Université de Poitiers
UFR Sciences Fondamentales et Appliquées - Licence 3 Informatique
UE : Programmation Orientée Objet & IHM

RAPPORT DE PROJET - JEU D'AVENTURE TEXTUEL

"DUNGEON ADVENTURE"

Année universitaire 2025 - 2026

Réalisé par :
- Adam SIBÉ
- Léonard DUHEM
- Lucas VADE-GRELET

Date de remise :
05 décembre 2025

Sommaire

Résumé

Ce projet, réalisé dans le cadre de l'UE Programmation Orientée Objet et IHM en Licence 3 informatique, consiste à concevoir et développer Dungeon Adventure, un jeu d'aventure textuel inspiré des classiques du genre. Le joueur incarne un aventurier piégé dans un donjon labyrinthique et doit progresser de salle en salle afin de trouver des objets, affronter des ennemis et atteindre le boss final, seule issue pour s'échapper vivant.

Le jeu se déroule intégralement dans la console : le joueur interagit en saisissant des commandes textuelles permettant de se déplacer, examiner l'environnement, ramasser ou utiliser des objets, combattre des créatures et gérer son inventaire.

Le développement a été structuré selon les principes de la programmation orientée objet, en mettant en place une architecture modulaire incluant des classes dédiées aux lieux, aux objets, aux personnages, au héros et aux commandes. Le moteur d'interprétation permet de traiter les instructions du joueur et de maintenir l'état du jeu au fil de l'aventure.

Ce rapport présente la documentation utilisateur, la documentation développeur avec le diagramme de classe commenté, ainsi que la répartition des tâches.

Documentation utilisateur - Dungeon Adventure

Installation

Pré-requis

Pour jouer à Dungeon Adventure, il faut disposer de :

- **Java 21** installé sur la machine
- Un terminal (Windows, MacOS, ou Linux)

Pour vérifier votre version de Java, tapez dans votre terminal : java -version

Installation du jeu

1. Télécharger et extraire l'archive **duhem_sibé_vadegrelet.zip**
2. Ouvrir un terminal et se placer à la racine du projet
3. Pour compiler le projet :
   make compile
4. Pour lancer le jeu :
   make run
5. ou en exécutant le fichier ja**r java -jar game.jar**

Objectif du jeu

Le joueur doit explorer un donjon, survivre aux dangers et vaincre le boss final, condition unique pour s'échapper.

Pour y parvenir, il faudra :

- se déplacer entre les salles
- trouver des objets utiles
- combattre ou éviter les ennemis
- gérer l'état du héros (vie, objets …)

Fonctionnement général du jeu

Interaction console

Dungeon Adventure fonctionne intégralement dans un terminal.
Le joueur tape des commandes de la forme :
COMMANDE [arguments]
Une fois la touche Entrée pressée, le jeu interprète la commande, effectue l'action et affiche le résultat.

Commandes disponibles

Voici les principales commandes implémentées dans le jeu :

- La commande **LOOK <objet>**
  LOOK -> Affiche la description complète de la salle actuelle :
    - objets présent
    - personnages
    - sorties disponibles

  LOOK <objet> -> Affiche des informations sur l'objet ou l'élément demandé.
- La commande **GO <destination>**
  Permet de se déplacer vers une salle adjacente.
  Si la sortie existe et est accessible, le joueur s'y déplace.
  Sinon, un message d'erreur est affiché.
- La commande **TAKE <objet | all>**
  TAKE potion-> récupère l'objet nommée potion
  TAKE all -> récupère tout les objets d'une room
  Permet de ramasser un objet dans la salle.
  L'objet est ajouté à l'inventaire.
- La commande **DROP <objet>**
  Permet de lâcher un objet de votre inventaire.
- La commande **USE <objet> [cible]**
  USE potion -> le joueur utilise un objet seul.
  USE key boss <direction> -> permet d'ouvrir une porte vérouillé
- La commande **ATTACK <cible>**
  Le combat affiche les dégâts infligés et reçus.
- La commande **INVENTORY**
  Affiche tous les objets qui sont dans le sac à dos.
- La commande **STATS**
  Affiche le statut du joueur(HP, dégâts de base, l'arme équipée et les dégâts totaux)
- La commande **HELP**
  Affiche la liste des commandes disponibles.
- La commande **SAVE <nom de la save>**
  Créer un fichier de sauvegarde.
- La commande **LOAD <nom de la save>**
  Lance le jeu à partir de la sauvegarde.
- La commande **QUIT**
  Quitte immédiatement le jeu.

Déplacements dans le donjon

Chaque salle est reliée à d'autres par des sorties.
Les sorties peuvent être ouvertes ou verrouillées par une clé.

Système de combat

Lorsque le joueur attaque un ennemi :

- Le héros inflige des dégâts
- L'ennemi riposte (si il est vivant)
- Les points de vie sont mis à jour

Si les points de vie du héros atteignent 0 -> **Game Over**

Conditions de victoire / fin de partie

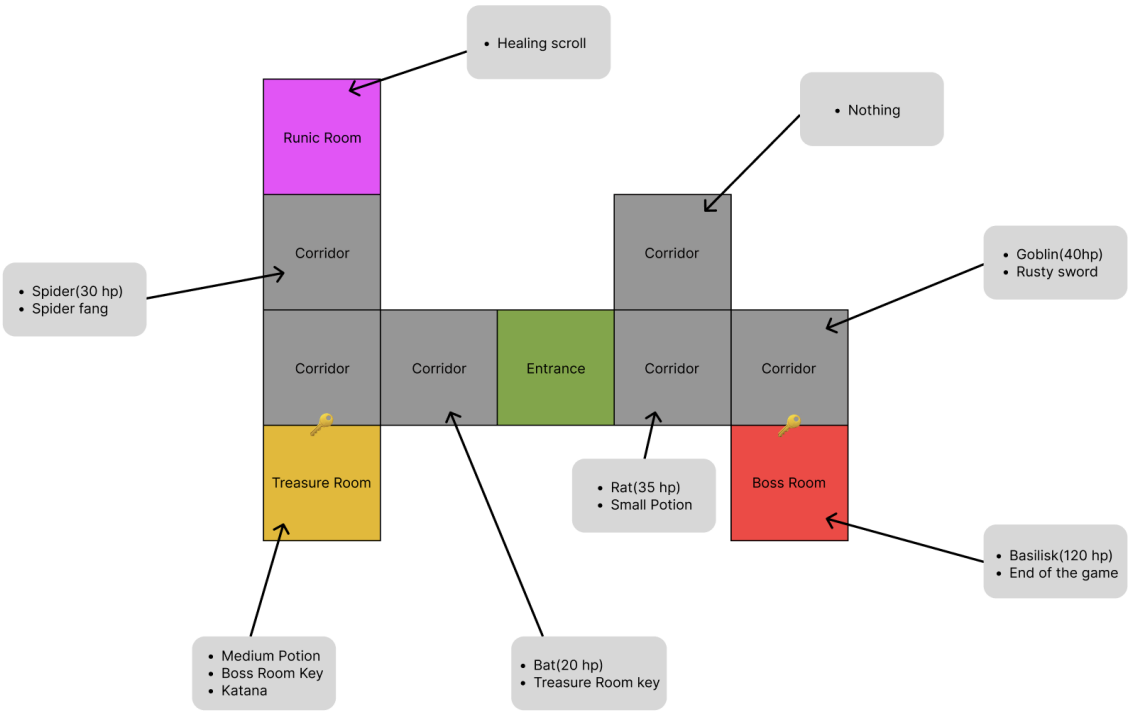
Victoire :
Le joueur gagne la partie lorsqu'il réussit à atteindre le boss final et à le vaincre.

Défaite :
Le jeu se termine si :

- les points de vie du héros tombent à 0
- le joueur choisit de **QUIT**

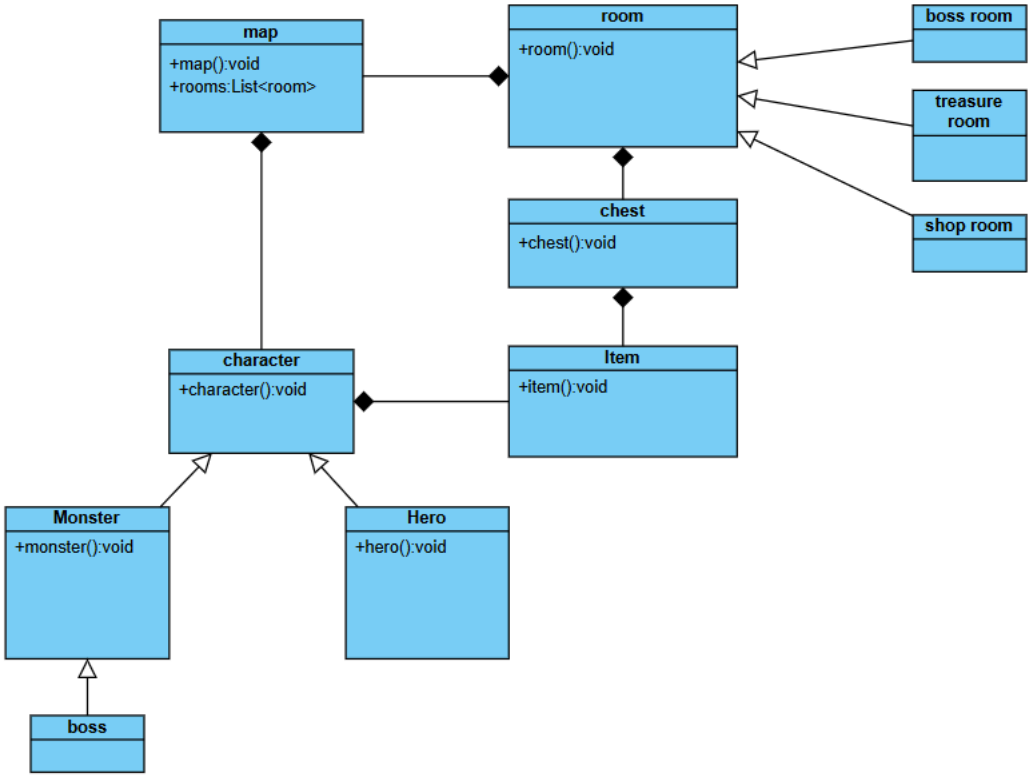
Un message final est toujours affiché.

Map



Documentation Développeur:

Nous avons décidé de commencer par un UML simpliste pour nous donner une idée générale de la conception de notre projet, voici l'UML:



Avant de décrire l'UML, je rappelle qu'il a été sujet à de nombreux changements et à été complété, la version finale sera décrite ultérieurement.

Nous avons d'abord décidé qu'il nous faudrait une carte qui contiendrait des rooms (comme un donjon).
Ces rooms pouvaient être simple, ou avoir une spécialité comme représenté par boss room, treasure room et shop room. Nous voulions faire en sorte qu'une room simple soit une classe mère dont boss, treasure, et shop héritent.
Notre map serait donc une liste de room, dont la durée de vie est liée à notre map.
(D'ailleurs toute entité du jeu serait liée à notre map, sans map, pas d'objets).

Nous avons ensuite décidé de créer des héros et des monstres. Nous avons commencé par créer une classe character, plus générale, qui permet d'y regrouper les attributs et méthodes communes à tous les personnages du jeu (par exemple les points de vies ou le fait d'attaquer …). De cette classe hérite les monstres, qui ont leurs propres méthodes, et boss, qui est juste un monstre avec un "pouvoir" que nous n'avions pas encore décidé.

De cette classe hérite également les héros. Nous avons hésité à créer une classe généraliste Heros de laquelle des classes comme "rogue","warrior" ou autres pourraient

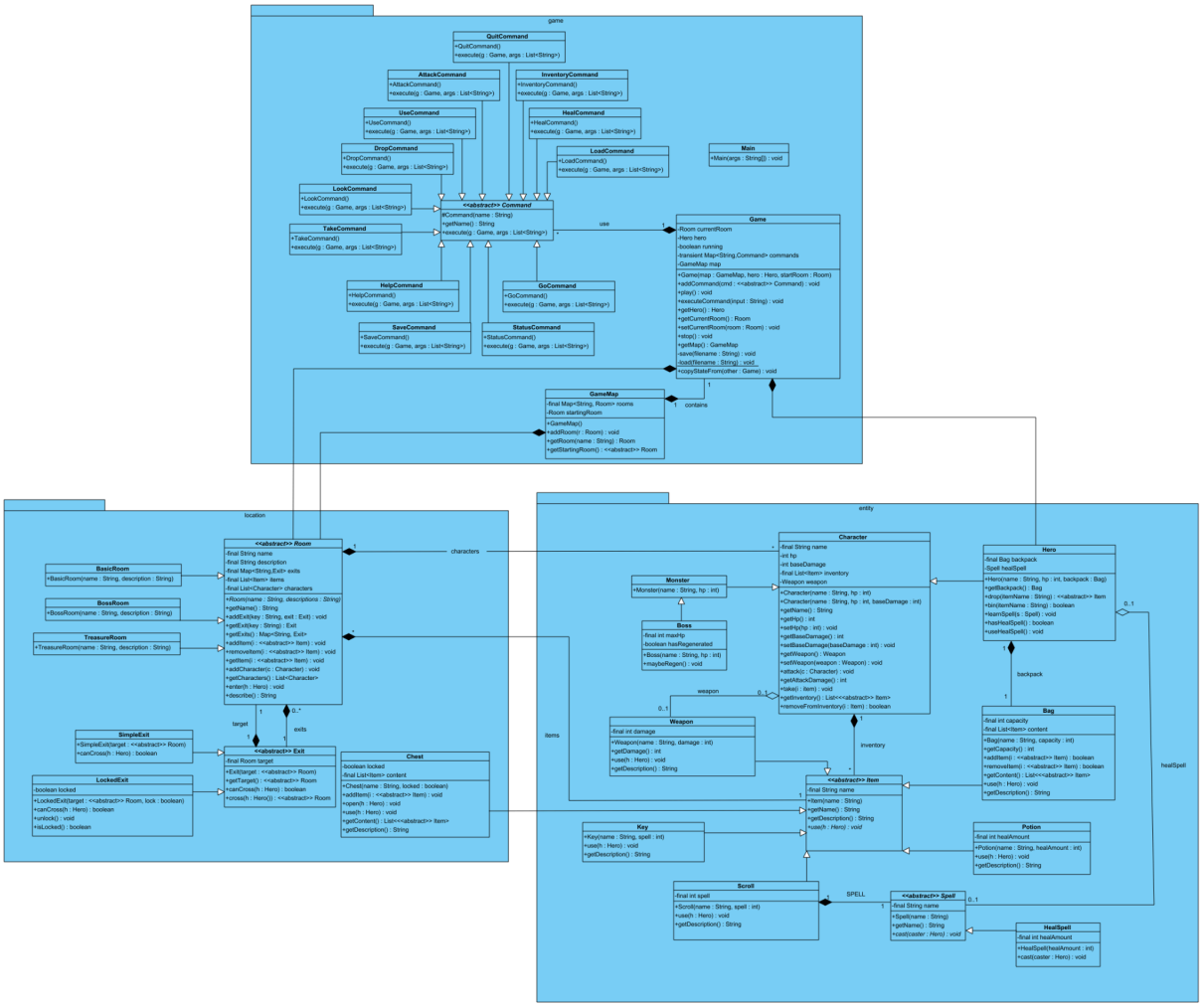
hériter, mais nous avons opter pour l'idée de ne créer qu'un Hero unique qui pourra se différencier par les items qu'il trouvera et utilisera/équipera pendant la partie.

Pour cela nous avons créé une classe Item ainsi que une classe Chest qui contiendrait des Item.

Voilà le projet initial avec lequel nous avons commencé à coder. Il est incomplet mais souvent l'UML complète le code et vice-versa.

Il nous a fallu ensuite réfléchir à des méthodes de base. Par exemple le fait de pouvoir changer de room, donc une méthode pour que notre héros puisse bouger. Également une méthode pour que le héros puisse équiper ou utiliser les objets qu'il trouvera pendant sa partie.
De plus avec un système de monstre, il nous faut des méthodes d'attaque, des attributs de vie et de dégâts etc… .

C'est en se posant toutes ces question que nous en sommes arrivé à notre UML final (en alternant code et complétion de l'UML) que voici:



Nous avons scindé notre projet en 3 packages distincts: game, location et entity.

- Game est le cœur du moteur du jeu, il gère les états, les commandes et la boucle de jeu.
- Location gère la map, les rooms, leurs connexions, et les éléments présents dans l'environnement
- Entity gère les personnages, l'inventaire et les objets utilisables

Parlons de game.

Game est la classe centrale du projet, elle :

-initialise la partie : on y crée la GameMap, on y crée le joueur (Hero) et on y crée les salles et leurs connexions.

-gère l'état courant : où on se trouve (getCurrentRoom), les monstres ou objets présents (commande LOOK)

-gère les commandes : voir page 4

Les commandes reposent sur la classe abstraite Command, agrégée à Game, qui définit une interface commune, et une méthode execute implémentée par toutes les commandes concrètes.

Chaque commande héritée de Command correspond à une action que le joueur peut exécuter:

- interaction avec l'environnement : LookCommand, TakeCommand, DropCommand,UseCommand
- gestion du héros : InventoryCommand,StatusCommand
- gestion du système: SaveCommand,LoadCommand,Quitcommand
- combat : AttackCommand
- aide: HelpCommand

Cette façon d'implémenter nous permettrait si l'envie nous prend de rajouter une commande sans modification du code déjà existant.

Pour structurer le monde nous avons la classe GameMap , liée à Game par agrégation. Elle stocke un ensemble de salles (room: Map<String, Room>) et une salle de départ (startingRoom). Elle possède également des méthodes utilitaires comme l'ajout de room (addRoom) ou récupérer une room(getRoom).
Nous avons séparé game et gameMap afin que leurs fonctions soient distinctes, l'un s'occupe du jeu et l'autre gère les détails topologiques des rooms.

Parlons de location.

La classe abstraite Room est la fondation de notre monde. Elle représente le lieu qui contient : nos personnages, les items, les coffres, les connexions à d'autres salles ainsi que leurs descriptions . Plus précisément elle contient les méthodes d'ajout de tous les objets mentionnées ci-dessus pour pouvoir créer une room a notre convenance.
Room possède également des méthodes fondamentales : décrire la salle, déterminer si une sortie est possible, interaction avec le joueur…

Room étant abstraite, elle possèdes des sous classes de rooms qui diversifient l'aventure. BasicRoom est une room simple, BossRoom contient le boss et est bloquée par une porte verrouillée (point que nous aborderons sous peu), Treasure room contient un coffre.
Ces rooms possèdent des entrées/sorties. Celles-ci sont représentées par la classe abstraite Exit en agrégation avec Room (durées de vie liées).
Elle implémente la traversée de salles ainsi que la possibilité qu'une porte soit verrouillée grâce à la méthode canCross. Si canCross est false alors l'Exit n'est pas une SimpleExit mais une LockedExit , ce qui introduit la mécanique de clés. Utiliser une clé passera l'attribut locked à false et rendra accessible la sortie.

Parlons d'entity

Ce package regroupe les classes liées au joueur, aux monstres, aux objets et à l'inventaire. C'est ici que nous gérons le gameplay.

Character et Item sont les classes pilier de entity, elles sont en agrégation avec room car leurs durées de vie sont liées, pas de room implique pas de personnage/item.

La classe Character est une classe abstraite qui représente toutes nos entités vivantes (entités possédant des HP).
Un character contient des points de vies HP, éventuellement une arme, il aura un inventaire et aura des méthodes d'action : attaquer, recevoir des dégâts, utiliser un objet, prendre un objet, changer d'arme… .
Ces attributs et méthodes unifient le comportement commun entre le joueur (Hero) et les ennemis (Monster).

La classe Hero qui étend Character ajoute la capacité à utiliser des sorts, ainsi que la gestion d'inventaire avec des méthodes comme drop().
Les Monster n'auront pas de méthodes supplémentaires, sauf le boss qui possède un pouvoir spécial, une méthode qui lui regénérera des HP.

Vient ensuite la classe abstraite Item, qui implémente tous les objets du jeu. Chaque objet a un nom, une description et une méthode use (chaque objet est donc utilisable).
Les sous classes d'Item spécialisent les comportement:

- Weapon ajoute un attribut damage, et est utilisé par les personnages pour augmenter leurs dégâts.
- Key permet de déverrouiller les LockedExit
- Potion apport un effet de soin(healAmount)
- Scroll qui contient un Spell de soin, donc un attribut Spell spell, que le héros peut apprendre avec sa méthode learnSpell et qui lui permettra d'utiliser la magie pour se soigner

La classe Spell étant abstraite pour permettre de pouvoir rajouter, si on le veut, d'autres types de spell. Nous avons décidé de ne faire qu'un sort de soin pour l'instant.

Pour gérer les Item que le Hero pourra porter, nous avons la classe Bag, qui représente l'inventaire. L'inventaire à une capacité maximum d'Item et une liste d'Item(seuil par capacity).Cette classe possède des attributs de base qui permettent d'ajouter ou d'enlever un Item, de regarder les Item dans le Bag ainsi que d'utiliser les Item.

Interactions des modules

game <~> location
    les commandes agissent sur les salles courantes (Take, Look …)
game <~> entity
    les commandes modifient le héros et son inventaire(Use, Drop…)
location<~> entity
    les rooms contiennent des item et des character

Nous pensons que notre architecture permet un modèle cohérent grâce à la séparation des concepts du jeu en classes bien définies; de plus la modularité permet d'ajouter des types de spells, de room ou de monstre naturellement.

Répartition des tâches

L'organisation du travail au sein de notre groupe s'est faite principalement par répartition des packages, conformément à la structure du projet. Au fil de l'avancée, certains besoin ont nécessité des contributions croisées afin d'assurer la cohérence globale du jeu.

Organisation initiale

- Lucas : package game
- Adam : package location
- Léonard : package entity

Contribution finale

Lucas :
- package game ≈ 80%
- package entity ≈ 40%
- package location ≈ 40%
- réalisation du diagramme UML final
- bug fix divers

Adam :
- package location ≈ 60%
- réalisation des tests
- rédaction du rapport

Léonard :
- package game ≈ 20%
- package entity ≈ 60%
- game design et structure global du donjon/organisation de la map
- rédaction du rapport et du README