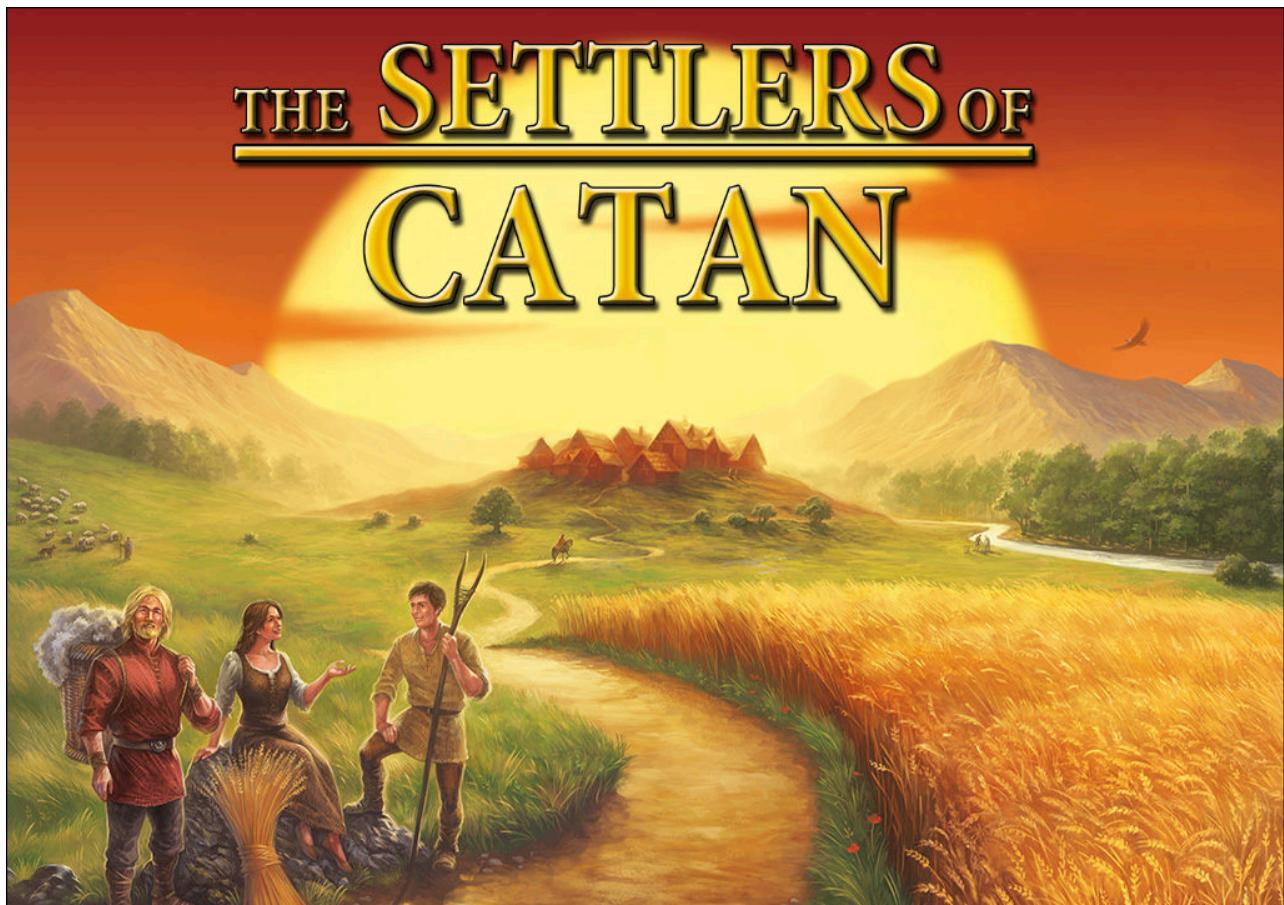


# Rapport Projet POOIG

## *Les Colons de Catane*

HU Yingqin (N° étudiant : 22014759)  
ZHANG Xirui (N° étudiant : 22116183)



---

## Introduction

Dans le cadre du projet POOIG, nous avons réalisé une implémentation en java du jeu de société, « Les Colons de Catane ». Le plateau représente une île avec de diverses ressources abondantes, île de Catane, les joueurs pourront collecter des ressources, et y construire des infrastructures. Les constructions des colonies et des villes, ainsi que l’acquisition de certaines cartes de développement apporteront des points de victoires, et le premier qui atteint 10 points sera le gagnant.

Ce rapport sera divisé en trois parties. Nous allons d’abord présenter les règles et les fonctionnalités que nous avons réalisées, ainsi que les difficultés rencontrées pendant ce processus et leurs solutions. Nous visualiserons ensuite le modèle des classes. Enfin, la dernière partie s’agira des éléments que nous n’avons pas réussi à implémenter.

# I. Règles et fonctionnalités réalisées

## 1.1 Éléments basiques du jeu

En lançant le jeu, on peut paramétriser le jeu : choisir le nombre des joueurs (3 ou 4 personnes) et le type de joueurs (humain ou AI), ainsi que personnaliser les noms des joueurs.



Figure 1

En cliquant sur « Nouveau jeu », on entre dans le plateau du jeu.

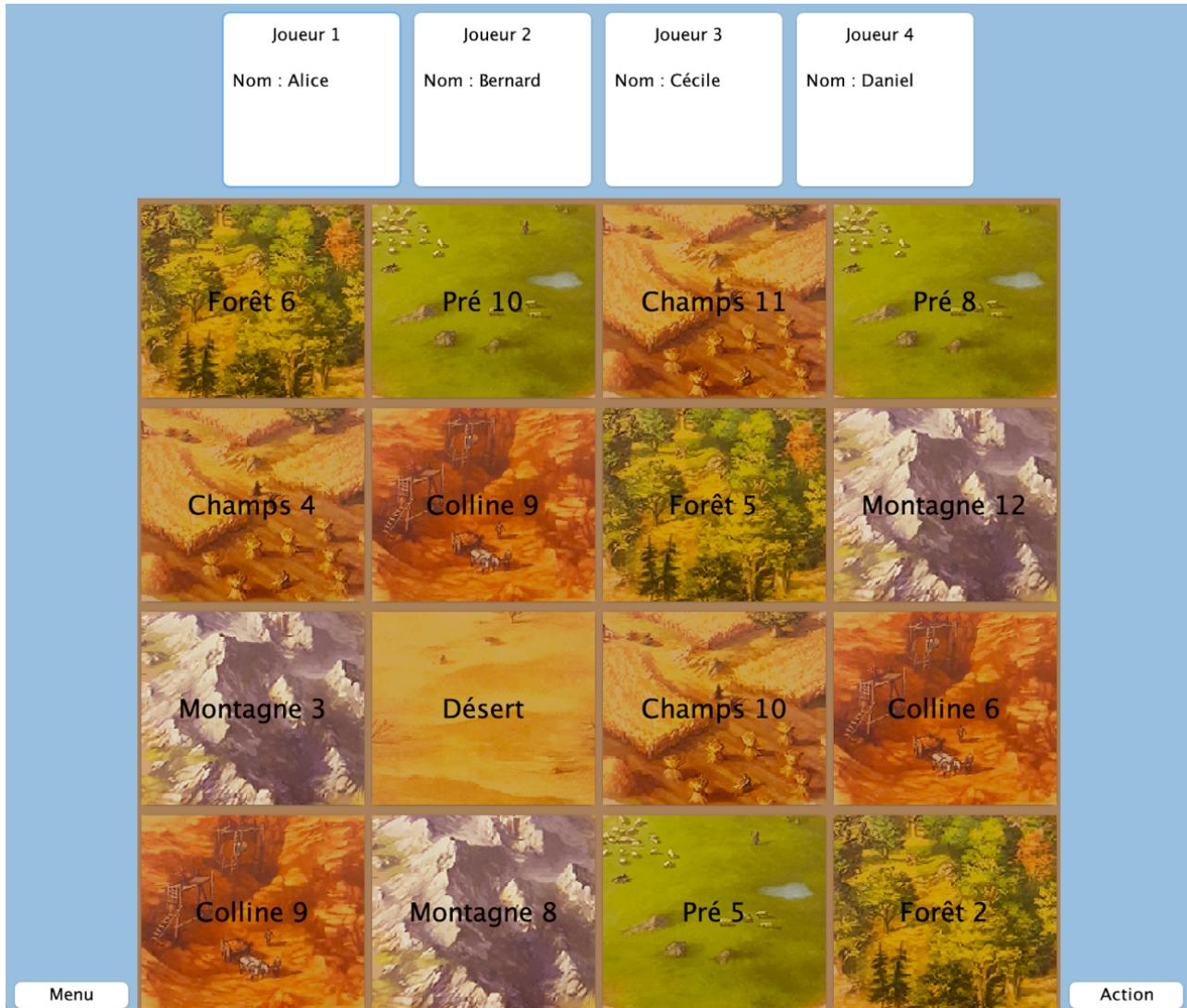


Figure 2

C'est un plateau carré en version simple. Le bleu clair au fond représente la mer. Les carrés en haut affiche les informations des joueurs, pour le moment, nous n'avons réussi qu'à afficher les noms des joueurs selon la personnalisation lors du paramétrage du jeu.

Les cases qui représentent des types de terrain différents doivent être distribuées de façon aléatoire, et nous avons écrit des méthodes pour le réaliser, cependant, elles n'ont pas fonctionné. Nous avons donc présenté ici dans la figure 2 le résultat idéal de la distribution des cases.

## 1.2 Les règles basiques

Nous avons implémenté les règles basiques : construction des routes, colonies et villes, consultation et gestion des ressources, événement de voleur en cas de 7 aux dés, gestion des cartes spéciales de développement et échange des ressources via les ports. Nous avons aussi implémenté un IA qui fait des choix aléatoires, des méthodes pour calculer le point de victoire, notamment pour les bonus comme « route la plus longue », « chevalier le plus puissant » et certaines cartes de développement. Parmi ces fonctions, la carte « route la plus longue » suscite notre intérêt particulier.

## 1.3 Route la plus longue

Ce bonus de deux points de victoire est distribué au joueur dont les routes consécutives sont les plus longues. Il y a deux détails auxquels nous devons faire attention : premièrement, il faut calculer le nombre des routes consécutives, au lieu du nombre total des routes ; deuxièmement, en cas de changement de détenteur, l'ancien détenteur de cette carte va perdre les bonus acquis.

Nous avons donc créé une méthode routeLaPlusLongue() pour juger si un joueur remplit les conditions.

```

public boolean routeLaPlusLongue(Plateau plateau) {
    Joueur nomJoueur = null;
    int maxPre = 0;
    for (Joueur joueur : plateau.getRoute().values()) {
        int maxSec = 0;
        for (Point2D[] routePre : plateau.getRoute().keySet()) {
            for (Point2D[] routeSec : plateau.getRoute().keySet()) {
                if (routePre[1] == routeSec[0]) {
                    maxSec++;
                }
            }
            if (maxSec > maxPre) {
                maxPre = maxSec;
            }
        }
        nomJoueur = joueur;
    }
    if (maxPre > nbRouteLaPlusLongue) {
        joueurRouteLaPlusLongue.setScore(joueurRouteLaPlusLongue.getScore() - 2);
        joueurRouteLaPlusLongue = nomJoueur;
        joueurRouteLaPlusLongue.setScore(joueurRouteLaPlusLongue.getScore() + 2);
        nbRouteLaPlusLongue = maxPre;
        return true;
    } else {
        return false;
    }
}

```

Figure 3

Nous avons enregistré toutes les routes dans un `HashMap<Point2D[ ], Joueur>`, chaque route est représentée par un array `Point2D[ ]` avec deux coordonnées (un point de début et un point final). Dans cette méthode, pour juger si deux routes sont consécutives, nous allons examiner si le point de début de la deuxième route est conforme au point final de la première route. Et on peut calculer ainsi le nombre des routes consécutives.

En même temps, le variable membre `joueurRouteLaPlusLongue` sera aussi mis à jour.

## 1.4 La surveillance de la souris

Pour une meilleure expérience de jeu, nous avons également mis en place une fonction de surveillance de la souris. Par exemple, lorsque la souris passe sur un bouton, la bordure du bouton devient jaune. Lorsque la souris clique sur un bouton, le texte de ce bouton devient plus gros.

On implémente d'abord l'interface `MouseListener` sur la classe `MenuVision`. Cette interface permet de recevoir les événements souris (pression, relâchement, clic, entrée et sortie) sur un composant. Après son implémentation, on utilise la méthode `addActionListener()` et redéfinit ses méthodes qui nous intéressent. Dans notre programme, nous avons utilisé les méthodes suivantes :

```
void mouseClicked(MouseEvent e)  
void mouseReleased(MouseEvent e)  
void mouseEntered(MouseEvent e)  
void mouseExited(MouseEvent e)
```

Et voici les effets que nous avons réalisé.



Commencer le jeu

Bouton en état normal



Commencer le jeu

Quand la souris passe sur le bouton



Commencer le jeu

Quand on clique sur le bouton

## II. Représentation graphique du modèle des classes

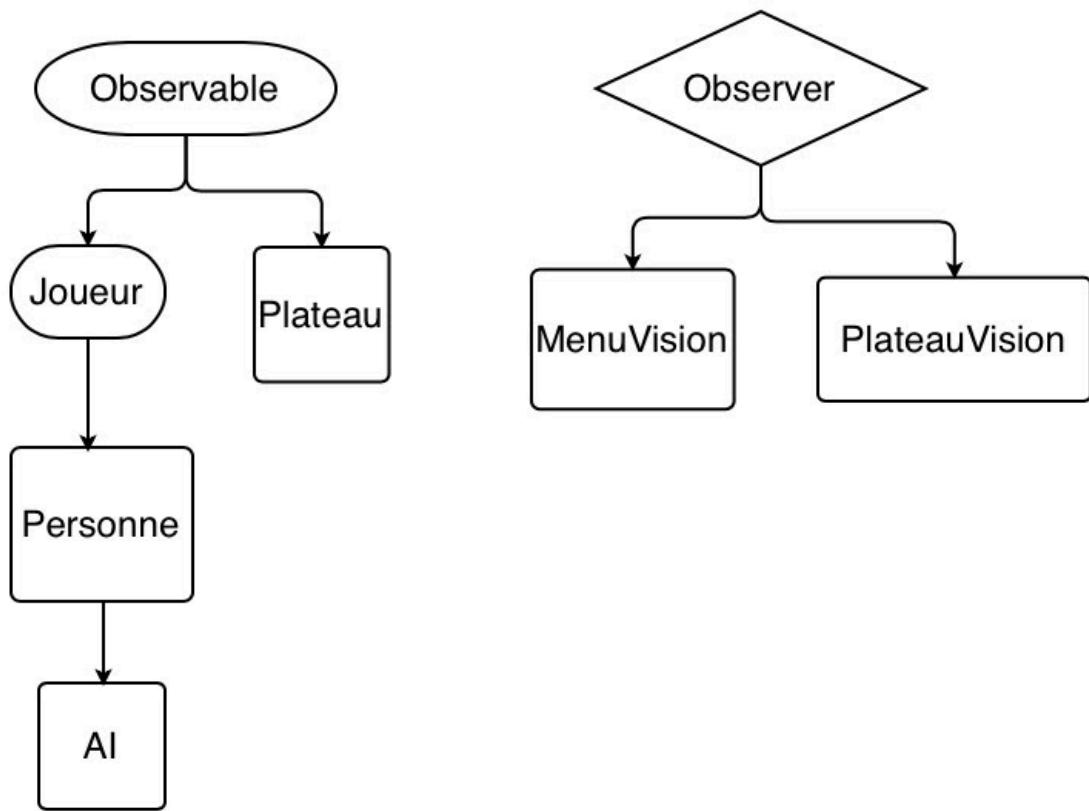


Figure 4

Dans notre programme, nous avons divisé trois packages : Catane (pour le modèle), GUI (pour les vues), Observer (Pour le pattern d'observateur). La structure principale de notre modèle est représentée par le graphe ci-dessus. L'ovale représente classe abstraite, losange représente l'interface, le carré représente les classes.

### **III. Eléments non implémentés**

#### **3.1 Sauvegarder et charger : Serializable et Thread**

Nous avons essayé de réaliser la fonctionnalité d'archivage et de chargement du jeu. Bien que nous n'ayons pas réussi, nous avons trouvé les méthodes pour l'implémenter : l'interface Serializable et la classe Thread.

L'interface Serializable est un procédé introduit dans le JDK version 1.1 et définie dans le package java.io pour mettre en œuvre des opérations de sérialisation sur les classes Java. L'interface Serializable ne possède pas de méthodes ou de champs, mais sert uniquement à identifier la sémantique de la sérialisation. Dans le processus de sérialisation, les membres publics et privés de l'objet (y compris les noms de classe) sont convertis en un flux d'octets, qui est ensuite écrit dans un flux de données et stocké sur un support de stockage, ce qui signifie généralement un fichier. L'utilisation de l'interface de sérialisation présente deux avantages. D'une part, l'objet sérialisé est enregistré dans un état binaire, ce qui permet d'obtenir l'indépendance de la plate-forme et d'obtenir l'objet par dessérialisation sans avoir à se préoccuper des exceptions d'affichage des données dues à des problèmes de plate-forme. D'autre part, la sérialisation peut s'appliquer facilement à quasiment tous les objets. Cependant, toutes les classes du JDK ne sont pas sérialisables : en particulier, les classes liées aux éléments du système ne le sont pas (Thread, OutputStream et ses sous-classes, Socket, Image, ...).

La classe *Thread* est définie dans le package java.lang. Elle implémente l'interface Runnable.

```
public static void sleep(long millis) throws InterruptedException
```

Thread.sleep() est utilisé pour suspendre l'exécution du thread actuel, en informant le planificateur de threads que le thread actuel est construit dans l'état d'attente pour la période de temps spécifiée. Une fois le temps d'attente écoulé, le thread repasse à l'état Runnable et attend que l'UC planifie à nouveau l'exécution. La durée réelle de la mise en veille des threads dépend donc du planificateur de threads, qui est effectué par le système d'exploitation.

Dans notre travail, nous avons implémenté l'interface Serializable aux classes que nous jugeons nécessaires, mais malheureusement, nous n'avons pas pu réaliser ses fonctionnalités.

### **3.2 Représentation des changements d'état sur l'interface graphique : observer pattern**

Pendant l'implémentation de l'interface graphique, une question se pose : comment les changements d'état peuvent-ils être transmis sur l'interface graphique ? Après des recherches, nous estimons qu'Observer pattern pourrait nous aider sur cet objectif.

L'observateur est un modèle de conception comportemental. Il spécifie la communication entre deux objets : observable et observateurs. Un observable est un objet qui notifie aux observateurs les changements de son état. Par exemple, les acquéreurs peuvent être considérés comme des observateurs et le prix des immobiliers comme des observables. Le processus par lequel l'acquéreur porte son attention sur les changements de prix du logement est le patron de l'observateur. En java, on peut utiliser la classe Observable et l'interface Observer pour mettre en œuvre la fonctionnalité du patron de l'observateur. Les classes observées doivent hériter de la classe Observable, et chaque observateur doit implémenter l'interface Observer.

Dans notre programme, nous avons partiellement implémenté ce design pattern. Nous avons d'abord créé une interface `Observer` qui comprend une méthode `update()` avec deux paramètres : l'un consiste à l'instance observée, et l'autre indique le contenu de la modification. Nous avons ensuite créé une classe abstraite `Observable` qui a une liste d'observateurs dans le champ, une méthode pour notifier les mise à jour aux observateurs le cas échéant, et deux méthodes pour ajouter et supprimer les observateurs, ainsi qu'un getter et un setter. Les classes `Plateau` et `Joueur` héritent de la classe abstraite `Observable`, parce qu'elles seront les objets à observer. Enfin, nous avons implémenté l'interface `Observer` dans la classe `PlateauVision` et `MenuVision` et redéfini la méthode `update()` pour mettre à jour toutes les informations. Mais nous n'avons pas pu implémenter toutes les fonctionnalités, la méthode `update()` redéfinie reste à compléter et améliorer.

### 3.3 D'autres fonctionnalités

Selon les consignes, nous pourrions créer IA avancé, implémenter la négociation ou inventer d'autres règles. Cependant, nous n'avons pas pu les réaliser. D'une part, c'est parce ce jeu ne nous est pas familier, d'autre part, c'est à cause du manque de capacité.

Ce projet constitue un vrai défi pour nous qui sommes débutantes en java, et nous sommes parfaitement conscientes que notre travail est encore loin d'être satisfaisant et il reste beaucoup à améliorer. Nous en avons toutefois profité, parce que nous avons beaucoup appris et réussi à surmonter quelques difficultés au cours du travail sur ce projet. Quel que soit le résultat, tant que nous avons fait des progrès, notre travail a sa valeur.