Tales of Liwa – RPGASTRAL

Dole Claire
Lien vers le dépôt github : https://github.com/ClaireDole/Tales-of-Liwa---Projet-L2

I. Introduction

Ce projet a lieu dans un contexte d'amélioration de compétences et de connaissances en Java. Il a pour but principal de développer un moteur de jeu 2D.
Pour cela on s'appuie sur la librairie LibGdx.
Ce jeu est basé sur un principe de rpg comme les premiers Zelda.
Le second objectif est de permettre la modification des terrains, des ennemis, des PNJ, des collectibles ainsi que des points de téléportation entre différentes cartes directement sur l'éditeur de carte Tiled.
Ainsi, il est possible de modifier :

- joueur : sa position de départ sur la worldmap
- ennemis : leur position, leur type, le nombre, l'obligation de tuer l'ennemi pour mener à la victoire du joueur
- PNJ : leur position, leur type, le nombre, le message qu'ils affichent quand on veut intéragir avec eux
- collectibles : ils se déclinent en trois classes : les armes, les tenues et les potions. Pour tous, on peut modifier la position et le nombre
- téléportation : la position, le nombre, la carte vers laquelle le TP permet d'aller
- terrain : modification totale possible des différents terrains, de leur représentation visuel, s'il représente un obstacle ou non
- ajout de carte

Précision : tous les sons et images utilisées dans ce projet sont libres de droit ou autorisent l'utilisation à but non commerciale.

II. Présentation du projet

Technologie et outils utilisés

- LibGDX pour le développement du moteur de jeu
- Tiled : pour la création et la configuration des cartes
- IDE : Eclipse
- Gradle : pour la gestion des dépendances, la compilation et la création d'un .jar de notre librairie
- Github / Fork : pour la gestion de l'avancée du développement et des différentes versions
- Audacity : pour la modification des sons lorsque cela a été nécessaire

Fonctionnalités implémentées

Documentation – De la documentation javadoc a été générée pour ce projet. Elle est stockée dans le dossier core\\doc du projet.

PNJ - Les pnj peuvent parler au joueur lorsque celui-ci appuie sur R. S'il s'agit d'un PNJ de type « Elfe » alors le joueur peut se faire soigner en appuyant sur la touche S à côté du PNJ. Il récupère alors toutes ses stats de base.

Potions – Les potions sont récupérées automatiquement lorsque le joueur se trouve sur la même case. Il est impossible de savoir à l'avance si la potion est un bonus ou un malus et sur quelle statistique elle va agir. Il s'agit d'une prise de position au niveau du game design. Une fois récupérées, les potions disparaissent de la carte.

Tenue – Les tenues sont récupérées lorsque le joueur se trouve sur la même case. Si le slot tenue du joueur est vide alors la tenue est récupérée automatiquement. Sinon, un écran s'ouvre, proposant un choix au joueur entre les deux tenues disponibles. Celle que le joueur ne prend pas sera laissée à la position sur la carte.
Il y a 6 tenues différentes, chacune ayant ses propres fonctionnalités. On retrouve l'augmentation des PV, du Mana, un bonus d'attaque, une annulation des effets de terrains, la possibilité de marcher sur les cases terrain eau et pouvoir convaincre les ennemis humain de fuir à tous les coups (voir les fonctionnalités des ennemis).

Armes – Les armes sont récupérées quand le joueur est sur la même case. Si le slot main droite est libre et que l'arme n'utilise qu'un seul slot (une main) alors arme récupérée. Si le slot main droite est pris mais que le slot main gauche est libre et l'arme n'utilise qu'un seul slot alors l'arme est prise. Si les deux slots sont libres et que l'arme utilise deux slots (arme à deux mains) alors l'arme prise. Dans les autres configurations, on propose au joueur de choisir ce qu'il préfère. Pour l'aider, une description ainsi que la puissance des armes concernées sont affichées. Les armes non prises sont remises sur la carte à l'endroit du choix.
Les armes ont des portées et des dégâts différents. Le sceptre consomme du mana pour attaquer.

Ennemis – Selon le type d'ennemis les fonctionnalités ne sont pas exactement les mêmes. Pour rester général, certains attaquent le joueur uniquement si celui-ci les a attaqués et d'autres l'attaque de façon automatique dès que le joueur est à portée. Dans ce dernier cas, un cooldown de l'attaque du monstre a lieu pour permettre au joueur d'effectuer des actions.
Certains ennemis ont un cycle de mouvement. Ils se déplacent selon un motif en même temps que le joueur.
Un type d'ennemi (volant) est résistant aux différentes armes mais peut être tué en un coup avec l'arc.
Si les ennemis correspondent à des humains (on parle ici des mousses, brigands, voleurs et chef) alors il est possible d'essayer de les convaincre de fuir. Pour cela, il faut être à la limite de la portée de l'ennemi en question et appuyer sur la touche Z. Chaque type d'ennemi à un taux différent et la fonctionnalité utilise l'aléatoire pour décider de la réussite ou non. En cas de réussite, l'ennemi disparaît. Sinon, il reste. Cette fonctionnalité a une chance de marcher une seule fois.

Terrain – Le terrain roche volcanique effectue des dégâts au joueur à chaque case mais ne peux pas le tuer.

Joueur – Il peut se déplacer grâce aux flèches, attaquer grâce à A et E, convaincre un ennemi avec Z, discuter avec un PNJ avec R, se soigner grâce à un elfe avec S. Il peut aussi lancer des menus : M permet d'ouvrir un menu d'aide avec les différentes fonctionnalités et X un menu de statistiques et d'inventaires.

Condition de victoire – Tous les ennemis obligatoires ont été tués ou ont fui. Cela comprend tous les ennemis humains sauf les voleurs. Tous les orcs et les volants sont obligatoire. Le troupeau de slime au sud est du village est obligatoire. Le slime qui s'est infiltré dans la maison est obligatoire.

Condition de défaite – La mort du joueur.

Sons – Des effets sonores ainsi qu'une musique de fond ont été implémentés.

Quitter – la possibilité de quitter le jeu à n'importe quel moment en appuyant sur la touche Q a été implémentée. Bien que la fonctionnalité marche, un bug persistant est reporté. C'est ce bug qui induit à la fermeture du jeu. Il s'agit soit d'une BufferError soit de la JVM qui a une RunTimeException

Design Pattern – Le pattern observer est utilisé pour gérer les différents évènement du jeu. Le pattern MVC a été utilisé avec les packages (séparation en model, view et controler) et en s'assurant que le code de chaque classe respecte les règles du MVC.

Configuration et ajout avec l'éditeur de carte Tiled

Organisation du dossier asset.

Les cartes et les fichiers de tuiles sont dans le sous-dossier carte. Les images utilisées par les fichiers de tuiles sont dans le sous-dossier terrain du dossier carte.

Gestion des cartes

Chaque carte doit avoir une propriété de type String « name ». Chaque nom doit être unique pour différencier les cartes dans le code.

Les cartes sont chargées grâce au loader de fichier tmx fourni par LibGdx.

Attention : pour charger les cartes, on scan le dossier asset\\carte à la recherche de tous les fichiers .tmx à charger. Il faut donc que le path corresponde bien au path que l'ordinateur ! (classe RpgMain ligne 91)

Ainsi de nouvelles cartes peuvent être ajoutées dans le dossier carte et traitées.

Les cartes sont interprétées par une classe nommée TiledParser. La classe gère la récupération des données à partir de la carte. Elle récupère la taille de la carte, la taille des tuiles, les différents terrains et objets ainsi que leurs propriétés.

Ces informations sont stockées dans un TiledModel. La classe TiledModel décrit les différentes données récupérées afin qu'elles puissent être utilisées dans le code. Elles sont majoritairement dans des array liste pour faciliter leur manipulation.

Le rendu des cartes s'effectue lors de la création de chaque gamescreen. On utilise un renderer spécifique fourni par LibGdx.

Les terrains

On utilise des tilelayer pour définir les terrains. Chaque tilelayer doit avoir comme propriété « franchir » qui est un booléen. Si la case est cochée alors le joueur peut marcher sur les cases correspondant au layer. Sinon, ces cases sont définies comme des obstacles.

Les tilelayer « Eau » et « Volcanique » doivent respecter cette écriture pour leur nom. Cela permet de créer les cases d'eau qui ont un traitement à part de même que pour les cases volcaniques.

Le reste de l'organisation est à l'appréciation du créateur de carte.

Les objets

On utilise des objectlayer. Un objectlayer correspond à une classe d'élément du jeu. Les noms de ces objectlayer doivent être comme définies dans les cartes fournies.

Chaque element a les propriétés de type int « X » et « Y » correspondant à leur position. Il peut être nécessaire de faire des ajustements entre les position proposées dans Tiled et le

rendu dans le jeu. Par exemple sur la carte worldmap, le « X » correspond bien à l'abscisse proposée lors du survol de la carte. Mais pour le « Y » il faut faire 50-y+1 car l'origine du repère de la carte est en haut à droite alors que dans le jeu l'origine est en bas à droite.

PNJ

Sur l'objectlayer NPC. Le nom de chaque objet doit être « Humain » ou « Elfe » en fonction du type de PNJ souhaité. Si le PNJ a un message de dialogue alors on créé la propriété de type String « msg ».

Tenues

Sur l'objectlayer Tenues. Les noms possibles sont « Apparat du kistune », « Bénédiction de Susanoo », « Eclat de Tsukuyomi », « Broche d'Izanami », « Armure de Bishamonten » et « Protection d'Amaterasu ».

Armes

Sur l'objectlayer Armes. Les noms possibles sont « Arc », « Hâche », « Sceptre », « Bâton » et « Epée ».

Potions

Sur l'objectlayer Potions. Les noms possibles sont « Bonus » et « Malus ». Chaque objet possède deux propriétés supplémentaires, « Quantité » de type float et « Type » de type String.

« Quantité » correspond au nombre de points en bonus ou en malus.

« Type » peut prendre trois valeurs : « Mana », « PV » ou « Attaque ». Cela correspond à la statistique influencée par l'objet.

Ennemis

Ils se divisent en deux catégories les monstres (layer Monstres) et les humains (layer EnemyHumans).

Chaque objet doit avoir la propriété de type booléen « obligatoire ». Si la case est cochée alors l'ennemi est à battre pour gagner.

Les noms pour les monstres sont « Volant », « Orc », « Gobelin » et « Slime ».

Les noms pour les humains sont « Chef », « Brigand », « Voleur » et « Mousse ».

Teleportation

Sur l'objectlayer Teleport. Ils permettent la liaison entre différentes cartes ainsi que de donnée la position du player lors de l'arrivée sur la carte.

Le nom d'un objet correspond au nom de la carte d'arrivée sauf pour un unique objet qui correspond au point de spawn du joueur au début du jeu. Attention, ce point de spawn ce fait forcément sur la carte principale worldmap. Autrement cela n'aurait pas de sens du point de vue du game design.

Ils possèdent des propriétés de type int « destX » et « destY » correspondant aux coordonnées du point d'arrivée.

Compilation et exécution

- **Prérequis** : Java version 17

  Dans la classe RpgMain à la ligne 91, le chemin doit correspondre au chemin du dossier asset\\carte sur l'ordinateur.

  Dans la classe Lwjgl3Laucher la taille de la fenêtre doit-être 800*800.
- **Compilation :** Sur Eclipse. Créer une nouvelle configuration. Choisir Gradle Task. Ajouter la tâche build. Mettre ${workspace_loc:/RPGASTRAL-parent} comme working directory.
- **Exécution :** Sur Eclipse. Créer une nouvelle configuration. Choisir Java Application. Mettre RPGASTRAL-lwjgl3 comme Projet. Choisir fr.rpgastral.lwjgl3.Lwjgl3Launcher comme Main class.

Dans l’onglet Arguments.
Mettre ${workspace_loc:RPGASTRAL-lwjgl3/assets} comme working directory.

Lors de la tâche de compilation, gradle créer des fichiers .class de tous les fichiers sources et à la fin il donne un fichier .jar qui correspond à notre projet sous forme de librairie.
Ce fichier .jar est mis dans les librairies du laucher de LibGdx. Ainsi, lorsque l’on exécute la configuration d’éxécution avec le laucher de LibGdx, notre projet est pris en compte comme une librairie par ce fichier .jar

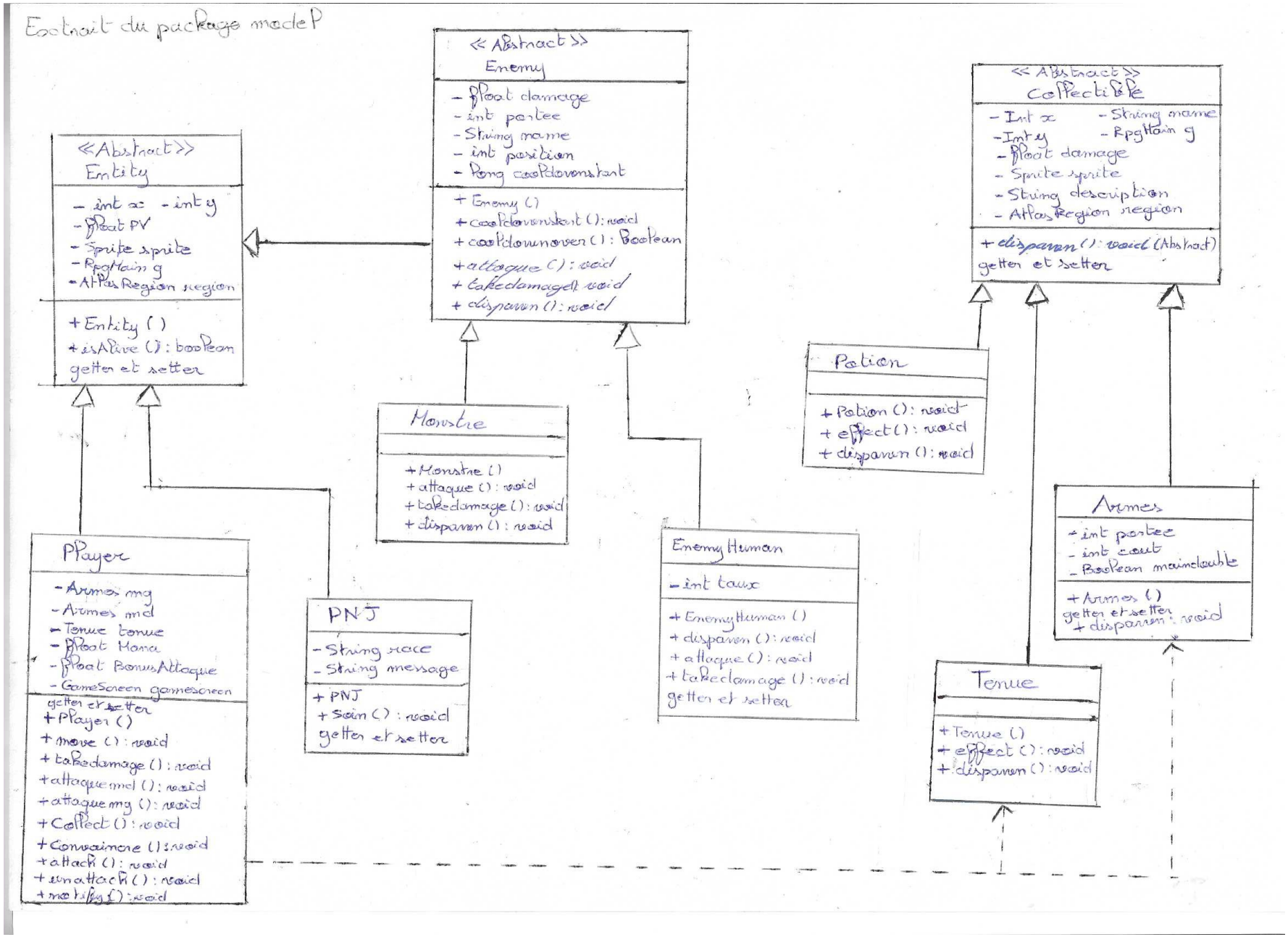
III. Présentation technique du projet

Utiliser et étendre la librairie du moteur de jeu

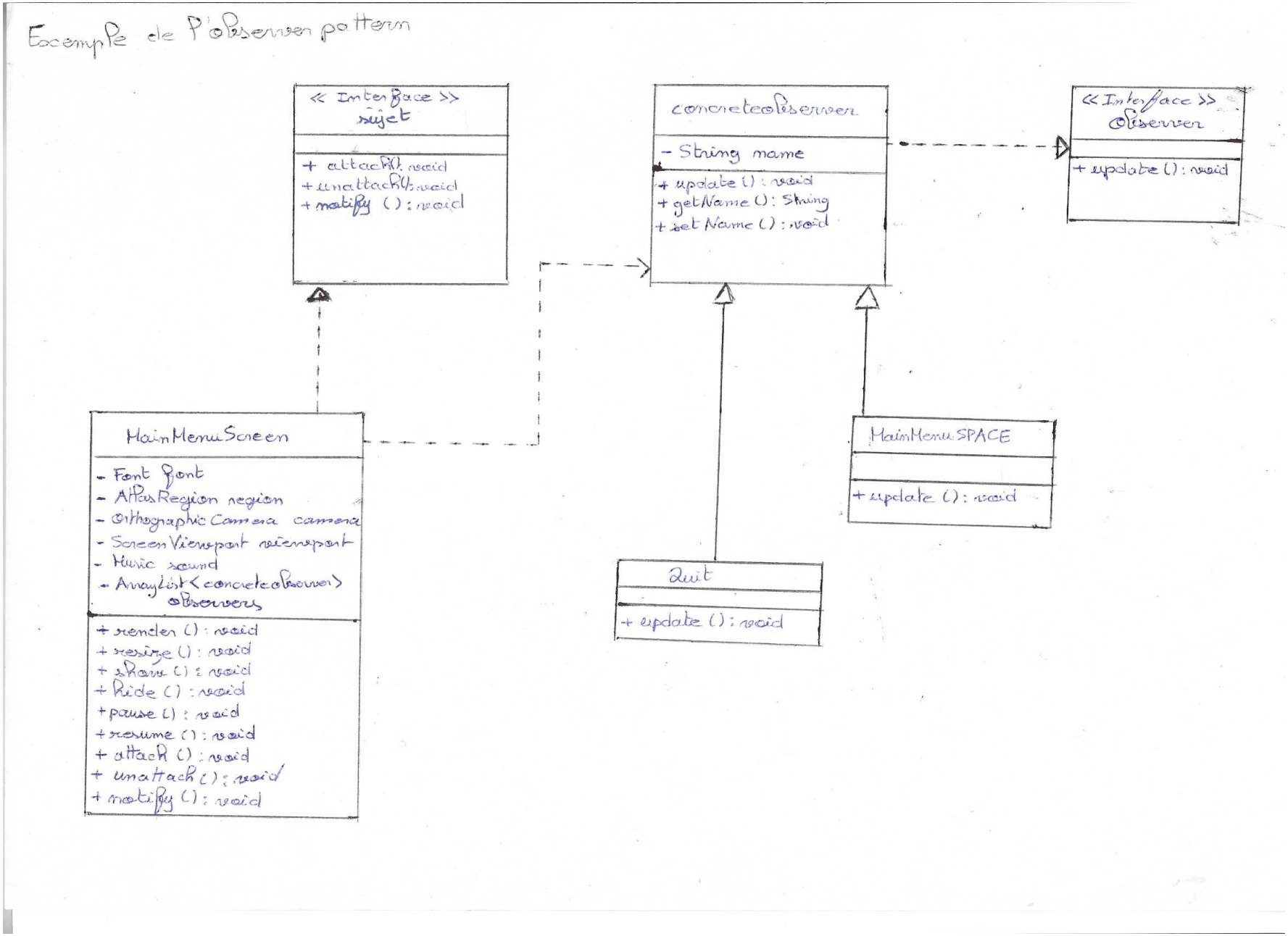
Pour ajouter un nouveau personnage on hérite de la classe Entity ou de la classe PNJ en fonction des besoins.
Pour ajouter un nouvel objet on hérite de la classe collectible. Pour ajouter un nouveau type d’armes ou de tenue on modifie le constructeur de la classe en question (Tenue ou Arme) pour ajouter un nouvel objet possible.
Pour ajouter un nouveau type de terrain on modifie le constructeur de la classe Terrain.
Pour ajouter un ennemis on hérite de la classe Enemy ou on modifie le constructeur de la classe EnemyHuman ou de la classe Monstre dépendant des besoins.
L’ajout d’une nouvelle classe par exemple bâtiment qui n’est ni un ennmis, ni une entité ni un collectible s’ajoute dans le package model.
Il faudra modifier le TiledModel pour ajouter les nouvelles listes correspondantes aux nouvelles classes.
L’ajout de nouvelles classes implique la modification de la classe GameScreen permettant l’affichage des objets, de la classe TiledParser permettant la récupération des nouvelles informations des cartes tiled avec l’utilisation des nouvelles classes. Des concreteobservers peuvent nécessiter une modification selon les fonctions à implémenter de ces nouvelles classes.

Architecture générale du moteur de jeu

La classe principale est la classe RpgMain qui hérite de la classe Game. Elle gère la création et le chargement du jeu. Elle se sert des classes loader, TexturePackerHelper et TiledParser. Elle s’occupe de charger les assets et les cartes Tiled. Elle interprète aussi ces dernières.
Dans le package model on retrouve toutes les classe décrivant un élément du jeu : les tenues, les potions, le joueur, les ennemis, etc.
On passe très souvent une instance de RpgMain dans les différentes classes afin d’avoir accès à des éléments comme la liste des tiledmodels ou des gamescreen ou encore le SpriteBatch du jeu.
Dans le package view on retrouve des classes qui implémentent l’interface Screen ainsi que l’interface sujet.
Il y a un observer de mise en place. L’interface sujet permet d’indiquer que l’objet est observable. L’interface Observer décrit les méthodes obligatoires d’un observer. La classe concreteobserver implémente observer et permet de donner un nom distinct à chaque observer. Les différentes classes dans controler.observerpattern.concreteobserver héritent toute de la classe concreteobserver. La classe Event permet de décrire les évènements qui sont transmis aux concreteobservers par les classes observables. Ainsi chaque Screen envoie des évènements à certains concreteobservers qui agissent en fonction. Afin de simplifier le code, la classe Player implémente l’interface sujet. Cela permet de gérer plus facilement certaines fonctionnalités.
Ci-dessous des schémas UML de certaines parties du projet.


Extrait du package modeP
≪Abstract≫
Enemy
- float damage
- int portee
- String name
- int position
- Long coolDownInstant
+ Enemy ()
+ coolDownInstant (): void
+ coolDownOver (): Boolean
+ attaque (): void
+ takeDamage (): void
+ disparen (): void
≪Abstract≫
Entity
- int x
- int y
- float PV
- Sprite sprite
- RpgMain g
- AtlasRegion region
+ Entity ()
+ isAlive (): boolean
getter et setter
≪Abstract≫
Collectible
- Int x
- String name
- Int y
- RpgMain g
- float damage
- Sprite sprite
- String description
- AtlasRegion region
+ disparen (): void (Abstract)
getter et setter
Potion
+ Potion (): void
+ effect (): void
+ disparen (): void
Monstre
+ Monstre ()
+ attaque (): void
+ takeDamage (): void
+ disparen (): void
Armes
- int portee
- int cout
- Boolean maineloable
+ Armes ()
getter et setter
+ disparen (): void
PNJ
- String race
- String message
+ PNJ
+ soin (): void
getter et setter
EnemyHuman
- int taux
+ EnemyHuman ()
+ disparen (): void
+ attaque (): void
+ takeDamage (): void
getter et setter
Tenue
+ Tenue ()
+ effect (): void
+ disparen (): void
PPlayer
- Armes mg
- Armes md
- Tenue tenue
- float Mana
- float BonusAttaque
- GameScreen gamescreen
getter et setter
+ Player ()
+ move (): void
+ takeDamage (): void
+ attaquemd (): void
+ attaquemg (): void
+ Collect (): void
+ Convaincre (): void
+ attach (): void
+ unattach (): void
+ mettre (): void

Exemple de l'observer pattern



IV. Conclusion et Perspectives

Les objectifs principaux du projets ont été atteints.
Les défis notables rencontrés ont été la prise en main de LibGdx avec la gestion du rendu (camera, viewport, batch, spritebatch, texture, etc), la mise en place d'une organisation MVC, l'interprétation des cartes Tiled et le scan des fichiers pour la gestion de plusieurs cartes Tiled.
On pourrait améliorer la factorisation du code ainsi que son organisation générale et sa clarté. L'organisation MVC n'est pas parfaitement exécutée. De plus, la documentation pourrait être complétée. On pourrait ajouter un objet à lire dans la maison permettant d'afficher les missions du joueur et les principales caractéristiques des ennemis.
Un design pattern singleton pourrait être implémenté pour la classe RpgMain permettant d'éviter de toujours passer en argument l'objet RpgMain créé au début de l'exécution.
Du point de vu du game design on pourrait implémenter des donjons permettant de récupérer des armes puissantes et des tenues pour éviter le simple parcours de la carte. Des missions annexes pourraient aussi être proposées au joueur tel que récupérer un objet et le ramener à son propriétaire ou sauver un PNJ.