

Rapport - Jeu d'aventure sur console

Étudiants: Paul Jouet, Loup Pigeaud-Chessé, Thomas Calberac

UE : Programmation Orientée Objet

Date : 6 décembre 2025

Encadrant : Samuel Peltier

Table des matières

Introduction et contexte du projet	3
Axe 1 : Architecture et conception	3
Modèle architectural global.....	3
Structure des paquets.....	4
Diagramme de classe UML	5
Axe 2 : Qualité du code et pratiques respectées.....	8
Gestion personnalisée des exceptions.....	10
Améliorations possibles	11
Axe 3 : Tests.....	11
Test fonctionnels	11
Test structurels	12
Conclusion des tests	12
Axe 4 : Expérience utilisateur	13
Interface console.....	13
Découvrabilité	13
Manuel Utilisateur	14
Conclusion	15
Organisation de l'équipe	15
Le mot de la fin	15

Introduction et contexte du projet

Ce projet fait partie du cours de programmation orientée objet. L'objectif était de développer un jeu d'aventure textuel en console à l'aide de Java, en mettant l'accent sur les principes de la programmation orientée objet et les bonnes pratiques de développement.

Le jeu offre une expérience interactive dans laquelle le joueur explore un monde virtuel et interagit avec des objets et des personnages à l'aide de commandes textuelles. Ce rapport détaille les aspects techniques et méthodologiques du développement.

Axe 1 : Architecture et conception

Modèle architectural global

Notre modèle repose sur la séparation des tâches. Nous pourrions parler de couches :

La couche d'affichage :

- Console.java : gestion des entrées/sorties et affichage

La couche Contrôleur :

- PlayerController.java : coordination des joueurs
- commands/ : gestion des commandes

La couche Entités :

- acteurs/ : joueur et PNJ
- locations/ : carte
- inventory/ : gestion des objets
- exits/ : mécanisme de navigation

Au-dessus de toutes ces couches, nous avons l'état du jeu, qui contient tout cela et le contrôle. Par exemple, pour les contrôleurs, nous avons une file d'attente dans l'état du jeu qui nous permet de savoir à quel contrôleur c'est le tour.

Cette séparation des responsabilités permet l'évolutivité du code. Chaque couche a une responsabilité clairement définie, ce qui réduit le couplage entre les composants.

Structure des paquets

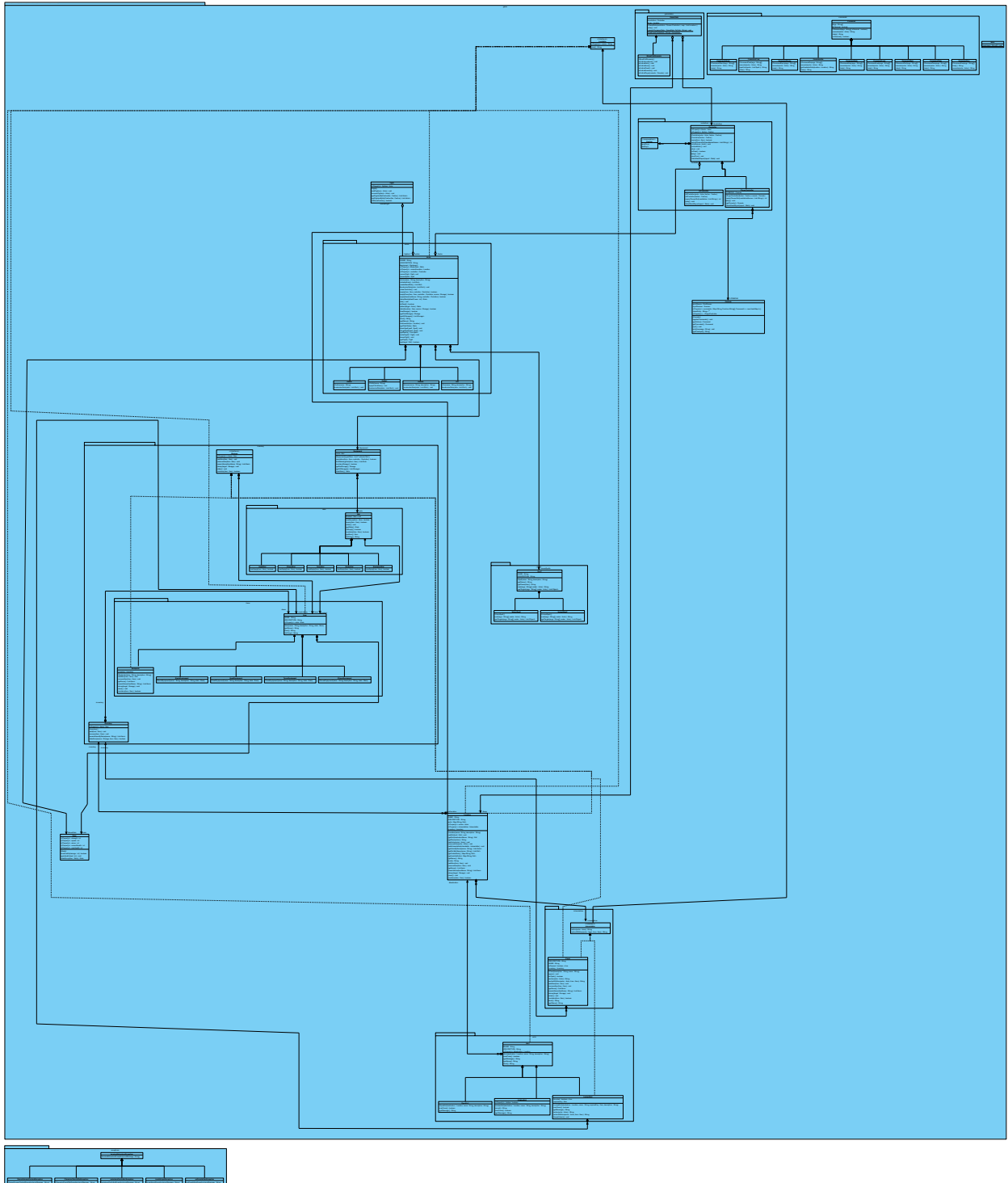
Voici une représentation graphique de la structure de nos paquets :

terminal.adventure.game

└─ actors/	# Personnage du jeu
└─ commands/	# Commandes du joueur
└─ controllers/	# Contrôleurs de personnage
└─ exits/	# Sorties/Système de navigation
└─ gamestates/	# Etat du jeu
└─ interactables/	# Tous les éléments interactifs l'implémente(interface)
└─ inventory/	# System d'inventaire
└─ items/	# Objets en jeu
└─ slots/	# Slot d'équipement
└─ Equipment.java	
└─ Inventory.java	
└─ Storage.java	
└─ spells/	# Système de sort et de magie
└─ Console.java	# Affichage/Gestion de la console
└─ Fight.java	# Système de combat
└─ Location.java	# Lieux dans le jeu
└─ Lookable.java	# Tous les éléments observables l'implémente (interface)
└─ Main.java	
└─ Stats.java	# Statistique d'un Objet

Vous pourrez trouver dans le dossier .zip la JavaDoc générée.

Diagramme de classe UML



Reprenons les particularités principales de notre conception.

Une grande partie de notre conception se fait autour des contrôleurs, les contrôleurs sont soit un IAControleur qui dirige un NPC soit un PlayerController qui dirige le héros du jeu. L'idée derrière ces contrôleurs est de permettre non seulement à tout moment de changer de corps, mais aussi d'implémenter des réactions IA des ennemis/alliés.

Dans l'état du jeu (gamestates), on retrouve une file de tous les Controller représentant le tour par tour chacun fait une action pour le PlayerController il y a donc des commandes qui passent un tour et d'autres d'informations qui peuvent s'enchaîner.

Le PlayerController est le seul en capacité à interagir avec la console, il récupère une commande saisie, demande à la commande de s'exécuter en lui fournissant le personnage qu'il contrôle puis récupère le message d'exécution de la commande. Ensuite, il demande à la console de l'afficher.

Notre conception vise à limiter chaque classe à une action/une fonctionnalité ici la commande est uniquement une classe d'affichage et de gestion des Entrées-sorties tandis que le Controller lui a pour unique fonctionnalité d'exécuter des commandes à son personnage.

Pour ce qui est de la conception d'un personnage. Un personnage a des attributs classiques, mais aussi des slots qui ensemble forme un équipement représentant les emplacements où l'on peut équiper des objets. Un personnage possède par défaut un sac à dos qui est un objet qui implémente une interface Storage qui va s'occuper de toute la gestion des inventaires et du déplacement d'un objet d'un Storage à un autre.

Un personnage a des Stats ces Stats ont fait l'objet d'une profonde réflexion dans le groupe et nous avons pris la décision d'en faire une classe regroupant les statistiques qui peuvent être celle d'un personnage ou bien d'un objet.

Un personnage implémente comme un lieu et une sortie l'interface Lookable qui représente toutes les entités que l'on peut observer. (tous ceux sur qui on peut faire la commande LOOK)

Pour ce qui est des commandes, nous gérons depuis la console jusqu'au Playercontroller une commande hérite de la classe abstraite Command imposant 2 méthodes executes qui exécute la commande et renvoie un message d'exécution et help qui renvoie un message d'utilisation de la commande de là, on peut ajouter retirer aisément des commandes, car même casté en tant que Command elle reste parfaitement utilisable. La console fait le travail de reconnaître la commande saisie et envoie une instance de commande au PlayerController.

Les spells fonctionnent à peu près de la même manière que les commandes dans le sens où il y a une Spell abstraite mère et toutes les Spell ont une action sur une target, target pouvant être littéralement n'importe quoi. Ici, on a implémenté uniquement des sorts qui agissent directement sur l'état d'un objet, mais rien ne nous empêche d'en ajouter de nouvelles agissant sur d'autres entités.

Notre gestion des controllers nous amène aux combats là aussi élément important du jeu, un combat est représenté par une classe on crée une instance lors d'un combat dans lequel des controllers agissent tour à tour avec des fonctions propres à la perte de point de vie, d'attaque, de mort. Encore une fois les controllers nous permettent une certaine liberté malgré la complexité

de leur utilisation, car nous pouvons à tout moment rejoindre un combat lancé entre deux IA et inversement.

Le système des exits étaient véritablement imposé par le sujet bien que nous ayons trouvé la solution proposée étrange. En effet devoir dire, je vais à salle des coffres et que l'on nous dise et bien vous passer par la porte, n'est pas des plus naturelles. Nous pensions plutôt à l'inverse cependant nous avons respecté ce choix du sujet alors une exit est une classe abstraite, nous avons implémenté des lockedExit sur lesquels on peut soit envoyer un sort pour l'ouvrir ou bien utiliser une clé. On implémente aussi une hiddenExit invisible par la commande look jusqu'au lancement d'un sort qui la fait apparaître. Encore une fois, l'implémentation réalisée nous permet d'étendre ce système sans problème.

La sérialisation ce fait donc dans le gamestate, nous pouvons aussi bien sauvegarder une partie qu'en importer une.

Axe 2 : Qualité du code et pratiques respectées

Suite aux erreurs commises dans un autre projet Java, nous avons essayé de suivre les règles de la méthode SOLID afin que la conception de notre jeu soit aussi extensible et claire que possible.

S - Single Responsibility Principle

Exemple : séparation stricte des responsabilités

```
// CommandLook: uniquement de l'observation
public class CommandLook extends Command {
    public CommandLook(String[] args);
    @Override
    public String execute(Actor actor){
        // Responsabilité unique : fournir une information visuel
    }
    public String help();
}
```

O - Open/Closed Principle

Exemple : Extension sans modification

```
// Classe de base fermée à la modification
public abstract class Command {

    protected String[] args;
    protected boolean isTerminal;

    // Structure fixe, ne change pas
    public abstract String help();
    public abstract String execute(Actor actor);

}

// Nouvelles commandes sans modification du code existant
public abstract class CommandNewCommand {

    @Override
    public abstract String help(){
        return "help to launch the command";
    }
    @Override
    public abstract String execute(Actor actor){;
        //Specific implementation
        return "you cast a spell";
    }

}
```

L - Liskov Substitution Principle

Exemple : Interchangeabilité des commandes

```
// Toute sous-classe de Command peut remplacer Command
List<Command> availableCommands = new ArrayList<>();
    new CommandLook(new String[]{}),
    new CommandGo(new String[]{"north"}),
    new CommandTake(new String[]{"sword"})
);

for (Command cmd : availableCommands) {
    // Elles fonctionnent tous de la même manière
    String result = cmd.execute(player);
    String help = cmd.help();
}
```

I - Principle of Interface Segregation

Exemple : interfaces minimalistes et spécifiques

```
// Lookable interface: uniquement pour ce qui peut être observé
public interface Lookable {
    String look(); // Méthode unique et spécialisée
}

// Il en va de même pour toutes les interfaces que nous créons
```

D - Dependency Inversion Principle

Exemple : dépendances abstraites

```
// La console gère l'affichage
public class Console{

    private PlayerController player; // Abstraction
    // La console envoie un événement à PlayerController.

    ...
}

// PlayerController implémente l'abstraction
public class PlayerController {
@Override
protected void play() {
    // Réaction aux événements
    // Demande à la console d'afficher le résultat du traitement des
    commandes qu'il a appelé
}
```

Gestion personnalisée des exceptions

Utilisation contextuelle :

```
@Override
public String execute(Actor actor) throws InvalidInputException {

    if (args.length == 0) {
        // Message d'erreur détaillé avec aide intégrée
        throw new InvalidInputException("Where do you wanna GO ?\n" +
            this.help() +
            this.getAvailableExits(actor.getCurrentLocation()));
    }

    Location currentLocation = actor.getCurrentLocation();
    Exit exit = currentLocation.getExit(this.args[0]);

    if (exit == null) {
        // Message d'erreur détaillé avec aide intégrée
        throw new InvalidInputException("There is no location named \"" +
            String.join(" ", args) +
            "\" around here.\n" +
            this.getAvailableExits(actor.getCurrentLocation()));
    }

    if (!actor.go(exit)) {
        // Message d'erreur détaillé avec aide intégrée
        throw new InvalidInputException(exit.getMessage());
    }

    Location newLocation = exit.getDestination();
    return ("You walk through " + exit.getName() + "... \n") +
        ("You've arrived at: " + newLocation.getName() + " \n" +
            newLocation.look());
}
```

Améliorations possibles

Bien que nous nous soyons efforcés de concevoir notre projet de manière à ce qu'il soit aussi extensible que possible, nous sommes conscients qu'il manque des interfaces potentielles. Nous pourrions ajouter des interfaces d'abstraction à plusieurs endroits du projet, mais en raison de contraintes de temps, cela n'a pas encore été fait. Cependant, notre implémentation nous permettra de le faire à l'avenir.

Axe 3 : Tests

Test fonctionnels

Les tests fonctionnels sont des tests réalisés en parallèle de l'implémentation de la fonction. Ils ont été réalisés par un développeur différent de celui qui l'implémente.

Voici un exemple de test fonctionnel :

```
@Test
public void testClearController_NoController_NothingHappens() {
    // Precondition: Actor has no controller
    assertNull("Actor should have no controller initially",
actor.getController());
    assertEquals("No unbind calls initially", 0,
controller1.getUnbindCallCount());

    // Execution
    actor.clearController();

    // Verification
    assertNull("Actor should still have no controller",
actor.getController());
    assertEquals("No unbind calls should be made", 0,
controller1.getUnbindCallCount());
}
```

Test structurels

Les tests structurels sont des tests qui permettent entre autres de détecter du code mort. Nous les avons réalisés juste après avoir implémenté la méthode.

Voici un exemple de test structurel :

```
// Test 1: Controller is null → No action taken
@Test
public void testClearController_ControllerIsNull_NoAction() {
    // Preconditions: actor has no controller
    assertNull("Actor should not have a controller initially",
actor.getController());
    assertEquals("Controller unbind should not be called", 0,
controller1.getUnbindCallCount());

    // Store initial state
    Controller initialController = actor.getController();

    // Execution
    actor.clearController();

    // Verifications
    assertNull("Actor should still not have a controller",
actor.getController());
    assertEquals("Actor controller unchanged (null)", initialController,
actor.getController());
    assertEquals("Controller unbind should not be called", 0,
controller1.getUnbindCallCount());
}
```

Conclusion des tests

La partie test est véritablement le point faible de notre projet. Par manque de temps, nous n'avons pas pu réaliser une grande quantité de tests. L'objectif pour nous était alors de montrer que nous avons compris comment réaliser les tests notamment que nous avons compris la différence entre les tests structurels et les tests fonctionnels.

Axe 4 : Expérience utilisateur

Interface console

Retour immédiat : résultats après chaque commande

Messages d'erreur : explicatifs, avec suggestions

Help() contextuelle : fonction help() intégrée à chaque commande

Cohérence : même modèle pour toutes les interactions

Découvrabilité

Pour rendre le jeu aussi agréable que possible, nous avons essayé d'envoyer des messages pour chaque action effectuée par l'utilisateur, qu'elle soit bonne ou mauvaise, afin de le guider dans ses actions. Nous pensons que c'est important, car dans un jeu comme celui-ci sur console, il est souvent difficile de comprendre ce que l'on doit faire et ce que l'on peut faire...

Voici un exemple de ce que nous renvoyons à l'utilisateur :

```
// Commande Inconnue
> OPEN door
"Unknown command. Type HELP for available commands."

// Argument manquant
> GO
"Where do you wanna GO ?"
Available exits from here:
- Forest
- Cave
```

Manuel Utilisateur

scenario :

Le royaume est en proie à la sécheresse ! Après une vague de répression contre le culte du sang ordonnée par le roi, le chef du culte a maudit le royaume afin qu'il ne connaisse aucune pluie pendant le siècle à venir. Cette situation ne peut plus durer, le roi vous a chargé, vous, le héros, de trouver une solution à ce problème !

Vous vous rendez donc à la base principale du culte afin de trouver et de tuer son chef, et ainsi libérer le pays de cette terrible malédiction !

En chemin, vous rencontrerez divers membres du culte, des monstres, mais aussi des alliés qui vous aideront dans votre quête. De nombreux secrets se cachent dans les vieux murs de la base du culte...

Commandes :

- **HELP** : montre les commandes utilisables
- **LOOK** : montre tout ce qui est visible dans le lieu actuel ou
- **LOOK <args>** : montre la description de l'argument donné
- **GO <location name>** : essayez de d'aller au lieu donné en passant par une sortie s'il y en a une
- **USE <things>** : essayez d'utiliser un objet quelque chose qui pourrait se déverrouiller ex : un coffre...
- **TAKE <item>** : permet de ramasser un objet qui traînerait sur le sol du lieu où vous vous trouvez
- **EQUIP <item>** : équipe un item de votre inventaire sur vous
- **FIGHT <enemy>** : permet d'entrer en combat contre un ennemi ATTENTION !!! un ennemi peut déjà être en combat contre un joueur dans ce cas vous rejoindrez l'équipe du joueur de votre camp. Il en va de même pour vos ennemis, un ennemi peut rejoindre votre combat à n'importe quel moment.
- **QUIT** : permet de quitter le jeu

Conclusion

Organisation de l'équipe

Nous avons réalisé ce projet à 3, voici comment nous nous sommes réparti les rôles (Attention cette répartition n'est qu'exhaustive, en réalité nous avons énormément changé la conception au cours du projet ce qui nous a mené à tous reprendre une partie de ce que quelqu'un avait déjà fait) : Paul Jouet s'est chargé principalement de la gestion d'un combat ainsi que celle des inventaires/items. Loup Pigeaud-Chessé s'est quant à lui principalement concentré sur l'état du jeu la sérialisation ainsi que des contrôleurs. Enfin, Thomas Calberac s'est chargé des commandes consoles, sorts et tests. Nous estimons que chacun a contribué à part égale dans le projet.

Le mot de la fin

Ce projet de jeu d'aventure en console nous a montré l'importance de la conception avant l'implémentation, de l'écriture de la documentation en continu pour le travail en équipe, et des tests réguliers pour remonter les problèmes et améliorer l'implémentation. Au-delà de l'aspect technique, il montre la possibilité de créer un jeu complet avec des moyens limités, amenant à un sentiment d'accomplissement pour l'entièreté de l'équipe.

Le code produit constitue une base solide qui pourrait :

- Servir de référence pour de futurs projets similaires
- Être étendu pour inclure des fonctionnalités plus avancées
- Illustrer des concepts avancés de programmation orientée objet
- Former la base d'un projet open-source éducatif

En définitive, ce projet représente non seulement la mise en œuvre des compétences enseignées, mais aussi la capacité à les appliquer de manière ludique pour résoudre des problèmes concrets et produire un logiciel fonctionnel, robuste et évolutif. Au nom de l'équipe de développeur, le seul regret que nous pouvons avoir est le manque de temps pour faire grossir le projet et le rendre encore plus extensible et plus performant.