



**Rapport final de projet**

# Projet de semestre 3 : Programmer un jeu de Ricochet-Robots

**Réalisé par**

Dorian LAURANCY

Aloys VERNHET

Weslie RABESON

Arnaud DUCRET

**Sous la direction de**

Alain MARIE-JEANNE

Pour l'obtention du DUT Informatique

Année universitaire 2017 - 2018

## GRILLE DE RELECTURE DU RAPPORT

Cocher au moyen d'une croix chacun des items après vérification.

*- Cette grille est à insérer au début du rapport -*

| | *Qualité et cohérence du propos* | |
|---|---|---|
| | **Le titre du rapport** indiqué sur la page de couverture décrit précisément le projet. | |
| | **Les mots du glossaire** sont identifiables dans le texte au moyen d'un astérisque. *(« Rechercher » / « Remplacer »)* | |
| | L'annonce du plan dans **l'introduction** correspond bien au plan réellement suivi. | |
| | Le rapport présente un 4° de couverture un **résumé** en **français ET** en **anglais** reprenant le contenu du rapport | |
| | **Table des matières** | **La table des matières** insérée automatiquement a été mise à jour une dernière fois avant l'impression. |
| | | Il n'y a pas de **titres « orphelins »** dans le plan. Ex 1.1.1 (et pas de 1.1.2) |
| | **Figures & diagrammes** | Toutes les **figures et diagrammes** sont numérotés et légendés avec un titre précis. Ils apparaissent dans la table des figures. |
| | | Chaque figure et/ou diagramme est commenté et il y a un **renvoi** dans le texte indiquant au lecteur quand il doit la/le regarder (« cf. figure 1 »). C'est une illustration du texte et non l'inverse. |
| | | **La table des figures** insérée automatiquement a été mise à jour une dernière fois avant l'impression. |
| | **Annexes** | Toutes les **annexes** sont nommées et numérotées. |
| | | Il y a un **renvoi** dans le texte indiquant au lecteur quand il doit regarder chacune des **annexes**. |

| | *Mise en forme du propos* | |
|---|---|---|
| | **Mise en page** | La police de caractère est de taille standard (Times, 12 pt). |
| | | L'alignement est justifié. |
| | | La présentation du texte est aérée : marges et interligne raisonnables ; espace avant / après le paragraphe ; espace après les titres et sous-titres. |
| | **En-têtes et pieds de page** | L'en-tête comporte le nom du rapport et le nom des auteurs (sauf sur les avant-textes et les post-textes qui n'ont pas d'en-tête). |
| | | Les pieds de page comportent les numéros de pages. |
| | **Numérotation des pages** | La page de couverture et la quatrième de couverture ne sont pas numérotées. |
| | | Les avant-textes sont numérotés en chiffre romain. |
| | | Le corps du rapport est numéroté en chiffre arabe (la numérotation reprend à 1) |
| | | Les post-textes sont numérotés en chiffre romain (la numérotation reprend à 1). |
| | **Les titres** | Les titres des grandes parties et des différents avant-textes et post-textes figurent en haut des pages (saut de page). |
| | | Les titres des parties sont numérotés sous la forme 1.1.1 et sont indentés. |
| | | Le titre n'est pas suivi d'un signe de ponctuation (« : » ou « . ») |

| | *Correction de la langue : orthographe et usages* | |
|---|---|---|
| | **L'intégralité du texte est passée au correcteur orthographique, a été relue par une tierce personne et a été corrigée.** Le texte ne comporte plus aucune faute. | |
| | Les prénoms et noms des personnes sont indiqués dans l'ordre **Prénom + NOM** (en majuscule). | |
| | **Monsieur** est abrégé « M. » et non « Mr » comme en anglais. *(Ctrl+F « Rechercher »)* | |
| | **Les chiffres sont écrits en lettres** (« dix heures », « vingt-cinq employés »), sauf exception (chiffre suivi d'une unité de mesure par exemple : 10Mo, 10km/h…) | |
| | **Relecture orthographique** (facultatif) *Indiquez les mots dont l'orthographe est à vérifier pour le rapport.* | *Ex : base de données, programmation multi-agents…* |
| | | |
| | | |
| | | |
| | | |

| **Diagrammes techniques (uniquement si nécessaire)** | | |
|---|---|---|
| **Section** | **Objectif** | **Diagrammes UML** |
| Analyse des besoins | Objectifs de la section :<br><br>- Identifier clairement le contexte dans lequel le logiciel à produire va s'insérer<br>- Identifier les besoins des utilisateurs | Diagramme des packages<br><br>Diagramme de cas d'utilisation<br><br>Diagramme d'activité |
| Conception | L'objectif de cette section est de donner une vue logique du logiciel à produire. Nous devons identifier toutes les entités du domaine; les processus qui vont agir sur ces entités; et les règles d'interaction (règles métier) | Diagramme de classes<br><br>Diagramme d'objets<br><br>Diagramme de séquence<br><br>Diagramme d'activités<br><br>Diagramme de collaboration<br><br>Diagramme d'état-transition |
| Réalisation | L'objectif de cette section est de présenter l'architecture interne du logiciel réalisé en termes de composants ainsi l'architecture de déploiement. | Diagramme de composants<br><br>Diagramme de déploiement |
| Test et Validation | L'objectif de cette section est de montrer comment le logiciel réalisé a été validé et testé dans son environnement de fonctionnement. | Diagramme de déploiement |

# Remerciements

Nous adressons nos remerciements aux personnes qui nous ont aidé dans la réalisation de ce projet.

En premier lieu, nous remercions M. Chollet, professeur à l'IUT de Montpellier, qui nous a enseigné les méthodes agiles.

Nous remercions également M. Prost, professeur à l'IUT de Montpellier, pour nous avoir enseigné la programmation orientée objet.

Nous présentons nos remerciements à M. Haettel, professeur à l'IUT de Montpellier, pour ses enseignements sur les graphes orientés.

Nous adressons de plus nos remerciements à M. Bougeret et M. Poupet, professeurs à l'IUT de Montpellier, pour leurs cours sur les algorithmes récursifs.

Nous remercions aussi Mme Messaoui, professeure à l'IUT de Montpellier, pour avoir fourni les directives de la rédaction de projet.

Nous voulons aussi remercier M. Sabatier, professeur à l'IUT de Montpellier, pour nous avoir éclairé sur la méthodologie de la conception d'applications et les diagrammes UML.

Nous souhaitons particulièrement remercier M. Marie-Jeanne, professeur à l'IUT de Montpellier. En tant que tuteur de projet, il nous a guidé dans notre travail et nous a aidé à trouver des solutions pour avancer.

# Sommaire

# Table des figures



# Glossaire

**Graphe orienté :** Un graphe orienté est une structure mathématique composée de sommets et d'arêtes qui relient certains sommets entre eux. Une arête a un sommet d'origine et un sommet d'extrémité, ce qui signifie qu'une arête d'un sommet A vers un sommet B n'est pas la même arête que celle allant du sommet B vers le sommet A. Cela introduit une notion de sens, d'où le terme "orienté".

**Listener (écouteur) :** Un écouteur est une partie du programme qui sera exécuté seulement lorsqu'elle reçoit un événement (clic de la souris, clavier, exception…).

# Introduction

Grâce à l'essor du numérique, de nombreux jeux de plateau peuvent être implémentés sous la forme de jeux vidéos, et le Ricochet-Robots en fait partie.

Programmer un jeu de Ricochet-Robot permettrait d'ajouter de nouvelles règles et, entre autres, de pouvoir résoudre le jeu automatiquement.

Nous avons deux approches au jeu Ricochet-Robots, qui sont l'approche mathématique et l'approche par la programmation.

L'approche par la programmation prend en compte la génération d'une partie de Ricochet-Robot Classique en permettant à un ou plusieurs joueurs de jouer une partie. Il faut également améliorer le modèle classique pour générer des parties aléatoires, et également permettre à un joueur de personnaliser son propre plateau. Le jeu devra comprendre une interface graphique, et optionnellement implémenter le jeu sur plateforme mobile (Android).

L'approche mathématique concerne la conception d'un algorithme de résolution automatique d'une partie standard de Ricochet-Robots.

# 1. Cahier des charges

Nous présenterons dans cette partie les différents aspects qui sont compris dans le cahier des charges.

## 1.1. Contexte et définition

Le but est de programmer un jeu de Ricochet-Robots. Le jeu se déroule sur un plateau d'un nombre n*n de cases et de murs. Sur les cases sont placés des pions appelés "robots" et des objectifs. Le jeu se déroule en tours.

Au début du tour, on tire au hasard un objectif. Le but est de déplacer le robot de la couleur de l'objectif sur celui-ci en un minimum de mouvements (Déplacer les autres robots compte également pour 1 coup). Un robot ne peut se déplacer que tout droit jusqu'à atteindre un obstacle (un bord du plateau, un mur ou un autre robot).

Le joueur ayant trouvé le plus rapidement la solution annonce son nombre de mouvements et lance un compte à rebours. Si les autres joueurs ne trouvent pas de meilleures solutions avant la fin du temps imparti, il remporte la manche.

## 1.2. Analyse de l'existant

### 1.2.1. Jeu de société

**Jeu de plateau Ricochet-robots**

Jeu de société qui comprend différentes pièces (cf. figure 1): on a un plateau de 16x16 cases avec 8 planches bifaces avec une encoche dans un coin et une marque de couleur, une plaque centrale pour lier 4 plateaux ensemble, avec des pions pour les robots de différentes couleurs (rouge, bleu, jaune, vert + argent pour la Variante), 17 jetons objets de quatre couleurs et un multicouleur pour déterminer les cases à atteindre, un sablier, des tuiles pour marquer la position de chaque robot (donc rouge, bleu, jaune et vert) et un mode d'emploi.

Le tirage d'un jeton de couleur détermine le robot à déplacer, et on peut alors gagner la partie avec ce robot.

⇒ Il y a différent plateaux disponibles dans cette version du jeu, on peut donc personnaliser les parties

*Figure n°1 : plateau de Ricochet-Robots*

### 1.2.2. Applications mobile

**Ricochet Robots - Sandro Montanari**

Implémentation classique du jeu sur une plateforme mobile, avec un plateau de dimension 16x16. La case à atteindre (avec sa position et sa couleur) est générée aléatoirement, les murs sont placés aléatoirement. On joue avec l'un des robots (rouge, bleu, jaune, vert) en fonction de la couleur de la case à atteindre. Le jeu a un chronomètre et un compteur de mouvements.

⇒ Dans cette implémentation, il n'y a qu'un seul plateau de disponible (16x16), ce qui rend le gameplay inconfortable car les cases sont trop petites pour le mobile.

**Ricochet racer - DilithiumLabs**

Implémentation du jeu sur une plateforme mobile, avec des petites voitures en guise de robot: le but est le même, il faut déplacer la voiture vers sa place de parking, en s'aidant des autres voitures de police. On ne gagne qu'avec une seule voiture (voiture de course), avec 3 voitures de polices. Les plateaux (8x8) sont déjà préparés et sont organisés par niveau (du plateau le plus simple au plateau le plus compliqué). Le jeu compte le nombre de mouvements et chronomètre le joueur et attribue des trophées en fonction de la performance du joueur.

⇒ Cette implémentation n'est pas personnalisable : Les plateaux sont tous prédéfinis et les robots sont placés dans des cases prédéfinies également.

### 1.2.3. Jeu en ligne

**www.ricochetrobots.com/RR.html**

Implémentation du jeu en flash, avec un plateau traditionnel (comme le jeu de plateau), avec 4 robots (rouge, bleu, vert et jaune). 5 modes de jeu sont proposés : le "mode normal" (le jeu classique), le "mode chrono" (2 minutes pour compléter un maximum de parties), le "mode difficile" (un plateau est généré à partir de 30 autres plateaux qui comportent au moins 10 mouvements possibles), un "mode chrono compétitif" (comme le mode chrono mais contre d'autres joueurs), et le "mode difficile du jour" (comme le mode difficile mais avec au moins 12 mouvements possibles et sans limite de temps). Les différents contrôles sont :

Une fois les robots sélectionnés, on peut les déplacer sur le plateau avec la souris.
Il y a un compteur de mouvements ainsi qu'un chronomètre.

⇒ Le plateau n'est pas personnalisable dans cette application mais les différents modes de jeu, la taille du plateau ainsi que le placement aléatoire des robots offre beaucoup de possibilités de jeux différents.

### 1.2.4. Conclusion sur l'analyse de l'existant

Nous avons appris de l'analyse de l'existant que le jeu Ricochet-Robots ne se restreint pas à un jeu de plateau simple, mais qu'il existe sur différentes plateformes, et sous des formes qui diffèrent du jeu de plateau.

L'idéal serait de pouvoir personnaliser le plateau, ou du moins en proposer plusieurs, et de placer les robots aléatoirement. Si nous souhaitons porter l'application sur mobile, il faudra vérifier qu'elle puisse bien s'adapter à ce format (ne pas faire de trop grands plateaux).

## 1.3. Description des besoins fonctionnels

### 1.3.1. Contexte

Nous travaillons sur le jeu Ricochet-Robots dans l'optique de réaliser un algorithme de résolution. Nous devons rendre le jeu jouable pour n'importe quel utilisateur, c'est-à-dire que nous devons implémenter une interface graphique. L'algorithme de résolution servira à montrer la solution la plus efficace (c'est-à-dire la plus rapide) au joueur avant, pendant ou à la fin d'une partie.

### 1.3.2. Besoins utilisateurs

L'utilisateur de ricochet-robot doit pouvoir effectuer diverses actions. Il lui est possible de paramétrer la partie (choix du plateau), de lancer une partie (avec les choix de la mettre en pause, de la reprendre, de la quitter ou de la résoudre automatiquement), de déplacer un robot et enfin de finir la partie (en la gagnant).

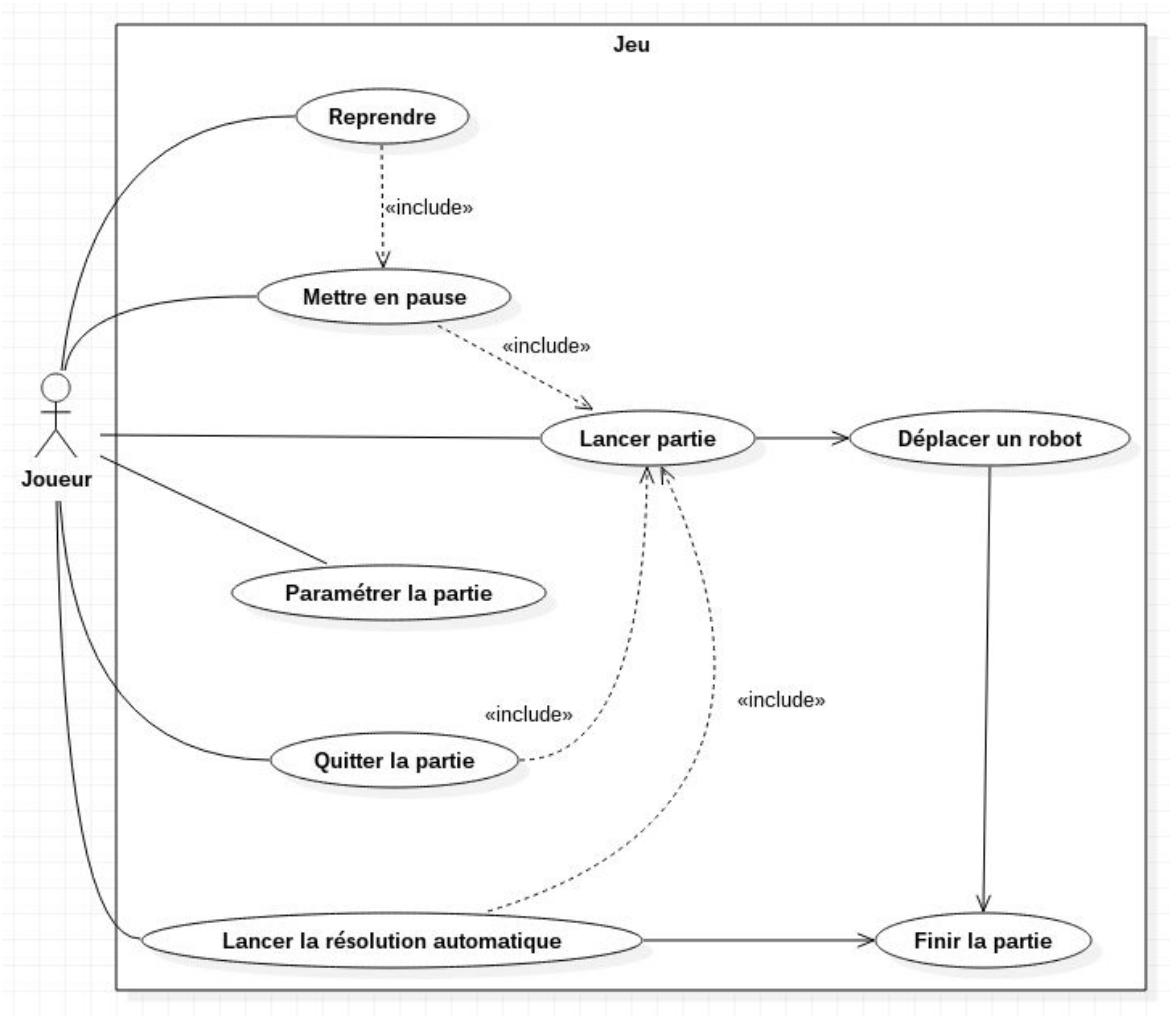



*Figure n°2 : Diagramme de Cas d'utilisation*

## 1.4. Analyse des besoins non fonctionnels

Pour ce projet, il nous a été imposé la programmation de l'algorithme de résolution d'une partie de Ricochet-Robots. Nous n'avons pas reçu de contraintes logicielles ou matérielles, nous devons seulement permettre à un utilisateur de jouer au jeu.

# 2. Rapport technique

## 2.1. Conception

### 2.1.1. Conception générale

Pour mener à bien ce projet, nous avons opté pour de la Programmation Orientée Objet avec le langage de programmation JAVA.

Étant donné les délais relativement courts qui nous ont été imposés, JAVA a été un choix logique car c'est l'un des premiers langages que nous avons utilisé à l'IUT et c'est celui que nous maîtrisons le mieux. De plus, il permettra une portabilité simplifiée sur mobile, qui est un de nos objectifs secondaires que nous aimerions réaliser si le temps nous le permet.

Pour définir un chemin vers l'objectif, nous avons choisi d'adopter une structure en graphe orienté (voir définition dans le glossaire).

La modélisation du plateau par un graphe orienté est assez naturelle car d'une case (sommet), on peut se déplacer (via les arêtes) vers 2 à 4 cases différentes, sans pour autant qu'il soit possible de réaliser le chemin inverse. Cela permet ainsi d'utiliser les cours de première année et les algorithmes associés aux graphes notamment pour la gestion des cases, et l'algorithme de résolution.

La figure 3 est un exemple de graphe orienté (à droite) modélisant un plateau (à gauche) relativement simple de 9 cases. On peut ainsi voir les déplacements possibles d'un robot en fonction de la case sur laquelle il se trouve.

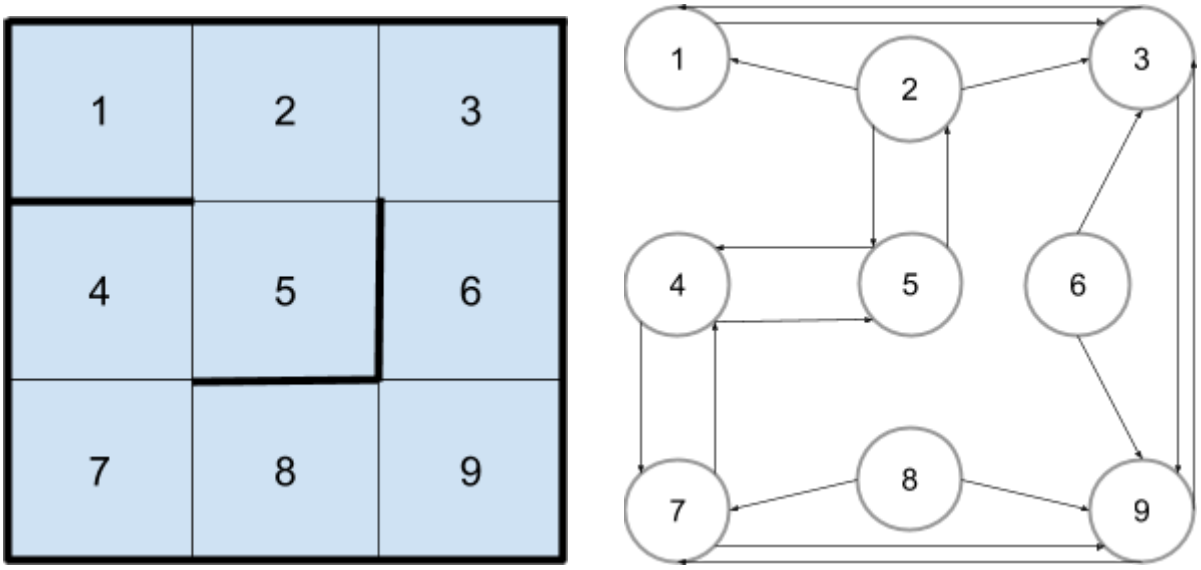



*Figure n°3 : Schéma des déplacements possibles d'un robot pour une grille 9x9*

### 2.1.2. Structure du programme

Le modèle de base est constitué de 4 classes principales : Case, Plateau, Robot et Partie (voir Figure 4).

**La classe "Case"**

C'est au travers de cette classe que nous modélisons le graphe orienté. Voici ses spécificités :

- Une Case possède au maximum 4 Cases suivantes (on les appellera CaseNext) ;
- Une Case suivante est la prochaine Case atteignable par le Robot.

Une Case qui ne possède pas de CaseNext (CaseNext == null) est considérée comme une Case Vide, c'est à dire une Case qui n'existe pas sur le plateau et où le Robot ne peut pas aller. Lorsqu'une CaseNext est vide, cela est interprété par le programme comme un Mur.

Une Case possède également un Robot (ou pas) et ses position X et Y (Les coordonnées ne sont pas utiles pour résoudre le jeu sur terminal, mais seront utilisées dans l'interface graphique).

**La classe “Robot”**

Un Robot possède une couleur (qui lui permet d’être différencié des autres Robots), et une Case (CaseActuelle). Il possède également un attribut “sélectionné” qui permet de repérer s’il a été sélectionné par le joueur ou non.

La méthode deplacerRobot (haut, bas, gauche ou droite), permet de placer le robot sur la Case suivante correspondante. Il ne se passe rien si la Case suivante est vide (Mur).

**La classe “Plateau”**

Cette classe permet de coordonner les Robots avec les Cases.

Un Plateau est composé d’un tableau de Cases de taille n*n. Il possède ensuite un tableau de Robots et les coordonnées de l’objectif à atteindre.

La méthode genererPlateau va permettre au Plateau de se créer entièrement, en instanciant les Cases et en leur attribuant les bonnes CaseNext, et en positionnant les Robots et l’objectif à atteindre.

La méthode toString() affichera les cases, les Robots et l’objectifs sur le terminal.

**La classe “Partie”**

La partie est le centre de l’application : elle fait le lien entre le Plateau, les autres classes, l’interface graphique et le joueur. Elle possède par exemple un compteur de coups et un Chronomètre.

La partie permet de définir la forme du Plateau : La taille du Plateau, l’emplacement des murs et de l’objectif, le nombre de Robots, etc.

Afin de disposer d’une grande variété de Plateaux différents, la classe Partie est abstraite. En effet, ce sont les classes filles de Partie qui définissent le Plateau, chacune ayant un plateau ou un affichage du jeu différent (Terminal ou Interface Graphique). La superClasse ‘Partie’ n’est donc pas instanciable, puisqu’elle ne possède pas les informations nécessaires à la création d’un Plateau.

En revanche, la Partie hérite des méthodes de l’interface graphique et de la classe FlecheClavierListener (Écouteur du clavier), ce qui lui permet de gérer les procédures pour les communications entre le joueur et la machine.

**Jeu sous terminal**

Dans la première version de l’application, le jeu est jouable sous terminal, sans la classe gérant l’interface graphique. Le joueur utilise les flèches du clavier pour déplacer le Robot, ainsi que certaines touches. Par exemple, La touche ECHAP permet de mettre fin au jeu.

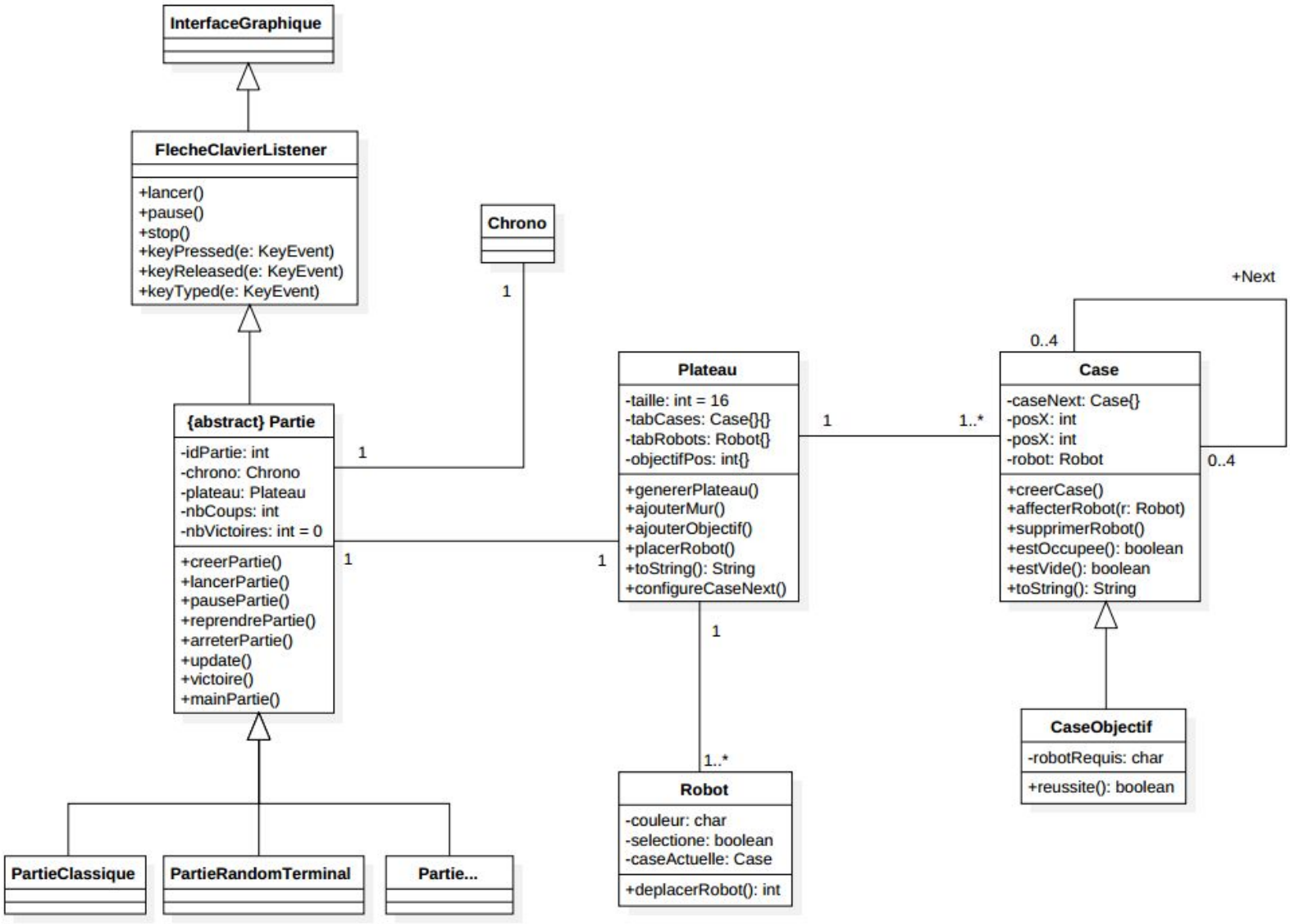



*Figure n°4 : Diagramme de classes global de l'application*

## 2.1.3. L'Interface Graphique

L'Interface graphique a été ajouté plus tard dans le projet. Elle est constituée de 3 classes principales : Fenetre, Grille et CaseGrille. Ces classes héritent de la librairie javax.Swing et peuvent implémenter les interfaces Observable et Observateur.
⇒ Voir figure n°5

**La classe "CaseGrille"**

Cette classe, qui hérite de javax.Swing.JButton, permet de gérer la partie graphique d'une Case. La plupart de ses attributs servent à changer l'affichage de la case (où sont les murs, si elle possède un Robot…). Lorsqu'elle est cliquée, elle transmet ses coordonnées à la grille, qui la transmettra au programme général pour la traiter.

**La classe “Grille”**

Cette classe possède un ensemble de CaseGrille. Elle gère la partie graphique de la grille en entier. Elle sert essentiellement de transfert d’informations entre les CaseGrille et le reste du programme. Lorsqu’une CaseGrille est cliquée, elle reçoit les coordonnées de cette case, et les transmet alors à la classe Fenetre, qui traitera cette information.

**Les interfaces “Observable” et “Observateur”**

Ces deux interfaces sont utilisées pour la communication entre les classes. Lorsqu’un événement a lieu dans une classe qui implémente Observable, la méthode updateObservateur est appelée. Cette méthode met en forme l’information et appelle ensuite la méthode update de l’Observateur. La méthode update permet alors de traiter l’information reçue.

Par exemple si la Grille reçoit un événement : une caseGrille est cliquée. Elle appelle alors la méthode ‘updateObservateur’, qui va mettre en forme l’information. La méthode ‘update’ de la classe Fenetre est alors appelée et l’information est alors reçue et traitée par la Fenetre.

**La classe “Fenetre”**

C’est la partie centrale de l’Interface Graphique. Cette classe réunit les principales fonctionnalitées pour faire apparaître la fenêtre et ses composants. Pour cela, elle hérite de la classe javax.Swing.JFrame qui gère ces fonctionnalités.

Elle possède un attribut de type Grille, qui permettra de gérer toutes les CasesGrilles et de recevoir les coordonnées des cases cliquées par le joueur.

La Classe Partie héritera de cette classe.

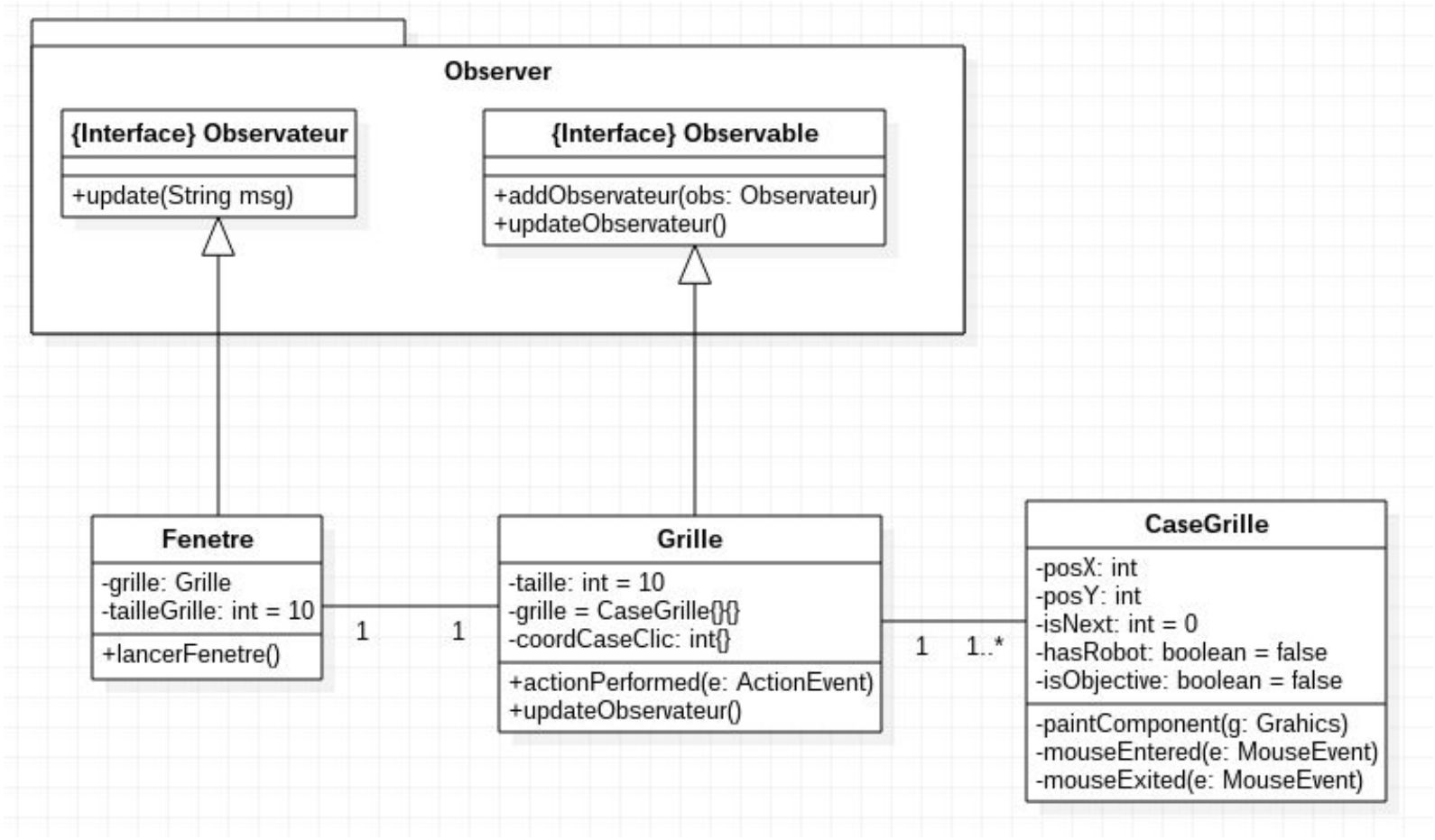



*Figure n°5 : Diagramme de classe de l'interface graphique*

### 2.1.4. Gestion des collisions avec les Murs

L'un des éléments les plus importants du projet est la possibilité de déplacer les Robots. Comme les Robots vont tout droit jusqu'à atteindre un obstacle, la collision avec les murs devient un problème très important (cf. Figure 6).

Le modèle de graphe orienté que nous avons adopté permet de répondre de manière simple à ce problème. En effet, un Robot se déplace en fait de CaseNext en CaseNext, mais il lui est interdit de se placer sur une Case vide (une case Vide est une case qui ne possède aucune CaseNext). Les murs seront alors modélisés comme des cases Vides.

Les Cases de parts et d'autres du Mur auront pour CaseNext une case vide, et comme le Robot ne peut pas entrer sur une case vide, il ne bougera pas si le joueur ordonne de se déplacer dans la direction du Mur.

⇒ Ce modèle aura tout de même un défaut : A chaque fois qu'un Robot bouge ou qu'on ajoute un mur, il faudra redéfinir chacune des CaseNext de la ligne / colonne, ce qui peut être difficile à implémenter et utiliser plus de ressources de la machine qu'avec un autre modèle.

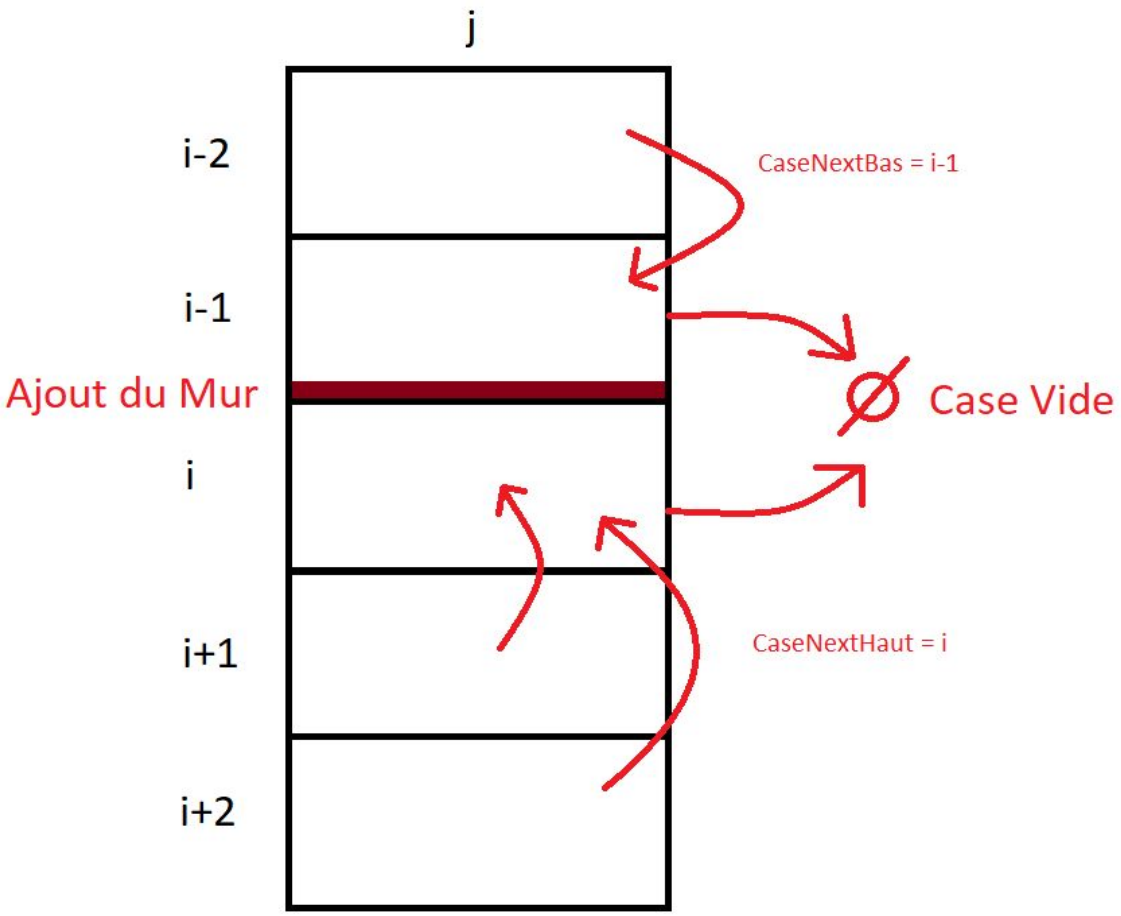



*Figure n°6 : Schéma de la gestion de la collision des Robots avec les Murs*

### 2.1.5. Conception de l'algorithme de résolution

L'algorithme de résolution a pour but d'indiquer un chemin à suivre en partant de la position du robot, jusqu'à l'objectif, et ce en un minimum de coup possible. On notera qu'il peut exister plusieurs chemins différents permettant de rallier l'objectif en ce même nombre de coup minimum. Auquel cas, l'algorithme indiquera le premier chemin qu'il trouvera.

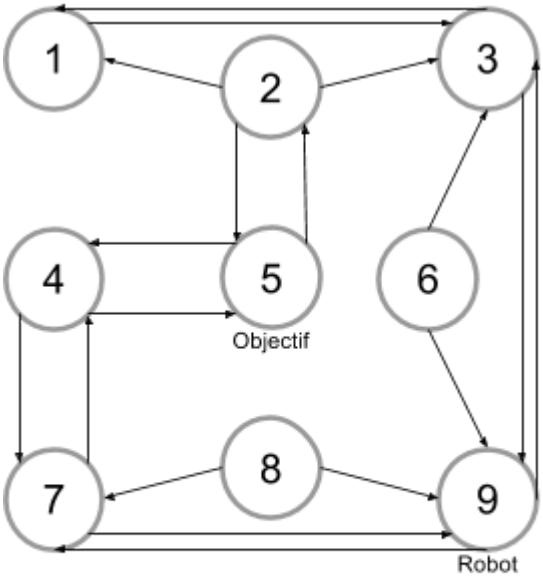
Le principe est de vérifier, en partant de la position du robot et en se déplaçant d'une case à la fois, si la case atteinte est l'objectif. Pour cela, on va marquer les cases atteignables en un déplacement en vert, et les cases déjà vérifiées en rouge, en sachant qu'une case ne doit pas être marquée en vert plus d'une fois. Cela évite de revenir à une case par un chemin plus long.

Pour garantir que le chemin trouvé est le plus court, il faut vérifier les cases vertes dans l'ordre dans lequel elles ont été marquées.

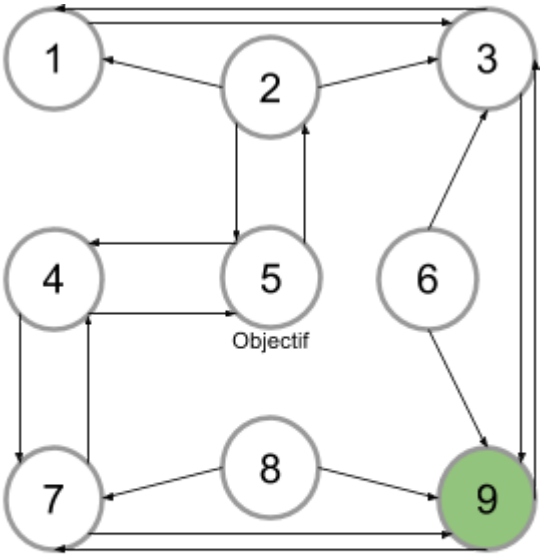
L'algorithme sauvegarde pour chaque case marquée le chemin qui a été pris jusqu'à leur marquage.

Un exemple étant souvent plus parlant, nous allons montrer le fonctionnement de l'algorithme de résolution avec un seul robot, nous allons reprendre le graphe de la figure 3 (dans la partie 2.1.1) modélisant le plateau de 9 cases (cf. figure 7).

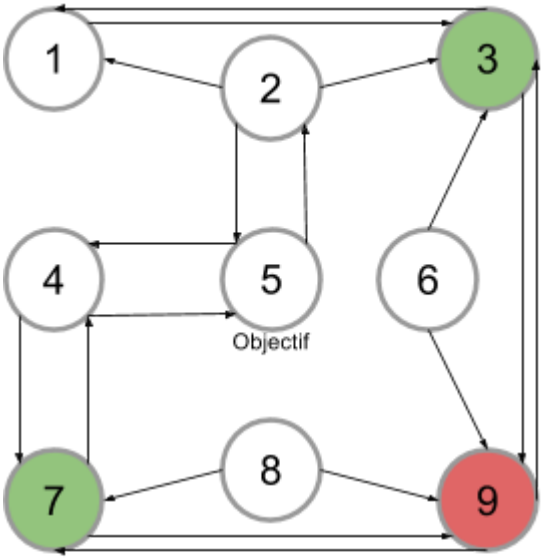
Supposons que la partie commence avec le robot sur la case 9 et que l'objectif est la case 5.



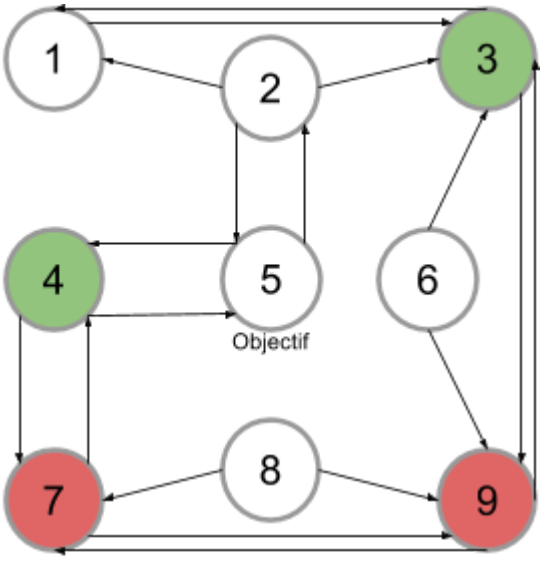

La première case marquée en vert est celle où se trouve le robot.



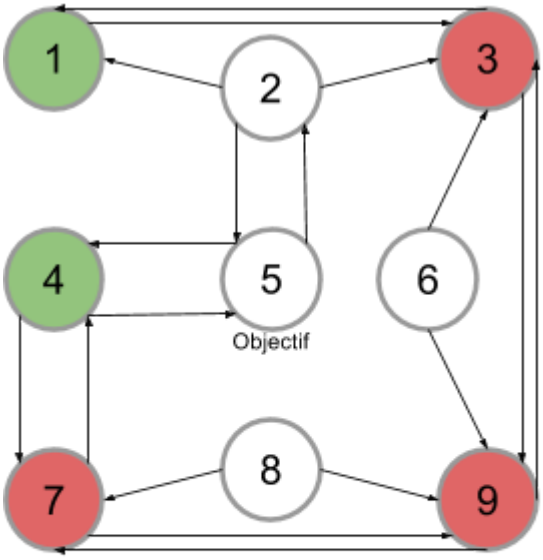

Cette case n'est pas l'objectif. On la marque donc en rouge, et on marque ses voisines en vert.



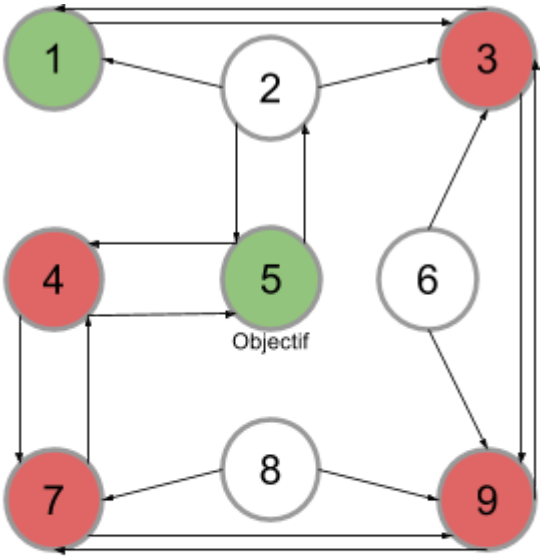

On vérifie ensuite une case en vert parmi les premières marquées. Les cases 3 et 7 ont été marquées en même temps, on choisit donc arbitrairement celle à marquer vérifier parmi les deux, ici la 7. Comme ce n'est pas l'objectif on marque en vert ses voisines non coloriées..



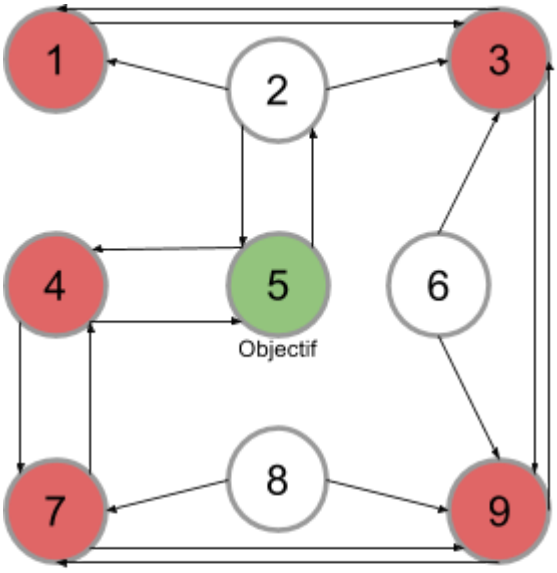

La case suivante à vérifier est celle marquée en vert depuis le plus longtemps. Ici, on n'a donc pas d'autre choix que de vérifier la case 3. Comme ce n'est pas l'objectif, on la marque en rouge, et ses voisines non marquées, en vert.




La case suivante à vérifier est la 4, qui a été marquée avant la 1. Ce n'est toujours pas l'objectif, on la marque donc en rouge et ses voisines non marquées, en vert.




On réalise la même opération pour la case 1, qui est la plus anciennement marquée. On peut remarquer qu'elle n'a pas de voisine non marquée.




La seule case verte restante est la case 5. En la vérifiant, on voit que c'est l'objectif, ce qui stoppe l'algorithme. Comme on a stocké le chemin jusqu'à elle au fur et à mesure des marquages de cases, on peut le renvoyer.

Notons que dans le cas où l'objectif n'a pas été atteint et que plus aucune case verte ne reste, cela veut dire que l'objectif n'est pas atteignable.

*Figure n°7 : Schéma des étapes de l'algorithme de résolution*

## 2.2. Réalisation

⇒ Le lien du code source peut être retrouvé en Annexe 1

### 2.2.1. Génération d'un plateau

La première version du logiciel consistait à générer un plateau de manière aléatoire. Ce modèle était facilement programmable mais nous l'avons vite abandonné car il donnait très souvent des plateaux sans solutions. Nous avons décidé de définir les plateaux manuellement

⇒ Voir le code en Annexe 2

Un plateau sera généré grâce à la méthode genererPlateau() de la Classe Plateau (cf. Annexe 2.1). Cette méthode prend en argument un tableau d'entiers à deux dimensions (murs), ainsi que les coordonnées de l'objectif.
Le tableau 'murs' est un tableau d'entier de la même taille que le Plateau. Chaque case du tableau représente une Case du plateau, et leur valeur indique le type de mur qui entoure cette Case :

- 1 : La case comporte un mur à **gauche** et un en **haut**
- 2 : La case comporte un mur en **haut** et un à **droite**
- 3 : La case comporte un mur à **droite** et un en **bas**
- 4 : La case comporte un mur en **bas** et un à **gauche**
- Autre : La case ne comporte aucun mur

La méthode va tout d'abord générer un plateau sans aucuns murs grâce à la méthode genererPlateauSansMur (cf. Annexe 2.2). Elle va ensuite placer l'objectif, en transformant la Case aux coordonnées [posXObjectif, posYObjectif] en CaseObjectif. Enfin, elle va attribuer les murs 1 par 1 grâces aux méthodes AjouterCoin(), qui appellent elles-mêmes les méthodes AjouterMur.

Si on prend l'exemple de la méthode AjouterMurHaut (cf. Annexe 2.3) : Cette méthode ajoute un mur au dessus de la case dont les coordonnées sont données en paramètre.

Tout d'abord, elle change la CaseNextHaut de la case et la remplace par une case vide. Elle fait de même avec la Case au dessus si et seulement si la Case ne se trouve pas sur la première ligne.

Elle définit ensuite chacune des CaseNext de la colonne j : L'attribut CaseNextBas de chacune des cases au dessus du mur pointe sur la première Case au dessus du mur en [i , j - 1], et l'attribut CaseNextHaut de chacune des cases au dessous du mur pointe sur la case [i , j].

### 2.2.2. Affichage du jeu sous Terminal

La première version de notre application affichait le jeu sous terminal (cf. Figure 8), et ne comptait pas d'interface graphique. Le joueur contrôlait le Robot à l'aide du Clavier.

⇒ Voir le code en Annexe 3

La Classe plateau est dotée d'une méthode toString (cf. Annexe 3) qui va transformer le plateau sous forme de chaîne de caractères :

- Les Murs du haut et bas sont représentés par le caractère : ' _ '
- Les Murs latéraux sont représentés par le caractère : ' | '
- Les Cases sont représentées par un point : ' . '
- Le Robot est représenté par la lettre : ' R '
- L'objectif est représenté par la lettre : ' O '

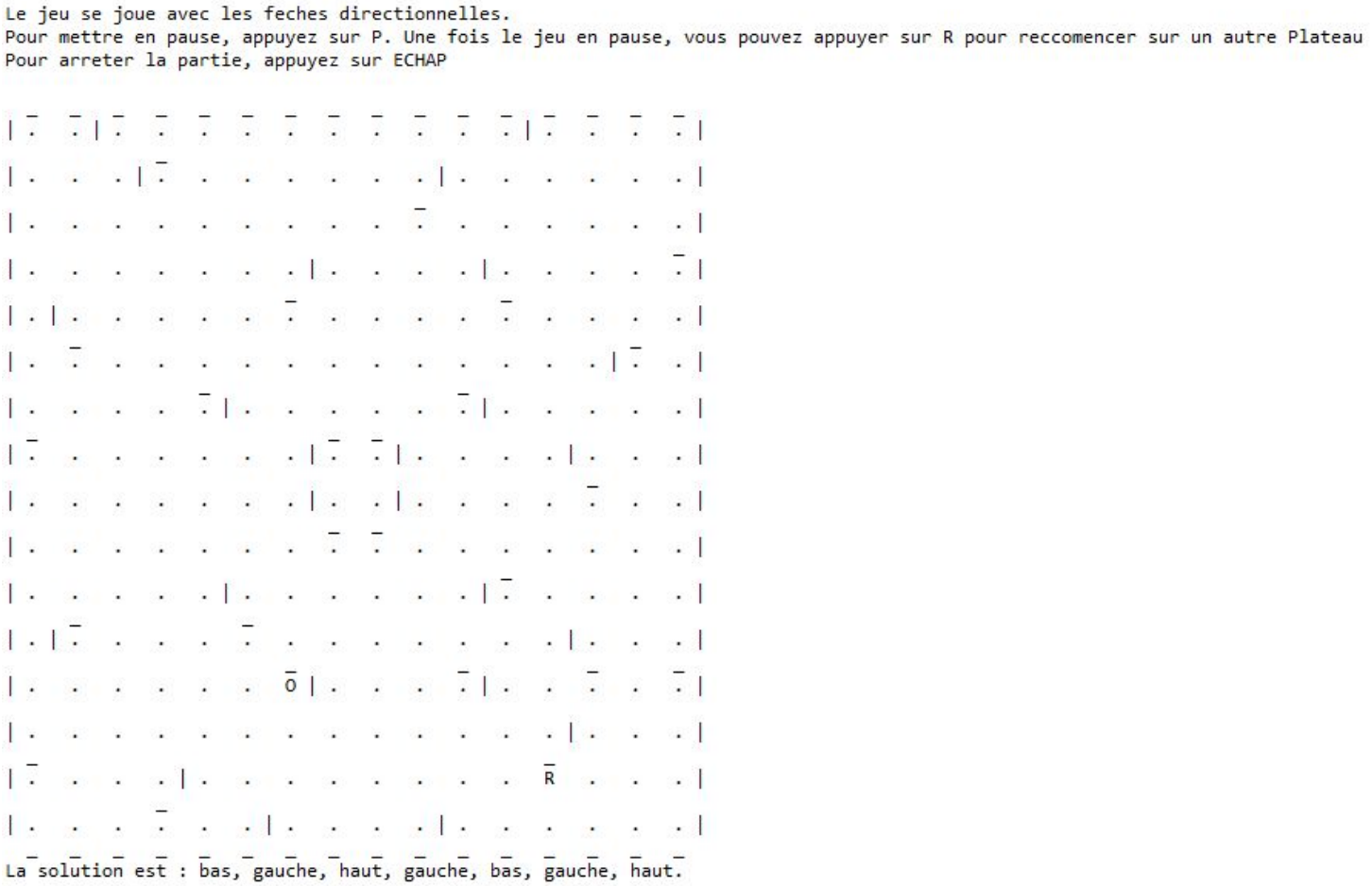



*Figure n° 8 : Apparence du jeu sous forme textuelle*

La méthode toString() (cf. Annexe 3) va étudier chaque case et l'insérer dans la chaîne de caractères.

Pour chaque Case, elle va déterminer si elle possède des murs sur l'un de ses côtés. Si c'est le cas, le caractère du mur sera inséré. S'il n'y a pas de mur, alors un caractère vide (espace) est inséré.

La Case sera affichée grâce à la méthode toString() de la Classe Case. Si la classe possède un robot, son caractère sera ' R '. Si c'est un objectif, son caractère sera ' O '. Sinon, son caractère sera ' . '.

A chaque déplacement de Robots, le programme principal appellera la méthode toString() du Plateau, et affichera la chaîne de caractère obtenue sur le terminal.

### 2.2.3. Contrôle du Robot avec le clavier

Sous la version Terminal, le Robot est contrôlé avec le clavier. Le programme utilise ce que l'on appelle un écouteur (listener) du clavier. Un écouteur est un morceau du programme qui ne s'active que lorsqu'il reçoit un événement (Ici : une touche du clavier qui est pressée).

La méthode appelée lorsqu'une touche est pressée est keyPressed, de l'interface java.awt.event.KeyListener.

⇒ Voir le code en Annexe 4

Cette méthode identifie tout d'abord la touche pressée puis effectue des actions en fonction de cette touche :

- Si le joueur appuie sur **ECHAP**, alors la partie est arrêtée automatiquement ;
- Si il appuie sur **P**, alors la partie se met en pause (Ou reprend) ;
- Si il appuie sur une **touche directionnelle**, alors le robot est déplacé grâce à la méthode deplacerRobot() de la classe Robot : Cette méthode supprime le robot de l'ancienne Case, puis l'ajoute sur la Case suivante (si elle n'est pas vide) ;
- A chaque déplacement de Robot, le programme vérifie si celui-ci se trouve sur l'objectif (Si c'est le cas, la partie se termine) ;
- A la fin d'une partie, une nouvelle partie est proposée au joueur, il peut choisir oui ou non grâce aux touches **O** et **N.**

### 2.2.4. Réalisation de l'algorithme de résolution

⇒ Voir le code en Annexe 5

Voici l'algorithme de résolution pour une partie avec un seul robot.
**Explications** :

On commence par créer trois listes :

- marked, qui contiendra les cases "marquées en vert" ;
- checked, qui contiendra les cases déjà vérifiées (cases "marquées en rouge") ;
- path, qui contiendra le chemin vers les cases marquées, à partir de la case de départ du robot.

Un chemin de path est une suite de chiffres entre 0 et 3, correspondant aux directions à prendre : 0 correspond à gauche, 1 en haut, 2 à droite et 3 en bas.

Path et marked contiennent le même nombre d'éléments, et les éléments au même indice dans les deux listes correspondent, c'est-à-dire que le chemin pour aller à la troisième case marquée est le troisième chemin de la liste path.

La première case marquée est celle où se trouve le robot. À la ligne suivante (ligne 339), on ajoute dans path le chemin qui doit être parcouru pour atteindre cette case, c'est à dire le chemin vide car on se trouve déjà sur cette case.

À la ligne 341, on stocke la case de l'objectif dans une variable pour y accéder simplement par la suite.

Après cela, on crée une variable finalPath qui contiendra le chemin vers l'objectif, puis une variable found qui indiquera si l'objectif a été trouvé.

Une fois ces variables initialisées, on va entrer dans une boucle dont on ne sortira que lorsque l'on aura trouvé l'objectif ou lorsque les cases marquées auront toutes été vérifiées. Cette dernière condition ne pourra arriver que si l'objectif n'est pas atteignable.

Voici comment se déroule un tour de boucle :

Ligne 347, on stocke la première case marquée (que l'on appellera case courante) dans une variable pour y avoir accès plus rapidement.

Premier cas : La case courante est l'objectif : On indique grâce à la variable found que l'objectif a été trouvé. On stocke ensuite le chemin vers la case courante (qui est l'objectif) dans la variable finalPath créée à cet effet précédemment.

Second cas : La case courante n'est pas l'objectif. Dans ce cas, pour chacune des 4 voisines de la case courante : si cette case voisine n'est ni marquée, ni vérifiée, ni un mur (symbolisé par une case vide), alors :

- On ajoute la case aux cases marquées ;
- À la ligne 360, on crée le chemin vers cette case à partir du chemin vers la case courante, qui est le premier de la liste path ;
- On y ajoute la direction à prendre depuis la case courante pour atteindre cette voisine ;
- On ajoute ce chemin à la liste des chemins.

Dans les trois cas, on termine le tour de boucle par l'ajout de la case courante à la liste des cases vérifiées, sa suppression des cases marquées, et enfin la suppression de son chemin de la liste des chemins.

Lorsque l'on sort de la boucle, on retourne finalPath, qui contient le chemin vers l'objectif si celui-ci a été trouvé, et qui sera vide sinon.

**Éléments de complexité**

Appelons n le nombre de cases du plateau.

La partie la plus complexe de l'algorithme est la boucle. On réalise au maximum n tours de cette boucle, car il est théoriquement possible de devoir marquer puis vérifier toutes les cases du plateau. Dans un tel scénario, sur les n tours de boucle, on entrerait n-1 fois dans le second cas décrit précédemment, ce qui entrainerait l'entrée dans une nouvelle boucle à 4 itérations.

Dans chacune de ces 4 itérations, on utilise la méthode *contains* deux fois (ligne 358). Cette méthode est relativement coûteuse car elle oblige un parcours partiel de la liste. Dans le pire des cas, l'élément se trouve à la fin de la liste et le parcours est alors total. Les listes marked et checked contiennent au maximum n éléments répartis sur les deux listes (une case ne peut pas être verte et rouge en même temps). Dans le pire des cas, ces deux appels à la méthode *contains* représenterait n opérations, ce qui amène à un total d'approximativement (n-1)*4*n opérations pour l'ensemble de l'algorithme pour un seul robot, soit une complexité en $n^2$.

### 2.2.5. Affichage graphique du jeu

⇒ Voir Annexe 6

L'interface graphique est codée à l'aide des librairies awt et Swing, deux librairies utilisées pour les interfaces graphiques de java.

L'interface graphique est composée de 3 partie (cf. figure 9) : La grille au centre (**1**), les boutons et compteurs à gauche (**2**), et un cadre affichant du texte en haut (**3**). Le jeu se contrôle entièrement à la souris.

**Partie 1**

La grille est composée des cases qui affichent chacune leur particularité (Murs, Robot…) :

- Les cases se mettent légèrement en évidence au survol de la souris, et les Cases suivantes se colorent en rouge ;
- Le Robot qui doit être amené sur l'objectif est un point bleu, les autres sont verts ;
- L'objectif est un cercle rouge ;
- Les murs sont des segments noirs sur les côtés d'une case ;
- Le Robot sélectionné est mis en évidence par des segments rouges autours.

Le joueur déplace le robot en cliquant sur les cases suivantes accessibles. Il peut également sélectionner un autre robot en cliquant dessus.

**Partie 2**

La section à gauche est composée des boutons (on pourrait y ajouter les compteurs de coups et le chronomètre). Le bouton “Recommencer” relance une nouvelle partie, avec un nouvel objectif et les robots placés dans de nouvelles cases. Le bouton “Solution” affiche la solution dans le cadre textuel du haut.

**Partie 3**

Le cadre en haut affiche du texte, il affiche initialement “Ricochet - Robots”, mais il peut également afficher la solution si on clique sur le bouton, et il affiche le mot “Victoire” si le Robot bleu a atteint l’objectif.

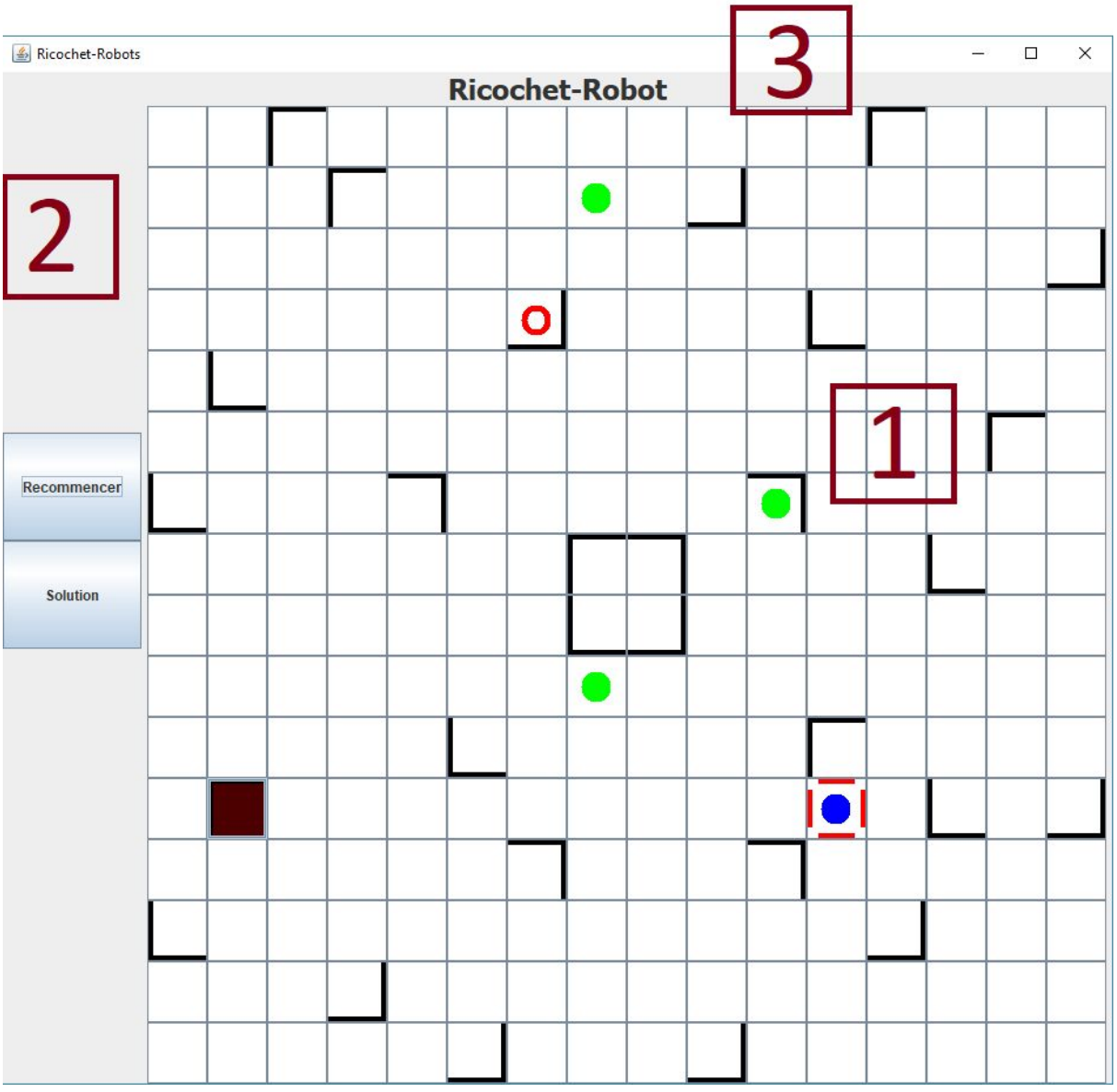



*Figure n° 9 : Affichage graphique d’une partie Classique à 4 Robots*

L’interface graphique est entièrement initialisée grâce à la méthode lancerFenetre() de la classe Fenetre (cf. Annexe 7). Cette fonction est appelée lors du lancement de la partie et initialisera tous les éléments de l’interface graphique :

- Définition la taille de la fenêtre ;
- Donne un titre à la fenêtre et initialise ses paramètres (ex : action de fermeture lorsqu'on clique sur la croix) ;
- Positionnement de la grille au centre de la fenêtre (qui a été initialisée précédemment) ;
- Création des boutons et ajoute leur écouteurs (pour effectuer des action lors du clic d'un bouton) ;
- Positionnement des boutons à gauche ;
- Création et positionnement du panneau du haut ;
- Rend la fenêtre visible : L'interface graphique est alors prête.

# 3. Résultats

## 3.1. Installation

### 3.1.1. Installation de JAVA

Le jeu est programmé en langage JAVA, et utilise donc un Machine Virtuelle Java (JVM). Il faut donc avoir Java installé pour pouvoir lancer le jeu.

**Installation pour Linux (64 bits) via un terminal :**

1. Ouvrir un Terminal.
2. Taper la commande ‘sudo apt-get install openjdk-8-jre’.
3. Taper le mot de passe administrateur si demandé.
4. Suivre les instructions d’installation.

**Installation pour Linux (64 bits) à l’aide d’un fichier binaire d’archive (format .tar.gz) :**

1. Accéder au site web http://java.com et trouver le bouton de téléchargement, puis choisir le lien de téléchargement “linux x64”.
2. Accepter le contrat de licence pour pouvoir télécharger le fichier.
3. Télécharger le fichier en vérifiant la taille du fichier pour s’assurer que le package du logiciel a été complètement téléchargé et n’est pas endommagé.
4. Ouvrir le terminal avec la commande Ctrl+alt+T, et accéder au répertoire choisi pour effectuer l’installation à l’aide de la commande “cd” (par exemple “cd /usr/java/”).
5. Déplacer le fichier binaire d’archive (format .tar.gz) dans le répertoire courant.
6. Décompresser l’archive TAR et installer JAVA avec la commande “tar zxvf *nom-de-l-archive*.tar.gz”.
7. Supprimer le fichier binaire d’archive (optionnel).

**Installation pour Windows**

1. Accéder à la page https://www.java.com/fr/download/manual.jsp et choisir le lien de téléchargement “Windows En Ligne” pour un téléchargement plus rapide.
2. Cliquer sur “Exécuter” sur la boîte de dialogue de téléchargement de fichier pour commencer le programme d’installation.
3. Cliquer ensuite sur “Installer” pour accepter les termes de la licence et poursuivre l’installation.
4. Poursuivre les boîtes de dialogue jusqu’au bouton “Fermer” de la dernière boîte de dialogue.
5. Le navigateur installé par défaut sur la machine s’ouvrira pour permettre la vérification du bon fonctionnement de JAVA.

### 3.1.2. Installation du jeu Ricochet-Robots

Pour installer le jeu, il suffit de télécharger une **archive exécutable .jar** (cf. Annexe 1). Si JAVA est installé sur l'ordinateur, alors il suffit de double-cliquer sur l'archive et le jeu se lancera.

## 3.3. Manuel d'utilisation

Ricochet-Robots se joue en sélectionnant un robot (le bleu, à amener à l'objectif, ou un vert) avec un clic gauche sur le robot souhaité.

Le robot choisi est alors encadré en rouge, et il est possible de le déplacer en cliquant sur une des destinations possibles (en haut, en bas, à gauche, ou à droite jusqu'à un obstacle).

Lorsqu'une destination possible est survolée par la souris, la case se colore en marron afin de l'indiquer.

Quand le robot bleu arrive sur la case objectif (cercle rouge), la partie est gagnée et il est possible d'en relancer une avec le bouton "recommencer" en cliquant dessus.

Si la partie devait s'avérer trop compliqué pour le joueur, celui-ci a la possibilité d'utiliser le bouton "solution" afin de voir affiché le chemin optimal à suivre pour finir le jeu.

# 4. Gestion de projet

## 4.1. Démarche personnelle

Pour ce projet, nous avons utilisé les systèmes d'exploitation Windows 10 et Ubuntu : ce sont les systèmes d'exploitation que nous utilisons tous les jours. Pour programmer, nous avons choisi le langage de programmation JAVA, comme précisé dans la partie de conception générale. Nous avons codé le jeu à l'aide des IDE Eclipse et Netbeans, ainsi que l'éditeur de texte Sublime Text.

Pour organiser notre travail, nous avons utilisé Google Drive pour le partage des documents (notamment pour la mise en place du cahier des charges). Pour la planification et le suivi de notre travail, nous avons utilisé un Trello. Pour la construction du jeu nous avons opté pour le site de versionning collaboratif Github, ainsi que l'application Gitkraken qui permet une gestion du versionning en temps réel.

Nous avons utilisé des applications de messagerie instantanée comme Skype, Discord et Messenger pour communiquer entre nous, ainsi que Gmail pour communiquer avec notre tuteur. Nous avons également organisé des rencontres régulières avec notre tuteur pour discuter de l'avancée du projet.

## 4.2. Planification des tâches

Nous avons planifié l'avancement de notre projet en cinq sprints de deux semaines et un sprint d'une semaine, pour reprendre la méthode agile Scrum. Nous avons rencontré notre tuteur à chaque fin de sprint.

Le premier sprint (du 23 Octobre au 7 Novembre) a été dédié à la conception et à une première phase de développement du jeu. C'est donc pendant ce sprint que nous avons réalisé le cahier des charges. Nous devions également présenter une première version du jeu avec un affichage sur le terminal en utilisant les touches du clavier.

Le deuxième sprint (du 8 Novembre au 21 Novembre) a été dédié à une première approche de l'implémentation d'une interface graphique et de l'algorithme de résolution pour un robot.

Le troisième sprint (du 22 Novembre au 8 Décembre) a été consacré à l'implémentation de l'interface graphique pour un robot, et de l'algorithme de résolution pour un robot, et qui affichait la solution sur le terminal.

Pendant le quatrième sprint (du 9 au 19 Décembre) et cinquième sprint (du 20 Décembre au 2 Janvier), nous nous sommes concentré sur la rédaction du rapport.

Lors du sixième et dernier sprint (du 3 au 10 Janvier), nous avons adapté notre interface graphique pour deux robots, et nous avons terminé la rédaction du rapport.

## 4.3. Bilan critique par rapport au cahier des charges

Notre tuteur nous avait proposé de coder l'algorithme de résolution pour trois voire quatre robots, mais nous avons été prévenus des contraintes techniques. Nous avons donc pu implémenter un algorithme de résolution uniquement pour un seul robot.

Nous voulions accessoirement mettre en place une application mobile sous Android (donc codé en JAVA) suite à l'analyse de l'existant, mais nous n'avons pas pu le faire par manque de temps.

# 5. Conclusions

## 5.1. Réalisation par rapport aux objectifs

Nous avions pour objectifs de porter le jeu de plateau Ricochet-Robots dans une application JAVA et d'y implémenter un algorithme de résolution. Nous avons donc pu coder une interface graphique afin que le jeu soit jouable pour un ou plusieurs joueurs, en jouant à l'aide la souris, ainsi que l'affichage de la meilleure solution possible en déplaçant un robot.

## 5.2. Perspective de développement ultérieur

Si nous continuons ce projet, nous pourrions y ajouter des fonctionnalités telles qu'un algorithme de résolution pour plusieurs robots, des compteurs de coups et de nombre de victoires sur l'interface graphique et également un chronomètre. Nous pourrions également y ajouter un menu principal et porter le jeu sous mobile. Nous pourrions également continuer l'interface graphique en affichant sur le plateau le parcours de la meilleure solution.

## 5.3. Bilan personnel

Pendant ce projet, nous avons pu appliquer ce que nous avions appris l'année précédente et lors de ce semestre : la programmation orientée objet, les graphes orientés, la conception d'applications, les algorithmes récursifs et les méthodes agiles.

D'un point de vue humain, nous avons appris à mieux communiquer en groupe en utilisant les outils de gestion de projet (Trello) et de versionning (Git).

# Bibliographie

## 1. Documentation en ligne

[1] “Ricochet Robots | Board Game | BoardGameGeek”, BoardGameGeek, [En ligne]. Disponible sur :
https://boardgamegeek.com/boardgame/51/ricochet-robots
[Consulté le : SPRINT 1]

[2] Sandro Montanari, “Ricochet Robots – Android Apps on Google Play”, Google Play, [En ligne], Disponible sur :
https://play.google.com/store/apps/details?id=shifty.ricochet.robots&hl=en
[Consulté le : SPRINT 1]

[3] DilithiumLabs, “Ricochet Racer – Android Apps on Google Play”, Google Play, [En ligne], Disponible sur :
https://play.google.com/store/apps/details?id=com.dilithiumlabs.ricochet&hl=en
[Consulté le : SPRINT 1]

[4] Michaels Freeman, “Mack’s Ricochet Robots site”, Ricochet Robots daily puzzle, [En ligne], Disponible sur :
www.ricochetrobots.com/RR.html
[Consulté le : SPRINT 1]

[5] Cyrille Herby, “Apprenez à programmer en Java”, OpenClassrooms, [En ligne], Disponible sur :
https://openclassrooms.com/courses/apprenez-a-programmer-en-java/notre-premiere-fenetre
[Consulté le : SPRINT 2]

## 2. Librairies Java

[6] Oracle, “java.awt (Java Platform SE 7 )”, *Oracle*, [En ligne], Disponible sur :
https://docs.oracle.com/javase/7/docs/api/java/awt/package-summary.html
[Consulté le : **SPRINT 3**]

[7] Oracle, “javax.swing (Java Platform SE 7 )”, *Oracle*, [En ligne], Disponible sur :
https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html
[Consulté le : **SPRINT 3**]

[8] Oracle, “java.util (Java Platform SE 8 )”, *Oracle*, [En ligne], Disponible sur :
https://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html
[Consulté le : **SPRINT 3**]

## 3. Installation de Java

[9] Oracle, “Instructions d’installation de Java pour Linux 64 bits”, *Java by Oracle*, [En ligne], Disponible sur :
https://www.java.com/fr/download/help/linux_x64_install.xml
[Consulté le: 23-dec-2017]

[10] Oracle, “Comment télécharger et installer manuellement Java sur mon ordinateur Windows ?”, *Java by Oracle*, [En ligne], Disponible sur :
https://www.java.com/fr/download/help/windows_manual_download.xml
[Consulté le : 23-dec-2017]

# Annexes

## Annexe 1 : Code source, Javadoc et exécutables

**Lien du projet GitHub :**
https://github.com/D-A-W-A/Ricochet-Robots.git

**Lien de la Javadoc (Egalement présente sur gitHub) :**
https://github.com/D-A-W-A/Ricochet-Robots/blob/master/Javadoc%20-%20Ricochet-Robots.rar

**Lien de l'exécutable du programme :**
Télécharger l'un de ces exécutables et double-cliquer pour jouer au jeu (Si Java est installé)

- Dossier des exécutables :
  https://github.com/D-A-W-A/Ricochet-Robots/tree/master/Exécutables

- Partie classique à 1 Robot :
  https://github.com/D-A-W-A/Ricochet-Robots/blob/master/Ex%C3%A9cutables/Ricochet%20-%20PartieClassique%20-%201%20Robot.jar?raw=true

- Partie classique à 4 Robots:
  https://github.com/D-A-W-A/Ricochet-Robots/blob/master/Ex%C3%A9cutables/Ricochet%20-%20PartieClassique%20-%204%20Robot.jar?raw=true

- Partie petite à 1 Robot :
  https://github.com/D-A-W-A/Ricochet-Robots/blob/master/Ex%C3%A9cutables/Ricochet%20-%20PartiePetite%20-%201%20Robot.jar?raw=true

- Partie petite à 4 Robots :
  https://github.com/D-A-W-A/Ricochet-Robots/blob/master/Ex%C3%A9cutables/Ricochet%20-%20PartiePetite%20-%204%20Robot.jar?raw=true

# Annexe 2 : Code de la génération d'un plateau

## 2.1. Méthode genererPlateau() de la classe Plateau

```
/**
 * Genere un plateau grace a un tableau d'entiers qui definit les murs :<br>
 * - 1 : Un angle Gauche-Haut<br>
 * - 2 : Un angle Haut-Droite<br>
 * - 3 : Un angle Droite-Bas<br>
 * - 4 : Un angle Bas-Gauche<br>
 * - Autre : Une Case normale, sans aucun mur<br>
 * <br>
 * Prereq : Le tableau murs doit etre de la meme taille que le plateau
 *
 * @param murs
 *            : le tableau des murs
 * @param posXObjectif
 *            : La position x de l'objectif
 * @param posYObjectif
 *            : La position y de l'objectif
 *
 *
 */
public void genererPlateau(int[][] murs, int posXObjectif, int posYObjectif) {
    // Genere d'abord un plateau sans mur
    genererPlateauSansMur();

    // Transforme une case du tableau en CaseObjectif
    ajouterObjectif(posXObjectif, posYObjectif);

    // Ajoute les murs en fonction du tableau 'murs'
    for (int i = 0; i < murs.length; i++) {
        for (int j = 0; j < murs.length; j++) {

            // Ajoute un mur a gauche et en haut de la case [i,j]
            if (murs[i][j] == 1)
                ajouterCoinGH(i, j);

            // Ajoute un mur en haut et a droite de la case [i,j]
            if (murs[i][j] == 2)
                ajouterCoinHD(i, j);

            // Ajoute un mur a droite et en bas de la case [i,j]
            if (murs[i][j] == 3)
                ajouterCoinDB(i, j);

            // Ajoute un mur en bas et a gauche de la case [i,j]
            if (murs[i][j] == 4)
                ajouterCoinBG(i, j);
        }
    }
}
```

## 2.2. Méthode genererPlateauSansMur() de la classe plateau

```java
/**
 * Genere un plateau sans aucun murs sauf les bords du plateau
 * Attribue les CaseNext et initialise les coordonnees de chaque case
 */
public void genererPlateauSansMur() {
    Case caseNonVide = Case.creerCase();
    Case caseVide = new Case();

    // Tableau de cases non vides qui sera attribuee
    // temporairement a toutes les cases
    Case[] tNonVide = new Case[4];
    for (int i = 0; i < 4; i++) {
        tNonVide[i] = caseNonVide;
    }

    // Initialisation du tableau de Cases
    tabCases = new Case[taille][taille];

    // Ajout du plateau global, on remplit toutes les cases du tableau
    // par des variables de type Case
    for (int i = 0; i < taille; i++) {
        for (int j = 0; j < taille; j++) {
            tabCases[i][j] = new Case();
            tabCases[i][j].setCaseNext(tNonVide);
        }
    }

    // Ajout du bord du plateau haut
    for (int j = 0; j < taille; j++) {
        tabCases[0][j].setCaseNextHaut(caseVide);
    }

    // Ajout du bord du plateau gauche
    for (int i = 0; i < taille; i++) {
        tabCases[i][0].setCaseNextGauche(caseVide);
    }

    // Ajout du bord du plateau droit
    for (int i = 0; i < taille; i++) {
        tabCases[i][taille - 1].setCaseNextDroite(caseVide);
    }

    // Ajout du bord du plateau bas
    for (int j = 0; j < taille; j++) {
        tabCases[taille - 1][j].setCaseNextBas(caseVide);
    }

    // On remplit les attributs de chaque Case
    insererCoordonees();
    // Attribue les bonnes CaseNext
    configureCaseNextPlateau();
}
```

## 2.3. Méthode AjouterMurHaut(int i, int j) de la classe plateau

```
/**
 * Ajoute un mur sur le plateau aux coordonnes (i,j) et sur le haut de la case
 * et redefinit toutes les autres cases pour pouvoir interagir avec ce nouveau
 * mur <br>
 *
 * @param i
 *            la ligne de la case
 * @param j
 *            la colonne de la case
 */
public void ajouterMurHaut(int i, int j) {
    // Remplacement de la case suivante par une Case vide
    tabCases[i][j].setCaseNextHaut(new Case());

    // Si on ne se trouve pas sur la premiere ligne, alors on remplace la
    // CaseNext de la case au dessus par une case vide
    if (i > 0)
        tabCases[i - 1][j].setCaseNextBas(new Case());

    // On redefinis la caseNextBas de toutes les cases au dessus du mur
    if (i > 1) {
        if (!tabCases[i - 1][j].getCaseNextHaut().estVide()) {
            int k = 2;
            while (!tabCases[i - k][j].getCaseNextHaut().estVide()) {
                tabCases[i - k][j].setCaseNextBas(tabCases[i - 1][j]);
                k++;
            }
            tabCases[i - k][j].setCaseNextBas(tabCases[i - 1][j]);
        }
    }

    // On redefinis la caseNextHaut de toutes les cases en dessous du mur
    if (i < taille - 1) {
        if (!tabCases[i][j].getCaseNextBas().estVide()) {
            int k = 1;
            while (!tabCases[i + k][j].getCaseNextBas().estVide()) {
                tabCases[i + k][j].setCaseNextHaut(tabCases[i][j]);
                k++;
            }
            tabCases[i + k][j].setCaseNextHaut(tabCases[i][j]);
        }
    }
}
```

## Annexe 3 : Méthode toString() de la classe Plateau

```
/**
 * Genere une String du Plateau, les elements du plateau
 * sont representes par :<br>
 * <ul>
 * <li> . : Pour une Case </li>
 * <li> _ : Pour mur du haut ou bas </li>
 * <li> | : Pour mur de gauche ou droite</li>
 * </ul>
 */
public String toString() {
    StringBuilder s = new StringBuilder("");

    // Dessine le bord haut du plateau
    for (int i = 0; i < taille; i++) {
        if (tabCases[0][i].getCaseNextHaut().estVide())
            s.append("  _ ");
        else
            s.append("    ");
    }

    // Pour chaque ligne du plateau
    for (int i = 0; i < taille; i++) {
        // Passe a la ligne de Case du dessous
        s.append("\n|");

        // Affiche la Case + Les murs sur les cotes
        for (int j = 0; j < taille; j++) {
            if (tabCases[i][j].getCaseNextDroite().estVide())
                s.append(" " + tabCases[i][j].toString() + " |");
            else
                s.append(" " + tabCases[i][j].toString() + "  ");
        }

        // Gere les murs du dessus
        s.append("\n ");
        for (int j = 0; j < taille; j++) {
            if (tabCases[i][j].getCaseNextBas().estVide())
                s.append(" _  ");
            else
                s.append("    ");
        }
    }
    return s.toString();
}
```

## Annexe 4 : Méthode keyPressed : Action du clavier

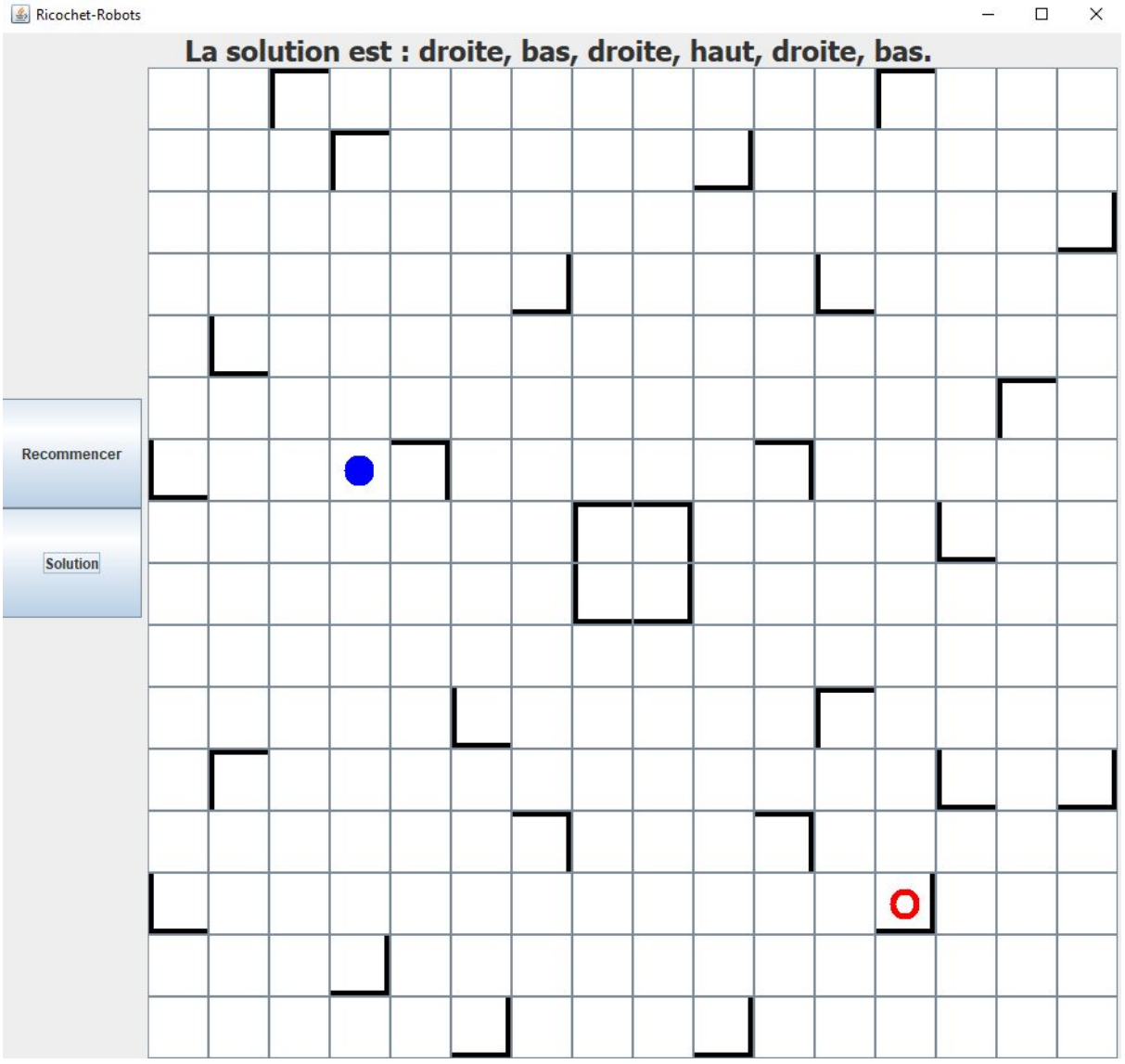
```
 * Listener du clavier. Il se comporte de 3 facons differentes : lors d'une partie arretee,
 * lors d'une partie lancee, lors d'une pause.<br>
 * Si le joueur appuie sur ECHAP, la partie s'arrete immediatement. <br>
 * <br>
 * Lors d'une Partie : <br>
 * == Si le joueur appuie sur une fleche directionelle, Le Robot est deplacee
 * dans la direction indiquee <br>
 * == Si le joueur Appuie sur P : la partie est mise en pause <br>
 * <br>
 * Lors d'une Pause : <br>
 * == Si le joueur appuie sur P, la partie reprend <br>
 * == Si le joueur appuie sur R, une nouvelle manche est lancee<br>
 * <br>
 * Lorsque la partie est arrettee :<br>
 * == Si le joueur appuie sur O, une nouvelle manche est lancee
 * == Si le
 * joueur appuie sur N, La partie se termine et se ferme.
 */
public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    if (code == KeyEvent.VK_ESCAPE) {
        arreterPartie();
    }
    if (etatPartie == 1) {
        if (e.getKeyChar() == 'P' || e.getKeyChar() == 'p')
            pausePartie();

        int deplacement = 0;
        if (code == KeyEvent.VK_DOWN) {
            deplacement = plateau.getTabRobots()[robotSelectionne].deplacerRobotBas();
        } else if (code == KeyEvent.VK_UP) {
            deplacement = plateau.getTabRobots()[robotSelectionne].deplacerRobotHaut();
        } else if (code == KeyEvent.VK_LEFT) {
            deplacement = plateau.getTabRobots()[robotSelectionne].deplacerRobotGauche();
        } else if (code == KeyEvent.VK_RIGHT) {
            deplacement = plateau.getTabRobots()[robotSelectionne].deplacerRobotDroite();
        }
        if (deplacement == 1) {
            nbCoups++;
            update();
        }
        if (plateau.getObjectif().reussite()) {
            victoire();
        }
    } else if (etatPartie == 2) {
        if (e.getKeyChar() == 'P' || e.getKeyChar() == 'p')
            reprendrePartie();
        else if (e.getKeyChar() == 'r' || e.getKeyChar() == 'r')
            mainPartieAux();
    } else if (etatPartie == 0) {
        if (e.getKeyChar() == 'O' || e.getKeyChar() == 'o') {
            etatPartie = 1;
            mainPartieAux();
        }
        if (e.getKeyChar() == 'N' || e.getKeyChar() == 'n')
            arreterPartie();
    }

}
```

## Annexe 5 : Code de l'algorithme de résolution

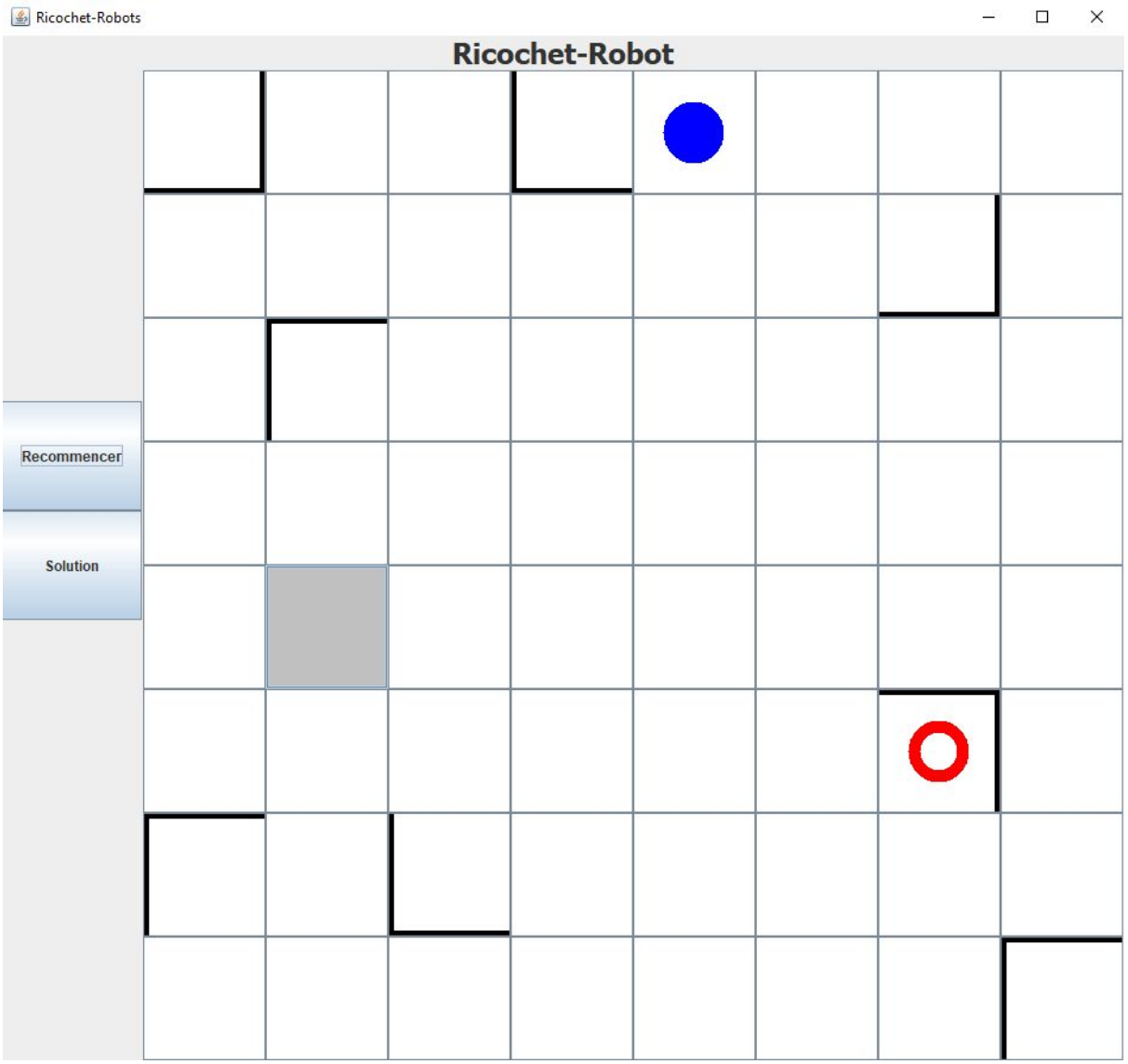
```
328     /**
329      * Les coups retournés sont des entiers sous la forme : 0 = gauche, 1 = haut, 2 = droite et 3 = bas
330      * Si la liste est vide, l'objectif n'est pas atteignable
331      * @return La liste des coups
332      */
333     protected LinkedList<Integer> solve1() {
334         LinkedList<Case> marked = new LinkedList<Case>();
335         LinkedList<Case> checked = new LinkedList<Case>();
336         LinkedList<LinkedList<Integer>> path = new LinkedList<LinkedList<Integer>>();
337 
338         marked.add(plateau.getTabRobots()[0].getCaseActuelle());
339         path.add(new LinkedList<Integer>());
340 
341         Case objectif = plateau.getObjectif();
342         LinkedList<Integer> finalPath = new LinkedList<Integer>();
343 
344         boolean found = false;
345 
346         while(!found && !marked.isEmpty()) {
347             Case current = marked.getFirst();
348 
349             if (current.equals(objectif)){
350                 found = true;
351                 finalPath.addAll(path.peek());
352             }
353 
354             else {
355                 for (int i=0; i<4; i++) {
356                     Case nextI = current.getCaseNext(i);
357 
358                     if(!nextI.estVide()&&!checked.contains(nextI)&&!marked.contains(nextI)) {
359                         marked.add(nextI);
360                         LinkedList<Integer> nextPath = new LinkedList<Integer>(path.peek());
361                         nextPath.add(i);
362                         path.add(nextPath);
363                     }
364                 }
365             }
366 
367             checked.add(current);
368             marked.remove();
369             path.remove();
370         }
371         return finalPath;
372     }
```

# Annexe 6 : Dispositions des plateaux

## 6.1. Plateau Classique de 16x16 cases



## 6.2. Petit plateau de 8x8 cases



## Annexe 7 : Méthode lancerFenetre() de la classe Fenetre

```
/**
 * Fonction a lancer pour initialiter l'interface graphique
 */
public void lancerFenetre() {
    // Obtiens et configure les dimentions de la fenetre
    Toolkit tk = Toolkit.getDefaultToolkit();
    Dimension screenSize = tk.getScreenSize();
    int screenHeight = screenSize.height;
    int screenWidth = screenSize.width;
    this.setSize(screenWidth / 2, screenHeight / 2 + screenHeight / 3);

    // Definis le titre et les parametres de la fenetre
    this.setTitle("Ricochet-Robots");
    this.setLocationRelativeTo(null);
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Met la grille au milieu
    this.getContentPane().add(grille, BorderLayout.CENTER);

    // Creation des boutons et ajout du listener this
    JButton recommencer = new JButton("Recommencer");
    recommencer.addMouseListener(this);

    JButton solution = new JButton ("Solution");
    solution.addMouseListener(this);

    // Creation du menu de gauche (Compose des boutons crees precedemment)
    JPanel menuGauche = new JPanel();
    for (int i = 0; i < 3; i++)
        menuGauche.add(new JPanel());
    menuGauche.add(recommencer);
    menuGauche.add(solution);
    GridLayout menulayout = new GridLayout(9, 1);
    menuGauche.setLayout(menulayout);

    // Ajout du menuGauche a gauche
    this.getContentPane().add(menuGauche, BorderLayout.WEST);

    // Creation du Titre
    JLabel titreLabel = new JLabel ("Ricochet-Robot");
    Font police = new Font("Tahoma", Font.BOLD, 24);
    titreLabel.setFont(police);
    titreLabel.setHorizontalAlignment(JLabel.CENTER);

    // Ajout du titre en haut
    this.getContentPane().add(titreLabel, BorderLayout.NORTH);

    // Affiche la fenetre (Pas de pack : pas besoin + fait bugger)
    this.setVisible(true);
}
```

# Résumé en français

Le projet Ricochet-Robots consiste en l'implémentation en JAVA du jeu de plateau "Ricochet-Robots". Le but est de permettre à un utilisateur de jouer à "Ricochet-Robots" à l'aide de la souris, et de donner la solution la plus efficace pour résoudre une partie. Ces deux parties du projet ont été pensées autour de la théorie des graphes orientés, ce qui nous a permis de lier les mathématiques à la conception d'un jeu. L'algorithme de résolution est capable de donner la solution la plus efficace (c'est-à-dire avec le moins de mouvements possible) en faisant bouger un ou deux robots.

L'interface graphique a été codée avec les librairies java.awt et javax.swing, et l'algorithme de résolution d'une partie est en JAVA. La gestion des cases du plateau a été codée avec des algorithmes récursifs.

## Mots-clés

JAVA, libraires, algorithme de résolution, graphes orientés, conception de jeu, algorithmes récursifs.

# Résumé en anglais

The project Ricochet-Robots consists in implementing the board game "Ricochet-Robots" in JAVA. The aim is to allow a user to play "Ricochet-Robots" using a mouse, and to give the most efficient solution to win. Both main parts of the projects were built around oriented graphs theory, which allowed us to link mathematics to game design. The solving algorithm is able to complete the game in the most efficient way (meaning with the least amount of moves) moving one or two robots.

The graphic-user interface was coded using java.awt and javax.swing libraries and the solving algorithm of the game is in JAVA. Box management on the board was coded using recursive algorithms.

## Key-words

JAVA, libraries, solving algorithm, oriented graphs, game design, recursive algorithms.