

Projet “carte aux trésors”

carbon^{IT}

Exercice pratique - La carte aux trésors



Table des matières

1.	Phase préliminaire	3
2.	Phase d'architecture	3
3.	Phase de développement	5
4.	Retour d'expérience sur le projet	6

1. Phase préliminaire

Avant de démarrer le projet, j'ai lu le sujet que j'ai ensuite condensé dans un fichier nommé « instructions.md ». Ce fichier a deux objectifs principaux. Dans un premier temps, il me permet d'assimiler le projet dans son intégralité en résumant les fonctionnalités principales de manière concise. Deuxièmement et pour terminer, il s'agit d'une trace écrite sur laquelle je peux revenir pendant la construction du projet pour revoir les fonctionnalités.

Cette étape terminée, j'ai ensuite architecturé le projet de sorte à minimiser les dépendances fonctionnelles et maximiser la modularité.

Le point d'entrée du projet se trouve dans le package main qui contient la classe Main. C'est ici qu'il est possible pour le développeur d'utiliser le « client package ».

2. Phase d'architecture

Pour répondre aux exigences de modularité et de minimisation de dépendances, j'ai adopté une architecture hexagonale en découpant le projet en plusieurs composants interchangeables. On se retrouve avec trois services principaux : IO (input output), core, client.

- **IO** : service de gestion de lecture et d'écriture de données
- **Core** : service de gestion des règles métiers (spécificités fonctionnelles du projet)
- **Client** : service d'interface pour les interactions avec le client (= dans notre contexte le développeur)

La dépendance la plus forte que l'on retrouve est celle du package **data** qui correspond aux structures de données propres à ce projet.

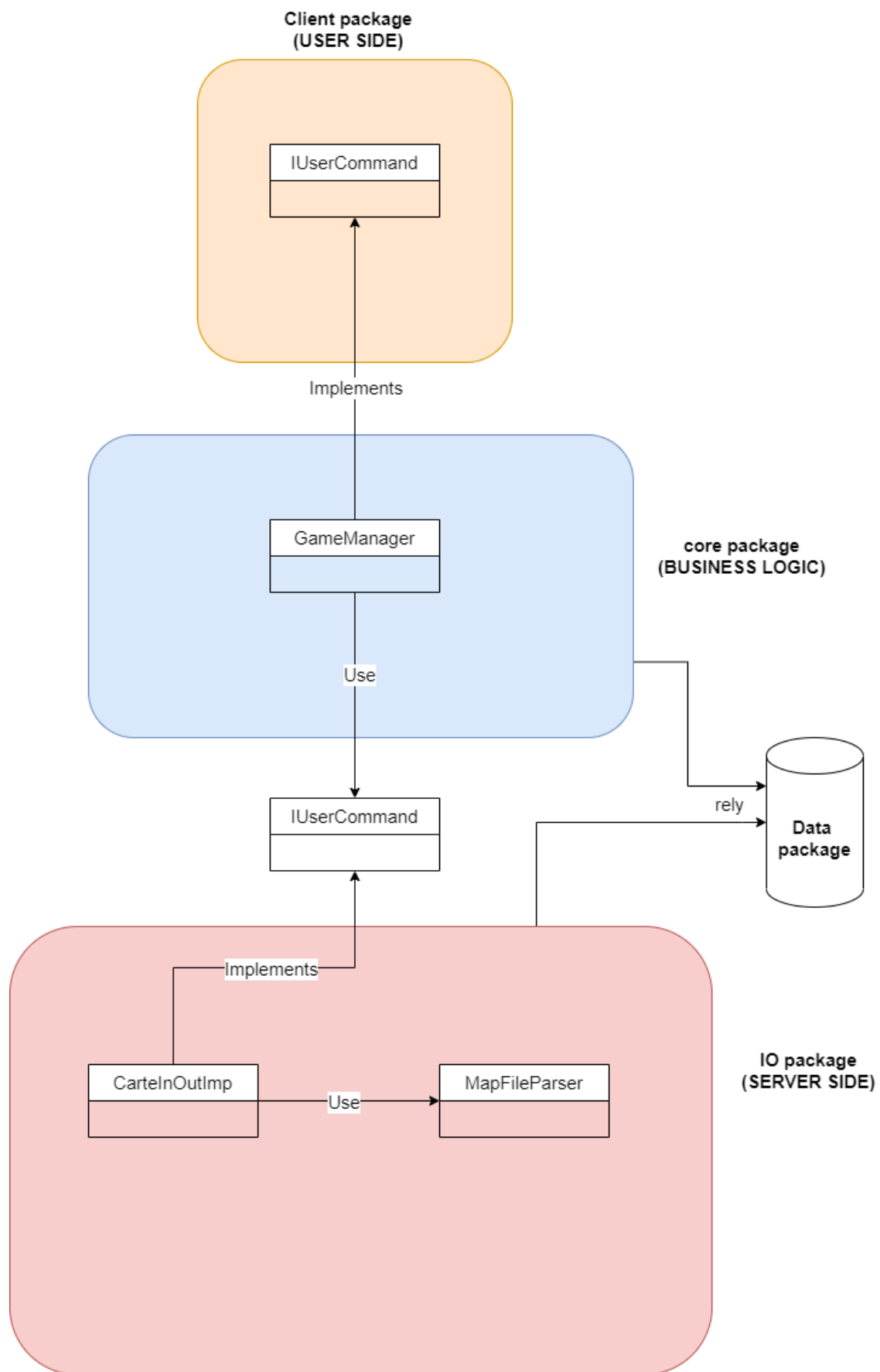


Figure 1 : schéma récapitulatif du fonctionnement

3. Phase de développement

Dans un esprit TDD, j'ai commencé par écrire les tests unitaires pour les différentes méthodes qui ont dû être testées pour assurer la pérennité du code dans le temps.

Ces tests unitaires sont rassemblés au sein du package **test** qui est lui-même composé de sous-dossiers respectivement associés aux trois autres packages cités plus haut.

Une fois les tests implémentés, je me suis lancé sur le développement du projet en le faisant package par package.

En tandem avec les tests unitaires, j'ai choisi l'outil **SonarQube** pour vérifier la qualité du code écrit.






Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
>  cartetresor	 84,5 %	2 294	421	2 715

Figure 2: code coverage metrics

Finished after 0,266 seconds

Runs: 47/47  Errors: 0  Failures: 0







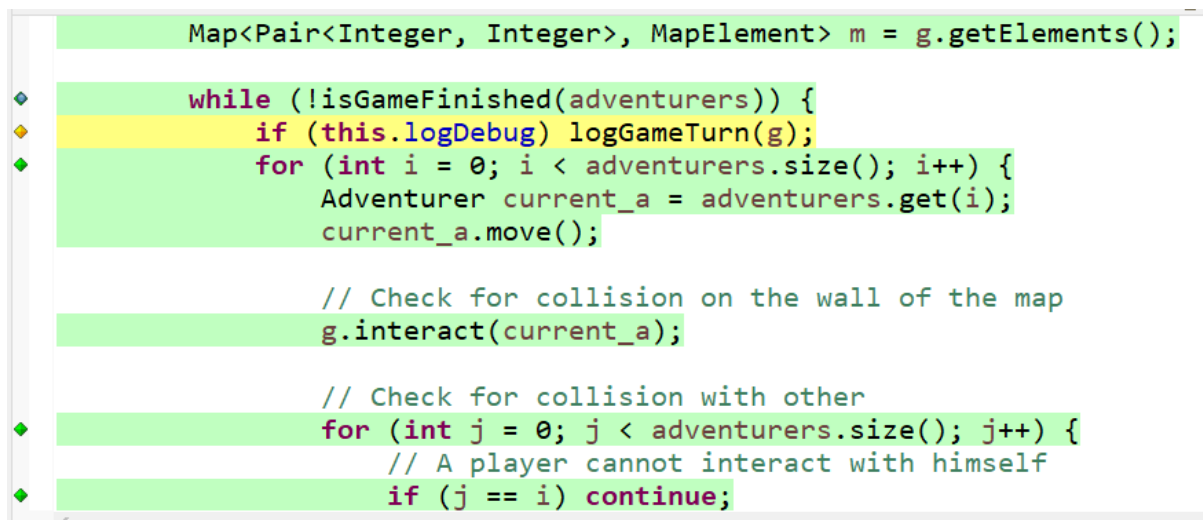
- >  CartelOTest [Runner: JUnit 5] (0,020 s)
- >  MapFileParserTest [Runner: JUnit 5] (0,044 s)
- >  GameManagerTest [Runner: JUnit 5] (0,055 s)
- >  DataDumpTest [Runner: JUnit 5] (0,003 s)

Figure 3: Unit testing metrics



```

Map<Pair<Integer, Integer>, MapElement> m = g.getElements();

while (!isGameFinished(adventurers)) {
    if (this.logDebug) logGameTurn(g);
    for (int i = 0; i < adventurers.size(); i++) {
        Adventurer current_a = adventurers.get(i);
        current_a.move();

        // Check for collision on the wall of the map
        g.interact(current_a);

        // Check for collision with other
        for (int j = 0; j < adventurers.size(); j++) {
            // A player cannot interact with himself
            if (j == i) continue;

```

Figure 4: Code coverage Syntax Highlighting

4. Retour d'expérience sur le projet

J'ai relevé plusieurs axes d'améliorations sur le projet :

- **Qualité du code** : plusieurs éléments soulevés par SonarQube peuvent être corrigés tel que la duplication de code (factoriser les éléments répétitifs à l'aide de méthodes)
- **Code coverage** : il est possible d'améliorer encore le pourcentage de couverture en ajoutant d'autres tests unitaires
- **Rapidité d'exécution** : il est possible de simuler plusieurs parties de manière successives. Il faut ajouter du parallélisme pour améliorer la rapidité de traitement (fait mais pas tester)
- **Aspect CI / CD** : il y a une perte de temps considérable en ce qui concerne les tests unitaires. Il aurait fallu intégrer un pipeline de test à chaque commit.