Rapport de projet – « Worms »

Dépôt

1. Introduction

Le projet Worms consiste en la réalisation d'un jeu inspiré de Worms, développé en Java. L'objectif principal est de créer une application fonctionnelle offrant deux interfaces distinctes

- Terminal (CLI)
- Graphique (GUI)

Une attention particulière a été portée à la structuration du code, à la maintenance et à la mise en place d'un processus de déploiement automatisé (CI/CD).

Le présent rapport détaille les fonctionnalités implémentées, les problèmes rencontrés, les corrections apportées et les pistes d'évolution possibles. Dans la section suivante, un schéma général de l'architecture complète du système est présenté dans la figure 1.Représentation graphique du modèle des classes pour le CORE :

Représentation graphique du modèle des classes pour le GUI est dans image 1 dans la section appendix et représentation graphique du modèle des classes pour le CLI est dans image 2 dans la section appendix

---

2. Gestion du terrain et du jeu

2.1 Création du terrain

- **Ticket GitLab** :
  (#10) Création de la classe PlayGround

  (#63) Génération des terrains

- **Merge Request associée** :
  (!5) MR pour ajouter les fichiers de terrain de jeu

  (!81) Générations de terrains

La classe initiale *PlayGround* (renommée **Terrain**) a été créée pour représenter la grille du terrain et ses dimensions.
Elle inclut des méthodes permettant de :

- Détecter le sol, l'eau et l'espace libre
- Détruire des cellules
- Afficher le terrain textuellement

Cette base a permis d'assurer le fonctionnement initial du jeu et de servir de fondation pour l'architecture MVC, l'affichage graphique ainsi que la gestion du terrain grâce au moteur physique, importé par la bibliothèque JBox2D.

2.2 Mise en place du modèle MVC

- **Ticket GitLab** :
  (#13) Implémentation du MVC de base pour le terrain Worms

- **MR associée** :
  (!13) Ajout de la classe TerrainView pour démarrer la structure MVC

Le terrain a été restructuré selon le modèle **Modèle–Vue–Contrôleur** :

- *TerrainModel* : le modèle représentant le plateau de jeu
- *TerrainView* : l'affichage graphique
- *TerrainController* : gestion des interactions utilisateur (ex. destruction de cellule)

  Les classes TerrainModel et TerrainCore ont été remplacées par la classe TerrainCore par la suite.

Des listeners assurent la mise à jour automatique de la vue lors des changements du modèle, garantissant une synchronisation complète.

2.3 Affichage des Worms

- **Ticket GitLab** :
  (#16) Ajout de l'affichage d'un Worm sur le terrain graphique

- **MR associée** :
  (!22) Ajout de l'affichage d'un Worm sur le terrain graphique

Cette étape permet de visualiser les Worms sur le terrain graphique, avec positionnement aléatoire et ajustement pour qu'ils reposent correctement sur le sol.

2.4 Affichage de l'inventaire

- Ticket GitLab : (#85) : Affichage d'image de l'inventaire
  (#56) : Affichage de l'inventaire

- MR associé : (!119) : Remplacement des texte de l'inventaire par des images
  (!63) : Ajout de l'affichage de l'inventaire (écrit)

L'inventaire s'affichait en continue et lorsque l'on sélectionne, il fallait viser pour tirer après.

2.5 Gestion du tour de jeu

- **Ticket GitLab** :
  (#20) Implémentation de la méthode round

- **MR associée** :
  (!33) MR pour completer la methode de round

  (!87) MR pour la résolution du saut de tour

La méthode *round* organise l'enchaînement des actions des Worms, garantissant le déroulement correct de la partie. Le développement a été réalisé de manière collaborative. Lors de l'implémentation on a remarqué un bug de saut de tour inventaire.

2.6 Gestion de la gravité, des mouvements et des collisions des worms

- **Ticket GitLab** :
  (#24) Gravité

  (#60) Physique

- **MR associée** :
  (!111) Intégration de la bibliothèque JBox2D

  (!115) Destructibilité du terrain

Le terrain est géré par JBox2D comme un corps statique ("World") dans lequel les autres éléments du jeu peuvent interagir en tant que corps dynamique ("Body"). Afin d'utiliser cette bibliothèque, il faut s'assurer que le terrain est bien relié à un objet World, et les corps à des objets Body importés de la bibliothèque. Les déplacements sont également gérés par la bibliothèque (en lui indiquant le comportement à adopter à l'aide de fonctions), l'état du World est mis à jour si nécessaire et les Worms peuvent se pousser et se déplacer même lorsque le terrain n'est pas plat.

3. Gestion de la partie

3.1 Organisation et logique de jeu

- **Ticket GitLab** :
  (#17) Création de Live Game

(#61) Implémentation du tour par tour

(#82) Système de pause

- **MR associées** :

  (!8) Création des classes de gestion de la partie

  (!68) Ajout du système de Tour par Tour

  (!123) Passage de tour

Le jeu initialise des parties Game selon les paramètres choisis par l'utilisateur. Cependant, pour que le déroulement se fasse sans accroc, une classe LiveGame a été créée pour retenir l'état courant des objets instanciés (terrains, worms, inventaire, …). LiveGame traite les actions d'un seul vers à la fois. Le tour peut être passé à l'aide d'un bouton visible sur l'interface graphique, ou grâce à un message de l'utilisateur dans la version terminal.

3.2 Déroulement d'un Round et Combat

- **Ticket GitLab** :
  (#78) Ajout des munitions

  (#87) Ajout du recul

  (#88) Ajout du vent

- **MR associées** :

  (!82) Mort au contact de l'eau

  (!79) Ajout du tir graphique

  (!103) Ajout des munitions et explosions

  (!116) Ajout du recul

  (!118) Ajout du vent

Le combat est divisé en Rounds de 15 secondes, au cours desquels il n'y a qu'un seul worm en activité. Celui-ci va pouvoir se déplacer, puis une fois une arme sélectionnée, enclenchera la phase de "Shoot" (soit de tir pour les armes Ranged, soit d'attaques pour les "Hand to Hand"), au cours de laquelle le joueur peut sélectionner la trajectoire de l'arme utilisée pour attaquer. Les armes "Ranged" ont un nombre de munitions limitées propres à chaque joueur et certaines ont la capacité de détruire le terrain avec une portée déterminée. Certaines armes appliquent un recul sur la cible, comme la 'Baseball bat'. Le vent a également une incidence réaliste sur la trajectoire du projectile tiré.

3.3 Affichages des informations de combat

- **Ticket GitLab** :
  (#65) Ajout des barres de vie et de temps

- **MR associée** :
  (!73) Ajout des barres de vie et de temps

  (!63) Ajout de l'inventaire au panel visuel

Sur l'écran de jeu figurent les informations suivantes : les barres de vie respectives des worms, l'inventaire des joueurs, le bouton pause, le bouton skip, le vent, le temps restant pour un round et le joueur actif, la trajectoire de l'arme utilisée, et pour les armes 'Ranged', le projectile est affiché tout au long de son tir.

4. Écran d'accueil

4.1 Lancement d'une partie (graphique) à l'aide d'un 'homescreen'

- **Ticket GitLab** :
  (#50) Implémentation de l'écran d'accueil

- **MR associées** :

  (!78) Première implémentation de l'accueil (Ébauche de l'écran d'accueil, pas encore fonctionnel)

  (!84) Avancée de l'accueil (Ajout d'image de fond)

  (!110) Accueil finalisé

L'écran d'accueil propose plusieurs options : le lancement d'une nouvelle partie selon les paramètres choisis par l'utilisateur (nombre de Worms, type de terrain, vent, objets de l'inventaire, nom des joueurs), ou bien le chargement de la dernière partie enregistrée dans un fichier texte.

4.2 Reprise d'une partie entamée

- **Ticket GitLab** :
  (#77) Ajout de la sauvegarde

- **MR associées** :
  (!101) Ajout de la sauvegarde (MR non finalisée car elle avait été débutée sur une version trop ancienne de dev)

  (!125) Correction et ajout de la sauvegarde

Il est possible, au début d'un round, de quitter une partie dont l'état sera automatiquement enregistré dans un fichier .txt. Le bouton 'Resume game' de l'accueil permet de charger cette partie et de la reprendre là où elle s'était arrêtée.

---

5. Architecture et maintenance du projet

5.1 Réorganisation de la structure et séparation des versions

- **Ticket GitLab** :
  (#23) Réorganisation de la structure du projet

- **MR associée** :
  (!34) Modification de la structure du projet

Deux répertoires distincts ont été créés pour séparer :

- la version terminale (*terminalversion*)
- la version graphique (*graphicalversion*)

Les packages et imports ont été mis à jour pour refléter cette organisation.

5.2 Réorganisation en modules Gradle et résolution de duplications

- **Ticket GitLab** :
  (#48) Réorganisation du projet en modules (core / cli / gui)

- **MRs associées** :
  (!59) Modularisation du projet au format core, cli et gui,
  (première MR, devenue obsolète en raison de nombreux conflits)

  (!60) Modularisation du projet en core, cli et gui,
  (MR principale)

(!76) séparation du fichier ShootTerminal.java en core et cli
(MR ultérieure pour une classe particulièrement difficile à séparer)

Cette refactorisation complexe a permis mutualiser les classes communes (*core*), et résoudre le problème de duplication de classes (*Shoot* et *Move*) présent dans plusieurs répertoires (cf ficket (#25) Classe en doublons dans chaque répertoire).

Cette tâche a nécessité un **temps considérable**, car chaque fichier déplacé entraînait des modifications en cascade dans d'autres fichiers.

5.3 Nettoyage et ajustements techniques

Quelques modifications mineures mais importantes ont été réalisées :

- **Ticket GitLab** :
  (#44) Réglage du build.gradle.kts

  (#46) Nettoyage du projet : suppression du JAR inutile

  MR associées :

  (!20) Renommage de plusieurs répertoires :
  renommage de répertoires pour respecter les conventions Java
  (!48) modification du fichier build.gradle.kts :
  correction du build.gradle.kts pour le lancement du JAR
  (!51) suppression du dossier app et modification de la structure du projet :
  suppression d'un JAR secondaire inutile

Ces ajustements ont contribué à la propreté, la cohérence et la maintenabilité du projet.

---

6. Intégration, Déploiement et exécutable

6.1 Continuous Integration (CI)

- **Ticket GitLab :**
  (#6) : Organisation des fichiers et pipeline

- **MR associé :**
  (!7) : MR Pour la pipeline

Le projet fonctionne à l'aide de l'outil Java Gradle qui permet de créer des build et gérer notre système de pipeline.

Le fichier build.gradle divise la pipeline en 3 jobs via la commande : "**./gradlew build**" :

- Build : Qui vérifie les erreurs au niveau du code;
- Test : Qui va vérifier si les tests (ajouter dans le dossier test) fonctionne bien;
- Deploy : Qui déploie le jeu.

Ce dernier va permettre également de générer des jar pour le déploiement continue.

6.2 Continuous Deployment (CD)

- **Ticket GitLab** :
  (#39) Mise en place du CD (Continuous Deployment)

  (#72) Finalisation du CD : Génération du JAR graphique

  MR associée :
  (!42) Ajout du CD :
  mise en place du CD pour le JAR terminal
  (!83) Finalisation du CD : Génération du JAR graphique :
  génération automatique du JAR graphique

Le pipeline CI/CD construit le projet avec **Gradle** et produit deux artefacts exécutables :

- *jeuCli.jar* pour la version terminale
- *jeuGui.jar* pour la version graphique

Ces automatisations garantissent la disponibilité permanente de versions fonctionnelles après chaque *push*.

---

7. Fonctionnalités avancées

7.1 Défilement de la carte

- **Ticket GitLab** :
  (#68) Implémentation du défilement de la carte

- **MR associée** :
  (!105) Ajout du défilement de la carte
  implémentation du déplacement de la caméra

Deux classes ont été créées :

- *Camera* : gère la position et les limites de la vue
- *CameraController* : permet de déplacer la caméra via le *drag* de la souris

Le terrain, les Worms et le fond se déplacent ensemble lors du défilement. Un camarade a aidé à résoudre un bug identifié dans la MR lors de l'implémentation.

7.2 Sauvegarde

- **Ticket GitLab :** (#77) : Ajout de la sauvegarde
- **MR associé :** (!125) : Ajout de la sauvegarde
- **Ancienne MR :** (!101) : Draft: Ajout de la sauvegarde

La sauvegarde se divise en 2 classes :

- Save : La classe qui permet de sauvegarder tous les paramètres de la partie dans un fichier texte (.txt);
- Load : La classe qui permet depuis un fichier texte, récupérer les état de la partie.

7.3 Corrections finales

- **Ticket GitLab** :
  (#83) Correction de bugs

  MR associée :
  (!110) bugfix: Jar + package
  correction des packages manquants et modification de la classe principale pour le JAR graphique (*Accueil.java* au lieu de *Start.java*)

- **Ticket GitLab** :
  (#95) Problème de lancement des JAR

  MR associée :
  (!131) bugfix : lancement des JARs
  Résolution d'un bug concernant le lancement des deux JARs qui ne prenaient pas en compte les dépendances.

Ces corrections assurent la stabilité et le fonctionnement correct des exécutables.

---

8. Documentation et Tests

8.1 : Documentation utilisateur

- **Tickets GitLab** :
  (#75) Création du README

- **MR associée(s)** :
  (!86) ajout du fichier README.md :
  création du README

Le fichier README permet de savoir comment compiler le projet, lancer les versions terminale et graphique, et explique les prérequis nécessaires, assurant une prise en main simple pour tout utilisateur.

8.2: Tests

- **Tickets GitLab :**

  (#34) Ajout de test pour le lancement de jeu

  (#80) Ajouter Tests pour la version GUI

- **MR associées**

  (!112) Ajouter les test pour les armes

  (!104) Ajouter les test de mouvement

Dans ce projet, des tests distincts ont été ajoutés au dossier /test pour chacune des classes clés et importantes afin de garantir leur bon fonctionnement. Les tests ont été divisés en trois catégories principales : Core, GUI et CLI. Des tests ont été ajoutés au projet par les membres de l'équipe dans de nombreuses MRs tout au long du projet. Ci-dessus figurent quelques-unes des principales MRs et leurs tickets correspondants.

---

9. Conclusion et perspectives

À ce stade, le projet *Ver de Terre* dispose :

- d'un **terrain fonctionnel** en mode textuel et graphique,
- d'une **architecture MVC** bien structurée,
- d'une **organisation modulable et maintenable**,
- d'un **CI et CD complet** produisant des JAR exécutables,
- et d'une **documentation prête à l'usage**.

Problèmes connus / limitations :

- difficulté à commencer à faire l'intégration continue. Au départ, nous avions essayé d'utiliser Maven comme outil de projet, mais avec de grosses difficultés et une réflexion, nous nous sommes rabattus sur la bibliothèque Gradle pour effectuer le script d'intégration continue.
- Nous avons certes décidé d'utiliser une bibliothèque pour gérer la physique dans notre jeu, mais on a eu du mal à commencer. En effet, au départ, nous avons eu du mal à implémenter la bibliothèque mais une fois cela fait, on a eu encore plus de mal à bien l'exploiter, cela nous a pris quelques jours pour bien comprendre cette bibliothèque.
- certaines optimisations visuelles ou mécaniques restant à réaliser.

Pistes d’extensions :

- ajout d’animations
- ajout du zoom
- création d’une IA (bot)
- ajout de nom aux vers et l'afficher dans le terrain à la place du nom du joueur
- ajout d’un temps au début du round qui permettrait au joueur de choisir quel worm utiliser
- Ajoutez une fenêtre séparée pour gérer l'audio du jeu.

A. Appendix

Image 1 : Représentation graphique de la version CORE

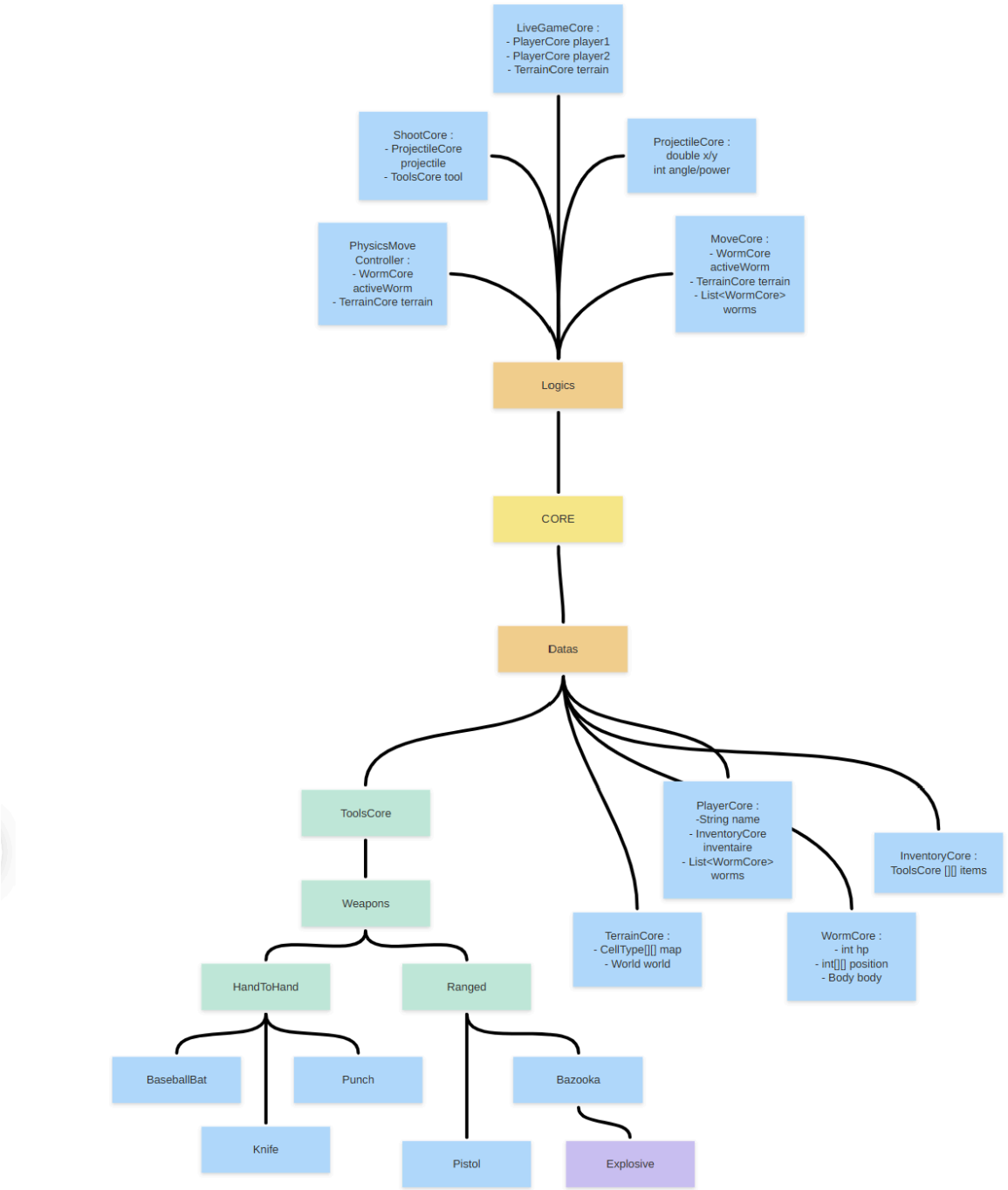


Image 2 : Représentation graphique de la version GUI

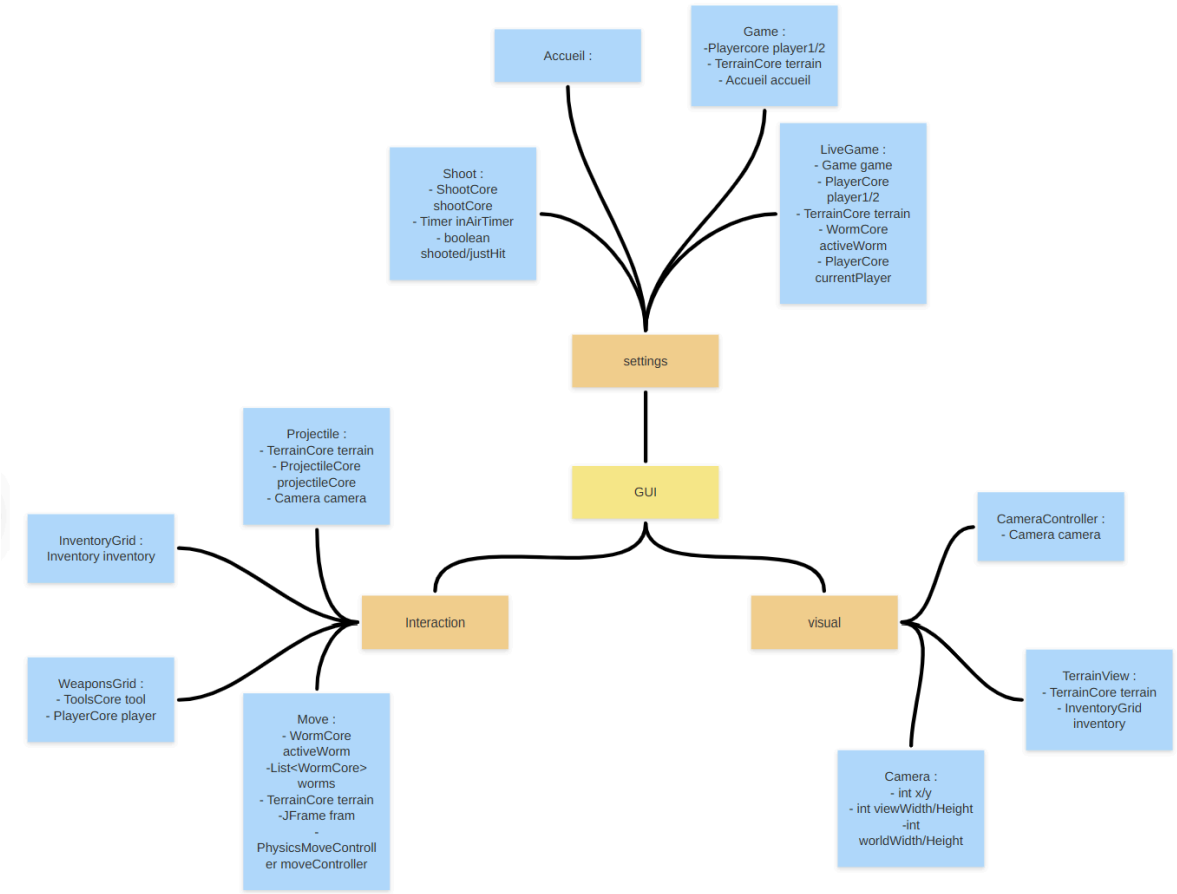
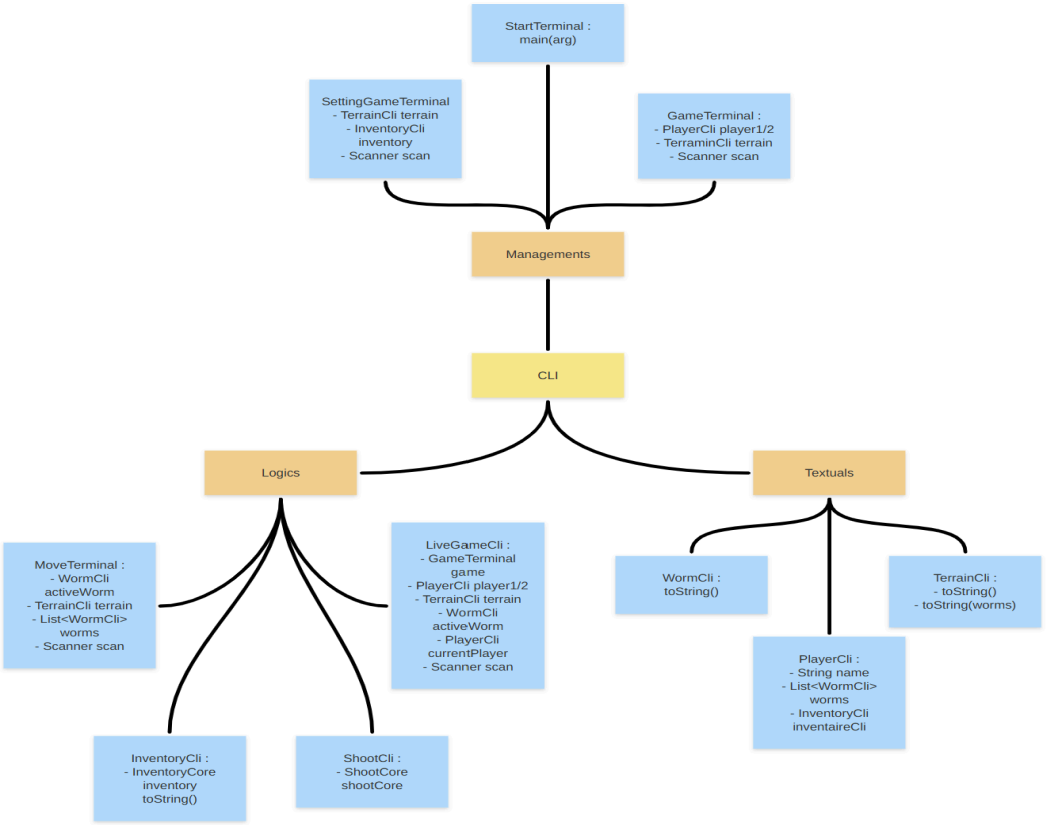


Image 3 : Representation graphique de la version CLI


StartTerminal :
main(arg)
SettingGameTerminal
- TerrainCli terrain
- InventoryCli
inventory
- Scanner scan
GameTerminal :
- PlayerCli player1/2
- TerraminCli terrain
- Scanner scan
Managements
CLI
Logics
Textuals
MoveTerminal :
- WormCli
activeWorm
- TerrainCli terrain
- List<WormCli>
worms
- Scanner scan
LiveGameCli :
- GameTerminal
game
- PlayerCli player1/2
- TerrainCli terrain
- WormCli
activeWorm
- PlayerCli
currentPlayer
- Scanner scan
WormCli :
toString()
TerrainCli :
- toString()
- toString(worms)
PlayerCli :
- String name
- List<WormCli>
worms
- InventoryCli
inventaireCli
InventoryCli :
- InventoryCore
inventory
toString()
ShootCli :
- ShootCore
shootCore