Rapport Projet TOB : civ++

Jérémie Carrez, Hinniger-Foray Nohé, Lazare Léo, Seghrouchni Anas, –
Kssim Aymane, Jeandon Emile, Kallel Ismail

Contents

1 Introduction

Ce rapport présente les différentes étape de création de notre projet TOB : civ++, un jeu incrémental en Java.
Le concept principal de ce jeu est de faire évoluer sa civilisation de l'antiquité à l'ère numérique avant qu'une immense météorite anéantisse tout le monde, et qu'il faille recommencer...
Au fur et à mesure des resets, le joueur peut néanmoins garder certains bâtiments uniques et donc leeurs bonus, ce qui lui permet de progresser de plus en plus rapidement et donc d'atteindre l'ère finale au bout d'un moment. Des informations supplémentaires sont disponibles dans le manuel utilisateur.

Ce rapport contient :

- les principales fonctionnalités et leur état de réalisation
- le découpage de l'application en sous-systèmes
- l'architecture et les diagrammes de classe de l'application
- les principaux choix de conception et réalisation
- l'organisation de l'équipe et la mise en oeuvre des méthodes agiles

2 Fonctionnalités et états de réalisation

Les principales fonctionnalités sont décrites ci-dessous

- Évolution des statistiques
  Le jeu comprend plusieurs statistiques que le joueur doit prendre en compte pour progresser plus rapidement et efficacement. Elles sont parfois liées entre elles et certaines influent sur d'autres (par exemple, si il n'y a pas assez de logement pour tout le monde, le bonheur diminue). Le joueur peut voir combien il possède de chaque statistique, et les augmenter en construisant des bâtiments (cf plus bas).

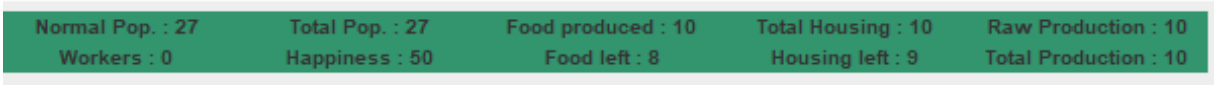

Figure 1: L'affichage de toutes les statistiques

  Il a donc fallu gérer une classe abstraite Stat qui avait les méthodes principales et communes à chaque statistique, puis appliquer des modifications pour les statistiques les plus complexes, ayant des liens avec d'autres statistiques. Nous avons utilisé la redéfinition pour certaines méthodes afin d'effectuer cela. Des classes de test Junit ont également été crées parallèlement au développement de ces classes afin de vérifier le bon comportement de ces dernières avant leur implantation dans le jeu.
  Afin de gagner des statistiques à la construction d'un bâtiment; nous avons également créé une classe Bonus, liée à chaque bâtiment achetable, que l'on peut fournir à La classe AllStats qui regroupent toutes les statistiques lors d'une partie et permet d'y accéder. Cette classe s'occupe de rediriger le bonus à la bonne statistique. Les bonus peuvent être "statiques" (augmentation d'un montant fixe d'une statistique) ou en pourcentage (i.e. augmentation d'un certain pourcentage de la statistique).
  Cette fonctionnalité a été entièrement développée et est par conséquent entièrement fonctionnelle, et ce depuis l'itération 2.

- Construire les bâtiments
  Afin de progresser plus rapidement, et de réussir à atteindre la dernière ère, le joueur doit construire différents bâtiments qui fournissent divers bonus aux différentes statistiques. Il existe différents types

de bâtiments (cf manuel utilisateur) qui donnent des bonus différents et qui résistent ou pas à la météorite (reset de la partie).
Afin de construire ces bâtiments, le joueur a accès à une liste (appelée Shop) qui contient tous les bâtiments actuellement constructibles. La communication entre la partie visuelle (Swing) et la partie logique se fait à l'aide de méthodes permettant d'obtenir directement les informations utiles pour l'affichage, sans que ce dernier n'ait à faire de vérification (par exemple regarder si un bâtiment est effectivement constructible).

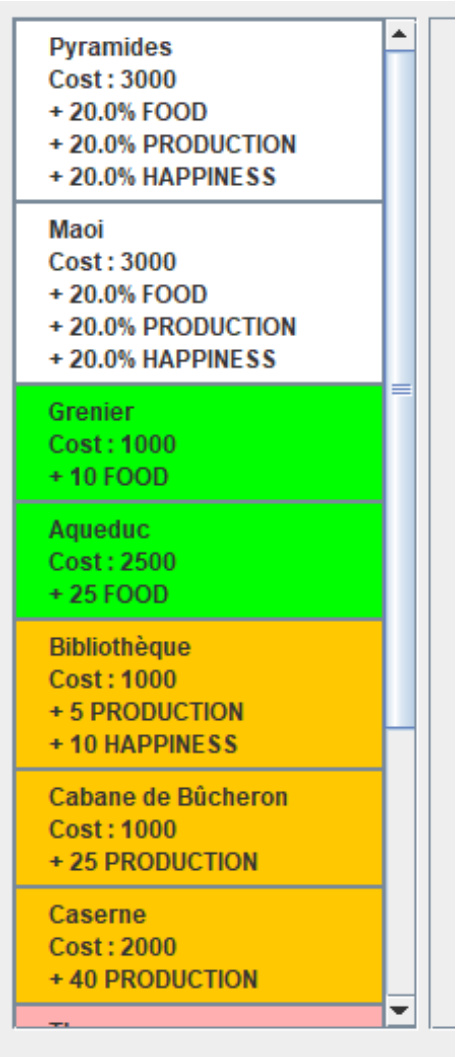

Figure 2: L'affichage des différents bâtiments constructibles.

Une autre liste contenant tous les bâtiments déjà construits est également affichée à la droite du Shop, et permet de voir les différents bonus s'appliquant sur les statistiques de la partie.
La liste de tous les bâtiments, avec leurs informations et bonus, est disponible au format json dans le fichier purchasables.json. Nous utilisons un module afin de les convertir en classe Java (cf découpage de l'application, item /Content).

Cette fonctionnalité est également entièrement fonctionnelle depuis l'itération 3.

- Afficher les bâtiments
  Les bâtiments uniques (ceux qui ne disparaissent pas après passage de la météorite) sont affichés sur l'île pour que l'utilisateur puisse encore plus facilement voir les bâtiments importants qu'il possède déjà (ceux là fournissant une partie importante des statistiques). Afin de connaître l'ère actuelle du

joueur il suffit de regarder l'Hôtel de ville actuel, qui évolue d'ère en ère, celui ci étant affiché en plein centre de l'île. Les différentes images ont été dessinées par nos soins.

Figure 3: L'île avec plusieurs bâtiments uniques construits

Cette fonctionnalité est opérationnelle depuis l'itération 3.

- Convertir la population en ouvrier

  Les ouvriers sont une forme spéciale de la population qui augment considérablement la vitesse de construction des bâtiments, mais qui ne peuvent pas se reproduire (cf manuel utilisateur). Le joueur peut convertir une de sa population en ouvrier en cliquant sur l'île, à condition qu'il possède encore au moins 2 population "normale".

  Cette fonctionnalité est opérationnelle depuis la première itération.

- Changer d'ère

  Afin de changer d'ère, l'utilisateur doit construire l'hôtel de ville suivant. C'est la classe Run qui vérifie l'hôtel de ville actuel afin de modifier l'ère de la partie si besoin. La classe Shop s'occupe de modifier la liste des bâtiments constructibles en fonction de l'ère de la partie actuelle. L'ère est également remise à l'antiquité lors du passage de la météorite.

  Cette fonctionnalité est opérationnelle depuis l'itération 2.

- Repartir de zéro

  Nous avons mis en place un timer qui, une fois arrivé à zéro, termine la Run actuelle est en recrée une autre avec les bâtiments uniques précédemment construits. C'est la classe Game qui s'occupe

de gérer cette partie en baissant la valeur de "tick" progressivement. La durée entre chaque tick est modifiable dans cette même classe afin de gérer l'équilibrage de la difficulté du jeu.

Le joueur a accès au temps restant avant arrivée de la météorite par le biais d'une barre de progression au bas de l'application :

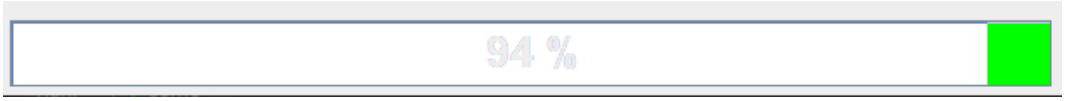

Figure 4: La barre de progression avant l'arrivée de la météorite... et de la mise à zéro de la Run.

Cette fonctionnalité est opérationnelle depuis l'itération 1. Cependant, nous avons récemment remarqué différents problèmes lors de la remise à zéro pour les autres fonctionnalités listées ci-dessus, notamment l'impossibilité de construire à nouveau les bâtiments non uniques une fois la partie remise à zéro.

3 Découpage de l'application en sous-systèmes

L'application est découpée en 6 dossier principaux.

- **/Content**, le contenu de l'application:
  Ce dossier à pour but de stocker le contenu du jeu, toutes les instances d'objets, de bâtiments ect, pour ne pas surcharger le coeur du jeu de classes. Ce dossier contient pour l'instant seulement un fichier purchasables.json. Ce fichier contient les caractéristiques de tout les "purchasables" (bâtiments achetables dans le "shop"). Voici par exemple le purchasable "pyramides" décrit en json:

```
{
"name": "Pyramides",
"bonuses": [
  {
    "stat": "FOOD",
    "staticUpgrade": 0,
    "percentUpgrade": 0.2
  },
  {
    "stat": "PRODUCTION",
    "staticUpgrade": 0,
    "percentUpgrade": 0.2
  },
  {
    "stat": "HAPPINESS",
    "staticUpgrade": 0,
    "percentUpgrade": 0.2
  }
],
"price": 3000,
"era": "ANTIQUITY",
"isPurchased": false,
"type": "UNIQUE"
},
```

- **/Dependencies**, les bibliothèques utilisées par l'application.
  Ce dossier permet de stocker les différents .jar des bibliothèques supplémentaires dont l'application dépend. Dans notre cas il n'y a que GSON.jar, une librairie dévelopée par google qui permet de parser des objets java en json et vice-versa. Nous reviendrons sur son utilisation plus loin.

- **/Logic**, le coeur logique de l'application.
  Ce package contient le "backend" logique de l'application. Cela est utilisé dans un désir de séprarer la partie visuelle et interactive de la partie fonctionnement du jeu. La classe principale de ce package est Game, c'est aussi la classe principale de l'application qui permets de la lancer. Game instancie les différents composants de l'application et leur permet de communiquer. Run est la classe qui définit une partie, avec une durée de vie limitée. Toutes les différentes statistiques sont aussi dans le package Logic. Pour finir, shop est l'une des classes les plus importantes, car une grande partie de l'interactivité viens du système de recherche et construction de batiements, ce qui est géré par la classe shop. Cette classe utilise la librairie GSON mentionnée plus haut afin d'instancier les objets purchasables directement dans un tableau depuis le fichier json.

- **/Sprites**, le pixel-art.
  Le dossier sprites contient l'ensemble des assets visuels que nous avons crées pour le jeu. Ces composants visuels ont été entièrement crées par l'équipe en pixel-art en utilisant le logiciel aesprite. Voici quelques batiments:

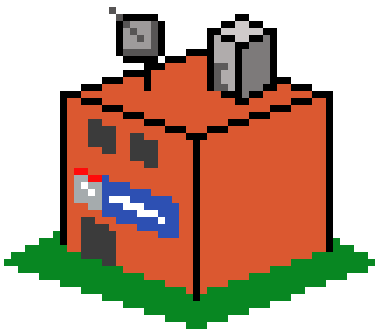

Figure 5: Le bâtiment N7

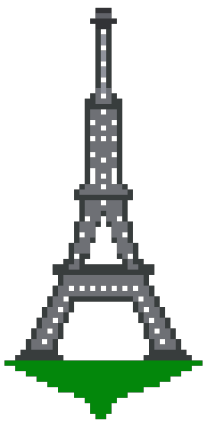

Figure 6: La tour Eiffel

- **/Utils**, utilitaires.
  Le package Utils contients des fonctions et scripts utilitaires qui n'on leur place dans aucun des autres packages. Il a au final assez peu servi, mais nous aurions pu refactoriser quelques scripts communs dans ce package.

- **/Visuals**, la partie visuelle et interactive du jeu.
  Comme son nom l'indique le package visuals contient tout le "front-end" du jeu, qu'il s'agisse de l'affichage ou de l'interation homme machine. Il est en charge d'afficher l'état du jeu, et de transmettre les actions du joueur à la partie logique. La partie visuelle utilise Swing pour fonctionner.

4 Architecture et les diagrammes de classe de l'application

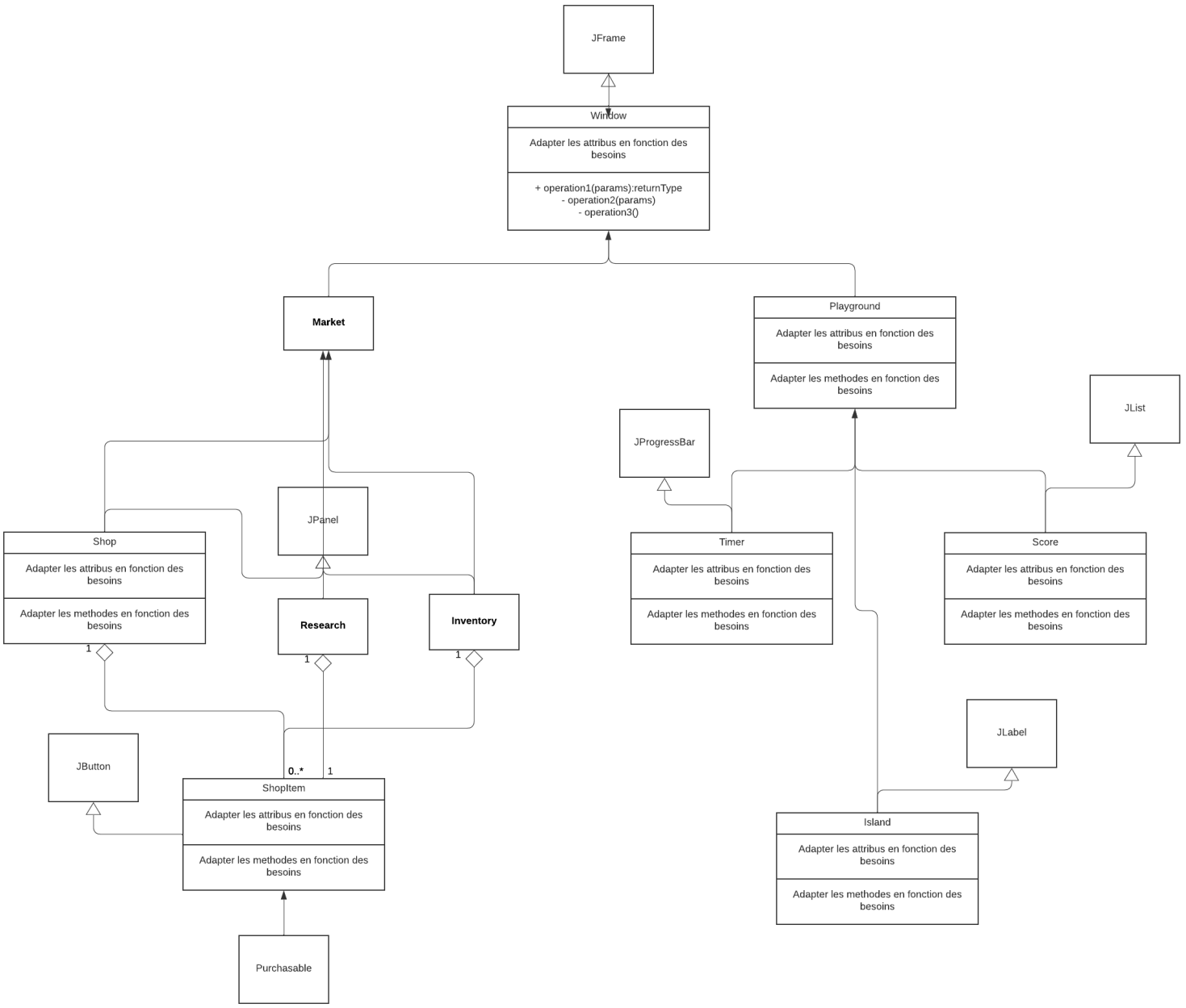

Figure 7: UML Affichage

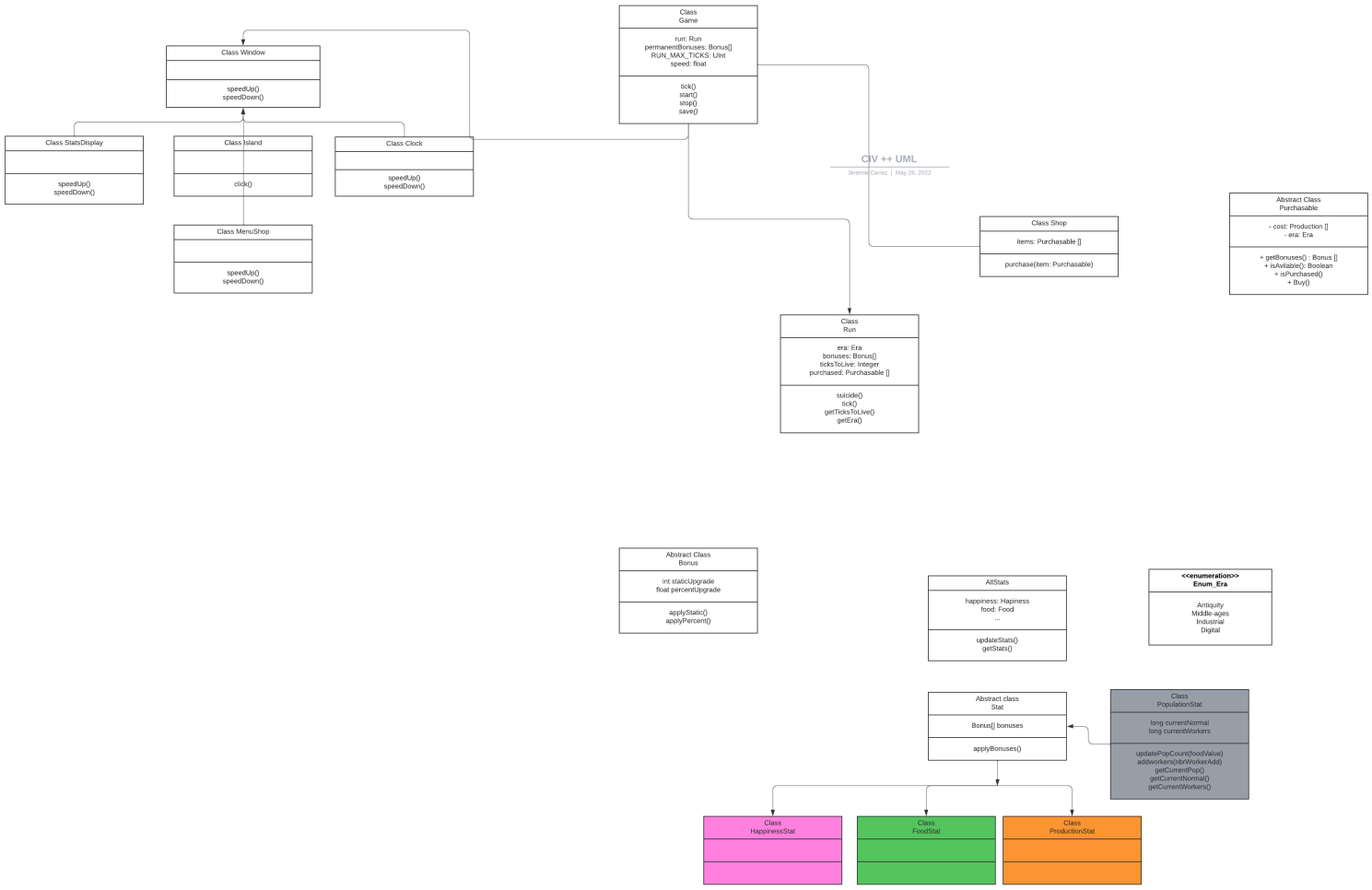

Figure 8: UML global

5 Choix de conception et réalisation

- **Système de ticks.**
  Afin de faire avancer le jeu, nous avons mis en place un système très commun dans les jeux vidéos: les ticks. Chaque tick correspond à une mise à jour du jeu (calcul des statistiques, temps restant ...). C'est donc la classe Game qui appelle une fonction tick à intervalle régulière (117ms à l'heure ou j'écris ce rapport). Chaque classe logique à une méthode tick (et certaines classes visuelles aussi), appelées par tick de Game. Les Run ont une durée de vie définie en tick (300 ticks) apres laquelle elle revient à 0, tout en conservant les bâtiments de type unique. Les shop utilise les ticks pour construire les bâtiments, en enlevant chaque tick une valeur du temps de recherche total définit par le purchasable (qui dépend des statistiques de la run).

- **Purchasables et GSON.**
  Pour gérer les objets achetables par le joueur (Purchasables), nous avons choisi de les représenter par une classe Purchasable, qui contient toutes les caractéristiques:

```
public enum BATIMENT {
            COMMUN,UNIQUE,HDV
    }

    private final String name;
```

```
private final Bonus[] bonuses; //ensemble des bonus donnés par l'achat
private final int price; //prix d'achat en production d'un achetable
private final Run.Era era; //ere minimum pour débloquer
private boolean isPurchased;
private final BATIMENT type; // catégorie de bâtiment
```

Plutôt que de créer une classe par purchasable, ce qui aurait pris beaucoup de fichiers, de temps et été très dur à maintenir, nous avons cherché une solution pour stocker toutes leur caractéristiques dans un même endroit. Nous avons opté pour GSON une librairie développée par Google, qui permet de lire et d'écrire des instances d'objets java au format JSON. C'est une libraire très simple d'utilisation mais pour autant très puissante, qui à partir de simplement une String JSON et du type crée une instance d'objet. Voici comment l'ensemble des purchasables sont instanciées dans un tableau (dans la classe Shop):

```
ArrayList<Purchasable> items;
String json = .... // lecture du ficher purchasables.json

items = gson.fromJson(json, new TypeToken<ArrayList<Purchasable>>(){}.getType());

// items contient désormais une instance de tout les purchasables
// décrits dans le fichier purchasables.json
```

6 Organisation de l'équipe et la mise en oeuvre des méthodes agiles

6.1 Découpage de l'application :

Nous avons commencé par découper l'application en deux grandes parties : la partie visuelle et la partie modèle. Nous avons ensuite établi les principales fonctionnalités à réaliser par chaque partie en réalisant des diagrammes de classe primitifs qui ont servi de guide pour le développement.

6.2 Détails des fonctionnalités :

Suit au découpage de l'application, et donc la division en sous-équipes, nous nous rencontrions pour organiser des réunions ou nous détaillons les fonctionnalités requises et le répartitions en tâches individuelles que nous attribuons à chacun des membres, tout en respectant la méthodologie agile, en attribuant des niveaux de difficulté pour chaque tâche et en gardant la communication entre sous équipes.

6.3 Sous-équipes

6.3.1 Equipe Visuelle

- Aymane Kssim : Architecture du paquetage visuel de l'application et implémentation du magasin du jeu.
- Leo Lazarre : Implémentation de la barre de scores et remplissage du JSON.
- Anas Seghrouchni : Art et implémentation de l'île de jeu en plus du compteur à rebours.

6.3.2 Equipe Logique

- Nohe Hinniger-Foray : Implémentation de l'architecture du paquetage Logique de l'application, écriture des tests unitaires ainsi que du magasin (côté Logique).
- Jeremie Carrez : Implémentation des fonctionnalites du jeu tels que l'évolution de la population, les bonus reçus et design du gameplay.

- Ismail Kallel : Equilibrage du jeu.
- Emile Jeandon : Implémentation des bâtiments.

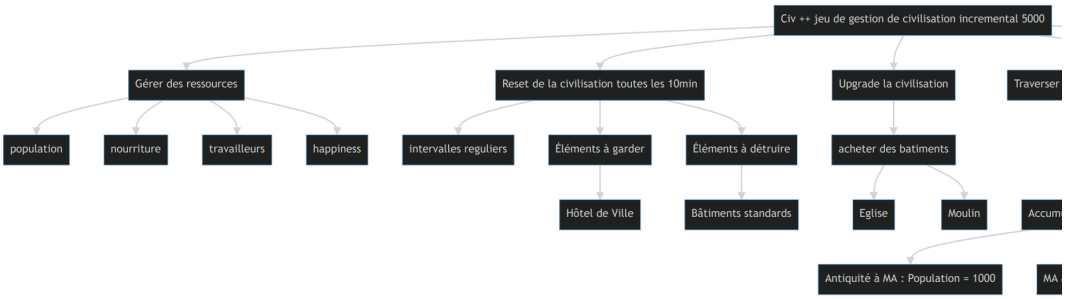

Figure 9: Backlog User partie 1

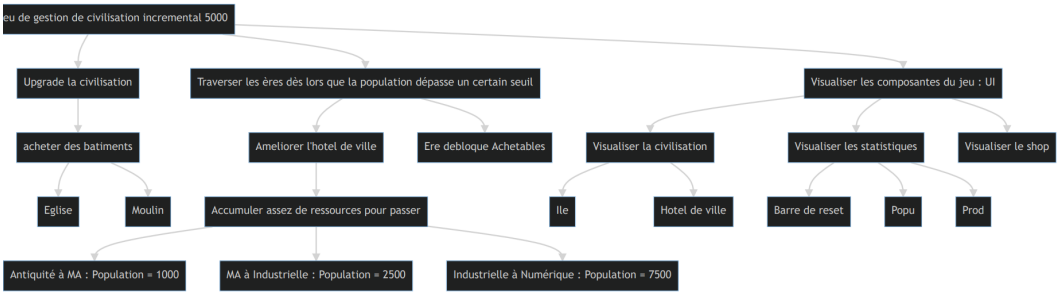

Figure 10: Backlog User partie 1

7 Conclusion

Malgré le fait que ce projet ne soit pas encore finalisé, les différents membres ont beaucoup appris de sa réalisation que ce soit au niveau technique comme au niveau des méthodes agiles et de l'organisation.