Rapport AP4B

Développement d'une application JavaFX en se focalisant sur la partie conception et programmation du coeur de l'application.

Antoine Laurant, Arnaud Michel, Antoine Perrin

Table des matières

Présentation

Projet Turing Machine - UTBM

Introduction

Ce projet rend hommage à Alan Turing, un génie des mathématiques et du cryptage qui a contribué à l'invention des ordinateurs. Inspiré du jeu de société Turing Machine, notre version numérique propose une adaptation unique et ludique pour les étudiants de l'UTBM.

Objectif du Projet

L'objectif de ce projet est de créer une version numérique du jeu Turing Machine, en s'inspirant du monde de l' **UTBM**. Dans cette adaptation, un **professeur** envoie un mail à ses élèves pour leur donner une **énigme** afin qu'ils puissent rentrer dans la salle d'examen grâce à un **mot de passe**. Les étudiants doivent proposer des mots de passe et l'application leur indique si les mots de passe proposés valident les critères de l'énigme. Pour éviter toute triche et pour rendre le jeu plus **stimulant**, les mots de passes seront à usage unique et généré en fonction de l'INE de l'étudiant. De plus lors du lancement de l'application, une difficulté est choisie par l'étudiant, ce qui influe sur la **complexité** de l'énigme et pour favoriser la prise de risque, un **multiplicateur** de points (variant de 1 à 1.8) sera appliqué sur la note de l'examen en fonction de la difficulté choisie.

Il est également possible de jouer à plusieurs, en **compétition** ou le **premier** qui trouve le mot de passe valide gagne. Dans ce cas l'INE n'est pas nécessaire.

Technologies Utilisées

Pour réaliser ce projet, nous avons utilisé les technologies suivantes :

- **Java**: pour la logique de programmation.
- **JavaFX**: pour l'interface utilisateur.
- **MVC (Model-View-Controller)**: pour structurer le code de manière modulaire et maintenable.

Fonctionnalités du Jeu

- **Connexion**: l'étudiant doit se connecter avec son INE pour accéder à l'énigme.
- **Choix de la difficulté**: l'étudiant peut choisir la difficulté de l'énigme.
- **Génération de mot de passe**: l'application génère un mot de passe unique en fonction de l'INE de l'étudiant.
- **Validation du mot de passe**: l'application valide le mot de passe proposé par l'étudiant.

Choix de conceptions

Afin de réaliser notre application, nous avons dû faire des choix de conception. Ces choix ont été faits en fonction des besoins de l'application et des contraintes techniques.

Model View Controller

Nous avons choisi d'utiliser le **modèle MVC** pour la conception de notre application. En effet, cette architecture permet de séparer les différentes composantes de l'application, ce qui facilite la maintenance et l'évolution de l'application. De plus le modèle MVC permet de rendre l'application plus modulaire et donc plus facile à comprendre et à maintenir.

Création d'une unique solution

Pour créer une solution unique en fonctions de différents critères, nous avons choisi de modéliser les différentes étapes de la création de la solution grâce à la **composition de fonctions**.

Mathématiquement, nous créons un tableau de **125** éléments, chaque élément étant une **combinaison possible**. Nous appliquons ensuite une série de fonctions récursivement pour **filtrer** les combinaisons en fonction des critères choisis par l'utilisateur. Et nous continuons ce processus jusqu'à ce qu'il ne reste qu' **une** seule combinaison. Si au bout de 6 étapes, s'il reste plusieurs combinaisons, nous revenons à l'étape précédente et recommençons le processus (donc, de manière récursive). Ce processus est développé dans un diagramme de séquence disponible **ici (Initialisation de la partie)**.

Modélisation mathématique du processus :

Initialisation :
$\Omega = \{(x_1, x_2, x_3) \mid x_i \in \{1,2,3,4,5\} \text{ pour } i = 1,2,3\}$
(ensemble de toutes les combinaisons possibles, taille 125)

Processus de filtrage :
$F = f_1 \circ f_2 \circ f_3 \circ f_4 \circ f_5 \circ f_6 : \Omega \to \mathcal{P}(\Omega)$
avec f_k : fonctions de filtrage appliquées séquentiellement

Résultat après chaque étape :
$\Omega_k = f_k(\Omega_{k-1}) \quad \text{avec } \Omega_0 = \Omega$
Si $|\Omega_k| = 1$ (une seule combinaison restante) et $4 \leq k \leq 6$, le processus s'arrête.
Si $|\Omega_k| \neq 1$ (il reste plusieurs combinaisons) ou $k = 6$ (6 étapes atteintes), alors :
Retour à l'étape précédente avec une nouvelle fonction de filtrage.

Utilisation de JavaFX

Nous avons choisi d'utiliser **JavaFX** pour la conception de l'interface graphique de notre application. JavaFX est un framework qui permet de créer des interfaces graphiques de manière simple et efficace. Par ailleurs, c'est une technologie qui est intégrée à Java, ce qui facilite son utilisation.

> Nous avons fait le choix de JavaFX plutôt que Swing, car JavaFX est plus moderne et plus adapté pour la création d'interfaces graphiques. De plus, JavaFX est plus performant que Swing et offre plus de possibilités en termes de design.

Diagrammes Use Case

Le diagramme de cas d'utilisation ci-dessous illustre les principales interactions entre l'utilisateur ("Joueur") et le système numérique dans le cadre du jeu. Il met en évidence les différents cas d'utilisation, organisés en parties, fiches de notes et manches, ainsi que leurs relations (includes, extends) pour une meilleure compréhension des fonctionnalités du système pour le client.

Diagramme de cas d'utilisation

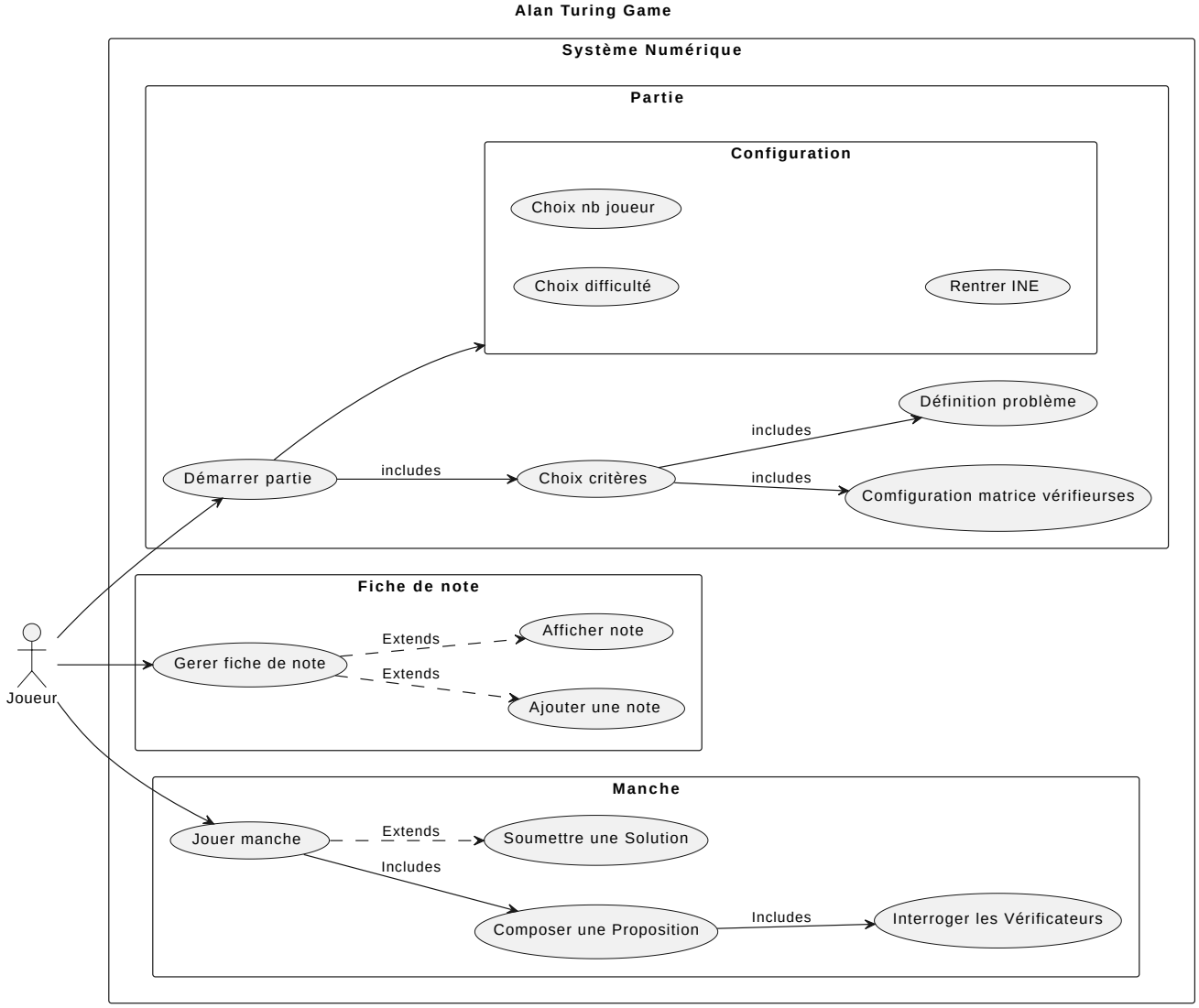


Diagramme de Classe

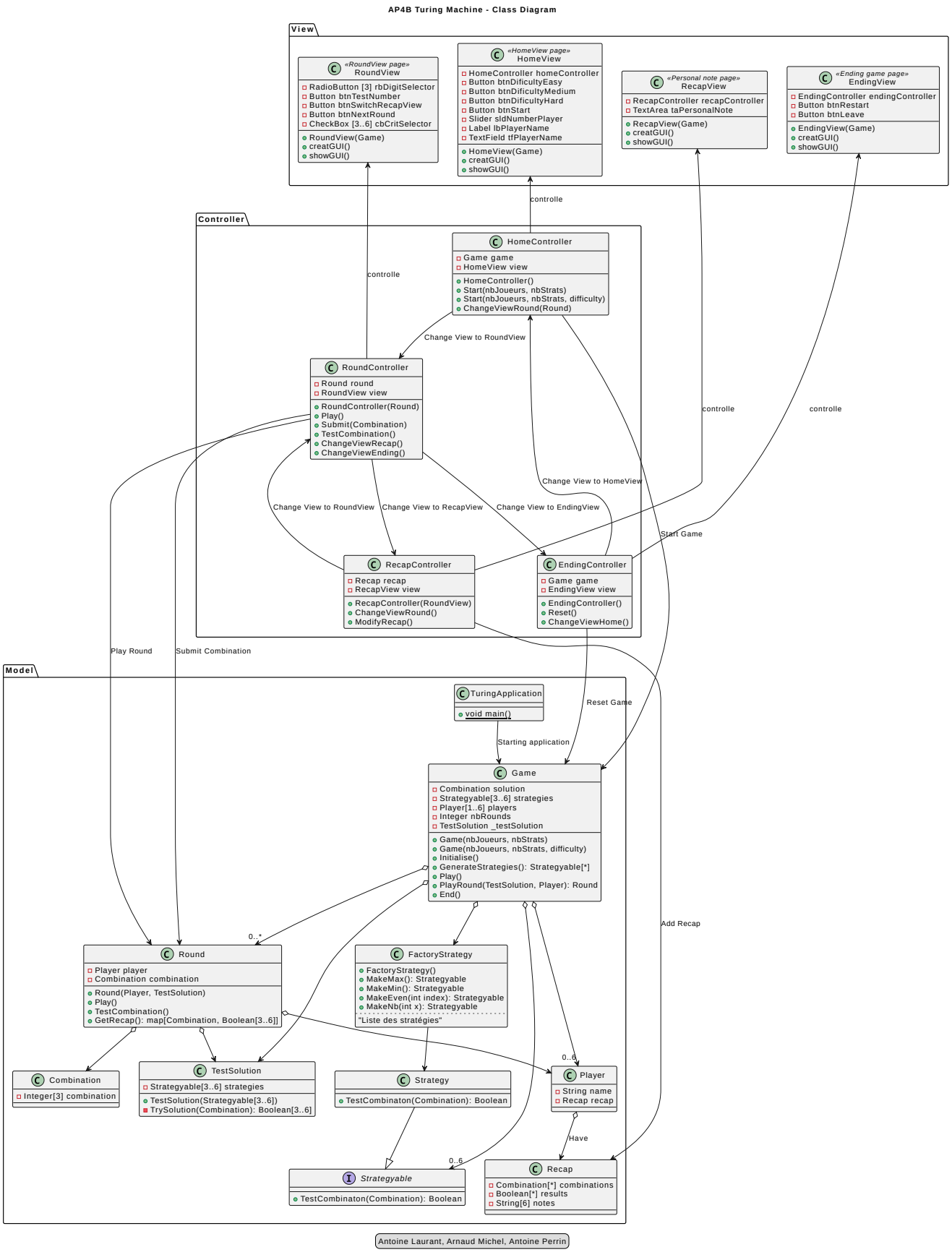
Explication

Le diagramme de classe ci-dessous illustre la structure de notre application.

Voici quelques points clés à retenir :

- `Strategyable` est une interface qui définit une méthode `TestCombination` pour tester une combinaison. Nous avons préféré utiliser une interface plutôt qu'une classe abstraite pour permettre une plus grande flexibilité dans la définition des stratégies. De plus elle ne contient pas d'attributs.
- `FactoryStrategy` est une classe qui permet de créer des stratégies. Elle contient des méthodes pour créer différentes stratégies (par exemple `MakeMax`, `MakeMin`, `MakeEven`, `MakeNb`, etc.). Cette classe permet de centraliser la création des stratégies et de faciliter l'ajout de nouvelles stratégies. Cela limite aussi beaucoup les dépendances entre les classes.
- `TestSolution` est une classe qui permet de tester une solution par rapport à une combinaison. Elle contient un tableau de stratégies et une méthode `TrySolution` qui teste une combinaison par rapport à chaque stratégie.
- `Player` est une classe qui représente un joueur. Elle contient un nom et un récapitulatif des combinaisons proposées par le joueur, ainsi que les résultats obtenus pour chaque combinaison.
- `Recap` est une classe qui représente le récapitulatif des combinaisons proposées par un joueur. Elle contient un tableau de combinaisons, un tableau de résultats et un tableau de notes pour chaque combinaison.

Diagramme de Classe


AP4B Turing Machine - Class Diagram
View
«RoundView page»
RoundView
RadioButton [3] rbDigitSelector
Button btnTestNumber
Button btnSwitchRecapView
Button btnNextRound
CheckBox [3..6] cbCritSelector
RoundView(Game)
creatGUI()
showGUI()
«HomeView page»
HomeView
HomeController homeController
Button btnDificultyEasy
Button btnDificultyMedium
Button btnDificultyHard
Button btnStart
Slider sldNumberPlayer
Label lbPlayerName
TextField tfPlayerName
HomeView(Game)
creatGUI()
showGUI()
«Personal note page»
RecapView
RecapController recapController
TextArea taPersonalNote
RecapView(Game)
creatGUI()
showGUI()
«Ending game page»
EndingView
EndingController endingController
Button btnRestart
Button btnLeave
EndingView(Game)
creatGUI()
showGUI()
controlle
Controller
HomeController
Game game
HomeView view
HomeController()
Start(nbJoueurs, nbStrats)
Start(nbJoueurs, nbStrats, difficulty)
ChangeViewRound(Round)
controlle
Change View to RoundView
RoundController
Round round
RoundView view
RoundController(Round)
Play()
Submit(Combination)
TestCombination()
ChangeViewRecap()
ChangeViewEnding()
controlle
controlle
Change View to HomeView
Change View to RoundView
Change View to RecapView
Change View to EndingView
Start Game
RecapController
Recap recap
RecapView view
RecapController(RoundView)
ChangeViewRound()
ModifyRecap()
EndingController
Game game
EndingView view
EndingController()
Reset()
ChangeViewHome()
Play Round
Submit Combination
Model
TuringApplication
void main()
Reset Game
Starting application
Game
Combination solution
Strategyable[3..6] strategies
Player[1..6] players
Integer nbRounds
TestSolution _testSolution
Game(nbJoueurs, nbStrats)
Game(nbJoueurs, nbStrats, difficulty)
Initialise()
GenerateStrategies(): Strategyable[*]
Play()
PlayRound(TestSolution, Player): Round
End()
Add Recap
0..*
Round
Player player
Combination combination
Round(Player, TestSolution)
Play()
TestCombination()
GetRecap(): map[Combination, Boolean[3..6]]
FactoryStrategy
FactoryStrategy()
MakeMax(): Strategyable
MakeMin(): Strategyable
MakeEven(int index): Strategyable
MakeNb(int x): Strategyable
"Liste des stratégies"
Combination
Integer[3] combination
TestSolution
Strategyable[3..6] strategies
TestSolution(Strategyable[3..6])
TrySolution(Combination): Boolean[3..6]
Strategy
TestCombinaton(Combination): Boolean
0..6
Player
String name
Recap recap
Have
0..6
Strategyable
TestCombinaton(Combination): Boolean
Recap
Combination[*] combinations
Boolean[*] results
String[6] notes
Antoine Laurant, Arnaud Michel, Antoine Perrin

Diagrammes de Séquence

Pour modéliser le comportement de notre application, nous avons réalisé 2 diagrammes de séquence. Le premier diagramme de séquence illustre le processus de génération des stratégies. Le second diagramme de séquence illustre le processus de déroulement d'une manche.

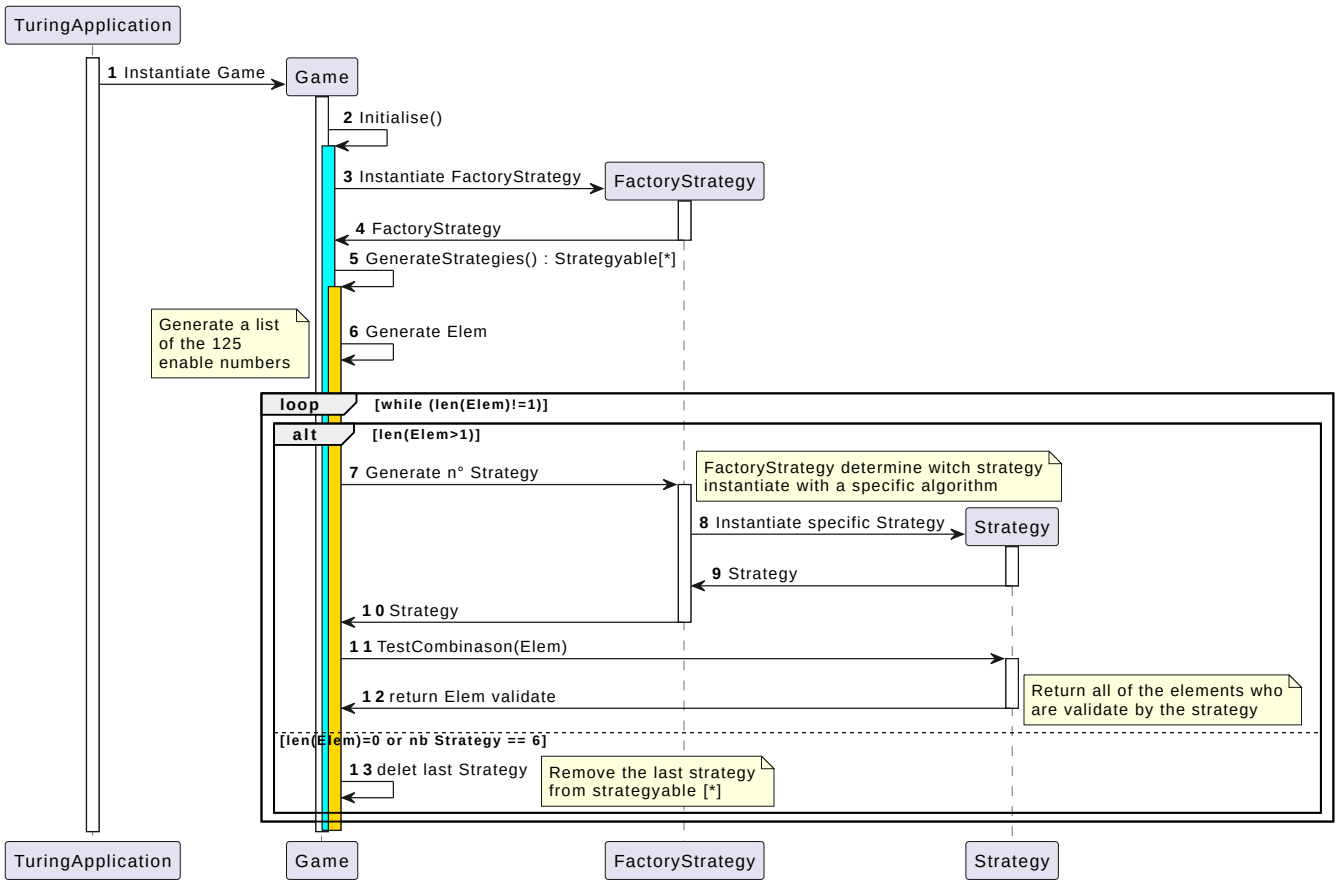
- Diagramme de l'intialisation de la partie (Initialisation de la partie)
- Diagramme de la manche (Description de la manche)

Nous avons choisi de modéliser ces processus sous forme de diagrammes de séquence pour faciliter la **compréhension** du **fonctionnement** de notre application. Ces diagrammes permettent de **visualiser** les interactions entre les différentes classes du système et de comprendre comment elles interagissent pour **réaliser** les fonctionnalités de l'application.

Initialisation de la partie

Le diagramme de séquence ci-dessous illustre les interactions entre les classes du système lors de l'initialisation de la partie. Il montre comment la classe `TuringApplication` instancie la classe `Game`, qui à son tour initialise le jeu en générant les stratégies et la solution. Il met en évidence le rôle de notre algorithmie de génération de stratégies, notamment la validation des stratégies par rapport à la solution.

Diagramme de séquence



Description de la manche

Description

La manche est une partie du jeu. Elle est composée de plusieurs éléments :

- Un **Round**: le model de la manche.
- Un **RoundController**: le contrôleur de la manche.
- Un **RoundView**: la vue de la manche.

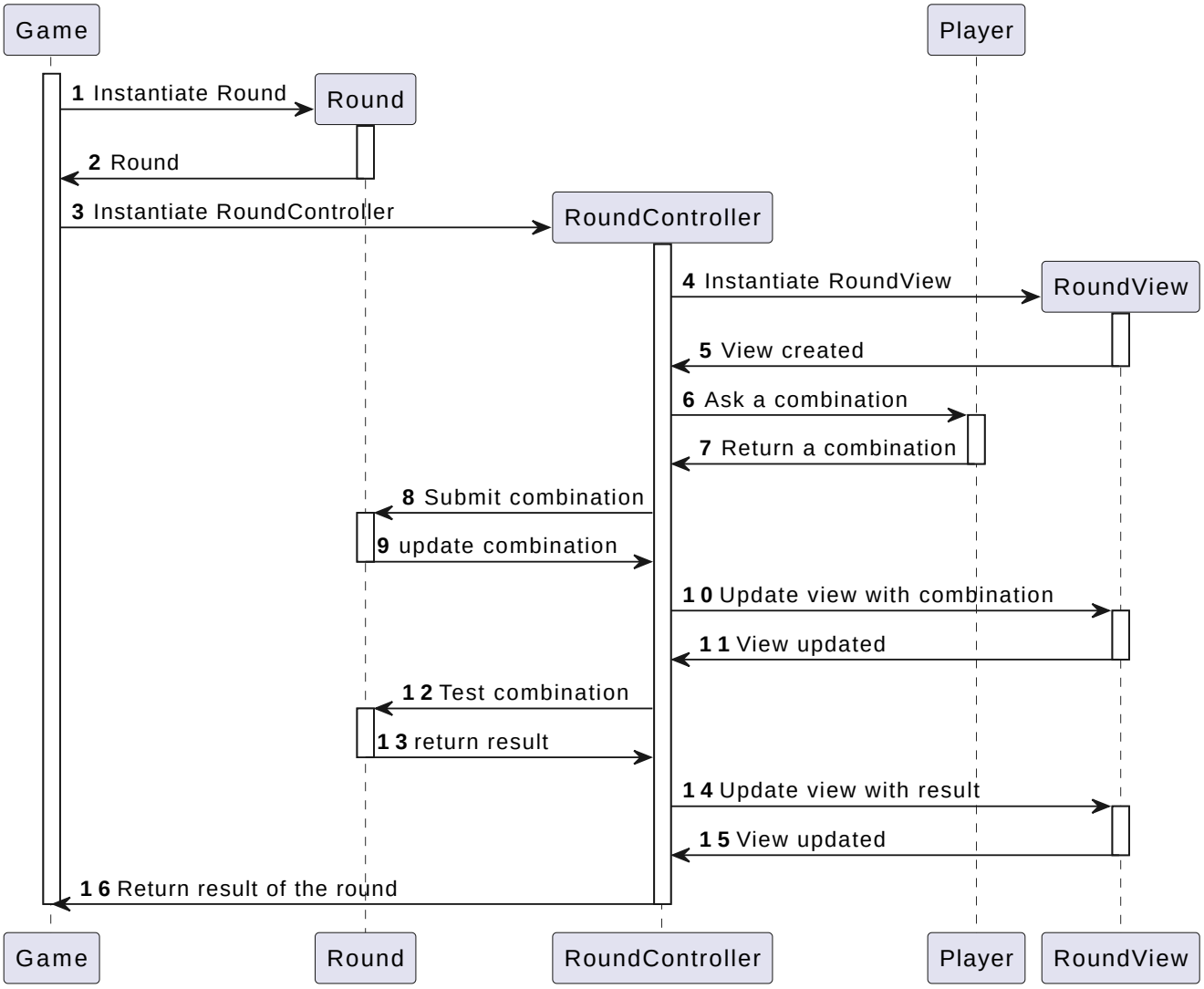
A chaque manche, un joueur doit proposer une combinaison. Cette combinaison est testée et le résultat est affiché à l'écran. Puis, le résultat de la manche est retourné au jeu.

Exemple

Un joueur propose la combinaison `123`. Pour un critère nombre de `1` le test renvoie `validé` (true), pour le critère le 3e nombre est le plus grand, le test renvoie `non validé` (false), et ainsi de suite pour le nombre de critères de la partie. Enfin, le résultat de la manche est retourné au jeu.

Diagramme de séquence

Manche


Game
Player
1 Instantiate Round
Round
2 Round
3 Instantiate RoundController
RoundController
4 Instantiate RoundView
RoundView
5 View created
6 Ask a combination
7 Return a combination
8 Submit combination
9 update combination
10 Update view with combination
11 View updated
12 Test combination
13 return result
14 Update view with result
15 View updated
16 Return result of the round
Game
Round
RoundController
Player
RoundView

Conclusion

Ce projet en Java, utilisant les bibliothèques JavaFX pour l'interface graphique et le modèle Model-View-Controller pour une meilleure accessibilité et modularité, nous a permis de nous familiariser avec la programmation orientée objet tout en appliquant les concepts vus en cours, tels que l'héritage et le polymorphisme.

Il a également stimulé notre créativité en adaptant le jeu aux couleurs de l'UTBM, conformément au descriptif de notre adaptation, qui transforme ce projet en un escape game revisité.