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 1.2 à 2) 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 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.

Répartition des Tâches

Arnaud Michel :

- Conception `théorique` de l'application
- Conception du diagramme de `séquence` sur l'initialisation de la partie
- Conception du diagramme de `classe`
- Implémentation des `vues`, des `contrôleurs` et la base des `modèles`.
- Relecture et `optimisation` de la génération de combinaisons.
- Design de l'interface graphique.
- Rédaction du rapport.

Antoine Laurant :

- Conception `théorique` de l'application
- Conception du diagramme de `classe`
- Conception du diagramme de `séquence` sur la gestion des manches
- Implémentation de l' **algorithme de génération de combinaisons**
  - Usine de stratégies

    - Test de solution
    - Récupération de la solution unique
- View et Controller de la fenêtre de fin de partie
- Relecture du rapport.

Antoine Perrin :

- Conception `théorique` de l'application
- Conception du diagramme `Use Case`
- Analyse et compréhension des règles du jeu pour l'adapter à l'UTBM
- Ajouts de stratégies
- Relecture complète du code et correction des bugs
- Écriture / montage vidéo de présentation
- Relecture du rapport.

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.

Evolution des choix de conception

Depuis la première version de l'application, l'architecture n'a pas grandement évolué.

Nous avons supprimé une classe que nous avions jugée inutile et nous avons ajouté quelques méthodes pour l'encapsulation des données. Nous avons aussi ajouter des Enumerations pour les différentes difficultés et les différents types de stratégies afin de rendre le code plus lisible et plus facile à maintenir.

Nous détaillerons ces modifications dans la partie **Diagramme de Classe (Diagramme de Classe)**.

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)**.

Modelisation mathematique du processus:

İnitialisation :

$\Omega = \{(x_1, x_2, x_3) \mid x_i \in \{1, 2, 3, 4, 5\} \text{ pour } i = 1, 2, 3\}$
(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 aprè s 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.

Nous avons utilisé Maven pour gérer les dépendances de notre projet. Maven est un outil de gestion de projet qui permet de gérer les dépendances, de compiler le code, de générer des rapports, etc.

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

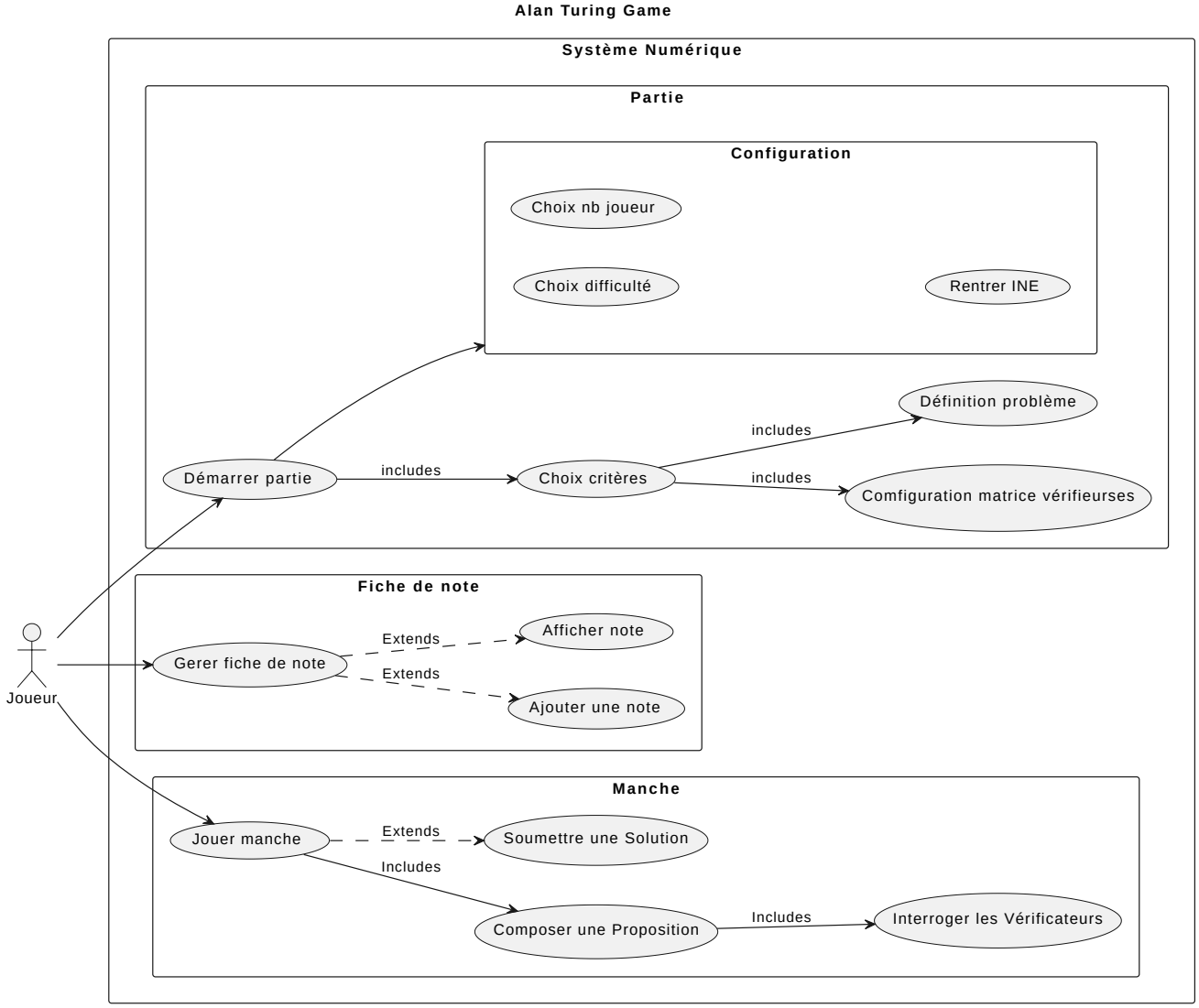


Diagramme de Classe

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.
- `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.

Modifications continue de la conception

Nous avons décidé de supprimer les classe `Round` et `TestSolution` car pour l'une nous pouvons récupérer les valeurs dans le controller. Et pour la seconde nous nous en sommes dispensé en utilisant une méthode de la classe `Game` pour tester les combinaisons.

Nous avons aussi ajouté des méthodes: Dans la classe `Game`:

- `generateAllCombinations` pour générer toutes les combinaisons possibles. (125 combinaisons)

- `filterCombinations` pour filtrer les combinaisons en fonction des stratégies choisies par les joueurs.
- `fillRemainStrategies` pour remplir les stratégies restantes avec des stratégies aléatoires.
- `isUniqueCombination` pour vérifier si une combinaison est unique.

Dans la classe `FactoryStrategy` nous avons ajoutés toutes les stratégies que nous voulions implémenter.

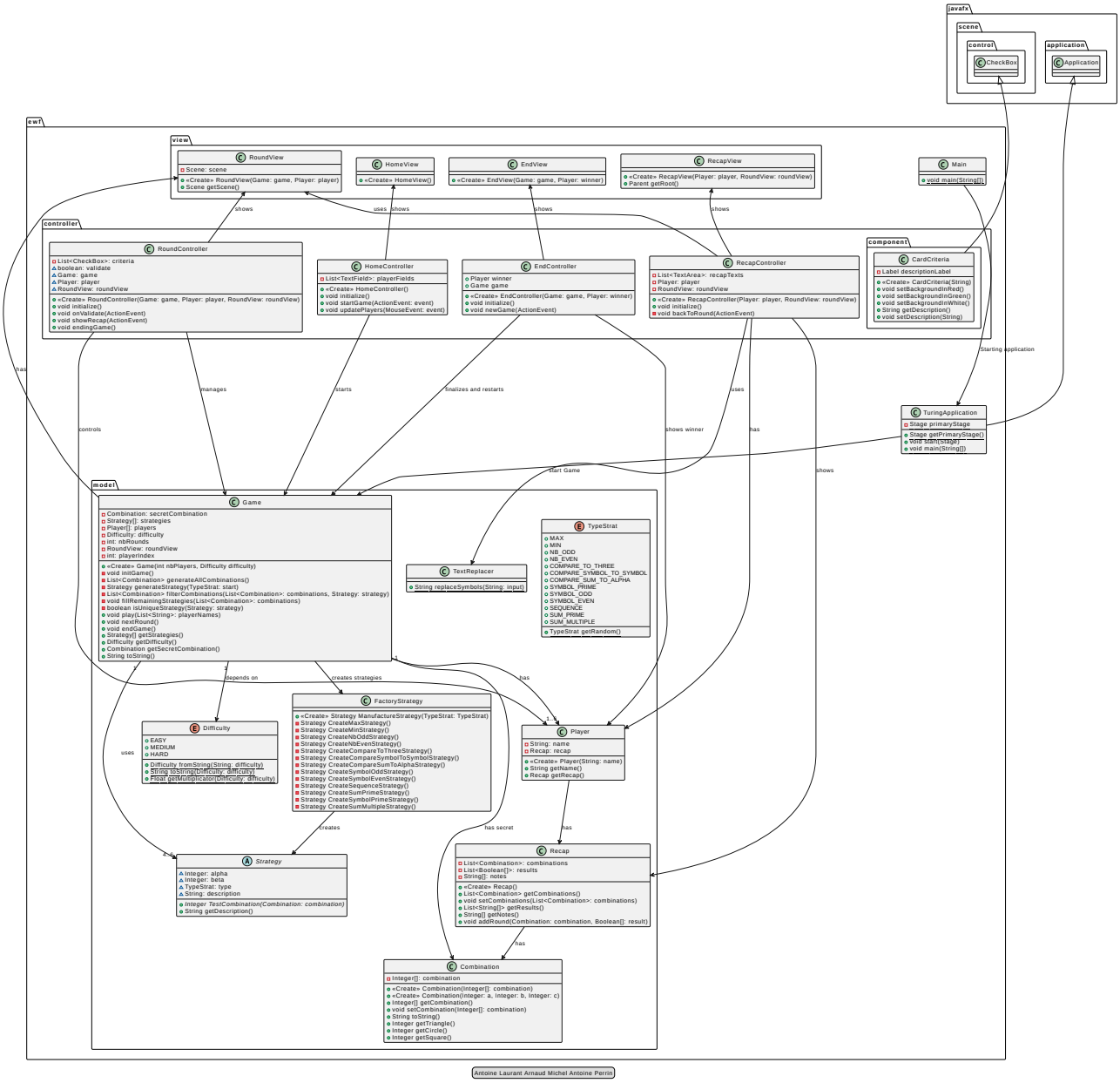
Pour la gestion des stratégies, nous avons supprimé la classe abstraite `Strategy` et nous avons transformé l'inteface `Strategyable` en classe abstraite pour pouvoir ajouter des méthodes. Ces classes sont implémentées dans la classe `FactoryStrategy` en redéfinissant la méthode abstraite `TestCombination`.

> ⚠ Pour la partie vue et contrôleur, nous avons ajouté des méthodes pour faciliter la manipulation des données et des élements interagissant avec l'utilisateur. Nous les avons simplement ajoutées sans les détailler ici car elles sont assez simples et ne nécessitent pas d'explications supplémentaires.

> ℹ Il y a eu aussi des ajouts de méthodes de type `getter` et `setter` pour les attributs des classes mais aussi des méthodes pour faciliter la manipulation des données. De la même manière, ces méthodes sont assez simples et ne nécessitent pas d'explications supplémentaires.

Diagramme de Classe

AP4B Turing Machine - Class Diagram



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.

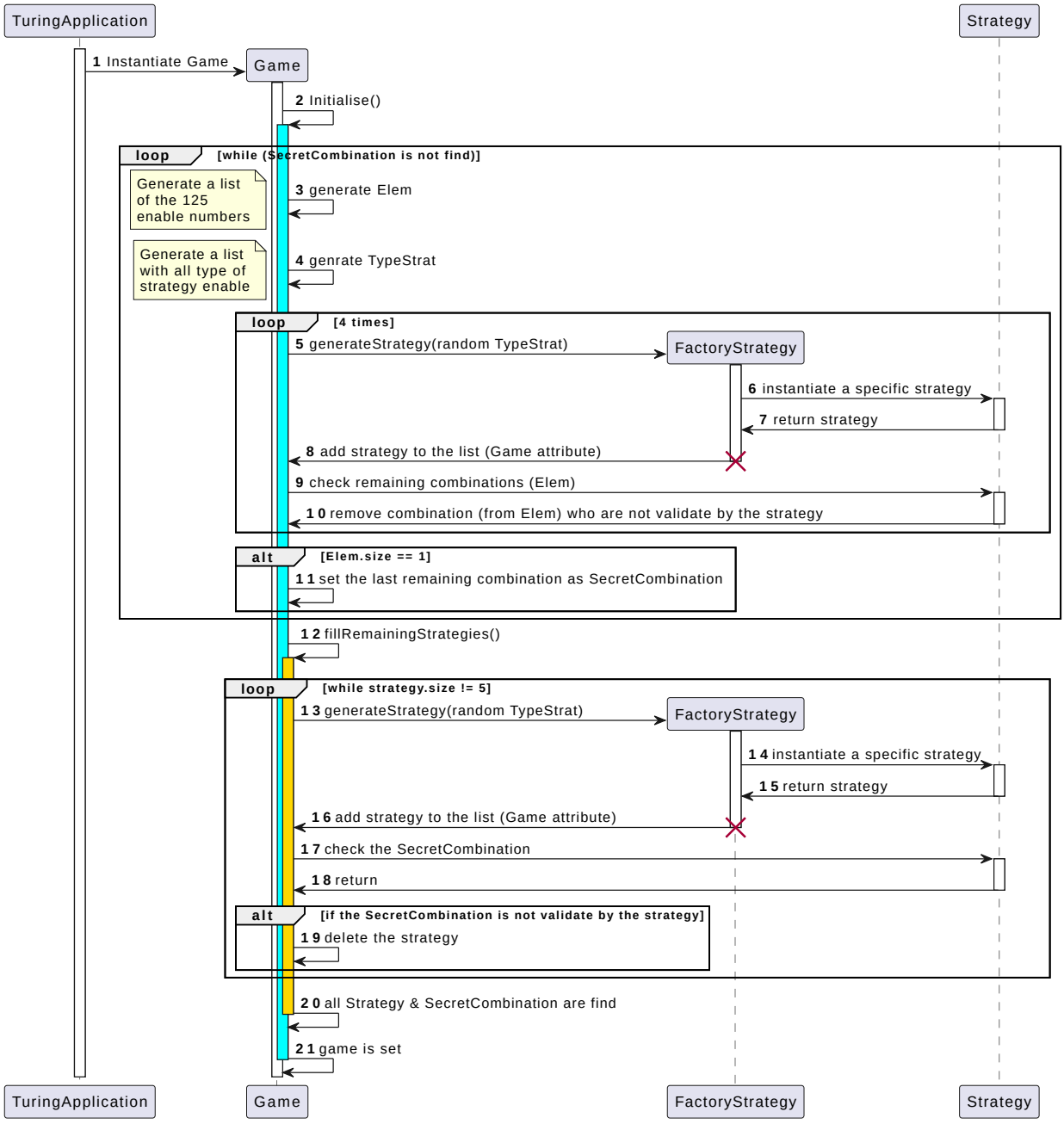
Initialisation de la partie

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.

Modifications continue de la conception

Cette partie n'a que peu évolué ormis certains ajouts de fonction afin de mieux séparer les responsabilités des méthodes.

Diagramme de séquence


TuringApplication
Game
Strategy
1 Instantiate Game
2 Initialise()
loop
[while (SecretCombination is not find)]
Generate a list of the 125 enable numbers
3 generate Elem
Generate a list with all type of strategy enable
4 genrate TypeStrat
loop
[4 times]
5 generateStrategy(random TypeStrat)
FactoryStrategy
6 instantiate a specific strategy
7 return strategy
8 add strategy to the list (Game attribute)
9 check remaining combinations (Elem)
10 remove combination (from Elem) who are not validate by the strategy
alt
[Elem.size == 1]
11 set the last remaining combination as SecretCombination
12 fillRemainingStrategies()
loop
[while strategy.size != 5]
13 generateStrategy(random TypeStrat)
FactoryStrategy
14 instantiate a specific strategy
15 return strategy
16 add strategy to the list (Game attribute)
17 check the SecretCombination
18 return
alt
[if the SecretCombination is not validate by the strategy]
19 delete the strategy
20 all Strategy & SecretCombination are find
21 game is set
TuringApplication
Game
FactoryStrategy
Strategy

Description de la manche

Description

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

- Un **Game**: le contenu de la partie.
- 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.

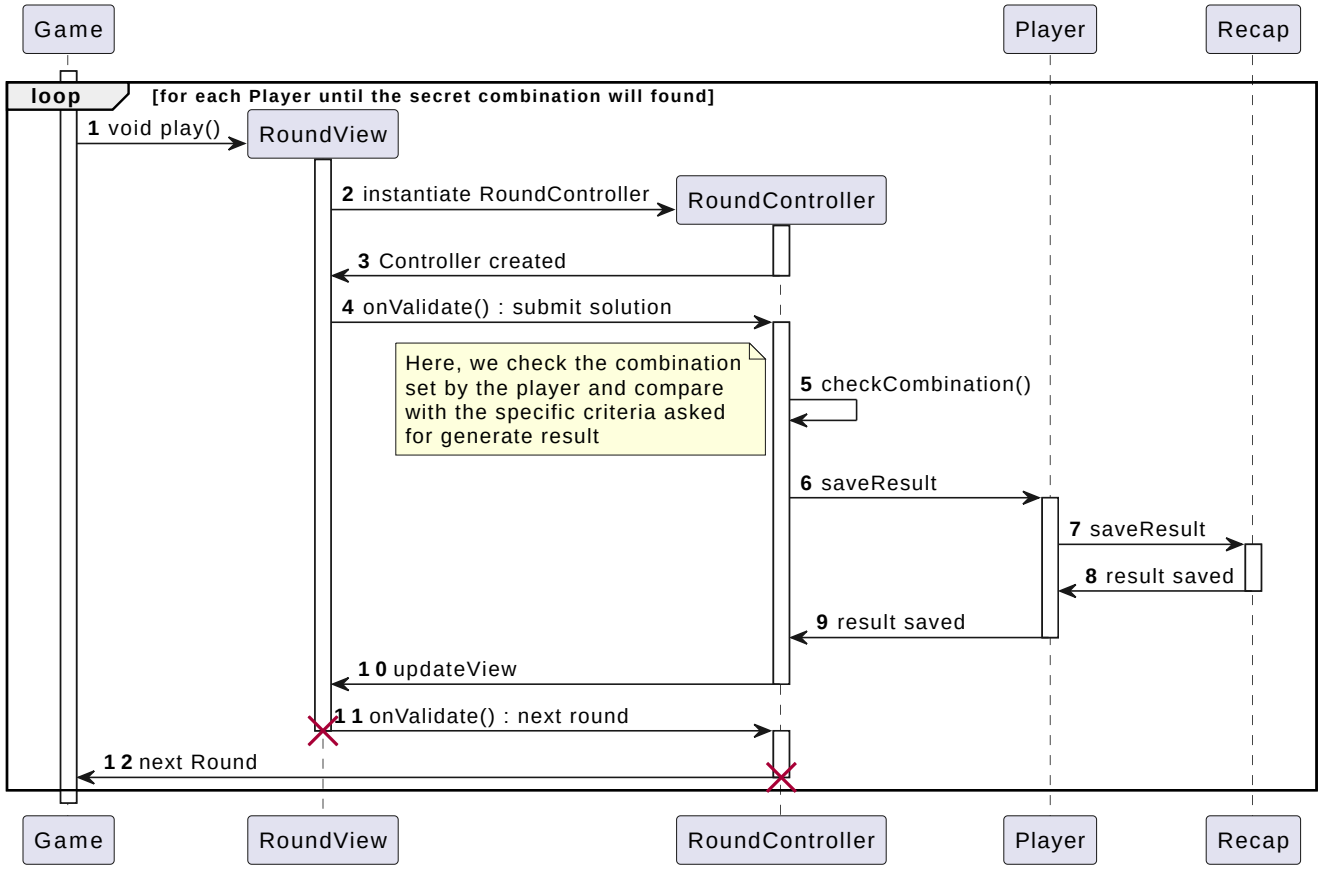
Modifications continue de la conception

Dans la première version de notre conception, nous avions prévu de créer une classe `Round` pour chaque manche. Cependant, comme nous n'avons pas besoin de stocker tout les résultats des manches, nous avons décidé de ne pas créer de classe `Round` et de gérer les manches directement dans le contrôleur de la manche. Et de stocker les résultats et les combinaisons dans la classe `Player`.

Exemple

Un joueur propose la combinaison `123`. Pour un critère nombre de `1` le test renvoie `validé` (true), pour le critère le 3^{e} 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


Game
Player
Recap
loop
[for each Player until the secret combination will found]
1 void play()
RoundView
2 instantiate RoundController
RoundController
3 Controller created
4 onValidate() : submit solution
Here, we check the combination set by the player and compare with the specific criteria asked for generate result
5 checkCombination()
6 saveResult
7 saveResult
8 result saved
9 result saved
10 updateView
11 onValidate() : next round
12 next Round
Game
RoundView
RoundController
Player
Recap

Implémentation

Première Partie

La première chose que nous avons implémentée est le **model**, fidèle à notre conception. Cela nous a permis de pouvoir directement ajouter les vues et les contrôleurs.

Mettre en place le modèle nous a permis de commencer à réfléchir à la manière dont nous allions **implémenter** les différentes fonctionnalités du jeu.

De plus nous avons mis en place la première vue, qui est la vue d'accueil du jeu pour que tout le monde ait les mêmes **bases** et **règles** pour commencer à programmer. Pour pouvoir gérer nos vues et ne pas s'embeter avec des dizaines d'attributs graphiques, nous avons décidé d'utiliser des fichiers **FXML** pour décrire nos vues. Ces fichiers permettent de décrire l'interface graphique de manière déclarative, ce qui facilite la **maintenance** et l' **évolution** de l'interface. Il suffit d'associer un fichier FXML à un contrôleur pour que l'interface soit **automatiquement** générée.

Deuxième Partie

Nous avons séparé le travail en **deux parties**, la première partie étant de mettre en place l' **algorithme** de génération de solution et la deuxième partie étant de mettre en place l' **interface graphique** liée avec la classe `Game`.

Ces deux taches pouvant être faites **indépendamment** l'une de l'autre, nous avons pu avancer plus **rapidement**.

Nous avons donc pu mettre en place les rounds alors même que l'algorithme de génération de solution n'était pas encore **fonctionnel**.

Troisième Partie

Nous avons rajouté des **fonctionnalités** au jeu, comme la possibilité de choisir la **difficulté** de l'énigme, de **valider** le mot de passe proposé par l'étudiant, etc.

Nous avons également ajouté la possibilité de jouer à **plusieurs** en **compétition**.

Tests unitaires

Nous avons mis en place un **test unitaire** pour générer des combinaisons et vérifier que l'algorithme fonctionne correctement (vérification de l'amplitude des combinaisons, de la génération de combinaisons uniques, etc.).

Utilisation

Pour utiliser notre application, vous avez deux possibilités :

Prérequis

Si vous souhaitez utiliser notre application, vous devez respecter les prérequis suivants :

> ⚠ Dans les deux cas, vous devez avoir installé Java `23.0.1` sur votre machine ou un environnement Java compatible. Vous pouvez télécharger Java sur le site officiel d'Oracle : Java SE Downloads (https://www.oracle.com/java/technologies/javase-jdk11-downloads.html)
>
> De plus, vous devez avoir installé JavaFX `23.0.1` sur votre machine. Vous pouvez télécharger JavaFX sur le site officiel d'OpenJFX : OpenJFX (https://openjfx.io/)
>
> Enfin, vous devez avoir installé Maven `3.8.3` sur votre machine. Vous pouvez télécharger Maven sur le site officiel de Apache : Maven (https://maven.apache.org/download.cgi) Pour vérifier que Java, JavaFX et Maven sont bien installés, vous pouvez taper les commandes suivantes dans un terminal :
>
> ```
> java -version
> javafx -version
> mvn -version
> ```
>
> Si les commandes renvoient des informations sur les versions installées, c'est que tout est bien installé.

Avec le code source

Pour utiliser notre application avec le code source, vous devez suivre les étapes suivantes :

1. Décompresser l'archive contenant le code source.
2. Ouvrir le projet dans votre IDE préféré (IntelliJ IDEA, Eclipse, NetBeans, etc.).

3. Compiler le projet avec Maven :

```
mvn clean install
```

4. Exécuter le projet avec Maven :

```
mvn javafx:run
```

5. Vous pouvez maintenant utiliser notre application.

Avec le .jar

Pour utiliser notre application avec le fichier `.jar`, vous devez suivre les étapes suivantes :

1. Décompresser l'archive contenant le fichier `.jar`.
2. Ouvrir un terminal et vous rendre dans le dossier où se trouve le fichier `.jar`.
3. Exécuter le fichier `.jar avec Java :

```
java -jar TuringMachine.jar
```

4. Vous pouvez maintenant utiliser notre application.

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.

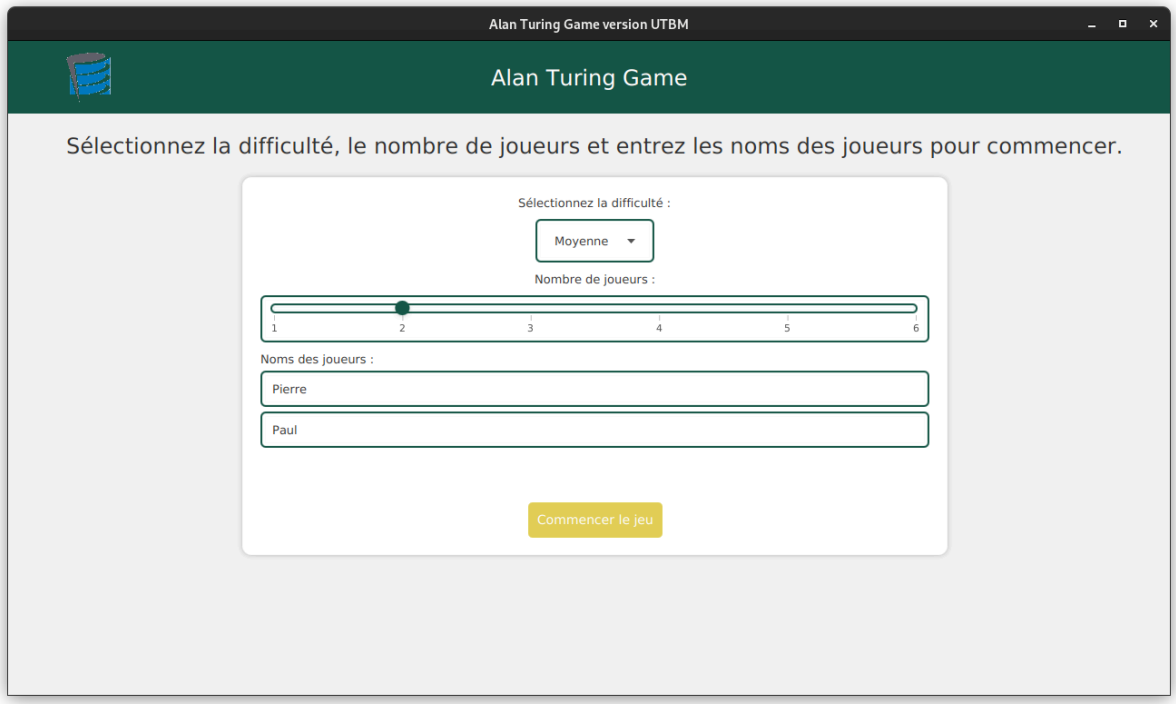
Pour conclure, nous tenions sincèrement à avoir un générateur de combinaisons unique (plutot que des parties fixes comme demandé) car chaque partie sera différente et chaque personne pourra rajouter des stratégies pour rendre le jeu plus complexe et plus intéressant.

Auteurs

- Arnaud Michel
- Antoine Laurant
- Antoine Perrin

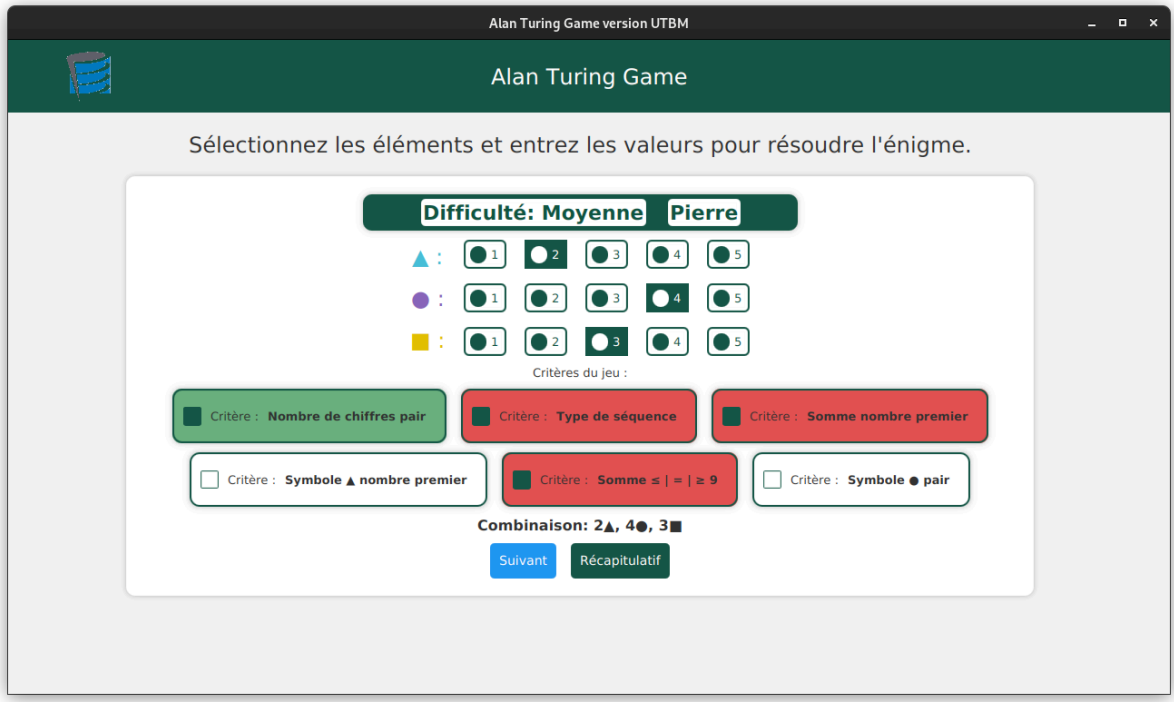
Sous la direction de Mme. **Fatima Zahrae EL-QORAYCHY**.

Annexes



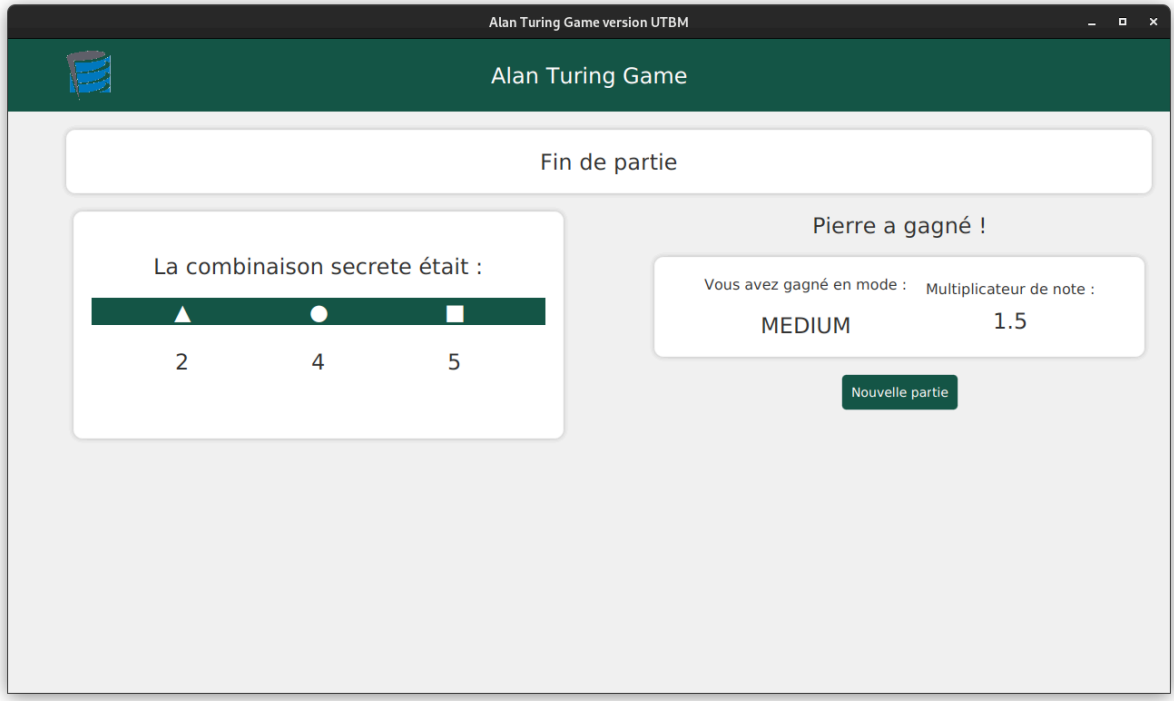
home-view

round-view-unvalidate.png



round-view-validate.png

recap-view.png


Alan Turing Game version UTBM
Alan Turing Game
Fin de partie
La combinaison secrete était :
2
4
5
Pierre a gagné !
Vous avez gagné en mode :
MEDIUM
Multiplicateur de note :
1.5
Nouvelle partie

end-view.png