Groupe 7
début : 17/06/19
màj : 19/06/19

Axel **GOURDIN**
Abou Bakr **KHEMISSI**
Mathieu **PELLAN**

Obliss

Sommaire

Introduction

Synopsis

Obliss, jeu plateforme inspiré grandement par le jeu-vidéo Terraria et par son proche cousin Minecraft. Obliss est à l'origine un jeu de type bac à sable où l'on peut casser des blocs et les poser et inclut un léger système de craft.

Vous êtes au contrôle d'un petit héro au nom inconnu, il est arrivé par malheureux hasard dans un monde médiéval-fantastique suite à un accident technique avec un micro-onde dans le monde contemporain.

Équipe

Dans cette partie nous présenterons les différents aspects de l'équipe et du projet rendu : la composition de l'équipe et de la répartition des tâches, des outils utilisés pour lesdites tâches.

Le groupe 7 est composé de trois développeurs : Axel GOURDIN, à la tête du développement algorithmique ; Abou Bakr KHEMISSI, assistance au développement et à la conception ; Mathieu PELLAN, à la tête de la conception.

Ce groupe a été fait ainsi par une alchimie déjà présente depuis le semestre dernier. Nous connaissions nos points forts et nos points faibles afin que chacun puisse épauler l'autre dans un domaine spécifique.

La méthode Agile s'est faite sans trop de problèmes au sein de notre groupe. Chaque sprint à été plus ou moins respecté à quelques jours près.

Développement

Cahier des charges

Axel	- menu - chargement map - mouvements - collisions - gestion environnement - binding hotbar - binding inventaire - binding craft
Abou Bakr	- document de conception - chargement map - rectification mouvements - rectification collisions - conception modèles - programmation modèles - test JUnit
Mathieu	- document de conception - character design - map design - répartition des tâches - chargement map - conception modèles - interface inventaire - craft modèle

Les tâches se sont réalisées au fur et à mesure des sprints.

Les sprints se sont répartis de cette manière :

- **Sprint 1 :** conception globale, *chargement de la map, premiers déplacements, graphic designs*
- **Sprint 2 :** *collisions et bordures de map*
- **Sprint 3 :** *début de conception modèles, gameloop et gravité, interface inventaire*
- **Sprint 4 :** *continuation de conception modèles, changement de map, début de craft*
- **Sprint 5 :** finition du changement de map, finition craft, hotbar, finitions et corrections du code, Tests JUnit

Logiciels utilisés

Nous avons utilisé de multiples logiciels pour mener à bien ce projet, que ce soit pour le management, pour le design et les graphismes réalisés ou pour la programmation.

En voici une liste non exhaustive :

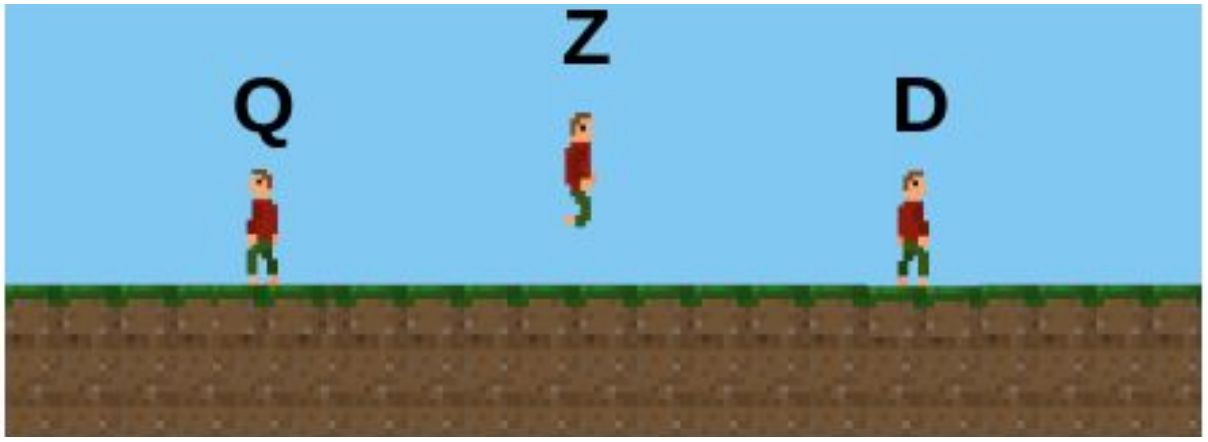
- **Eclipse :** *IDE pour la programmation Java et JavaFX*
- **SceneBuilder :** *interface JavaFX*
- **GitHub :** *management de rendu de contenu*
- **Trello :** *management des tâches*
- **yED :** *réalisations UML*
- **Piskel :** *graphic design*
- **Tiled :** *création des maps*
- **Google Docs :** *documents de conception et de rendu*

Utilisation

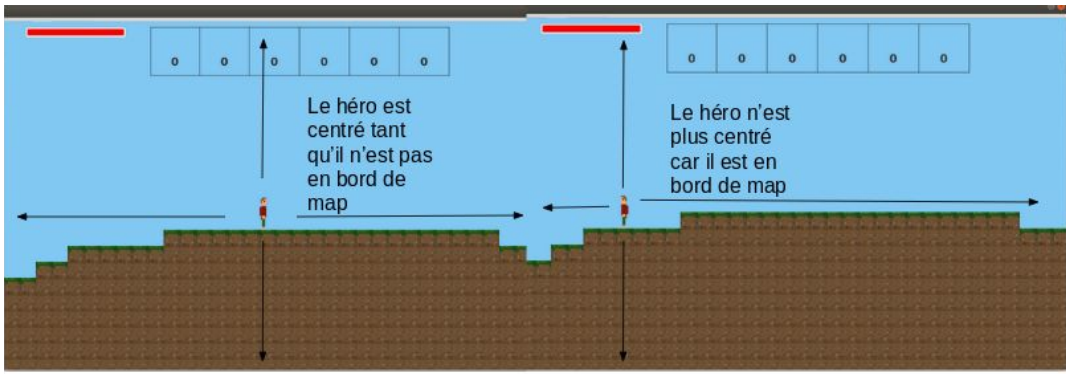
Mécaniques

Interface menu

Touches de déplacement



Fonctionnalité de déplacement

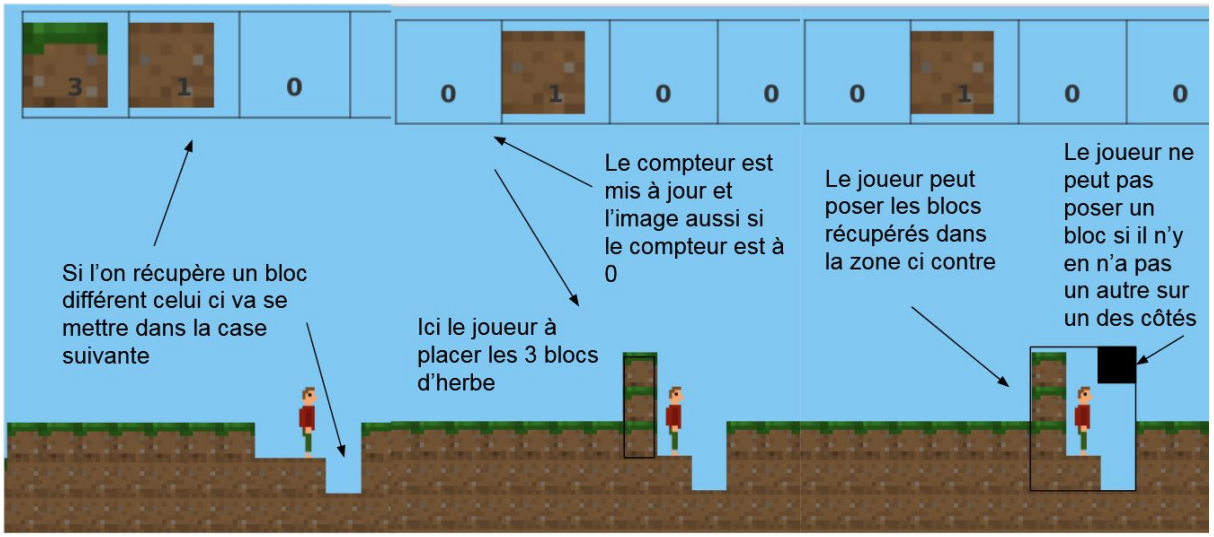


Au vu du chargement de la map, nous pouvons nous permettre d'avoir des bordures de map, lorsque le joueur est proche d'une bordure, la caméra n'est plus centrée sur lui.

Fonctionnalité d'attaque sur ennemie

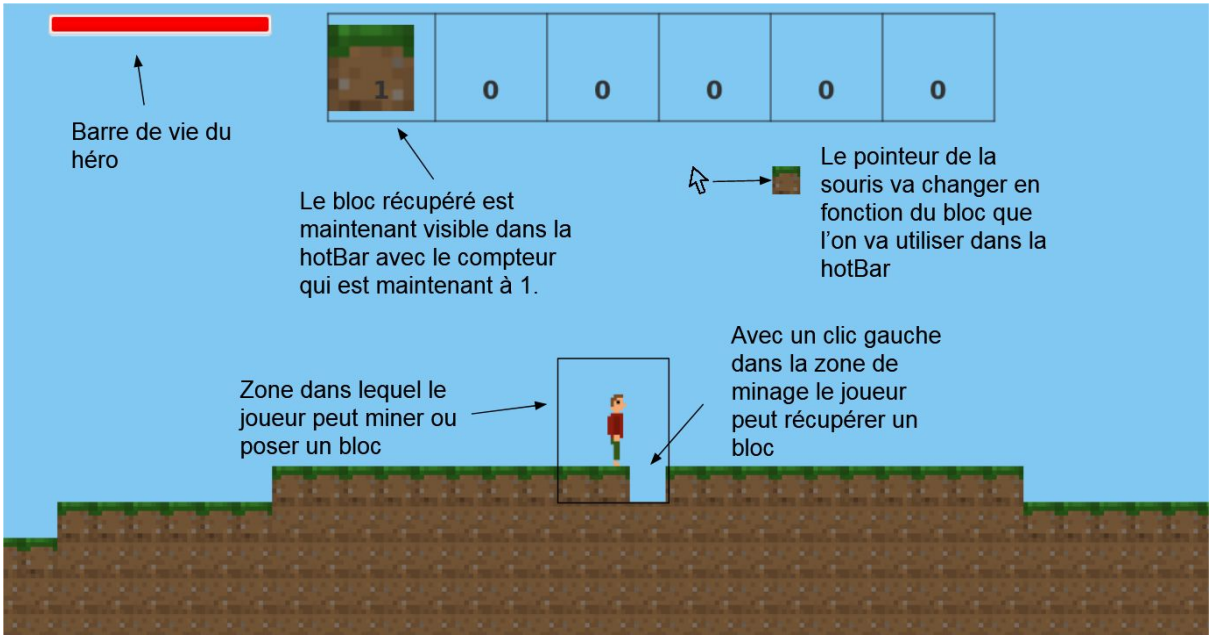
Au bout des quelques coups assénés, un monstre peut mourir.

Minage et récupération de tuile

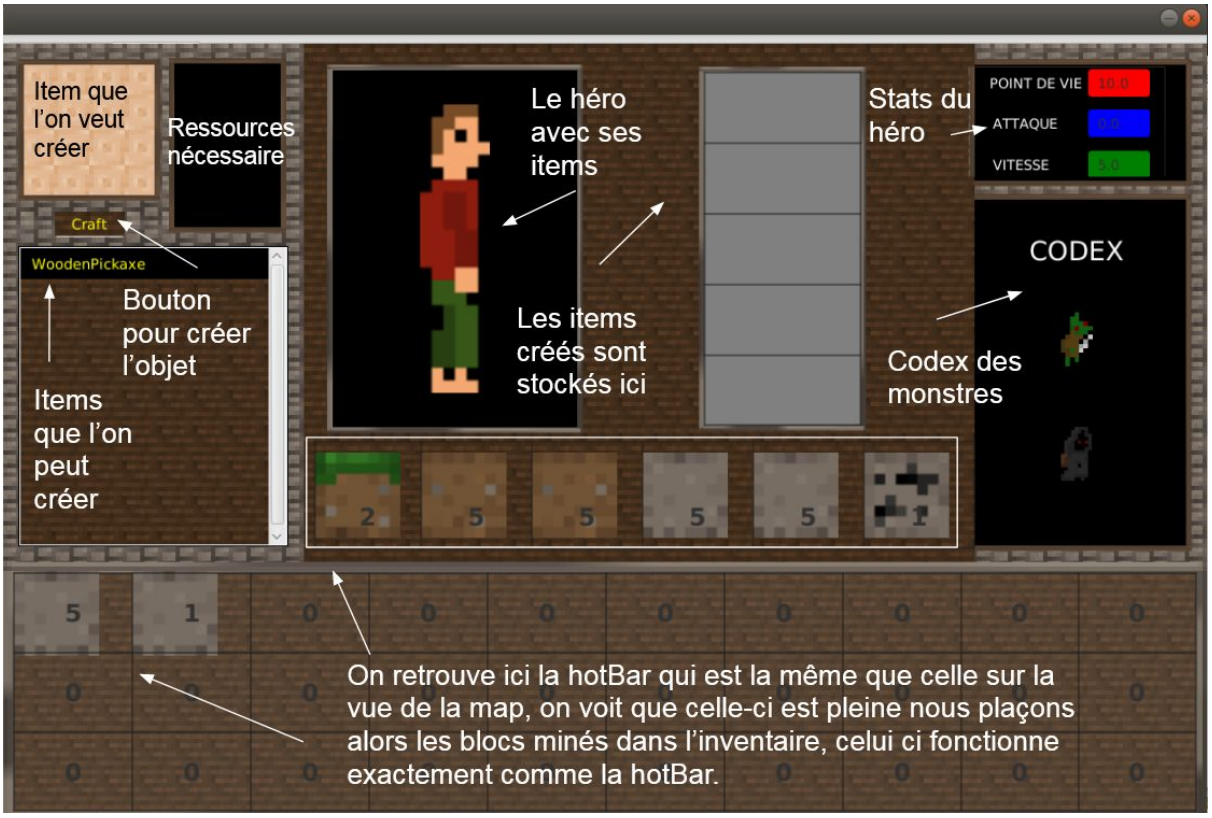


(voir diagrammes de séquence)

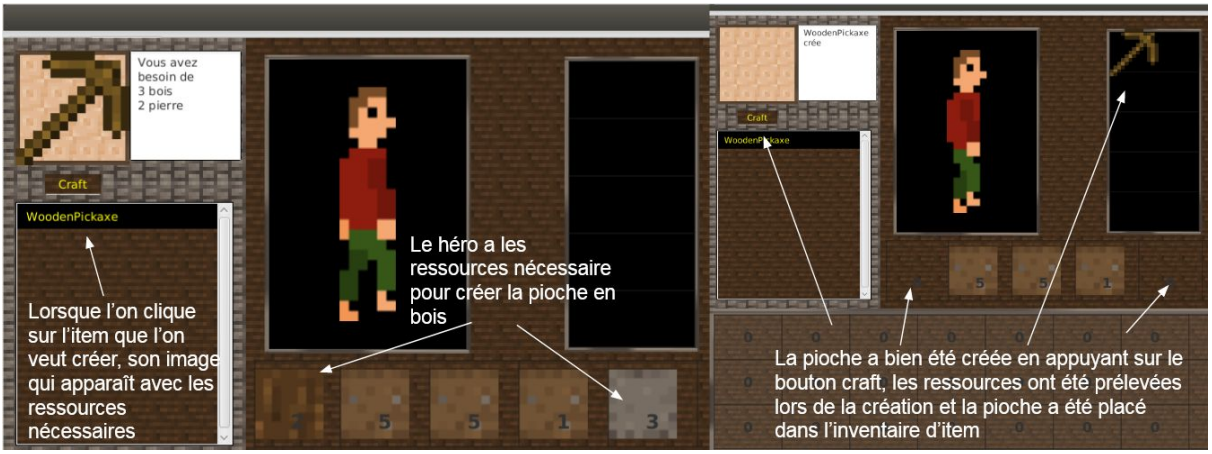
Interface joueur



Inventaire



Fonctionnalités d'inventaire



Fonctionnalités

Chargement de la map

Notre équipe a opté pour un chargement statique de la map à partir d'un fichier.csv. Une fois initialisée dans le contrôleur, la map va générer une OBlist de 100*100 tuiles.
Chacune de ces tuiles possèdent un ImageView, un ID et une position.

Craft

Le craft a dû être redéfini suite à la conception initiale des modèles. Dorénavant, nous devons avoir les ressources nécessaires dans l'inventaire. Une fois que le craft est réalisé, les ressources disparaissent (ainsi que les ImageViews) et l'item apparaît dans une zone dédiée (c.f mécaniques).

Le seul objet craftable pour le moment est la pioche en bois, elle nécessite du bois et de la pierre.

Diagrammes

Initialement, nous étions partis sur un concept bien défini vis-à-vis des modèles (voir les deux images ci-dessous).

Convenablement réparties et triées, les différentes classes étaient bien rangées dans des paquetages différents, nous avions une conception de modèle jusqu'à présent correcte.

Or, suite à des problèmes d'intégration trop nombreux, nous étions contraints de délaisser cette orientation conceptuelle et de passer à d'autres choses à cause d'un (manque) de temps imparti devenu réduit.

Initialement nous avions deux packages distincts : active_entities et inactive_entities.

Dans active_entities, il y a la classe Character dont tous les personnages du jeu hériteront. Sous la classe Character, il y a deux sous-classes abstraites : agressive et passive.

La classe agressive contient les sous-classes des personnages qui pourront se battre telles que Monsters ou Hero; et la classe passive contient ceux qui ne pourront pas attaquer comme la classe Animals ou NPCs (personnages non joueurs).

Toutes ces classes sont abstraites sauf Hero car nous ne voulons pas appeler directement ces classes mais les sous-classes qui en héritent. Ainsi les classes Chicken, Cow et Sheep héritent de Animals - Goblin, Dragon et Skeleton héritent de Monsters.

Dans inactive_entities , il y a la classe principale : Items. Tous les objets que le héros pourra collecter et fabriquer hériteront de cette classe.

Par exemple, il y a la classe abstraite Food qui permettra de créer les objets Kebab ou Tenders qui soigneront le héros .

La classe maneuverable permettra de créer tous les objets dont le héros pourra s'équiper tels que Tool, Armor ou Weapon.

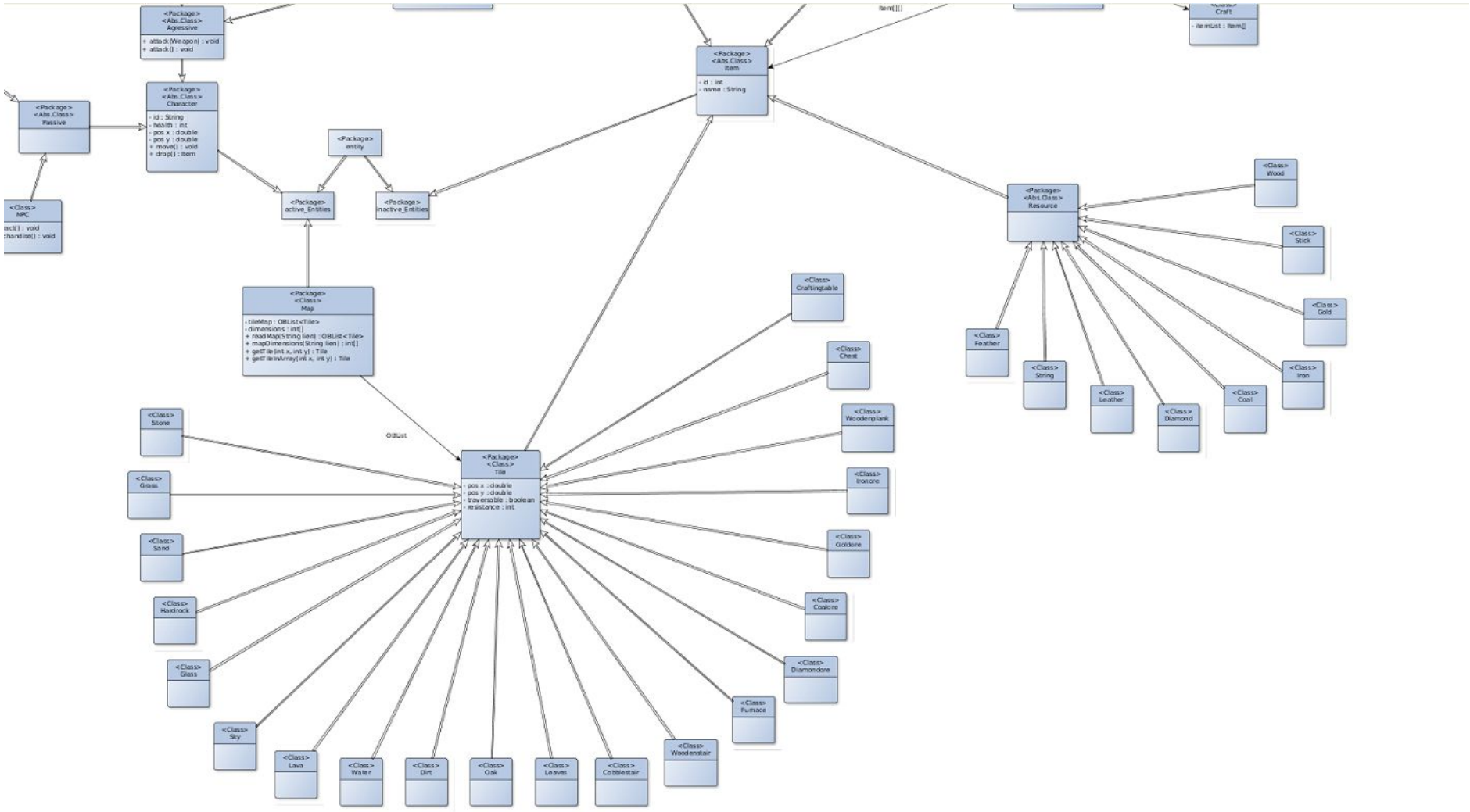
Ces classes sont également abstraites afin de n'utiliser que les méthodes uniquement sur les objets qui appartiendront à ces classes.

Dans Tools, on peut retrouver les objets Pickaxe, Axe et Hoe ; dans Armor, on retrouve : Boot et ChestPlate et dans Weapon, on retrouve Sword et Bow.

Il y a également la sous-classe Resource héritant d'Items qui permettra de fabriquer les différents items manœuvrables (Tool, Weapon, Armor). On trouvera par exemple les classes Gold, Diamond , Iron ...

Enfin, la dernière classe héritant d'Items est tuile qui sont toutes les tuiles présentes sur le terrain de jeu.
Ces tuiles pourront être collectée
s grâce aux outils afin de pouvoir fabriquer les ressources. Goldore, Diamondore, Ironore hériteront ainsi de tuile.

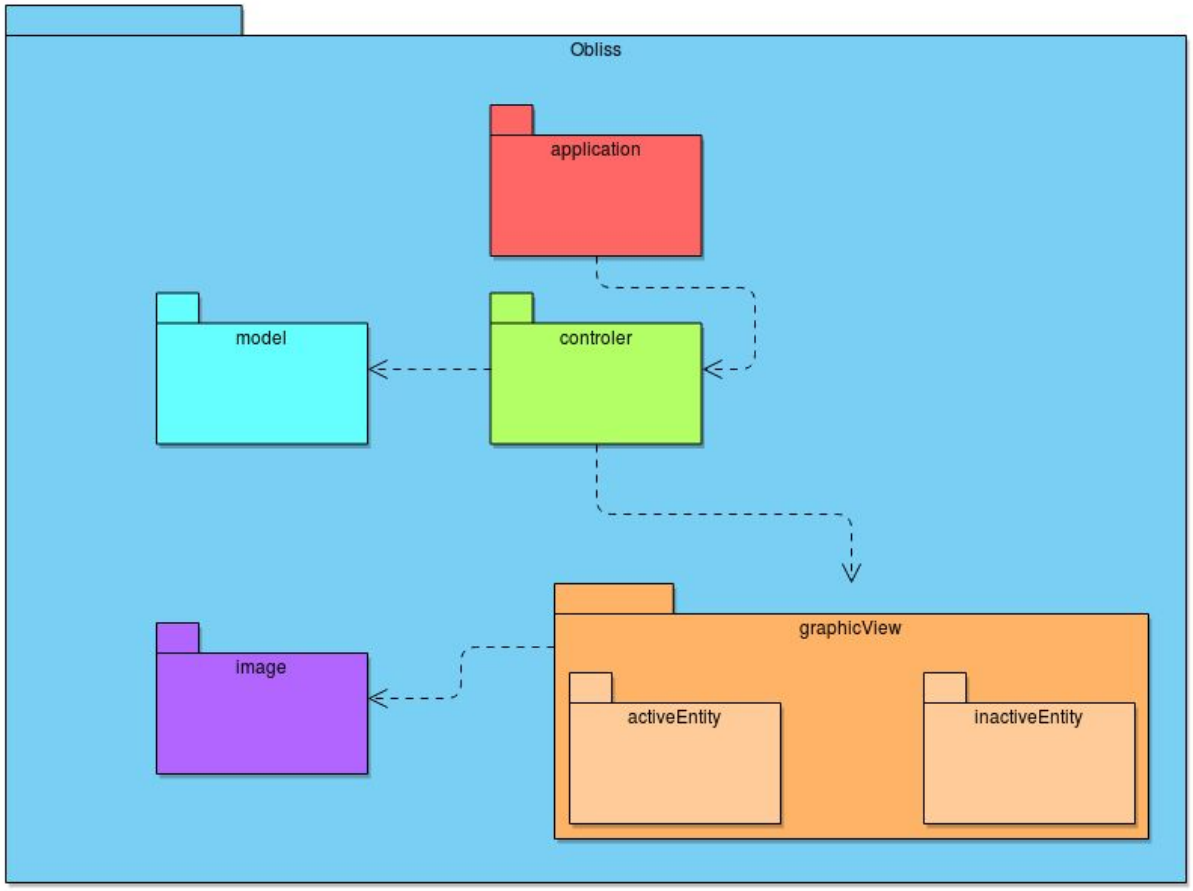
<Package>
Interface
<Class>
Player
+ eat(Food) : void
+ equip(Wearable) : void
+ use(Tool) : void
+ trade(Resource) : void
<Package>
<Class>
Inventory
- holder : Item[]
- wearables : Armor[]
- inventory : Item[]
+ stackItem() : void
<Package>
<Abs.Class>
Item
- id : int
- name : String
<Package>
<Abs.Class>
Wearable
- id : int
- name : String
- material : int
+ getCraft() : Resource[][]
<Package>
<Abs.Class>
Armor
- defensePoints : int
<Package>
<Abs.Class>
Weapon
- damage : int
<Package>
<Abs.Class>
Tool
+ canBreak() : boolean
<Package>
<Abs.Class>
Food
- pointsGiven : int
<Package>
<Abs.Class>
Resource
<Package>
<Abs.Class>
Agressive
+ attack(Weapon) : void
+ attack() : void
<Package>
<Abs.Class>
Monsters
- flying : boolean
<Package>
<Abs.Class>
Animals
<Package>
<Abs.Class>
Passive
<Package>
<Abs.Class>
Character
- id : String
- health : int
- pos.x : double
- pos.y : double
+ move() : void
+ drop() : Item
<Package>
Inactive Entities
<Package>
Entity
<Package>
Active Entities
<Class>
NPC
+ interact() : void
+ merchandise() : void
<Class>
Pig
<Class>
Cow
<Class>
Sheep
<Class>
Chicken
<Class>
Dragon
<Class>
Skeleton
<Class>
Necromancian
<Class>
Goblin
<Class>
Bucket
<Class>
Shovel
<Class>
Pickaxe
<Class>
Axe
<Class>
Hoe
<Class>
Dagger
<Class>
Sword
<Class>
Bow
<Class>
Wizardstaff
<Class>
Gauntlets
<Class>
Chestplate
<Class>
Helmet
<Class>
Boots
<Class>
Leggings
<Class>
Tunics
<Class>
Poncho
<Class>
Cloak
<Class>
Banana
item[][]


<<Package>>
<<Abs.Class>>
Agressive
+ attack(Weapon) : void
+ attack() : void
<<Package>>
<<Abs.Class>>
Character
- id : String
- health : int
- pos x : double
- pos y : double
+ move() : void
+ drop() : item
<<Package>>
<<Abs.Class>>
Passive
<<Class>>
NPC
<<Package>>
entity
<<Package>>
active_Entities
<<Package>>
inactive_Entities
<<Package>>
<<Abs.Class>>
Item
- id : int
- name : String
item[][]
<<Class>>
Craft
- itemList : Item[]
<<Package>>
<<Abs.Class>>
Resource
<<Class>>
Wood
<<Class>>
Stick
<<Class>>
Gold
<<Class>>
Iron
<<Class>>
Coal
<<Class>>
Diamond
<<Class>>
Leather
<<Class>>
String
<<Class>>
Feather
<<Package>>
<<Class>>
Map
- tileMap : OBList<Tile>
- dimensions : int[]
+ readMap(String lien) : OBList<Tile>
+ mapDimensions(String lien) : int[]
+ getTile(int x, int y) : Tile
+ getTileInArray(int x, int y) : Tile
OBList
<<Package>>
<<Class>>
Tile
- pos x : double
- pos y : double
- traversable : boolean
- resistance : int
<<Class>>
CraftingTable
<<Class>>
Chest
<<Class>>
Woodenplank
<<Class>>
Ironore
<<Class>>
Goldore
<<Class>>
Coalore
<<Class>>
Diamondore
<<Class>>
Furnace
<<Class>>
Woodenstair
<<Class>>
Cobblestair
<<Class>>
Oak
<<Class>>
Leaves
<<Class>>
Dirt
<<Class>>
Water
<<Class>>
Lava
<<Class>>
Sky
<<Class>>
Glass
<<Class>>
Hardrock
<<Class>>
Sand
<<Class>>
Grass
<<Class>>
Stone

Architecture globale

Packages

Diagramme de package

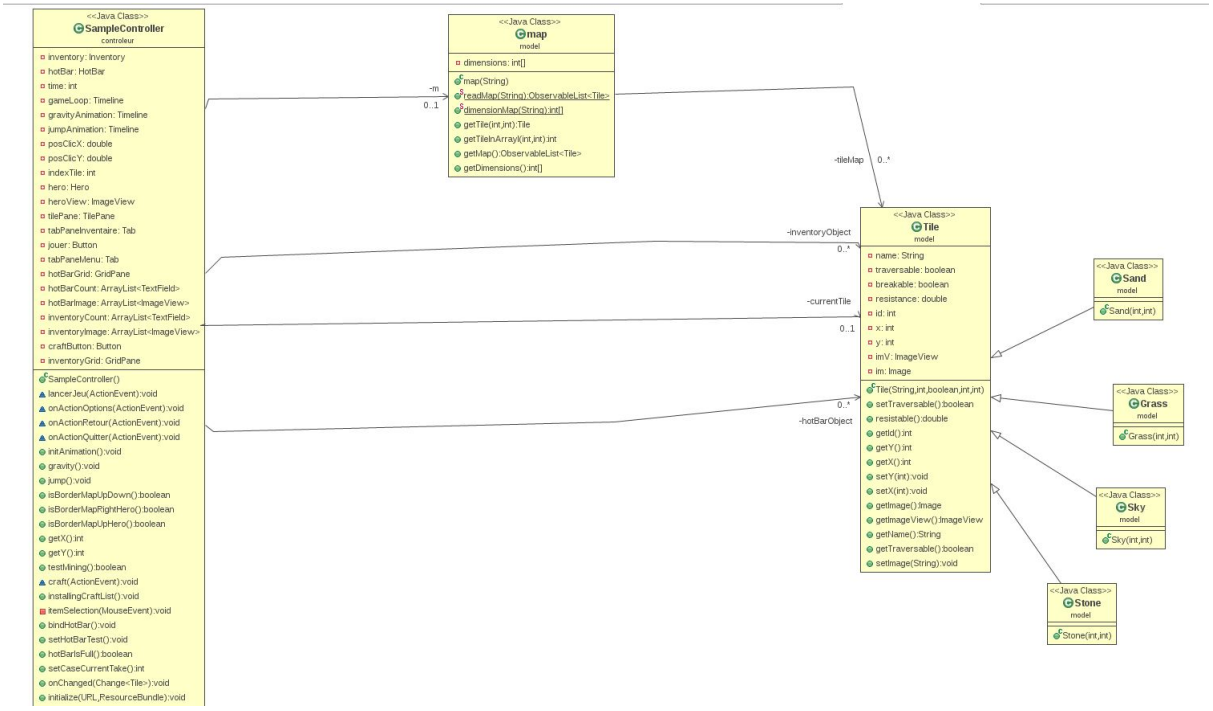


Type de classe

Contrairement aux modèles initiaux, avec des multiples classes abstraites majeures, nous n'avons pas pu implémenter notre conception de modèle, nous nous sommes alors pliés à de simples classes pour la majorité du programme.
Les seules classes abstraites sont les outils et sont très peu utilisées.

Classes

Diagramme des classes majeures

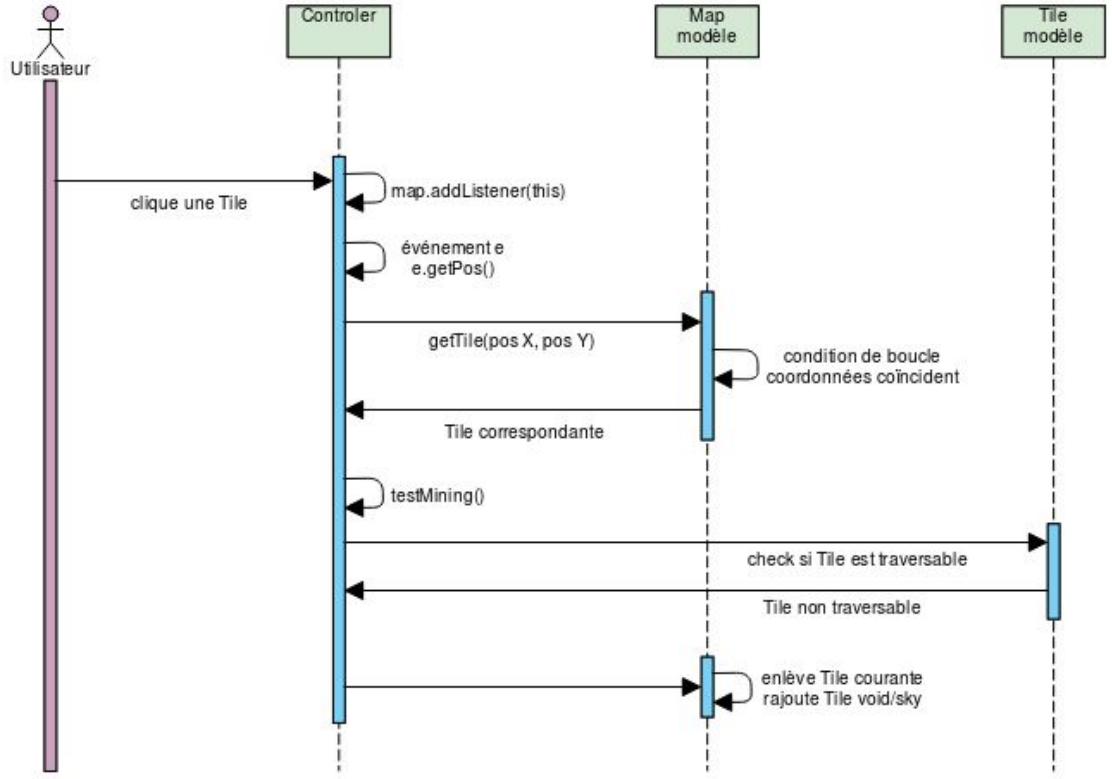


Nous avons décidé de représenter les classes ici présentes sur cet UML car ce sont les classes les plus importantes de notre projet.

La map est le biais entre les deux autres classes, le contrôleur appelle un objet map, qui lui, va aller créer une liste observable à partir de nombreuses tuiles.

Séquences

Diagramme de séquence 1 - Modification de map après le clique d'un joueur

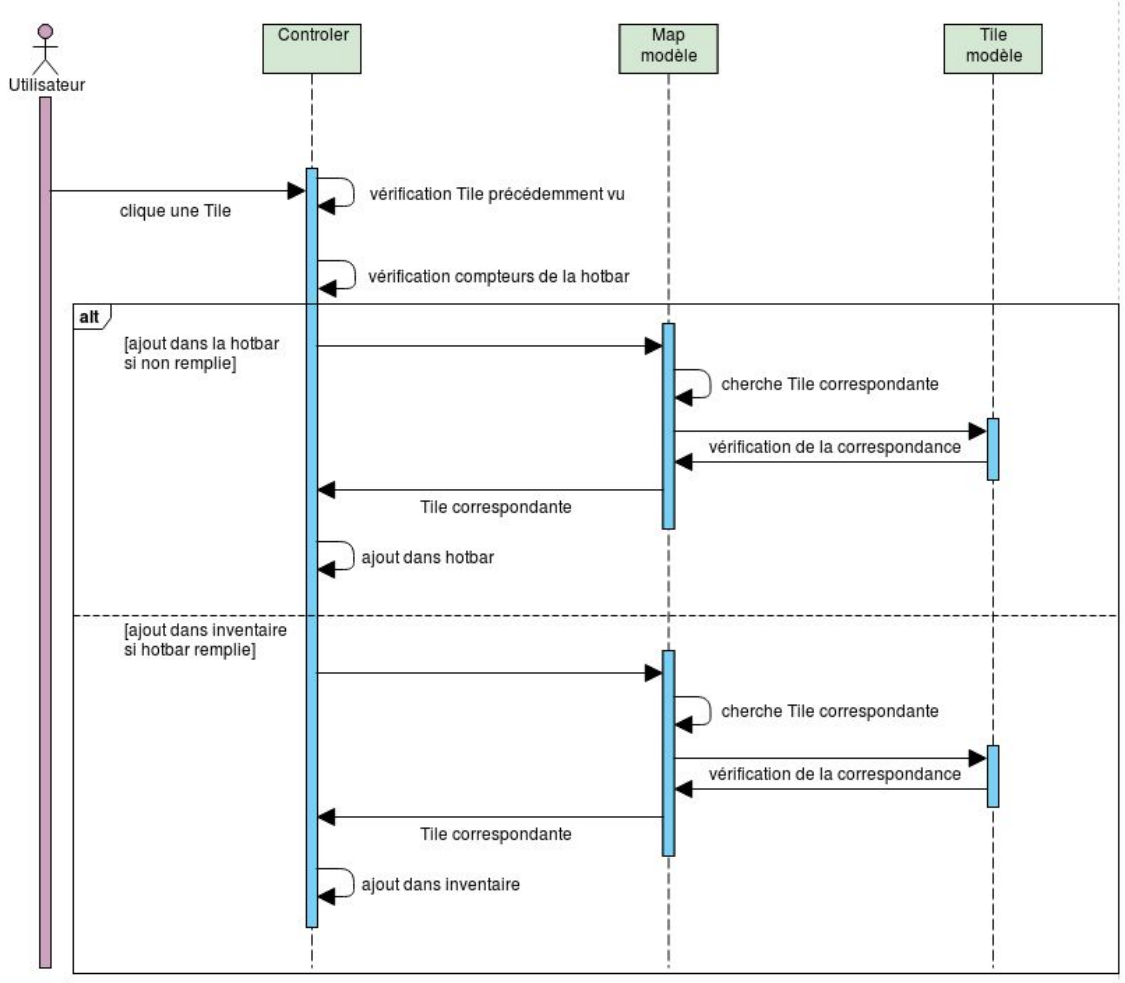


Dans ce diagramme de séquence est représenté l'action du changement de map :
lorsque le joueur clique, le listener ajouté à la map détecte le clic et enclenche un événement.

Le contrôleur récupère les coordonnées du clic afin de chercher la tuile correspondante dans la map.

Si les conditions de testMining() et la possibilité de franchir la tuile sont remplis, nous récupérons cette dernière afin de l'enlever de la map et qu'elle soit remplacée par un bloc de vide/ciel. L'imageView du tuile est alors modifiée par la suite dans le contrôleur.

Diagramme de séquence 2 - Récupération d'une tuile dans la hotbar et dans l'inventaire



Dans ce diagramme de séquence dans la continuation du précédent, nous récupérons la tuile cassée. Une fois la tuile récupérée et la map modifiée, nous vérifions l'état de la hotbar.

Si la hotbar n'est pas remplie, nous allons chercher la tuile et la retransmettre dans cette hotbar tout en prenant compte de l'image de la tuile ainsi qu'un compteur.

Si la hotbar est remplie, nous allons chercher la tuile et la transférer dans l'inventaire, encore une fois, en prenant compte de son image et de son compteur.

Diverses structures

Structures de données

private ArrayList<TextField> hotBarCount;

La hotBar possède une ArrayList de TextField qui va nous servir afin d'afficher le nombre d'items que l'on a dans chaque case.

private ArrayList<ImageView> hotBarImage;

Nous avons une ArrayList d'ImageView pour pouvoir afficher les items.

private Tile hotBarObject[];

Un tableau de tuiles pour les tester entre elles, afin de vérifier si l'on doit placer les items dans la hotBar ou dans l'inventaire si celle-ci est pleine...

private ArrayList<TextField> hotBarCountI;
private ArrayList<ImageView> hotBarImageI;
private Tile hotBarObjectI[];

Nous allons retrouver exactement la même chose qu'au-dessus, c'est juste pour afficher la hotBar aussi dans l'inventaire en parallèle.

private ArrayList<TextField> inventoryCount;
private ArrayList<ImageView> inventoryImage;
private Tile inventoryObject[];

Même principe que pour les cas précédents, l'inventaire sera utilisé seulement si la hotBar est pleine.

private ArrayList<ImageView> inventoryImageItem;
private Item inventoryItem[];

Nous retrouvons en dernier une ArrayList d'imageView et un tableau d'items pour pouvoir les utiliser lors du craft, nous avons alors une unicité des items et n'avons pas besoin de TextFields pour afficher le compte d'items.

Gestion des exceptions

Nous avons une seule exception primordiale, dans notre programme. Elle se situe au niveau du chargement du fichier.csv contenant la map. Si le chargement ne fonctionne pas alors le programme ne se lance pas.

```
try {
        m = new map("src/controler/map2.csv");
        m.getMap().addListener(this);
 } catch (IOException e1) {
        e1.printStackTrace();
 }
```

Test JUnit

Test JUnit pour getTile()

```
@Test
public final void test2() {
    /*creation d'une map moitié stone moitié sky*/
    tileMap =  FXCollections.observableArrayList();
    for (int i = 0; i < 51; i++) {
        for(int j =0; j<=51 ;j ++) {

            if (i<25) {
                tileMap.add(new Sky(i * 32, j * 32));
            }
            else {
                tileMap.add(new Stone (i*32 , j*32));
            }


        }
    }

    assertEquals("verif premier bloc de Sky ", 1, getTile(0,0).getId());
    assertEquals("verif milieu bloc de Sky ", 1, getTile(19*32,20*32).getId());
    assertEquals("verif dernier bloc de Sky ", 1, getTile(24*32,50*32).getId());
    assertEquals("verif premier bloc de Stone ", 5, getTile(25*32,0).getId());
    assertEquals("verif milieu bloc de Stone ", 5, getTile(35*32,20*32).getId());
    assertEquals("verif dernier bloc de Stone " , 5, getTile(50*32,50*32).getId());

}
```

Nous avons voulu faire un test JUnit sur la méthode getTile() car elle est primordiale pour une majorité des autres méthodes, on la retrouve pour le changement de map.