il est sans doute probable que le manque d'une catégorie Feature ait été à l'origine de ce mauvais découpage.

A) Sprint 0



Objectif du sprint : Découpage du projet en **Epics** et **US** (de manière grossière) et l'attribution de certains rôles.

Tout d'abord, afin de nous organiser au mieux pour la suite du projet et de s'approprier la méthode Agile, un Sprint de test, appelé **Sprint 0** a été réalisé. L'objectif principal étant de mettre en place les fondations pour un déroulement sans accrocs.

L'accent a été mis sur la préparation du code, chaque Epic avait un référent :

- Map → Cédric
- Personnage → Thomas/Jérémy
- Fenêtre de jeu → Eric
- Inventaire → Yanis

Des premiers diagrammes de classes ont été réalisés pour chaque composante indépendante afin de commencer la programmation au sprint suivant.
Le **Kanban** et sa structure ont également été décidés lors de ce sprint.

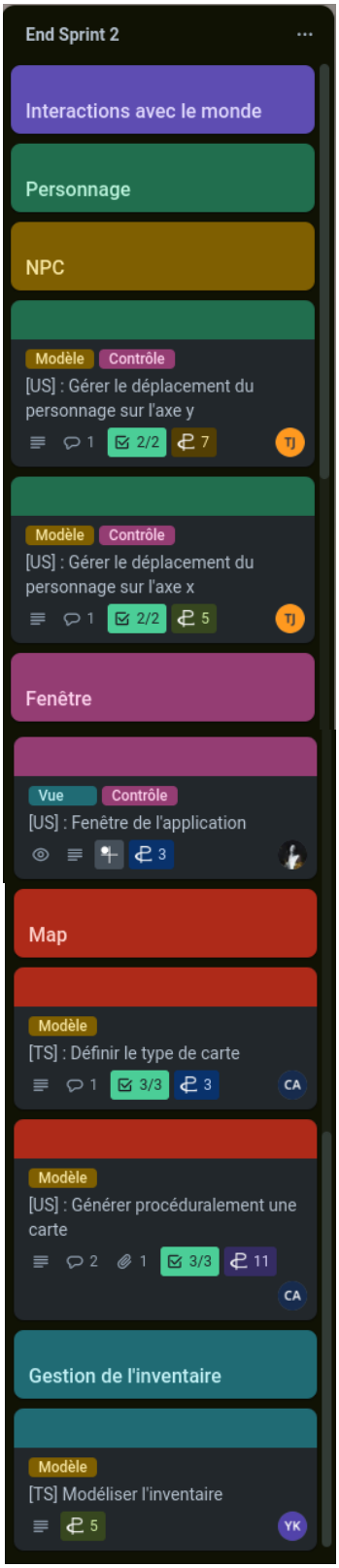
B) Sprint 1

Objectif du sprint : Création des composantes indépendantes et leur logique. Chiffrage des différentes user stories.

Ce premier vrai sprint avait pour but de commencer la programmation avec l'architecture donnée.

En vue du premier jalon prévu pour le **sprint 3**, l'avancement était satisfaisant, de grosses fonctionnalités ont commencé à être implémentées au niveau de la carte, mais également au niveau du personnage. La base du personnage (coordonnées, affichage) et de la carte (représentation dans un terminal) ont été terminées.

C) Sprint 2



Objectif du sprint : Continuer l'implémentation des composantes de manière indépendante.

Rien de particulier, l'avancement du projet s'est fait normalement et nous avons continué le développement des composantes.

Nous avons par contre, en matière de gestion de projet, mis en place les **Story Points** (ceux des sprints précédents ont été faits rétroactivement) qui nous ont beaucoup servi pour la suite et nous ont permis de prévisualiser et embarquer un nombre limité de **User Stories** pour les **sprints** suivants.

D) Sprint 3

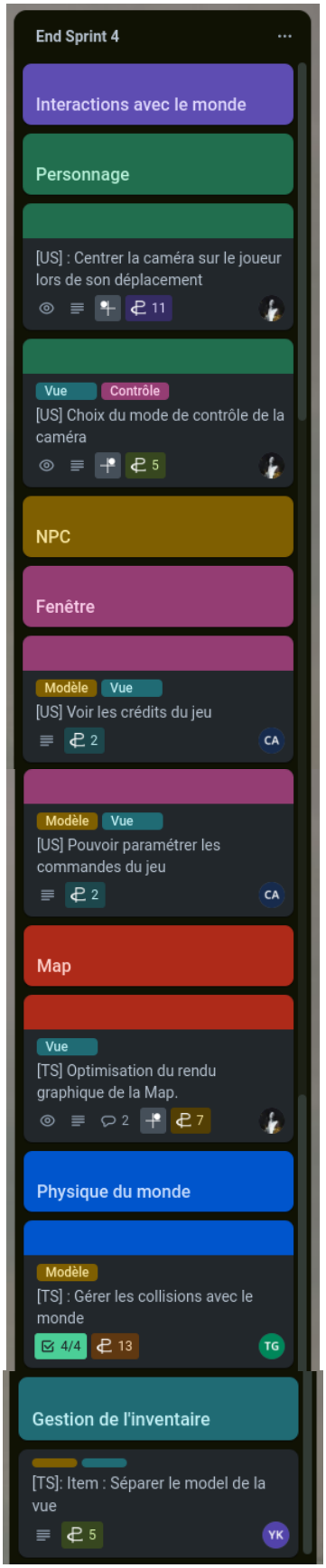
Objectif du sprint : Intégration de l'ensemble du travail fourni jusqu'à présent pour atteindre le **premier jalon** réévalué.

La partie la plus redoutée : mettre en relation les différentes composantes développées jusqu'à présent. Le **merging** des **US** était relativement simple étant donné que les composantes n'avaient aucune dépendance entre elles, mais leur liaison s'est avérée compliquée.

C'est dû à cette complexité qu'un rôle a également été attribué à un des membres, celui du responsable d'intégration. Ainsi, Jérémy était chargé de gérer l'assemblage des différentes parties en une application et jeu **Pixel Adventurer**.

L'inventaire a également été commencé lors de ce sprint avec pour responsable Yanis Kouidri.

E) Sprint 4



Objectif du sprint : Amélioration des fonctionnalités précédentes et optimisation.

Un **sprint** relativement tranquille où des améliorations de qualité de vie ont été développées, notamment au niveau de la caméra et des menus.

L'inventaire a également été ré-architecturé afin de respecter le **MVC**.

F) Sprint 5

Objectif du sprint : Mise en place des premières mécaniques de jeu.

Ce dernier sprint nous a permis de commencer le gameplay maintenant que le personnage pouvait se déplacer correctement sur la map avec la gestion des collisions et de la gravité.

Ainsi, nous avons développé la destruction de blocs et un inventaire fonctionnel dans lequel chaque bloc cassé était stocké.

L'interaction entre la souris et l'inventaire était également prévue, mais faute de temps n'ont pas pu être commencés.

3) Rétrospective

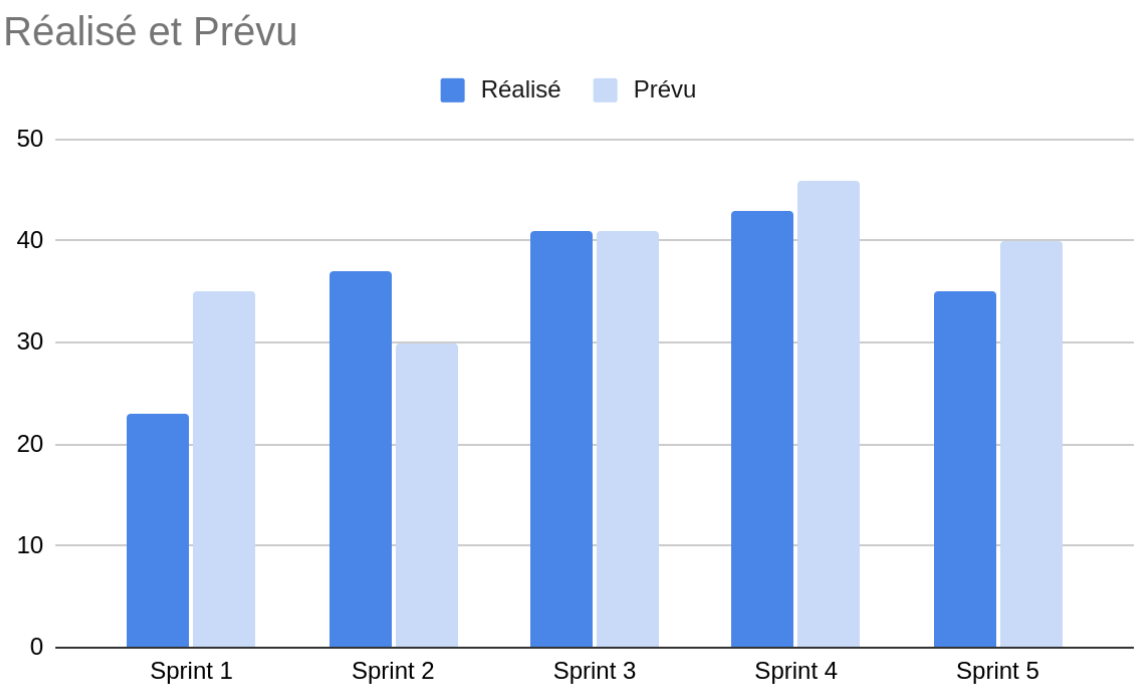


Diagramme de vélocité prévisionnel et effectif de chaque sprint.

Finalement, cette méthodologie **AGILE** flexible adoptée nous a été énormément bénéfique afin de pouvoir suivre correctement et nous retrouver dans le processus de développement de l'application. Avoir un **Product Owner** nous a permis de nous recentrer à chaque fois sur un objectif précis et de ne pas nous égarer dans des divagations non attendues par le client. Les **Story Points** attribués à chaque **User Story** et la priorisation grâce aux objectifs de sprint nous ont efficacement orientés sur nos choix d'inclusion d'**US** à chacun de nos **sprints**.

Nous pouvons voir qu'au fil du temps notre vélocité n'a cessé de croître et était de plus en plus en adéquation avec notre vélocité prévisionnelle (mis à part le **sprint 5** qui s'est déroulé pendant la période d'examens ce qui nous a empêché d'être aussi efficaces).

Sur un projet plus long, plus conséquent et avec plus de membres, la mise en place d'une structure plus rigide doit sans doute être plus efficace (évaluation plus stricte de vélocité et valeur métier, davantage de cérémonies…) mais dans le cadre de notre projet, il nous a semblé adéquat d'omettre certains outils pour pouvoir produire un programme avec plus de fonctionnalités. De plus, se concentrer sur les fondements de l'agilité avec une équipe qui ne s'y connaissait pas semblait être une approche intéressante.

VII. Conclusion

Les objectifs de ce projet étaient multiples. Premièrement, nous avons pu mettre en pratique tout ce que nous avons appris dans le module : **Conception et Programmation Objet**. Nous avons eu l'occasion de créer des classes, des interfaces, de mettre en place de l'héritage, d'appliquer des **patrons de conception**, de respecter la séparation du code en **modèle, vue** et **contrôleur**, d'utiliser **Java Swing**, etc.

Parallèlement, nous avons appliqué la méthodologie de gestion de **projet agile** "**Scrum**". Cela n'a pas été simple à appréhender, au début, nous avions l'impression de passer beaucoup de notre temps à faire de la gestion de projet et peu à coder. Cependant, avec le temps et l'expérience, nous avons apprivoisé la **méthodologie agile**. Nous avons pris l'habitude d'ouvrir des tickets sur **Trello**, de penser en termes d'**User Stories** et non de code.

Une fois la méthode agile bien ancrée, nous avons pu travailler en autonomie, chacun étant libre de choisir ses **US** dans le **backlog**, de créer une **branche** sur **Git** et de s'occuper de son code. Nous discutions tous les jours de l'avancée du projet et de nos éventuels problèmes, mais nous savions tout ce que nous avions à faire.

Malgré ces difficultés, à la fois d'un point de vue programmation et d'un point de vue gestion de projet, nous avons su les surmonter. Tous ces efforts nous ont permis de construire un jeu vidéo, pièce par pièce, sans utiliser de moteur de jeu et d'obtenir un résultat satisfaisant.

Notre projet n'est pas seulement un jeu vidéo dans l'espace, c'est aussi un moteur de jeu permettant de créer n'importe quel jeu vidéo en deux dimensions en adaptant certains paramètres et en changeant les ressources graphiques.

Nous vous remercions pour l'attention portée à la lecture de ce rapport.