

# MOCI

## Rapport RPG

Ewan Decima

Décembre 2024

### Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Design Pattern</b>	<b>2</b>
2.1	Singleton . . . . .	2
2.2	Factory . . . . .	3
2.3	Builder . . . . .	3
2.4	Prototype . . . . .	3
2.5	Visitor . . . . .	4
2.6	Strategy . . . . .	4
2.7	Observer . . . . .	5
2.8	Decorator . . . . .	5
2.9	Command . . . . .	5
2.10	State . . . . .	6
<b>3</b>	<b>Diagrammes</b>	<b>6</b>
3.1	UML . . . . .	6
<b>4</b>	<b>Interface graphique</b>	<b>7</b>
<b>5</b>	<b>Sources</b>	<b>9</b>

# 1 Introduction

Dans le cadre du développement de systèmes logiciels complexes, les design patterns jouent un rôle crucial en offrant des solutions éprouvées pour résoudre des problèmes de conception récurrents. Ces patterns facilitent la maintenance, l'évolutivité et la lisibilité du code tout en encourageant une architecture modulaire et flexible.

Ce rapport s'inscrit dans le cadre d'un projet visant à créer un prototype de jeu de rôle (RPG) simple. L'objectif principal est de mettre en œuvre divers patterns de conception pour répondre aux exigences du jeu, telles que la gestion des personnages, des équipes, des interactions, et des états. Chacun de ces patterns est utilisé pour résoudre un problème spécifique dans l'architecture du jeu, tout en maintenant un faible couplage entre les composants et en assurant une extensibilité future.

Le présent document explore en détail les patterns utilisés, leur rôle dans l'architecture globale du jeu, et leur contribution à la résolution des problèmes de conception.

## 2 Design Pattern

### 2.1 Singleton

En génie logiciel, le singleton est un patron de conception, appartenant à la catégorie des patrons de création, dont l'objectif est de restreindre linstanciation d'une classe à un seul objet.

- **Problème de conception :** Besoin de gérer des paramètres globaux du jeu de manière centralisée et unique, sans risque de créations multiples et potentiellement incohérentes.
- **Rôle dans l'architecture :**
  - Centraliser la configuration globale du jeu.
  - Contrôler les paramètres fondamentaux comme le niveau de difficulté.
  - Assurer une configuration cohérente et unique pour toute l'application.
- **Solution :** Le Singleton résout le problème de conception en :
  - Empêchant la création de multiples instances de configuration.
  - Fournissant un point d'accès global unique via une méthode statique.
  - Garantissant que tous les composants du jeu utilisent exactement la même configuration.
  - Contrôlant strictement l'accès et la modification des paramètres globaux.
  - Évitant les incohérences potentielles dues à des configurations multiples et divergentes.

## 2.2 Factory

Le pattern Prototype permet de copier des objets existants sans dépendre de leurs classes concrètes.

- **Problème de conception :** Besoin de créer rapidement des équipes similaires sans répéter un processus de création complexe.
- **Rôle dans l'architecture :**
  - Faciliter la duplication d'équipes types.
  - Réduire la complexité de création d'équipes multiples.
  - Permettre des variations rapides d'équipes existantes.
- **Solution :** Le Prototype résout le problème de conception en :
  - Fournissant un mécanisme de clonage pour les structures d'équipes.
  - Évitant la reconstruction complète d'équipes similaires.
  - Permettant la création rapide de variations d'équipes existantes.
  - Réduisant la redondance dans le code de création d'équipes.
  - Offrant une flexibilité maximale dans la génération de nouvelles équipes.

## 2.3 Builder

Le pattern Builder sépare la construction d'un objet complexe de sa représentation, permettant la même construction de produire différentes représentations.

- **Problème de conception :** Gestion de la création progressive et flexible d'équipes avec potentiel d'ajout de différents éléments.
- **Rôle dans l'architecture :**
  - Construire des équipes étape par étape.
  - Permettre l'ajout dynamique de personnages.
  - Préparer l'extensibilité pour l'ajout d'autres éléments (véhicules, ...).
- **Solution :** Le Builder résout le problème de conception en :
  - Décomposant la construction complexe d'équipes en étapes simples et contrôlées.
  - Permettant l'ajout progressif et configurable de personnages
  - Séparant la logique de construction de la représentation finale de l'équipe.
  - Offrant une grande flexibilité pour l'ajout de nouveaux types d'éléments
  - Simplifiant la création de configurations d'équipes complexes et variées.

## 2.4 Prototype

- **Problème de conception :** Le pattern Prototype permet de copier des objets existants sans dépendre de leurs classes concrètes.
- **Rôle dans l'architecture :**
  - Faciliter la duplication d'équipes types.
  - Réduire la complexité de création d'équipes multiples.
  - Permettre des variations rapides d'équipes existantes.

- **Solution :** Le Prototype résout le problème de conception en :
  - Fournissant un mécanisme de clonage pour les structures d'équipes.
  - Évitant la reconstruction complète d'équipes similaires.
  - Permettant la création rapide de variations d'équipes existantes.
  - Réduisant la redondance dans le code de création d'équipes.
  - Offrant une flexibilité maximale dans la génération de nouvelles équipes.

## 2.5 Visitor

Le pattern Visiteur permet d'ajouter de nouvelles opérations à des objets sans modifier leurs classes existantes.

- **Problème de conception :** Nécessité de réaliser des opérations complexes et variées sur des ensembles de personnages sans altérer leurs structures internes.
- **Rôle dans l'architecture :**
  - Effectuer des opérations transversales sur les personnages
  - Ajouter des comportements sans modifier les classes existantes
  - Implémenter des visiteurs spécialisés (buff, dommage, guérison)
- **Solution :** Le pattern Visiteur résout le problème de conception en :
  - Introduisant une séparation claire entre les opérations et la structure des objets.
  - Permettant l'ajout de nouvelles opérations sans modifier les classes de personnages.
  - Centralisant les algorithmes complexes dans des classes de visiteurs dédiées.
  - Offrant une grande flexibilité pour implémenter de nouveaux types d'opérations
  - Facilitant l'extensibilité du système de traitement des personnages.

## 2.6 Strategy

Le pattern Stratégie définit une famille d'algorithmes, encapsule chacun d'eux et les rend interchangeables.

- **Problème de conception :** Besoin de comportements de combat dynamiques et variés, avec possibilité de changement à l'exécution.
- **Rôle dans l'architecture :**
  - Implémenter des stratégies de combat différencierées.
  - Permettre des changements dynamiques de stratégie.
  - Varier l'agressivité et la défense des personnages.
- **Solution :** Le pattern Stratégie résout le problème de conception en :
  - Encapsulant des algorithmes de combat distincts.
  - Permettant le changement de stratégie à la volée sans modifier la structure du personnage.
  - Séparant les algorithmes de combat de la classe de personnage.
  - Facilitant l'ajout de nouvelles stratégies sans impact sur le code existant

- Offrant une grande flexibilité dans le comportement de combat.

## 2.7 Observer

Le pattern Observateur définit une dépendance un-à-plusieurs entre objets pour que tous les dépendants soient notifiés automatiquement.

- **Problème de conception :** Gestion des notifications et réactions aux changements d'état des personnages de manière découpée et flexible.
- **Rôle dans l'architecture :**
  - Notifier des événements (niveau up, mort).
  - Créer un système de réaction aux changements.
  - Maintenir un couplage faible entre objets.
- **Solution :** Le pattern Observateur résout le problème de conception en :
  - Établissant un mécanisme de notification automatique et découpé.
  - Permettant à plusieurs observateurs de réagir à un même événement.
  - Évitant les dépendances rigides entre les objets qui changent d'état et ceux qui réagissent.
  - Facilitant l'ajout de nouveaux types d'observateurs sans modifier les classes existantes.
  - Rendant le système plus modulaire et extensible.

## 2.8 Decorator

Le pattern Décorateur permet d'ajouter dynamiquement des responsabilités à un objet de manière flexible.

- **Problème de conception :** Besoin d'étendre les fonctionnalités des personnages de manière dynamique et réversible.
- **Rôle dans l'architecture :**
  - Ajouter des améliorations aux personnages.
  - Modifier dynamiquement les attributs et comportements.
  - Permettre l'ajout et la suppression de capacités.
- **Solution :** Le pattern Décorateur résout le problème de conception en :
  - Permettant l'ajout de fonctionnalités sans modifier la classe originale.
  - Offrant une composition flexible plutôt qu'une inheritance rigide.
  - Autorisant l'ajout et la suppression dynamique de capacités.
  - Facilitant la création de combinaisons complexes d'améliorations.
  - Maintenant une grande flexibilité dans la personnalisation des personnages.

## 2.9 Command

Le pattern Commande encapsule une demande sous la forme d'un objet, permettant de paramétriser des clients avec différentes demandes.

- **Problème de conception :** Nécessité de gérer et de contrôler les actions de jeu de manière flexible et extensible.
- **Rôle dans l'architecture :**

- Encapsuler les actions de jeu.
- Permettre l'annulation et la répétition d'actions.
- Gérer une file d'exécution des commandes.
- **Solution :** Le pattern Commande résout le problème de conception en :
  - Transformant chaque action en un objet autonome.
  - Permettant la mise en file d'attente et l'exécution séquentielle des actions.
  - Facilitant l'implémentation de mécanismes d'annulation et de répétition.
  - Découplant l'émetteur d'une requête de son exécuteur.
  - Offrant une grande flexibilité dans la gestion des actions de jeu.

## 2.10 State

Le pattern État permet à un objet de modifier son comportement lorsque son état interne change.

- **Problème de conception :** Besoin de gérer des comportements complexes et changeants des personnages selon leur état.
- **Rôle dans l'architecture :**
  - Gérer les transitions d'états des personnages.
  - Modifier dynamiquement le comportement selon l'état.
  - Implémenter des états comme Mort, Blessé, Effrayé.

## 3 Diagrammes

### 3.1 UML

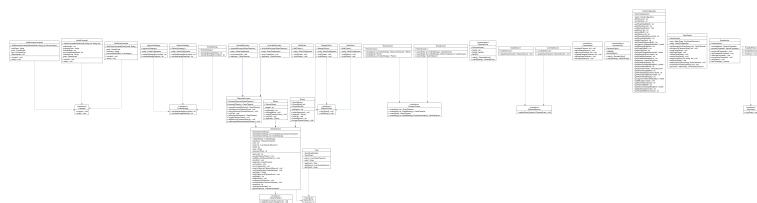


FIGURE 1 – Diagramme UML.

## 4 Interface graphique



FIGURE 2 – Écran titre du jeu, avec menu.

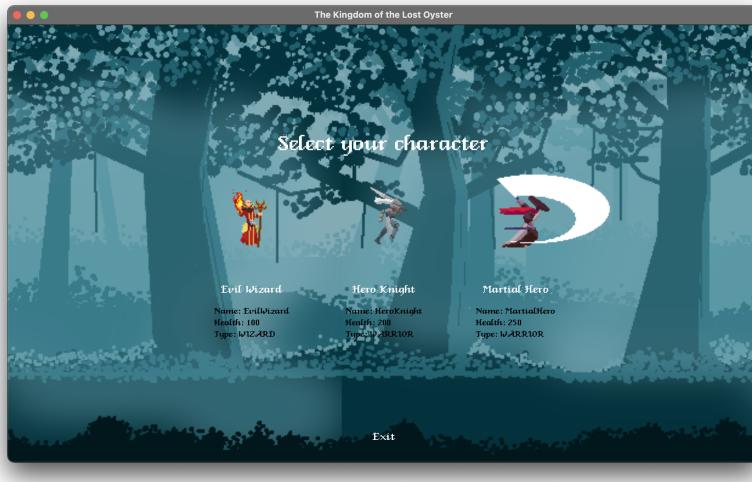


FIGURE 3 – Écran titre de sélection de personnages.



FIGURE 4 – Écran d'application des "Visitors".



FIGURE 5 – Écran titre de Game Over.

## 5 Sources

- <https://refactoring.guru/design-patterns/singleton>
- <https://refactoring.guru/design-patterns/factory-method>
- <https://refactoring.guru/design-patterns/prototype>
- <https://refactoring.guru/design-patterns/builder>
- <https://refactoring.guru/design-patterns/visitor>
- <https://refactoring.guru/design-patterns/strategy>
- <https://refactoring.guru/design-patterns/observer>
- <https://refactoring.guru/design-patterns/decorator>
- <https://refactoring.guru/design-patterns/command>
- <https://refactoring.guru/design-patterns/state>