Thomas Barseghian TP5

Hugo Leger TP5

Rapport de Conception : Évolutions et Architecture Finale - Bataille Navale

1. Introduction

Ce rapport présente l'architecture finale de notre implémentation de la bataille navale en Java. Suite au premier rendu, notre démarche de développement s'est concentrée sur le passage de la théorie à la pratique. Nous avons conservé la structure MVC (Modèle-Vue-Contrôleur) tout en ajustant certaines parties pour éviter la sur-conception et pour mieux gérer le flux d'exécution du jeu, notamment la gestion des tours et des attaques.

2. Évolutions Majeures et Refactoring

2.1. Simplification du système de Pièges (Suppression de la classe `Effect`)

Lors du premier rendu, nous avions théorisé un système complexe basé sur une classe générique Effect pour gérer les altérations d'état temporaires (comme l'effet du Tornado).

Constat : Lors de l'implémentation, nous avons réalisé que cette abstraction était disproportionnée pour le besoin réel. Créer une classe pour encapsuler un simple état binaire ajoutait de la lourdeur inutile au code.

Solution actuelle : Nous avons supprimé la classe Effect.

- L'état "sous l'effet d'une tornade" est désormais géré par un attribut booléen m_isTornadoed directement dans la classe Player.
- La logique de décompte des tours de l'effet est encapsulée directement dans la classe Tornado (attribut remainingScrambles), qui interagit avec le Grid et le Player.
- Gain : Cette approche respecte le principe KISS (Keep It Simple, Stupid) et réduit le couplage inutile.

2.2. Gestion du Flux de Jeu : Le Pattern Callback

L'un des défis majeurs a été la synchronisation entre l'action d'attaque (qui peut impliquer des animations ou des calculs) et la fin du tour.

Ajout : Nous avons introduit l'interface WeaponCallback et la méthode notifyFinished() dans la classe abstraite Weapon.

Fonctionnement :

- Lorsqu'une arme est utilisée (use()), elle effectue son action.
- Une fois l'action terminée, l'arme appelle m_callback.onAttackFinished().
- Le AbstractPlayerController implémente cette interface. Cela permet au contrôleur de ne passer au tour suivant (via TurnObserver) que lorsque l'arme a réellement fini son travail.

3. Architecture Détaillée

3.1. Le Modèle (Model)

Le modèle reste le cœur logique, mais il a été enrichi pour mieux stocker l'historique et les états.

- **Grid et Tile :** La structure granulaire a été conservée. Grid expose désormais des méthodes plus précises pour les observateurs (notifyTileHit), optimisant les mises à jour de la vue.
- **MoveData :** Nous avons ajouté la classe MoveData. Elle agit comme une structure de données dédiée à l'historique des coups (position, touche/raté, arme utilisée). Cela permet de séparer la logique de jeu de la mémorisation des actions passées (utile pour l'affichage et l'IA).
- **Hiérarchie des Objets :** Les interfaces ShipFactory, WeaponFactory et TrapFactory sont pleinement opérationnelles, permettant une instanciation flexible via le contrôleur ou la configuration.

3.2. Les Contrôleurs (Controller)

L'architecture des contrôleurs a été affinée pour mieux séparer la gestion globale du jeu de la gestion spécifique des joueurs.

- **GameController :** Il agit comme le chef d'orchestre global. Il implémente TurnObserver pour savoir quand un tour se termine, mais délègue la logique d'action aux sous-contrôleurs.
- **AbstractPlayerController :** Cette nouvelle classe abstraite factorise le code commun entre HumanController et AIController (gestion des observateurs, callbacks d'armes).
- **AIController vs HumanController :**
    - HumanController traduit les entrées UI (clics souris) en actions de jeu.
    - AIController utilise directement les stratégies de l'IA (attackWithTought ou attackRandom) définies dans la classe AI du modèle.

3.3. La Vue (View)

La vue suit strictement le pattern Observer pour se mettre à jour sans interroger le modèle en permanence.

- **Granularité des Observateurs :** Nous avons divisé les responsabilités. GridObserver écoute les changements sur la grille (tirs), tandis que PlayerObserver écoute les

changements d'inventaire (placement de bateaux, pièges). Cela évite de rafraîchir tout l'écran pour un simple changement de pixel.
- Une vue principale (MainVue) contient les autres vues pour que le changement de vue ne soit pas visible par le joueur.

4. Patterns de Conception Utilisés

En plus des patterns initiaux, l'architecture finale s'appuie sur :

1. **Observer :** De nouveaux Observer ont été ajouté depuis le premier rendu comme le TurnObserver qui est un observer situé dans le package “controller” car il sert à l'orchestration du jeu : il permet au HumanController de signaler au GameController que la séquence d'action est terminée, sans que le Modèle n'ait besoin d'être au courant de cette logique de tour par tour.
2. **Callback :** Pour gérer la fin asynchrone des attaques des armes via WeaponCallback.

5. Conclusion

Les ajustements réalisés pour ce second rendu ont permis d'optimiser le code, notamment en éliminant la sur-conception liée à la classe Effect au profit d'une logique plus directe, et en structurant rigoureusement la communication via des Interfaces et des Callbacks. Hormis cet ajustement nécessaire, l'architecture globale de notre Modèle s'est révélée très robuste : les choix de conception initiaux ont été validés par la phase d'implémentation, facilitant grandement la traduction du diagramme UML en code fonctionnel.