Introduction :

Dans le cadre de notre formation universitaire, nous avons réalisé un projet de développement de jeu vidéo en Java, en utilisant la bibliothèque LibGDX, spécialisée dans la création de jeux 2D.
L'objectif du projet était double :

Concevoir un jeu complet, fonctionnel et structuré professionnellement.

Appliquer les bonnes pratiques de programmation, de gestion de projet, et de qualité logicielle à travers les tests unitaires et d'intégration.

Nous avons développé un roguelike en 2D, un genre dans lequel le joueur doit survivre à des vagues successives d'ennemis, récupérer des bonus, et affronter un boss final.
La progression repose sur :

La maîtrise des déplacements et des tirs.

La gestion des ressources de vie et des boosts.

L'amélioration continue du score basé sur le temps de survie.

Au-delà de la simple réalisation technique, nous avons porté une attention particulière à :

La qualité du code (architecture claire et modulaire).

La scalabilité (facilité d'ajout futur de nouveaux ennemis, objets, niveaux...).

La robustesse (grâce aux tests automatisés et au travail collaboratif organisé).

Ce projet représente ainsi une expérience complète de conception logicielle, développement agile, et apprentissage collaboratif, avec la volonté de produire un résultat à la fois fonctionnel, propre et extensible.

Partie 2 : Optimisation, architecture et structure du projet

2.1. Organisation générale du projet
Notre projet repose sur une architecture solide et modulaire, suivant les principes de la programmation orientée objet.
Chaque élément du jeu (joueur, ennemis, projectiles, objets, map) a été clairement séparé en différentes classes, rassemblées dans des packages bien organisés :

entities/ : Joueur, ennemis, projectiles.

entities/items/ : Boosts et bonus (items récupérables).

managers/ : Gestionnaires de vagues (WaveController, WaveManager).

screens/ : Différents écrans du jeu (MenuScreen, GameScreen, DeathScreen).

utils/ : Outils transversaux (Timer, Constants, Position, Damageable, etc.).

world/ : Carte du jeu (GameMap).

➡️ Cette organisation permet de :

Faciliter la lisibilité et la compréhension rapide du projet.

Séparer clairement les responsabilités de chaque classe.

Permettre l'évolutivité : ajouter de nouveaux éléments est simple et sans casser l'existant.

2.2. Architecture détaillée
Notre projet suit une architecture dynamique en temps réel (update/render loop), typique des jeux vidéo :

Phase Rôle
update() Mettre à jour l'état du jeu (joueur, ennemis, projectiles, bonus, vagues) à chaque frame.
render() Dessiner tous les éléments visuellement à l'écran à chaque frame.
Chaque entité suit le même cycle : update → render.

Le rôle de chaque package :
entities/ :

Player.java : Gère l'état et les déplacements du joueur, les tirs, la gestion de vie et les boosts.

Projectile.java : Gère les tirs du joueur, leur vitesse, et leur collision avec les ennemis.

EnemyBase.java (interface) : Définit un contrat commun pour tous les types d'ennemis (update, draw, collisions).

entities/items/ :

Système de boosts modulables (santé, vitesse, projectiles).

Tous les boosts implémentent une interface ItemBase pour une gestion générique.

managers/ :

WaveController.java : Chef d'orchestre des vagues d'ennemis et du boss final.

WaveManager.java et ses dérivés : Gestion de différents types de vagues (jaune, bleu/jaune, rouge, full).

screens/ :

Gère le changement d'écrans entre le menu principal, le jeu, et l'écran de mort.

utils/ :

Timer : Chronomètre du jeu.

Constants : Fichier centralisé de toutes les constantes (HEALTH, SPEED, TIMER, etc.), évite les valeurs magiques dans le code.

Damageable : Système générique de vie/invincibilité.

Position : Enregistre simplement une position x,y.

MovementSpeed : Gère la vitesse et ses modifications.

Hitbox : Gère les collisions entre entités.

world/ :

GameMap : Gestion basique des murs (mur invisible aux bords) pour limiter le joueur.

2.3. Pourquoi cette structure est optimisée et évolutive
Nous avons appliqué plusieurs principes de qualité logicielle :

Responsabilités uniques : Chaque classe a un seul rôle clair.

Utilisation d'interfaces : (EnemyBase, ItemBase) → facilite l'ajout d’ennemis ou d’objets sans modifier l’existant (Open/Closed Principle).

Centralisation des constantes (Constants.java) : changement facile sans fouiller tout le projet.

Protection de l’état interne : Le joueur utilise des classes internes (Damageable, MovementSpeed) pour mieux encapsuler la logique métier (ex: invincibilité).

Gestion modulaire des vagues : Les vagues sont gérées indépendamment par des classes, permettant d'en rajouter facilement d’autres (ex: de nouveaux types d'ennemis).

Design extensible : L'ajout d'un nouvel objet (item) ou d'un nouveau type de projectile/ennemi est faisable en quelques lignes, sans casser la structure existante.

2.4. Points d'amélioration possibles
Même si notre structure est très solide pour un projet étudiant, voici des idées pour l'améliorer encore :

Gestion des collisions plus fine (ex: séparée en un système dédié, pour détecter collisions entre toutes entités dynamiquement).

Gestion d'animations plus évoluée (ex: utiliser une librairie externe pour améliorer les effets visuels).

Création d'un système d'événements (event bus) pour éviter des couplages directs entre classes dans certaines interactions (ItemManager → Player par exemple).

Optimisation des ressources : charger les textures une seule fois dans une classe "AssetManager" plutôt qu'à la main dans chaque entité.

Menus et options plus poussés (ex: sauvegarder la progression, paramètres de jeu).

Partie 3 : Fonctionnalités du jeu et communication entre les classes

3.1. Principe du jeu
Notre projet est un roguelike 2D dynamique dont l'objectif est de survivre le plus longtemps possible face à des vagues d'ennemis de plus en plus difficiles.
Le joueur peut :

Se déplacer dans les quatre directions (haut, bas, gauche, droite).

Tirer des projectiles pour éliminer les ennemis.

Récupérer des boosts (santé, vitesse, tir amélioré) en touchant des objets apparaissant toutes les 30 secondes.

Affronter un boss final après un certain temps de jeu (2 min 30).

Continuer à survivre après le boss avec des ennemis encore plus difficiles.

Le score est calculé en fonction du temps de survie.

3.2. Fonctionnalités principales et leur logique de codage

Fonctionnalité Comment c'est implémenté
Déplacements du joueur Gérés dans la méthode handleInput() de Player, en fonction des touches pressées (ZQSD ou flèches). Blocage aux bords de l'écran pour éviter de sortir de la carte.
Tirs du joueur Lorsque la touche ESPACE est appuyée, un objet Projectile est instancié dans la direction du dernier déplacement.

Vagues d'ennemis Le WaveController gère l'enchaînement logique des vagues, en fonction du temps (BasicWave, BlueYellowOnlyWave, RedOnlyWave, FullWave).
Boss final Après 2 min 30 de jeu, un BossDuck apparaît au centre de l'écran. Il doit être éliminé pour revenir aux vagues normales.
Boosts (bonus) Le ItemManager génère aléatoirement un HealthBoost, SpeedBoost ou ProjectileBoost toutes les 30 secondes. Le joueur doit toucher l'item pour obtenir son effet.
Collision Les collisions sont gérées par la classe Hitbox. Un simple overlaps() permet de savoir si deux entités se touchent (projectile avec ennemi, joueur avec boost ou ennemi).
Chronomètre La classe Timer suit le temps écoulé depuis le début du jeu pour : afficher le temps à l'écran, déclencher les vagues, et calculer le score.
Meilleur score À chaque fin de partie, le score est comparé à celui sauvegardé dans best_score.json via BestScoreManager. Le meilleur est affiché si battu.
3.3. Communication entre les classes
Nous avons conçu des communications claires et maîtrisées entre les classes :

GameScreen est le chef d'orchestre :

Il appelle toutes les mises à jour (update()) et tous les dessins (render()).

Il gère les interactions principales entre Player, WaveController, ItemManager, Projectile, EnemyBase.

Player communique avec :

Projectile pour créer des tirs.

ItemManager pour recevoir des boosts.

WaveController indirectement via les combats contre les ennemis.

WaveController utilise :

Des instances de WaveManager pour moduler les types d'ennemis en fonction du temps.

ItemManager utilise :

ItemBase pour gérer tous les boosts de manière polymorphique.

Collision entre Player et ItemBase pour déclencher les effets (applyEffect()).

EnemyBase est une interface, ce qui permet à toutes les classes d'ennemis (RedDuckEnemy, BlueDuckEnemy, BossDuck) d'avoir un comportement standardisé tout en étant spécialisées si besoin.

3.4. Technologies et outils utilisés
Java 17 : langage principal du projet.

LibGDX : bibliothèque utilisée pour tout ce qui est affichage, input et boucle de jeu.

JUnit 5 : pour écrire les tests unitaires et d'intégration.

Mockito : pour simuler (mock) des comportements dans certains tests difficiles (ex: collisions).

Git : pour la gestion de version et le travail en groupe.

Gradle : pour la gestion du build et des dépendances.

Partie 4 : Mise en place des tests et validation du projet

4.1. Objectif des tests

Afin d'assurer la stabilité, la fiabilité et la maintenabilité du projet, nous avons choisi de :

Écrire des tests unitaires sur les classes contenant de la logique métier importante.

Mettre en place des tests d'intégration pour vérifier les interactions entre les principales fonctionnalités.

Réaliser des tests manuels sur toute la partie graphique et les comportements complexes liés à l'affichage.

Notre objectif était de garantir que chaque fonctionnalité individuelle fonctionne correctement et que l'ensemble du jeu se comporte comme prévu en conditions réelles.

4.2. Organisation des tests

Nous avons organisé nos tests dans un dossier distinct :

swift
Copier
Modifier
src/test/java/com/mygdx/roguelikeproject

en respectant l'architecture du projet pour garder une cohérence entre le code de production et le code de test.

Nous avons utilisé :

JUnit 5 pour tous les tests unitaires et d'intégration.

Mockito pour "mock" (simuler) certaines classes difficiles à instancier (par exemple, pour simuler un Player dans les tests d'items).

Assertions classiques (assertEquals, assertTrue, assertFalse) pour valider les comportements.

4.3. Tests unitaires réalisés
Nous avons ciblé les classes critiques :

Classe testée	Ce qu'on a testé
Damageable	- Prise de dégâts, mort - Invincibilité temporaire après dégâts - Heal sans dépasser le max
Projectile	- Déplacement selon la direction - Système de boost du projectile - Sortie d'écran
Position	- Simplicité d'accès à x et y
ItemManager	- Spawn automatique d'items toutes les 30 secondes - Suppression des items collectés ou expirés
WaveController	- Enchaînement des vagues dans l'ordre - Apparition du boss - Reprise sur les vagues après mort du boss
Player	- Modification correcte de la vitesse - Activation/Désactivation du boost de tir - Dégâts et soins appliqués correctement
EnemyBase (interface)	- Simulation d'un comportement basique d'ennemi pour vérifier la compatibilité

✅ Tous les tests sont passés avec succès.
✅ Le projet reste fonctionnel après ajout des tests.

4.4. Tests d'intégration réalisés
Les tests d'intégration avaient pour but de vérifier les interactions entre plusieurs classes :

Test d'intégration	Ce qu'on a vérifié
Player + ItemManager	Le joueur peut récupérer un boost et l'effet est bien appliqué.
Projectile + EnemyBase	Un projectile détruit bien un ennemi au contact.
WaveController + EnemyBase	Les ennemis spawnent correctement dans les différentes vagues.
Timer + WaveController	Le changement d'état des vagues dépend bien du temps écoulé.

Pour certains de ces tests (notamment ceux dépendants de l'affichage ou du temps réel), nous avons utilisé des simulations en accélérant le temps (deltaTime) et en mockant certains objets.

4.5. Tests manuels
Certaines fonctionnalités graphiques n’étaient pas testables automatiquement avec JUnit :

Déplacement du joueur à l'écran.

Animation des sprites.

Apparition visuelle des boosts et leur collision réelle.

Affichage de la barre de vie du joueur et du boss.

Nous avons donc réalisé des tests manuels, en jouant au jeu dans différentes conditions :

Avec peu d'ennemis / beaucoup d'ennemis.

Avec boost activés / non activés.

En touchant tous les types de boosts.

En survivant jusqu'au boss puis au-delà.

Observation :
Le jeu se comporte de manière stable, fluide et conforme aux attentes même avec beaucoup d'entités à l'écran.

4.6. Utilisation de Constants.java
Pour sécuriser les valeurs importantes (ex: vitesse du joueur, dégâts, dimensions des objets, délais entre vagues), nous avons utilisé un fichier unique :

mathematica
Copier
Modifier
utils/Constants.java
Cela permet :

De modifier facilement les paramètres du jeu sans toucher à plusieurs classes.

De garder de la cohérence et réduire les erreurs humaines.

C’est une très bonne pratique sur des projets de cette taille et encore plus utile pour des évolutions futures.

Partie 5 : Retours d'expérience et pistes d'amélioration

5.1. Retours sur l'expérience du projet
Ce projet nous a permis d’acquérir une expérience concrète en développement de jeu vidéo 2D avec Java et LibGDX, en mettant en œuvre :

Des pratiques professionnelles de conception propre (clean code, architecture modulable).

L'importance de prévoir et anticiper les évolutions d'un projet pour éviter de devoir réécrire beaucoup de code.

Une rigueur nouvelle sur les tests unitaires et d'intégration, rarement pratiqués dans ce type de projet étudiant.

5.2. Difficultés rencontrées et solutions apportées
Implémentation des items
Une difficulté majeure a été l'ajout des objets "boosts" (santé, vitesse, tirs améliorés) :

Ils ont été intégrés tardivement dans le projet.

Cela a nécessité de modifier plusieurs classes déjà terminées (ex: GameScreen, Player), ce qui a mis en lumière l'importance de bien anticiper les spécifications dès la phase de conception.

Cette expérience nous a appris à prévoir des points d'extension dans le code, même pour des fonctionnalités qui pourraient arriver plus tard.

Collaboration avec Git
Un autre point important a été la gestion du travail en groupe via Git :

Certains membres n’avaient pas ou peu de compétences Git.

Nous avons dû mettre en place des règles simples (branches claires, commits réguliers) et former certains membres sur les bases de l'outil.

Ce projet nous a renforcés dans l'utilisation de Git en équipe et dans l'importance d'une bonne communication technique.

5.3. Points forts de notre projet
Notre projet est selon nous :

Bien structuré : Les packages (entities, managers, utils, screens) permettent de s'y retrouver facilement.

Évolutif : Grâce à une architecture pensée pour ajouter facilement de nouveaux ennemis, de nouveaux boosts, ou d’autres vagues sans tout casser.

Testé : Des tests unitaires sérieux et des tests d'intégration sur les fonctionnalités critiques assurent la stabilité du projet.

Optimisé : Utilisation de Constants.java pour centraliser les réglages, bonne gestion mémoire (dispose() propre).

5.4. Pistes d'amélioration
Même si notre projet est solide, voici ce que nous pourrions améliorer :

Amélioration possible Pourquoi ?

Gestion plus fine des collisions Aujourd’hui les collisions sont basiques (carrées) ; un système plus avancé (cercle, polygone) améliorerait la précision.
Ajout d’effets visuels et sonores Rendre le jeu plus immersif (explosions, sons de tirs, etc).
Système de niveaux / progression Au lieu de rester sur des vagues infinies, avoir des niveaux avec une difficulté croissante.
Optimiser l'affichage pour de très grandes vagues Implémenter un "culling" pour ne pas afficher les objets trop loin du joueur.
Tests automatiques de rendering Utiliser des outils spécifiques pour tester automatiquement le rendu graphique sans intervention humaine.
🎯 Conclusion de cette partie :
Nous avons non seulement réussi à produire un jeu fonctionnel et stable, mais surtout :

Nous avons acquis des compétences professionnelles sur l'architecture, les tests, et la collaboration.

Nous avons appris par l’expérience à mieux prévoir, coder plus proprement, et tester méthodiquement.

Ce projet peut servir de base solide pour de futurs développements plus complexes.

Conclusion générale

Au terme de ce projet de développement d'un jeu vidéo en Java avec la librairie LibGDX, nous avons pu concevoir un jeu 2D structuré, fonctionnel et testé, tout en respectant les bonnes pratiques de l'industrie.

Notre travail s'est appuyé sur :

Une architecture propre et modulaire, permettant une grande évolutivité future.

Un effort particulier sur les tests unitaires et d'intégration, garantissant une meilleure stabilité du projet.

Une réflexion poussée sur la conception logicielle, favorisant la maintenance et la compréhension du code.

✅ Objectifs atteints
Création d'un gameplay solide : un joueur évoluant sur une carte, affrontant différentes vagues d'ennemis avec des mécaniques de boosts (santé, vitesse, projectiles).

Intégration complète des fonctionnalités : gestion des collisions, des animations, de la difficulté progressive avec l'arrivée d'un boss, du score et des meilleurs scores sauvegardés.

Tests rigoureux : validation du fonctionnement correct de tous les éléments critiques (santé, vitesse, ennemis, vagues, etc).

Travail collaboratif efficace malgré les difficultés initiales sur Git.

🎯 Ce que ce projet nous a appris
Ce projet a représenté un véritable exercice de professionnalisation :

Anticiper les évolutions futures pour éviter de lourdes modifications (exemple : système d'objets ajouté tardivement).

Tester sérieusement chaque composant du jeu pour gagner en fiabilité.

Collaborer efficacement via des outils de gestion de version.

Concevoir un code évolutif en pensant dès le départ à la séparation des responsabilités.

🚀 Perspectives d'évolution
Ce projet constitue une base très solide pour aller encore plus loin :

Ajouter de nouvelles mécaniques de jeu (niveaux, compétences spéciales...).

Améliorer l'immersion (musique, effets sonores, graphismes enrichis).

Déployer des tests automatiques visuels pour valider l'affichage graphique.

Évoluer vers une version multijoueur locale ou en ligne.

✨ Bilan final
Nous sommes fiers d’avoir réalisé un projet :

Sérieux dans sa conception,

Ambitieux dans son architecture,

Et qui respecte les standards professionnels en termes de qualité de code et de stabilité.

Cette expérience nous a fait progresser techniquement, mais aussi humainement, dans la gestion d’un projet informatique complet.