AIDARA ASNA
WANG Haozhe

Backpack Hero – Manuel Développeur

Introduction

Ce manuel décrit l'architecture interne, les choix techniques et les principes d'implémentation utilisés pour développer notre version du jeu *Backpack Hero*.

Dans ce rapport, nous allons expliquer les choix effectués afin d'aboutir à la réalisation du Jeu **Backpack Hero**.

Architecture générale du Projet

1. Modèle (package modélisation)

Ce package contient toute la logique du jeu, indépendante de l'affichage.
Il regroupe les éléments suivants :

Le héros

Classe : Hero

Responsabilités :

- gérer les **statistiques** du héros :
  - points de vie (hp, maxHp),
  - énergie par tour (energie),
  - mana (mana, maxMana),
  - expérience (exp, expMax),
  - niveau (niveau),
  - or (or),
  - blocage / armure temporaire (block) ;
- appliquer les opérations de combat :
  - damage(int dmg) : applique des dégâts en tenant compte du bloc ;
  - heal(int val) : soigne le héros sans dépasser maxHp ;
  - defend(int b) : ajoute de la protection pour le tour ;
- gérer la **mana** :
  - recharge() : recharge la mana à maxMana ;
  - use(int val) : consomme de la mana ;
  - charge(int m) : augmente la mana actuelle (pierres de mana, etc.) ;
- gérer l'**énergie** :
  - startCombat() : initialise l'énergie et la mana en début de combat ;
  - rechargerCombat() : remet l'énergie à 3 au début de chaque tour ;

- cost(int cost) / ifcost(int cost) : vérifie et consomme l'énergie ;

- gérer le **sac** (Backpack sac) :
  - getBackpack() / sac() : accès au sac à dos ;
  - updateStatsFromBackpack() : met à jour les statistiques (notamment la mana max) en fonction du contenu du sac ;
- gérer la **position dans le donjon** via Position :
  - étage courant ;
  - coordonnées dans l'étape.

Le sac à dos

Classe : Backpack

Responsabilités :

- représenter le sac comme une **grille de cases** gérée par une Map<Coord, Item> :
  - les clés sont des coordonnées (Coord),
  - la valeur est soit null, soit un objet (Item) occupant cette case ;
- gérer la **capacité** du sac et son évolution :
  - capacité initiale : 3 lignes × 5 colonnes (phase 1) ;
  - possibilité d'ajouter des cases avec allouerCase(Coord c) pour les montées de niveau ;
- gérer le **placement des objets** :
  - peutPlacer(Item item) : vérifie si la forme de l'objet (plusieurs cases) rentre dans les cases libres ;
  - placer(Item item) : place physiquement l'objet dans le sac ;
- gérer les **malédictions** :
  - peutPlacerMalediction(Item item) et PlacerMalediction(Item item) ;
  - remplace les objets existants si nécessaire, selon les règles du sujet ;
- gérer la **monnaie** :
  - recherche d'objets Gold dans le sac ;
  - RefreshMonnaie() : supprime les pièces d'or dont la quantité est devenue nulle.

Le choix d'une Map<Coord, Item> plutôt qu'un simple tableau fixe permet :

- d'ajouter des cases au sac à dos sans restructurer toute la classe ;
- de manipuler facilement des formes d'objets complexes (armes qui occupent plusieurs cases).

Les objets

Interfaces : Item
Implémentations (ou classes les utilisant) : Weapon, Gold, ItemMalediction, etc.

Caractéristiques :

- Chaque objet possède une forme (List<Coord>) et une position (offsetCoord).
- Les objets appliquent leurs effets via des classes dédiées (Effect_Attack, Effect_Def, etc.).
- Le polymorphisme supprime le besoin de tester les types d'objets dans le code.

Classe : Weapon

- Attributs principaux :
  - name : nom de l'arme ;
  - cost : coût en énergie par utilisation ;
  - mana : coût en mana par utilisation ;
  - estConsommable : indique si l'objet disparaît après usage (ex : pierre de mana consommable) ;
  - forme : liste de cases occupées dans le sac ;
  - effects : liste d'objets Effect appliqués à l'utilisation.
- Méthode clé :
  - utiliser(Hero hero, Ennemi ennemi, Combat combat) :
    - vérifie l'énergie et la mana disponibles ;
    - consomme les coûts ;
    - applique chaque effet sur le héros, l'ennemi, ou le combat.

Système d'effets :

- Interfaces et classes comme Effect, Effect_Attack, Effect_Def, Effect_Charge, etc.
- Chaque Effect sait comment s'appliquer :
  - par exemple, Effect_Attack enlève des PV à l'ennemi,
  - Effect_Def augmente le bloc du héros,
  - Effect_Charge augmente la mana.

Les ennemis

Classes : Ennemi, EnnemisBase

Responsabilités :

- stocker les PV, dégâts, comportements,
- choisir une action (Action()),
- appliquer l'action sur le héros.

Combat

Classe : Combat

- regroupe :
  - un Hero ;
  - une liste d'ennemis ;
- fournit des méthodes pour :
  - récupérer les ennemis vivants ;
  - vérifier l'état du combat : héros vivant, ennemis morts, etc.
  - gérer le passage de tour (héros → ennemis → héros).

Enum : CombatState

- HERO_TURN / ENEMY_TURN
  Utilisé par l'interface graphique pour savoir qui doit jouer.

Le donjon

Classes : Donjon, Etape, Room, RoomBase, Coord, RoomType

Concepts :

- Un donjon est constitué de plusieurs **étapes**.
- Chaque étape contient une grille dynamique de **salles**.
- Chaque salle possède un type (ENEMY, TREASURE, MERCHANT, …) et une action associée (Enter).
- RoomBase fournit un ensemble de modèles réutilisables pour la génération aléatoire.

Le système d'entrée de salle

Classes : Enter_Combat, Enter_Market, Enter_Healer, Enter_Tresor, Enter_moveEtape, etc.

Idée proposée : Chaque type de salle possède sa propre classe **d'effet d'entrée**, évitant les gros switch/case dans Room.

2. Interface graphique (package App, controller, view)

Classe principale : Main.java

Responsabilités :

- gérer la boucle de rendu,
- afficher les différents écrans (menu, donjon, combat, marchand, trésor, guérisseur),
- traiter les événements clavier/souris,
- relier les actions graphiques au modèle (Hero, Combat, etc.).

Écrans gérés

Combat graphique

Le combat repose sur :

- un rendu du sac à dos dynamique,
- un affichage du héros et de l'ennemi,
- un état CombatState (tour du héros → tour de l'ennemi),
- une gestion par clic des objets du sac.

Le moteur graphique appelle le modèle (Weapon.utiliser, Ennemi.executerAction, etc.) sans contenir la logique de combat.

Génération du donjon (package modelisation.Generation)

Le donjon est généré aléatoirement :

- génération d'un nombre aléatoire de salles,
- affectation d'un type de salle basé sur des templates (RoomBase),
- placement des liaisons entre salles,
- mise à jour des salles voisines accessibles.

Objectifs :

- Variabilité d'une partie à l'autre,
- Structure cohérente et explorée progressivement par le héros.

Gestion des salles

Interface : Enter

- **Méthode : void apply(Hero h);**

Implémentations :

- Enter_Combat : crée un Combat et prépare un affrontement ;

- Enter_Market : ouvre un marchand, avec une liste d'objets à acheter ;
- Enter_Healer : soigne le héros ;
- Enter_Tresor : ajoute des objets au sac du héros ;
- Enter_moveEtape : permet de passer à l'étage suivant.

Choix techniques principaux

Séparation modèle / vue

Toute la logique du jeu est isolée dans modelisation, permettant :

- la maintenabilité,
- la réutilisation,
- le remplacement de l'interface graphique sans toucher au moteur du jeu.

Système d'objets flexible

Chaque objet gère ses effets lui-même via Effect, ce qui permet :

- d'ajouter facilement de nouveaux objets,
- de ne pas modifier Hero.use,
- de réduire la complexité du code.

Donjon basé sur des templates

Avantage :

- cohérence entre les parties,
- génération pseudo-aléatoire maîtrisée,
- séparation claire entre données (RoomBase) et logique (Generation).

Interface graphique : package App **(Main.java)**

Voici les étapes entamées pour obtenir l'interface graphique de nôtre jeu :

- initialiser le modèle du jeu (Hero, Donjon, etc.) ;
- dessiner les différents écrans :
  - **menu principal** ;
  - **carte du donjon** ;
  - **écran de combat** ;
  - **marchand** ;
  - **guérisseur** ;

- **trésor** ;
- levelup;
- Gagne/Mort;

- traiter les événements clavier/souris :
  - flèches et espace pour le menu ;
  - clic sur une salle pour se déplacer ;
  - clic sur un objet dans le sac pour l'utiliser en combat ;
  - clic sur les boutons ("Fin du tour", "Retour", "Continuer", "OK").

Conclusion

Ce rapport présente l'architecture et les choix techniques du projet *Backpack Hero*.
L'application repose sur une organisation claire entre le modèle, l'interface graphique et la génération du donjon.
Les mécanismes de polymorphisme, de modularité et de séparation des responsabilités permettent une évolution et une maintenance efficace du code.
Ce projet fournit ainsi une base solide pour toute extension future du gameplay.