UNIVERSITÉ
CAEN
NORMANDIE

ADVANCED RICOCHET-ROBOTS

CAJEAT Romain - 21808338

DEMOLINIS Rémy - 21702827

GARREAU Emmanuel - 21700336

GOURNAY Alexandre - 21806749

L2 Informatique - Groupe 1B

Rapport Conception Logicielle Avancée

2019-2020

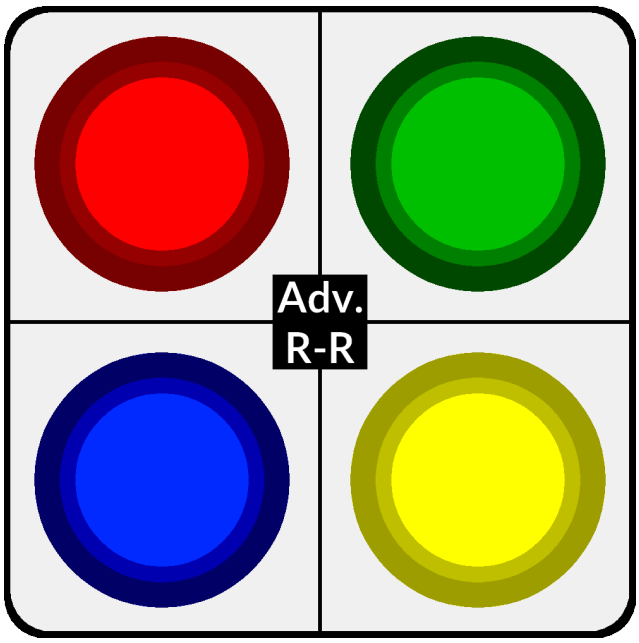


Table des matières

1 Objectifs du projet

1.1 Description du concept

Le concept est celui du jeu de société "Ricochet-Robots" qui est de trouver la séquence de mouvements qui permettra à un robot donné parmi quatre d'atteindre un objectif désigné sur une case du plateau de jeu. Cependant, les robots ne peuvent que se déplacer en ligne droite jusqu'à rencontrer un obstacle. Le but de ce projet est de développer un programme permettant de trouver une solution optimale pour toute situation du jeu.

1.2 Ce que nous avions à faire

Dans un premier temps, nous devions concevoir le moteur du jeu puis implémenter un algorithme de résolution naïf, appelé A*. Dans un second temps, il fallait alors proposer des méthodes d'optimisation de l'algorithme. Enfin, nous avons pu réaliser une interface graphique complète, offrant un bon nombre de possiblités à l'utilisateur. De plus, nous avons souhaité ajouter une partie jouable à l'application.

1.3 Ce qui existe déjà

Il existe déjà le fameux jeu de société cité ci-dessus sortie en 1999, ainsi que quelques versions web du principe de ce jeu. Il existe également l'algorithme A* que l'on peut probablement retrouver sur internet sous la forme de pseudo-code.

FIGURE 1 – Exemple d'une partie en cours du jeu de société. [1]

2 Fonctionnalités implémentées

Voici les fonctionnalités que nous avons implémentées :

- **Solveur :** Le solveur fonctionne avec l'algorithme A* dont la description se trouve dans la sous-section 4.1 en page 5 du rapport.
- **Interface Graphique :** Nous avons eu le temps d'ajouter une interface graphique au projet pour rendre l'application plus pratique et plaisante d'utilisation et de visualisation. Ainsi, vous pouvez jouer au jeu Ricochet-Robots, réaliser des expérimentation de résolution de terrain de jeu ou encore créer vos propres terrains confortablement.
- **Interface Terminale :** Nous avons préféré laisser la plupart des fonctionnalités dans l'interface graphique puisque beaucoup plus visuelle et pratique.
  Toutefois l'interface terminal n'a pas été délaissée, nous y avons implémenté le jeu Ricochet-Robots, se déroulant en tour par tour avec 4 pions.
- **Jouabilité :** Nous avons établi la possibilité de jouer des parties de Ricochet-Robots en plus du solveur.
- **Expérimentations :** Possibilité de tester le solveur avec Algorithme A* mais aussi de visualiser la solution trouvée.
- **Création de terrain(s) :** Possibilité de concevoir des terrains de toute taille, d'en placer les murs, pions et objectifs.

3 Organisation du projet

3.1 Déroulement du projet

Dans un premier temps, nous avons commencé par un diagramme de classes (provisoire aux vu de certains changements), pour avoir une idée plus claire quant à la réalisation du projet.

Nous sommes ensuite passés à la phase de développement des fonctionnalités principales, à savoir la création du plateau (en JSON), des murs, des mouvements possibles et des pions. Une fois les classes principales fonctionnelles, nous avons mis en place une interface terminale (avant l'arrivée de l'interface graphique) pour pouvoir visualiser le fonctionnement du jeu sans le solveur (la partie jouabilité).

Dans un autre temps, nous avons débuté l'élaboration de l'interface graphique en parallèle de celle du solveur avec l'algorithme A*, dans un souci de gain de temps.

Et finalement, nous avons entamé la phase finale du projet comprenant la réalisation de ce rapport, de la présentation pour notre oral et des dernières fonctionnalités à ajouter.

3.2 Diagramme de Gantt

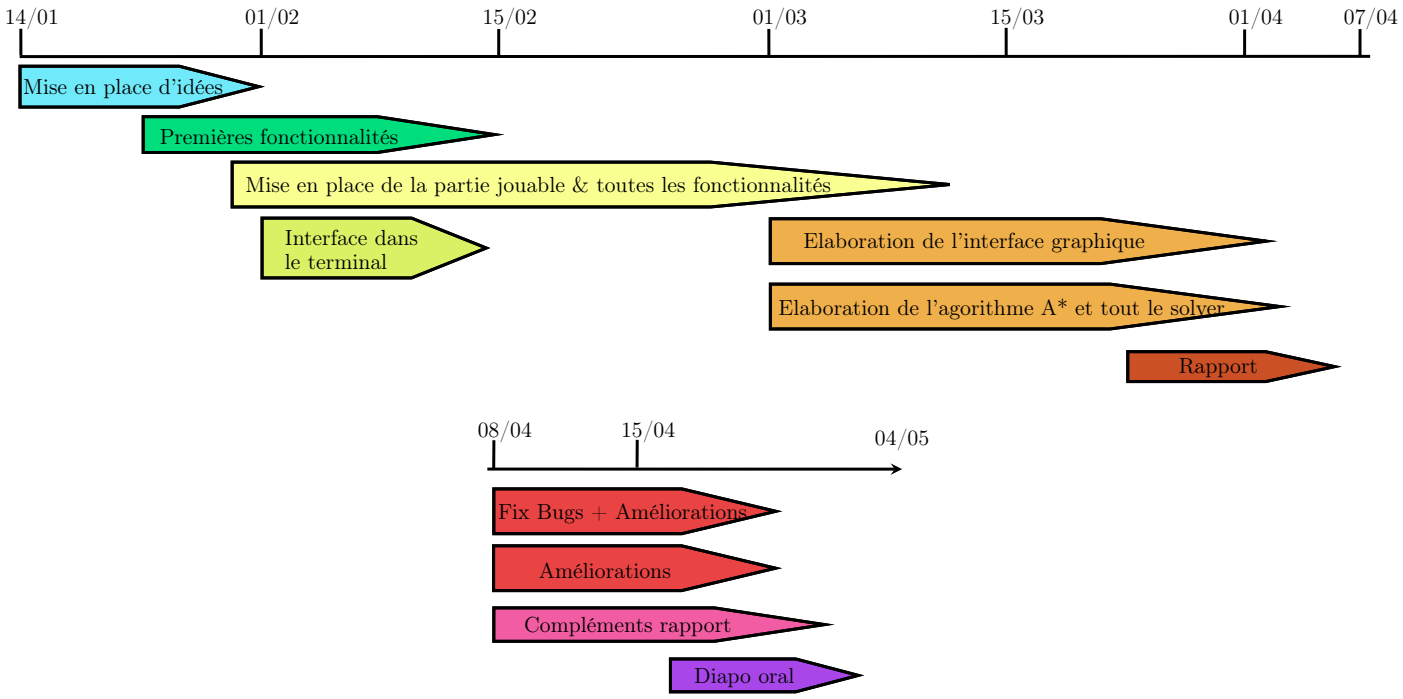


FIGURE 2 – Diagramme de Gantt de l'organisation du projet

3.3 Répartition des tâches

Afin de ne pas perdre de temps, nous avons beaucoup travaillé en parallèle (et en dehors des séances de Travaux Pratiques), en nous répartissant les tâches à réaliser en fonction de nos compétences personnelles et de nos envies.

- **Création des terrains :** Alexandre & Emmanuel & Romain
- **Algorithme A* :** Romain
- **Interface Graphique :** Alexandre & Rémy
- **Tests unitaires :** Emmanuel
- **JavaDoc :** Emmanuel & Romain
- **Parser :** Alexandre
- **Rapport & Diaporama :** Rémy & Emmanuel & Alexandre & Romain

4 Eléments Techniques

4.1 Algorithmes

Voici la liste des algorithmes principaux implémentés ainsi qu'une description de leur rôle :

- **Rotation :** C'est une méthode de la classe *"SubTerrain"*. Elle permet d'appliquer des transformations sur le sous-terrain appelé en fonction du nombre de rotations passé en paramètre.

- **Fusion :** C'est une méthode de la classe *"Terrain"*. Elle permet de fusionner quatre sous-terrains en un seul. La fusion se fait bien sûr sous forme d'un tableau à deux dimensions d'objets issus de la classe *"Case"*.

- **SolverRandom :** Le solverRandom se déplace en choisissant ses mouvements aléatoirement parmi ceux disponibles. Il respecte en revanche une règle : il ne revient en arrière que s'il n'a pas le choix (c'est à dire qu'il n'a que ce mouvement de disponible). Cet algorithme est aussi utilisé pour le placement de l'objectif du pion, lorsque les fonctions d'initialisation de la classe Terrain sont appelées.

- **SolverRandomWithMemory :** Ce solveur est une version améliorée du SolverRandom, respectant quelques règles supplémentaires qui lui font choisir en priorité des mouvements qu'il n'a pas déjà effectués. Ce joueur mémorise donc ses mouvements et donne la priorité à la découverte des nouveaux. De plus, ce joueur va effectuer les coups évidents (comme par exemple lorsque son objectif se situe à un coup).

- **Algorithme A* :** A* est un algorithme de recherche de chemin qui permet de calculer le chemin le plus optimal entre deux points dans un graphe. Proposé en 1968, celui-ci est une amélioration de l'algorithme de Dijkstra, crée 9 ans plus tôt en 1959. L'algorithme A* est utilisé dans le projet de Ricochet Robot pour résoudre le chemin entre un pion et son objectif associé.

- **Amélioration de l'algorithme A* :** Le jeu Ricochet Robot présente un problème pour l'algorithme A*. En effet, il faut résoudre plusieurs chemins qui peuvent dépendre les uns des autres. Les mouvements d'un pion peuvent impacter les mouvements d'un autre pion : il existe des obstacles dynamiques. Deux principes on été implémentés pour permettre une résolution complète. Premièrement, une permutation des objectifs à atteindre est mise en place, ce qui assure que l'algorithme le plus optimal est trouvé. Enfin, l'algorithme de résolution recherche en premier une solution avec A* avant de faire une recherche récursive en bougeant les autres pions disponibles sur le terrain de jeu. Il est similaire à un algorithme de parcours en largeur d'un arbre.

4.2 Structures de données

Voici la structure des dossiers de notre projet :

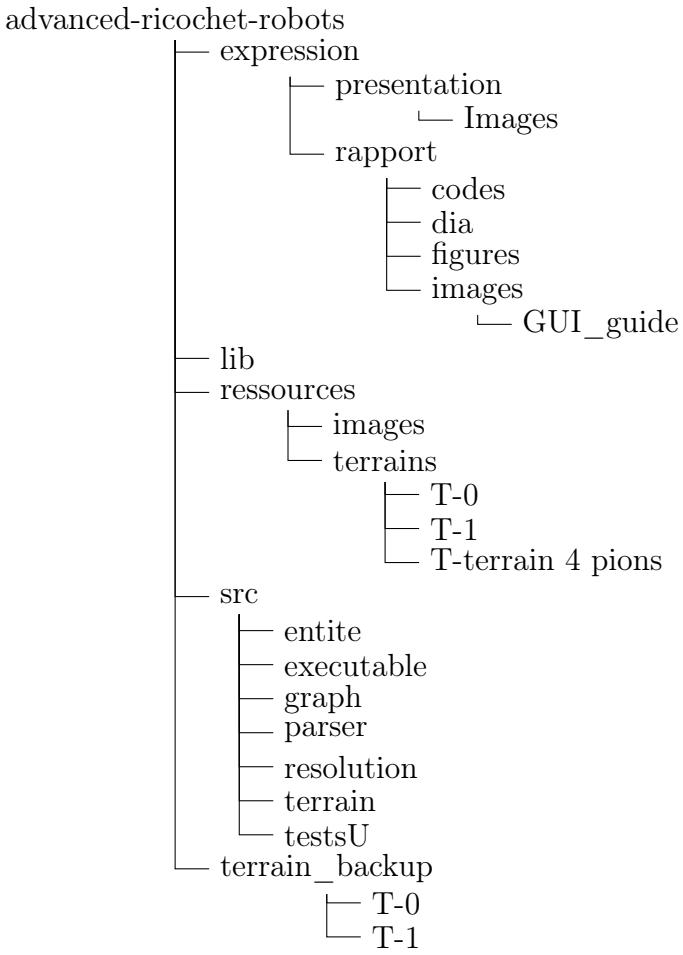


FIGURE 3 – Arborescence des dossiers

Pour représenter nos terrains, nous avons choisi d'utiliser des fichiers JSON. Un terrain est divisé en quatre sous-terrains. Ces derniers sont enregistrés sous les noms de T-*identifiant du terrain-numéro de la section*.json dans les dossiers T-*identifiant du terrain*, issus du dossier ressources/terrains. Le *numéro de la section* est un chiffre parmi 0, 1, 2 ou 3, correspondant respectivement aux sous-terrains nord-ouest, nord-est, sud-est et sud-ouest.

```
{
"cases": [
                [
                        {
                        "pion": {},
                        "objectif": {},
                        "murs": {
                                "nord": true,
                                "sud": false,
                                "est": false,
                                "ouest": true
                                }
                        }, ...

                ], ...
        ]
}
```

FIGURE 4 – Exemple d'une case dans un sous-terrain, possédant un mur nord et un mur ouest.

4.3 Bibliothèques

- **JSON-Simple**[1] **:** Nous nous sommes servis de la librairie JSON-Simple pour faciliter la manipulation des fichiers json afin de sauvegarder nos terrains.
- **JUnit**[1] **:** Quant à la librairie JUnit, celle-ci nous a permis d'effectuer les tests unitaires avec une organisation propre et structurée, en étendant notamment la classe *TestCase*. Ainsi, certaines méthodes sont exécutées avant et après chaque test, préparant l'environnement de travail.

1. Liens des bibliothèques dans la bibliographie section 9 à la page 18

5 Architecture du projet

5.1 Diagrammes de classe

Voici les explications sur le fonctionnement de chaque package (vous pouvez retrouver les diagrammes de classes en annexe page 19) :

- ➜ **Package Entité** (voir Figure 11) : Le package Entite a été conçu de façon naturelle et regroupe l'ensemble des pièces disposées sur un Terrain : Les pions ainsi que les objectifs.

- ➜ **Package Terrain** (voir Figure 12) : Le role du package Terrain est de regrouper les classes nécessaires à la création et représentation d'un terrain ainsi que des mouvements. La classe *SubTerrain*, représentant les quatres quadrants d'un terrain, permet au joueur de déplacer/échanger les sous-plateaux d'un Terrain avant d'y jouer.

- ➜ **Package Résolution** (voir Figures 13 & 18) : Ce package regroupe les classes nécessaires à la résolution du jeu Ricochet Robot. On y retrouve les résolveurs principaux : *ResolutionMultiple*, *ResolutionMultipleThreaded*, *SolverRandom* et *SolverRandomWithMemory*, ainsi que les classes annexes utilitaires utilisées par les classes principales.

- ➜ **Package Graph** (voir Figures 14 & 15) : Le package graph regroupe l'ensemble des classes utiles à l'interface graphique ainsi que la classe *TerminalGraphics* qui permet d'afficher un terrain dans le terminal en appelant la fonction *Display*, ou encore la classe *Surligneur* qui permet de sauvegarder une liste de mouvements et une couleur pour que l'interface les affiche sur le terrain avec la couleur spécifiée.
  Les classes de l'interface graphique se divisent en deux catégories : les classes qui contiennent les composants principaux (voir figure 14) et les classes représentant les terrains et permettant à l'utilisateur d'effectuer des taches spécifiques sur ces derniers (voir figure 15).

- ➜ **Package TestU** (voir Figure 16) : il contient les classes permettant d'effectuer des tests unitaires sur les classes des packages Entite et Terrain (partie backend) qui seront appelés et exécutés par la classe *SerieTests*.

- ➜ **Package Parser** (voir Figure 17) : La classe permet de charger des terrains depuis des fichiers JSON et de sauvegarder des terrains dans des fichiers JSON. La classe permet de charger et d'enregistrer des terrains en spécifiant leurs identifiants grâce aux fonctions *loadTerrain*, *loadTerrainAll* (pour récupérer tous les terrains disponibles ainsi que leurs identifiants) et *saveTerrain*. La fonction *createPath* permet de créer les dossiers de sauvegarde lors de l'enregistrement de terrain s'ils n'existent pas.

- ➜ **Package Executable** (voir Figure 19) : Ce package regroupe la classe principale *Launcher* permettant de lancer l'interface graphique et la classe *Launcher4JoueurTerminal* pour jouer au jeu dans le terminal.

5.2 Cas d'utilisation

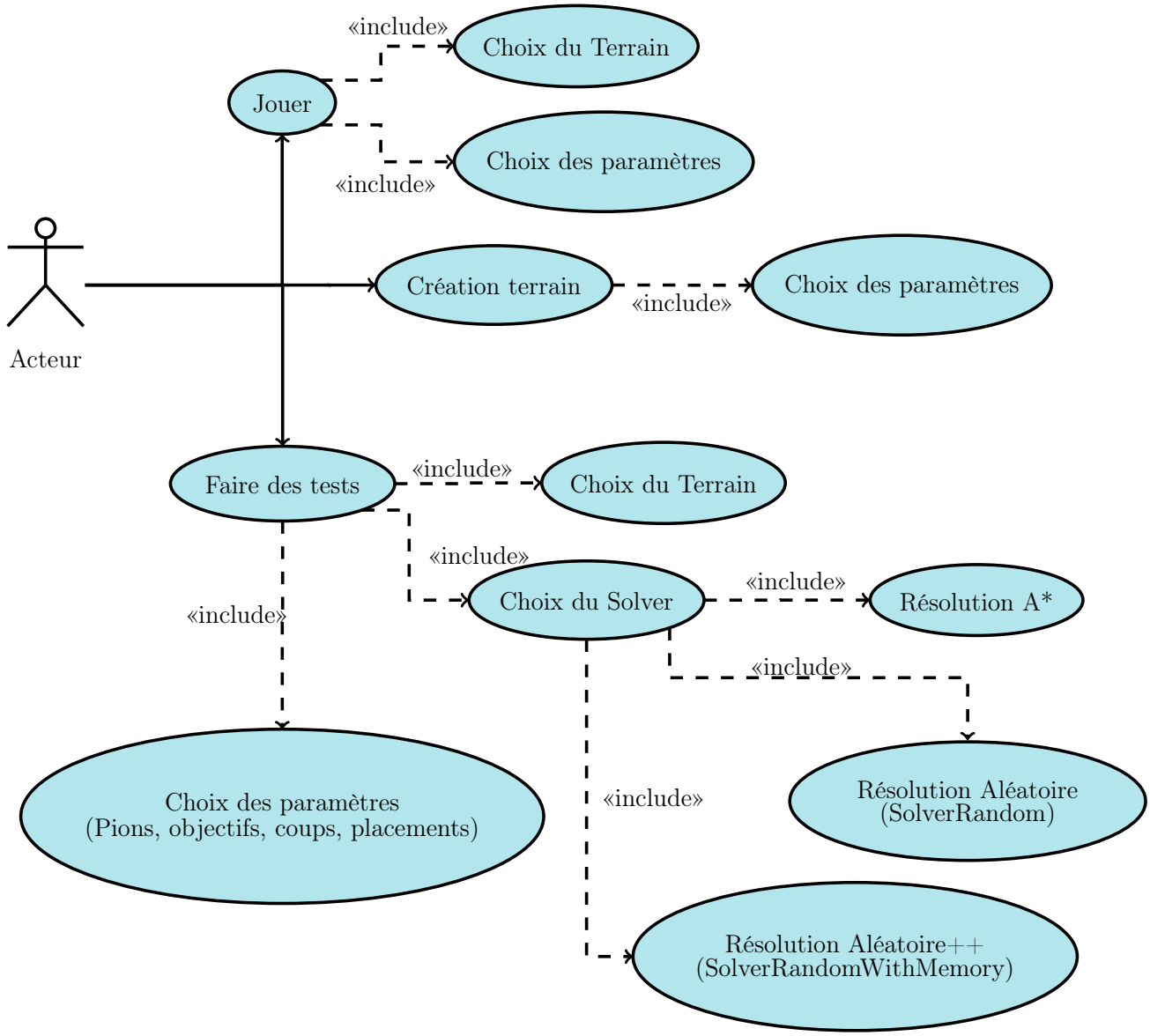


FIGURE 5 – Diagramme des cas d'utilisation

Pour des explications plus détaillées sur l'utilisation de l'application, voir partie 7 page 14.

6 Expérimentations et usages

6.1 Mesures de performances

Plusieurs améliorations de performance ont été effectuées. Voici les résultats :

6.1.1 MaxIter et MaxDepth

L'algorithme de résolution étant similaire à un algorithme de parcours d'arbre en largeur pour la résolution multiple, il est nécessaire d'implémenter des limites d'exécution, le but de ces limites étant d'éviter un cas de blocage. MaxIter et MaxDepth établissent respectivement une limite de nombre d'exécutions et une limite de profondeur de l'arbre.

Voici un tableau résumant les performances du solver. Pour avoir des résultats plus précis, chaque test a été effectué 10 000 fois :

Limite itérations	Nombre de ratés	Efficacité
256	124	98,76%
512	106	98,94%
1024	76	99,24%
2048	59	99,41%
4096	36	99,64%

TABLE 1 – Tableau des mesures des performances

6.1.2 SingleThreading et MultiThreading

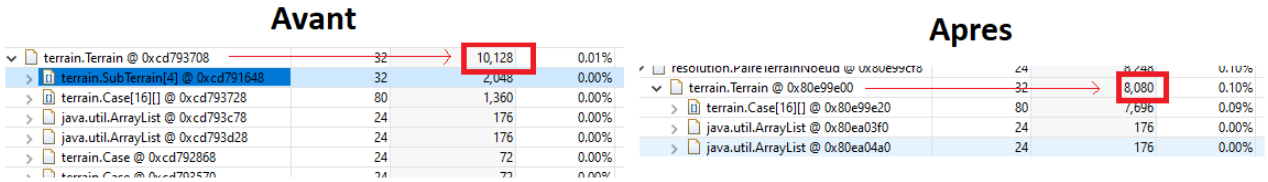
En informatique, un programme basique sur un seul thread est un programme qui exécute une tâche après l'autre. Bien que suffisant pour la majorité des cas, des programmes complexes peuvent nécessiter d'exécuter des tâches simultanément, par exemple un serveur web qui pourrait servir plusieurs clients en même temps.
Dans le cas du Ricochet Robot, le solveur tente de résoudre différentes permutations des couleurs d'objectifs afin d'obtenir une solution la plus optimale possible. Une solution de parallélisation se présente : exécuter les résolutions des permutations dans différents threads.

Après avoir effectué des centaines de milliers de tests de résolution en comparant 1 threads avec plusieurs threads, nous avons obtenu les résultats suivants :

	Résolution Singlethreaded	Résolution Multithreaded	Taux d'Efficacité
Moyenne :	1516ms	406.2ms	373.2%

6.1.3 DeepCopy et Taille des classes

L'algorithme de parcours en largeur utilise beaucoup de mémoire. C'est pour cela qu'il faut optimiser le code pour réduire l'empreinte en mémoire dans la machine virtuelle Java. Bien sûr, la machine virtuelle Java (JVM) dispose d'un système de récupération des déchets (GC, Garbage Collector), mais une optimisation dans le code est aussi nécessaire.
Une amélioration de la fonction de copie profonde a permis d'ignorer les SubTerrains d'un Terrain qui sont utilisés pour l'initialisation. Le programme VisualVM a été utilisé pour constater la réduction de taille : la classe Terrain est passée de 10128 Octets à 8080 Octets, soit 20% plus petite.



	Avant(Avg)	Après(Avg)	Taux d'Amélioration
ResolutionMultiple	1516ms	864.1ms	175.4%
ResolutionMultipleThreaded	406.2ms	326.2ms	124.5%

6.1.4 ArrayDeque vs LinkedList

En Java, une Queue n'est pas une classe mais une interface. Il faut donc soit l'implémenter nous même, soit utiliser l'une des classes implémentant cette interface. Les classes Java implémentant la Queue sont : AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, SynchronousQueue.

Nous avons donc utilisé une LinkedList au début car c'était un nom qui était familier mais en recherche d'améliorations de performance, une remise en cause de ce choix s'est présentée. En examinant la documentation de la classe ArrayDeque de plus près, il est stipulé que celle-ci serait plus rapide que la LinkedList en terme d'utilisation de Queue. Malheureusement, rien n'était présent sur la page de documentation de la LinkedList, ce qui nous aurait évité de faire des recherches.

Lors de la recherche d'expérimentations à ce sujet, seuls deux liens ont été trouvés, l'un étant hors-ligne, le site archive.org a du être utilisé. Voir [8] et [9]. C'est pour cela que nous avons effectué nos propres tests :

	LinkedList (Avg)	ArrayDeque (Avg)	Taux d'Amélioration
ResolutionMultiple	864.1ms	503.5ms	171.6%
ResolutionMultipleThreaded	326.2ms	231.3ms	141%

6.1.5 Conclusion : Avant et Après

	Avant(Avg)	Après(Avg)	Taux d'Amélioration Total
ResolutionMultiple	1516ms	503.5ms	301%
ResolutionMultipleThreaded	406.2ms	231.3ms	175.6%
Total SingleThread MultiThread	1516ms	231.3ms	655.4%

Bien que les améliorations entre chaque étape soient plus impressionnantes en Thread Unique (SingleThreaded) qu'en Thread Multiple (MultiThread), on peut constater une amélioration **totale**, c'est à dire en partant de la résolution à thread simple avec LinkedList et l'ancienne DeepCopy vers la résolution à thread multiple avec ArrayDeque et nouvelle DeepCopy, **de 655.4%**.

Voici deux conseils que l'on peut donc déduire de ces expérimentations :

- Dans le cas d'un algorithme utilisant beaucoup de mémoire, s'assurer que l'empreinte mémoire des classes restent minimales.
- Une interface est un contrat d'implémentation, et non de performance. Toujours vérifier l'ensemble des classes implémentant une interface et, si nécessaire, faire des expérimentations.

6.2 Tests unitaires

Les tests unitaires ont pour objectif la vérification du bon fonctionnement de certaines parties de code d'un projet. Ils ont un rôle essentiel, permettant de s'assurer que chaque cas possible a bien été pris en compte lors de l'écriture de la partie testée et ainsi éviter des bugs indésirables.
Pour cela, nous avons choisi d'utiliser la bibliothèque JUnit qui falicite grandement la rédaction et l'organisation de ces tests. En effet, elle nous donne accès à la classe **TestCase**. Nous avons pris le parti de tester toutes les classes de la section backend du projet :

- **CaseTest** correspond aux tests de la classe **Case**
- **MouvementTest** correspond aux tests de la classe **Mouvement**
- **ObjectifPrimaireTest** correspond aux tests de la classe **ObjectifPrimaire**
- **ObjectifSecondaireTest** correspond aux tests de la classe **ObjectifSecondaire**
- **PionTest** correspond aux tests de la classe **Pion**
- **SubTerrainTest** correspond aux tests de la classe **SubTerrain**
- **TerrainTest** correspond aux tests de la classe **Terrain**

Chacune de ces classes redéfinit deux méthodes de la classe TestCase :

- *setUp* : initialise les données d'entrée qui seront utilisées pendant les tests
- *tearDown* : retour à l'état initial de l'environnement

Un test de classe se déroule de la manière suivante pour chacune des méthodes testées :

1. Appel (implicite) de la méthode setUp
2. Application de la méthode testée
3. Vérification de la réussite de la méthode avec des **assert**
4. Appel (implicite) de la méthode tearDown

En utilisant correctement les assertions et en vérifiant tous les paramètres possibles, on s'assure ainsi que la méthode a été correctement réalisée et que la fonctionnalité du projet n'est pas remise en cause.
S'il est possible de lancer les classes des tests séparément, nous avons également conçu une classe **SerieTests** qui, grâce à la bibliothèque JUnit, va gérer l'exécution de chacune des classes une par une.

7 Manuel d'utilisation de l'interface graphique

Lorsque vous lancerez l'application, vous arriverez sur le menu principal (voir figure 6). Les autres parties de l'application seront détaillées dans les sections suivantes.

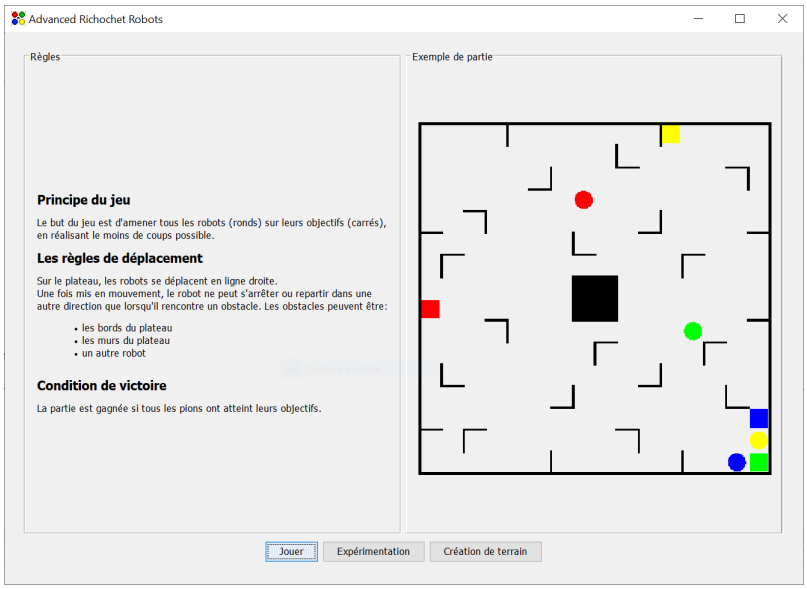


FIGURE 6 – Capture d'écran du menu principal de l'application. Vous trouverez les règles du jeu, une démo de partie, ainsi que 3 boutons (en bas de la fenêtre) vous permettant d'accéder aux différentes parties de l'application : jouer, expérimentation et création de terrain.

7.1 Jouer

Cette partie de l'application vous permet de jouer au jeu, selon les règles définies dans le menu principal.

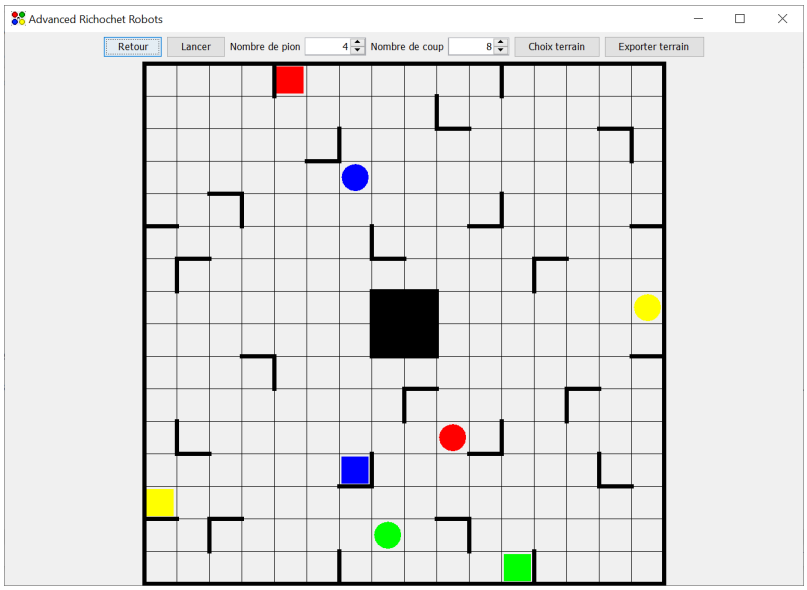


FIGURE 7 – Capture d'écran de la fenêtre de jeu de l'application.

En haut de cette fenêtre, vous trouverez 4 boutons pour : revenir au menu principal, lancer une partie, choisir un terrain, exporter le terrain, ainsi que 2 sélecteurs pour choisir le nombre de pions à placer sur le terrain et le nombre maximal de coups séparant le pion de son objectif. Pour jouer, il vous suffit simplement de cliquer sur le pion que vous souhaitez déplacer. Ses mouvements possibles s'afficheront et vous n'aurez qu'à cliquer sur l'un de ces mouvements pour le réaliser.

7.2 Expérimentation

Cette partie de l'application vous permet de réaliser des expérimentations (voir figure 8).

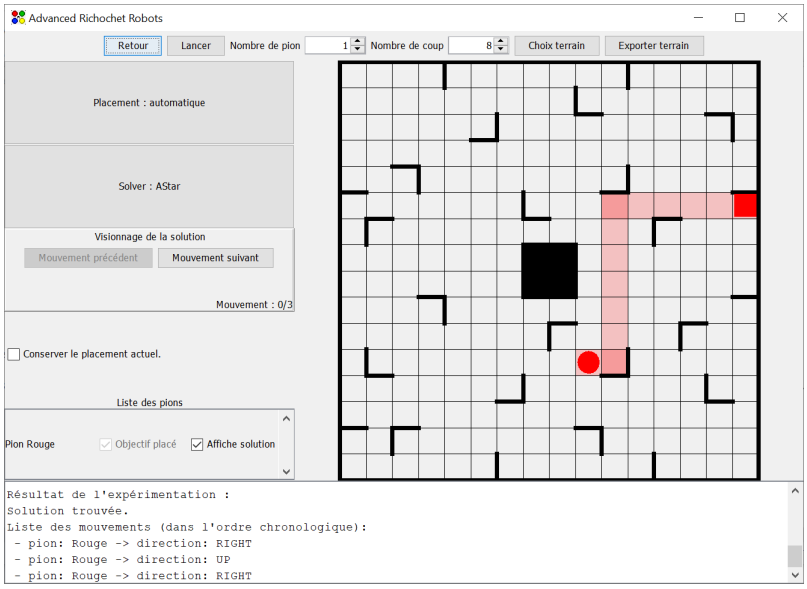


FIGURE 8 – Capture d'écran de la fenêtre d'expérimentation de l'application.

Le haut de cette fenêtre fonctionne de la même manière que celui de la fenêtre *jouer* (voir section précédente).
A gauche de la fenêtre, vous trouverez un panneau de contrôle pour l'expérimentation.
Le premier bouton est le bouton permettant de choisir parmi les 2 types de placement pour les pions et les objectifs, le *placement automatique*, l'application placera sur le terrain autant de pions que le sélecteur *nombre de pions* indiquera et leurs objectifs à autant de coups que le sélecteur *nombre de coups* indiquera, ou le *placement manuel* qui vous permet de placer vous-même les pions et leurs objectifs (voir figure 9).

Le second bouton vous permet de choisir le type de résolution, entre *AStar*, *Aléatoire*, *Aléatoire++* dont les détails techniques sont spécifiés dans la section suivante (voir section 4.1).
Le troisième élément de ce panneau de contrôle est la partie *visionnage de la solution*. Elle vous permet, lorsqu'une solution a été trouvée, de la rejouer en effectuant les coups, les un après les autres, en utilisant les boutons *mouvement précédent* et *mouvement suivant*.
Les deux derniers éléments de ce panneau de contrôle sont : une case à cocher vous permettant de conserver le placement actuel des pions et objectifs sur le terrain, et la liste des pions placés sur le terrain indiquant la couleur du pion, si ce pion possède un objectif et enfin lorsqu'une solution est trouvée, d'afficher ou non les mouvements de ce dernier.
Enfin, en bas de la fenêtre, vous trouverez la console qui affichera les résultats des expérimentations ainsi que les conseils pour le placement manuel évoqué plus haut.

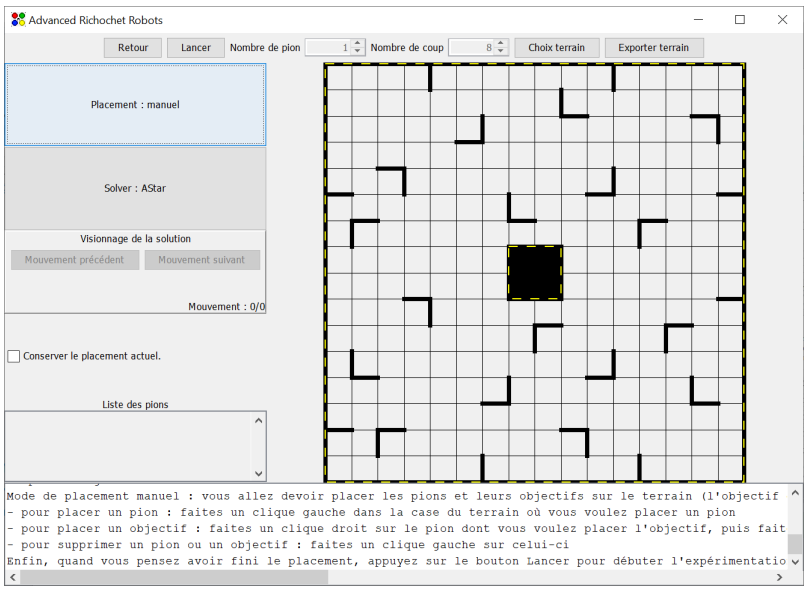


FIGURE 9 – Capture d'écran de la fenêtre d'expérimentation de l'application permettant de placer manuellement les pions et objectifs, les instructions pour le placement sont affichées dans la console en bas de la fenêtre et vous pouvez placer jusqu'à 4 pions.

7.3 Création de terrain(s)

Cette partie de l'application vous permet de créer vos propre terrain à partir de terrains déjà existant ou d'un terrain vide en choisissant sa taille à l'aide du sélecteur *largeur du terrain*, de placer les pions ainsi que leur objectif mais aussi les murs du terrains, puis d'exporter ce terrain pour l'utiliser ultérieurement, par exemple dans la partie expérimentation de l'application (voir figure 10).

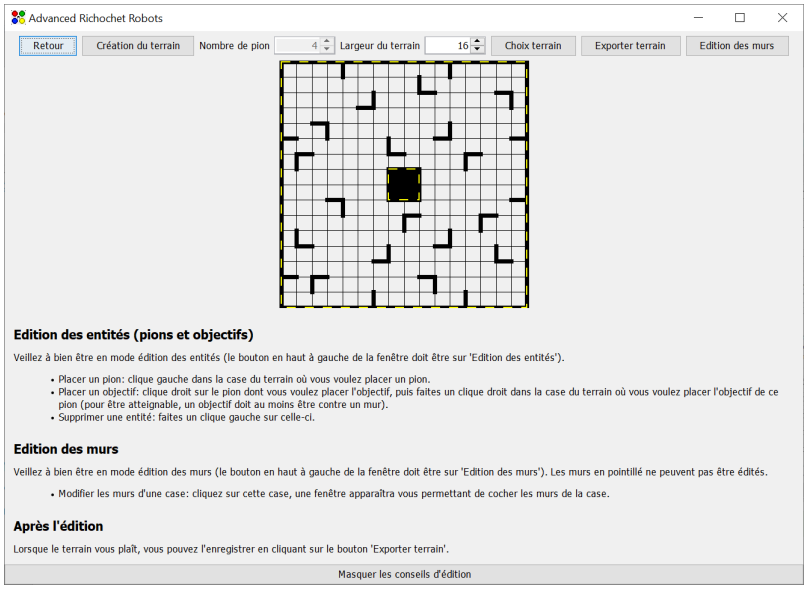


FIGURE 10 – Capture d'écran de la fenêtre de création de terrain(s) de l'application, les instructions pour la création du terrain s'affichent en bas de la fenêtre.

Conseil : pour qu'un changement sur un sélecteur s'applique, il faut appuyer sur *lancer* (ou *création du terrain*).

8 Conclusion

8.1 Récapitulatif des fonctionnalités principales

Pour plus d'informations, voir sous-section 2 à la page 3 du rapport.

- **Solveur A***
- **Interface graphique**
- **Interface terminale**
- **Jouabilité**
- **Expérimentations**
- **Création de terrain(s) :** possibilité de créer ses propres terrains.
- **Customisation des expérimentations :** placement libre et aléatoire des pions et objectifs pour lancer le calcul.

8.2 Propositions d'améliorations

Voici une liste d'améliorations possibles :

- Optimisation plus poussée du A*.
- Développer, pour l'interface terminale, des fonctionnalités similaires à l'interface graphique.
- Possibilité de sauvegarder une expérimentation (ou une partie) après fermeture.
- Ajout(s) de terrain(s).
- Mode sombre de l'interface graphique.

8.3 Apports personnel

Ce projet nous a beaucoup apporté en terme de connaissances, du fait qu'il a fallu découvrir de nouveaux outils comme le développement d'algorithmes décisionnels, et améliorer nos compétences en Programmation Orientée Objet. De ces faits, nous avons acquis une assez bonne connaissance du java sous de nombreuses formes, ce qui nous prépare encore mieux pour les années à venir et de futurs emplois.
Ce projet nous a également permis d'apprendre encore plus à travailler en équipe sur un projet, ce qui est important pour évoluer correctement au sein d'une entreprise.

9 Bibliographie

Références

[1] Ricochet Robots - Wikipédia : *https://fr.wikipedia.org/wiki/Ricochet_Robots*

[2] Algorithme A* - Wikipédia : *https://fr.wikipedia.org/wiki/Algorithme_A**

[3] A* (A Star) Search Algorithm - Computerphile : *https://www.youtube.com/watch?v=ySN5Wnu88nE*

[4] A* Pathfinding (E01 : algorithm explanation) - Sebastian Lague : *https://www.youtube.com/watch?v=-L-WgKMFuhE*

[5] Graph Data Structure 6. The A* Pathfinding Algorithm - Computer Science : *https://www.youtube.com/watch?v=eSOJ3ARN5FM*

[6] Librairie JSON-Simple : *https://code.google.com/archive/p/json-simple/*

[7] Librairie JUnit : *https://junit.org/junit4/*

[8] Java : ArrayDeque vs. LinkedList : *http://brianandstuff.com/2016/12/12/java-arraydeque-vs-linkedlist/*

[9] java.util.LinkedList performance : *https://web.archive.org/web/20180528224935/http://java-performance.info/linkedlist-performance/*

10 Annexe

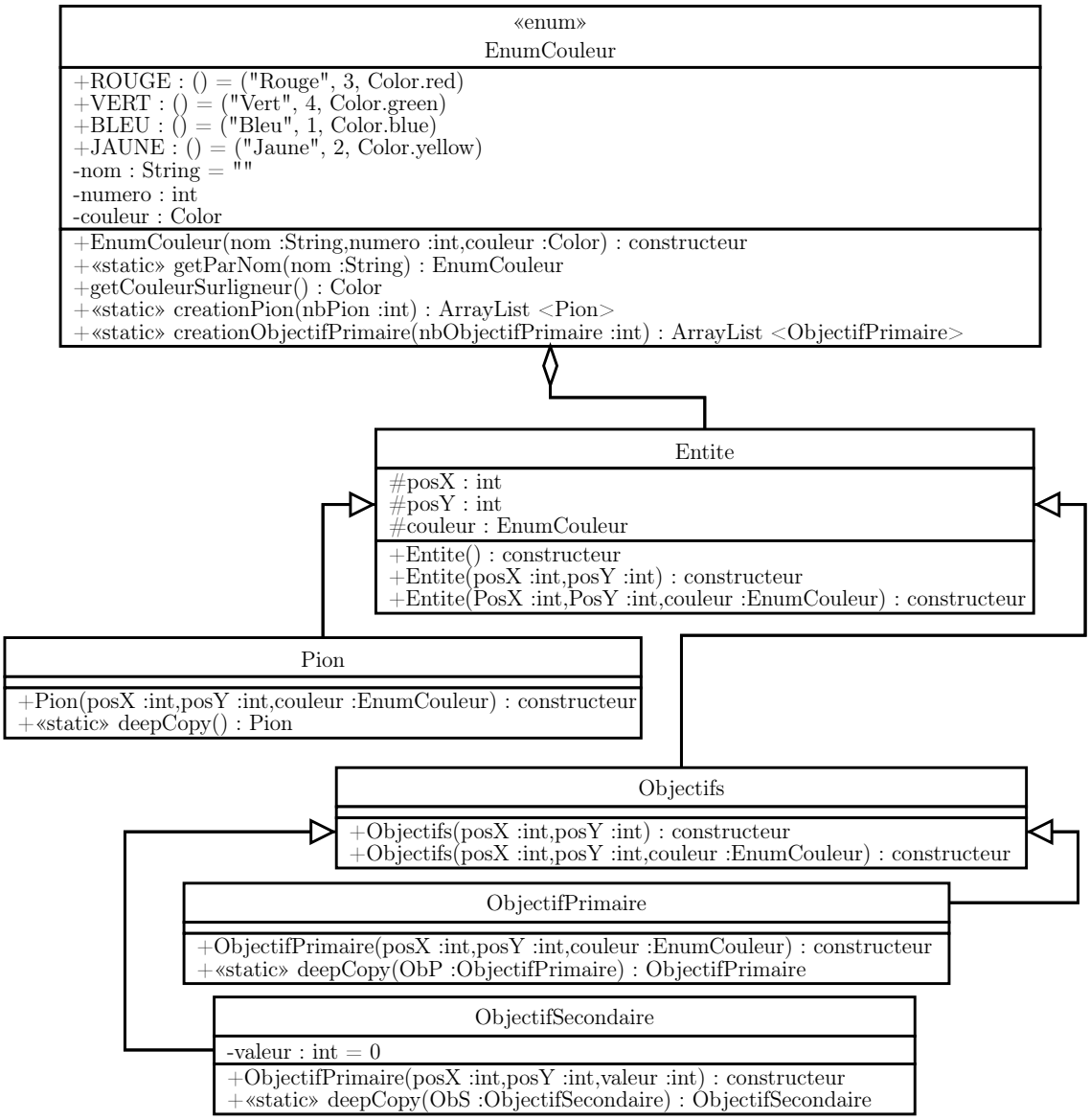


FIGURE 11 – Diagramme de classe du package : Entité

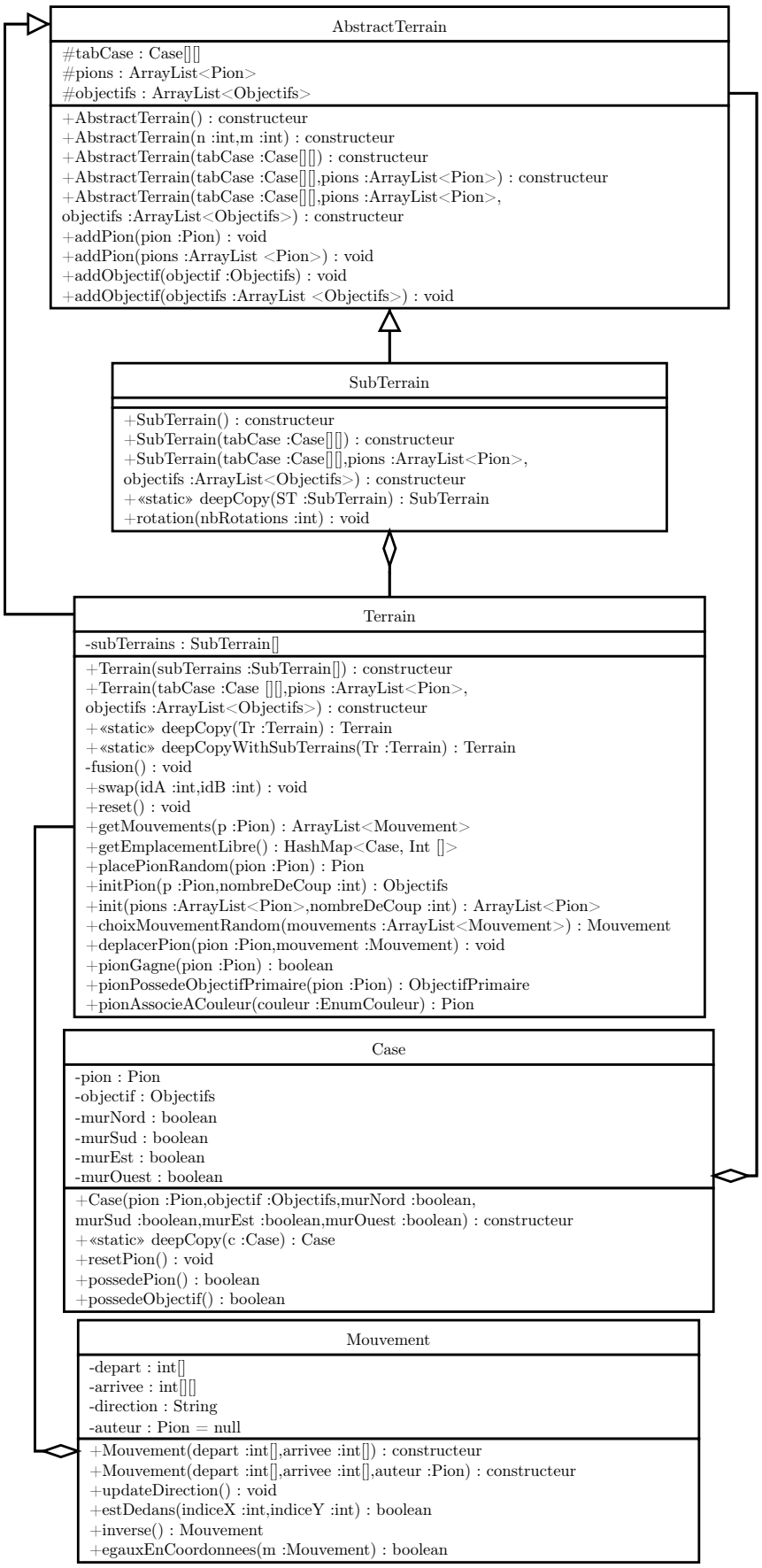


FIGURE 12 – Diagramme de classe du package : Terrain

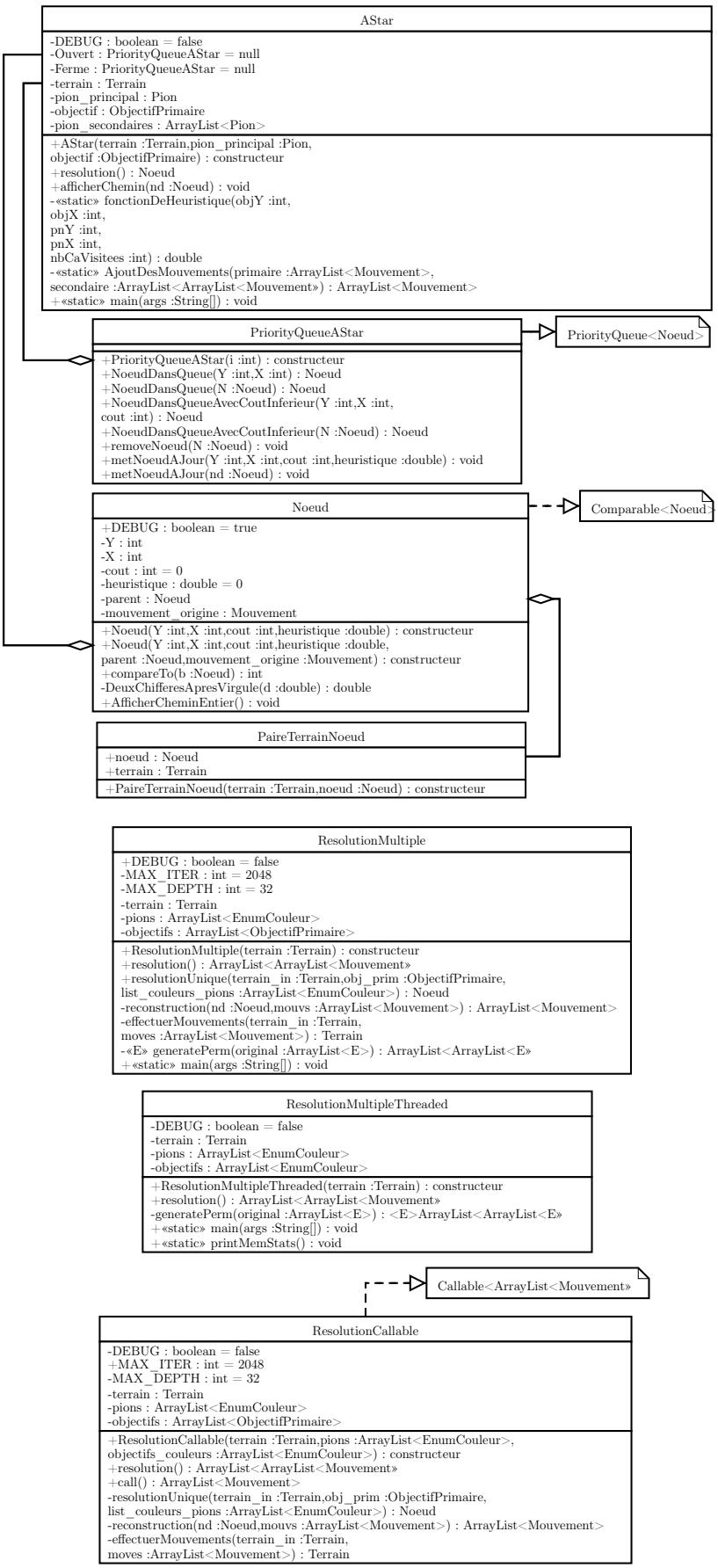


FIGURE 13 – Diagramme de classe de du package : Résolution (hors classes solvers.)

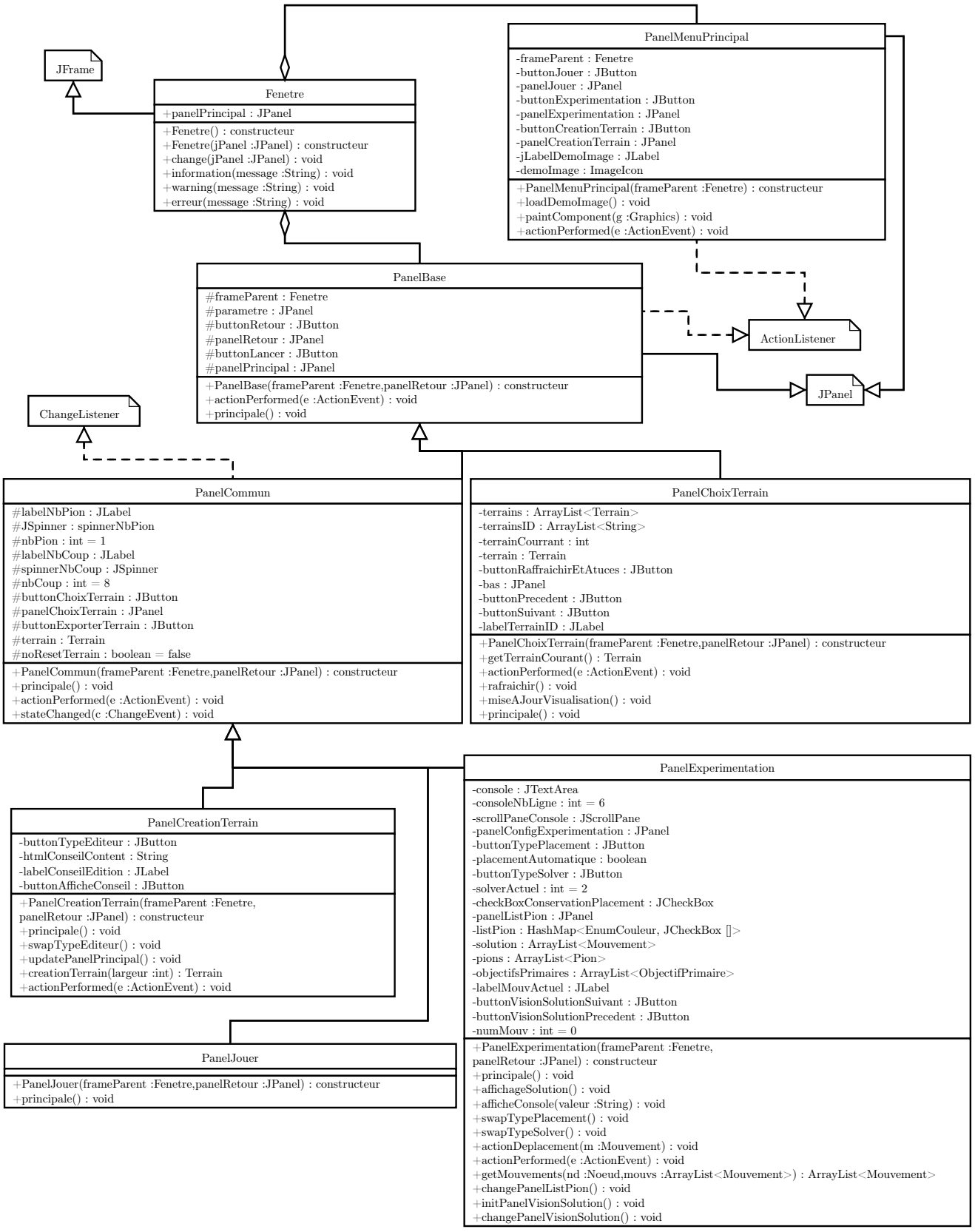


FIGURE 14 – Diagramme de classe du package : Graph (Partie 1 : dépendance avec classe Menu)

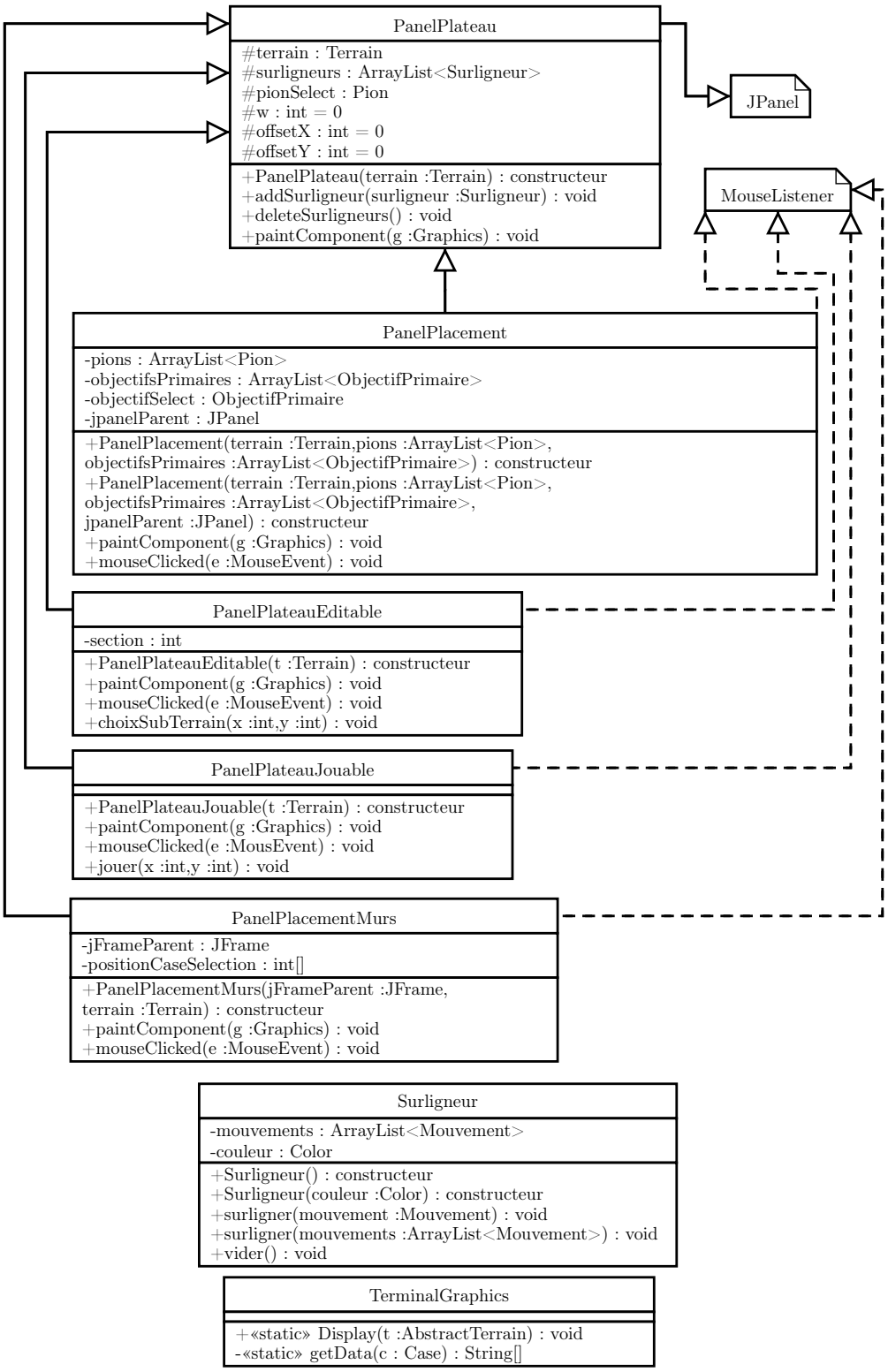


FIGURE 15 – Diagramme de classe du package : Graph (Partie 2 : Autres dépendances)

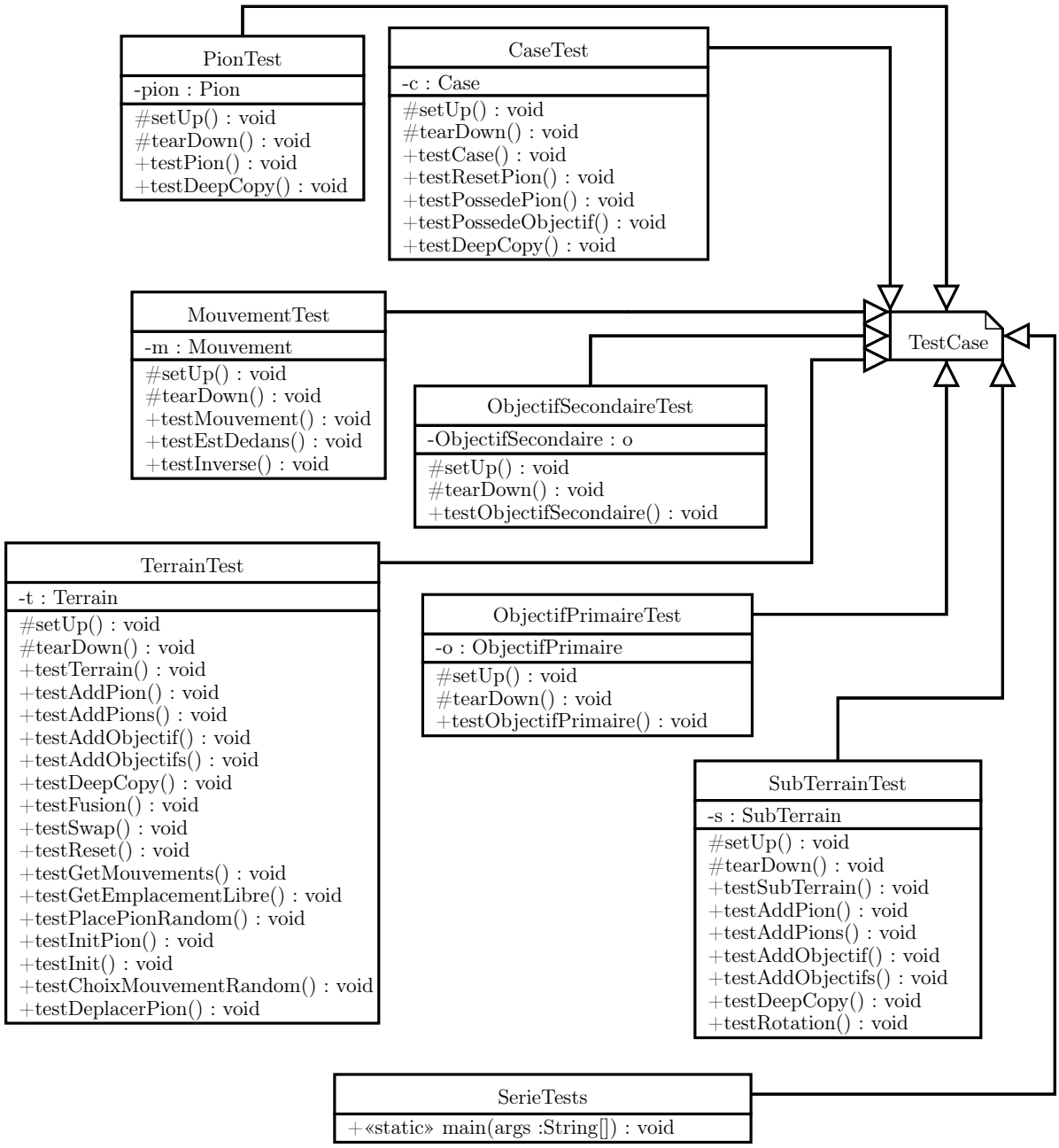


FIGURE 16 – Diagramme de classe du package : TestU

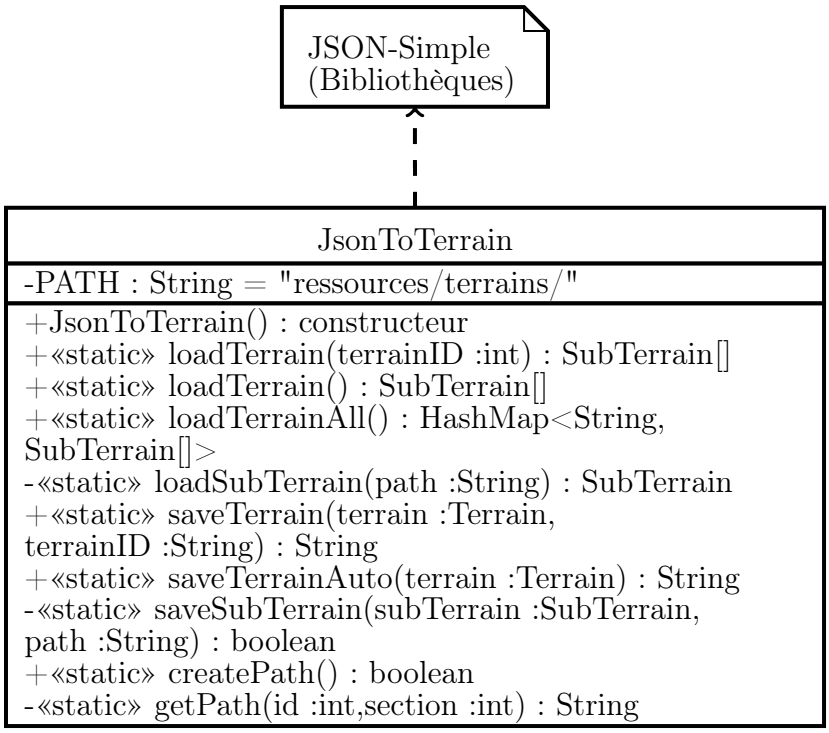


FIGURE 17 – Diagramme de classe du package : Parser

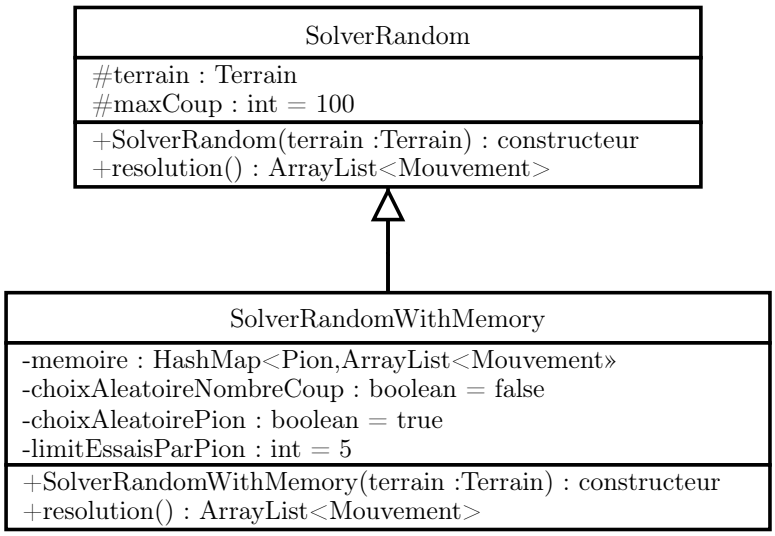


FIGURE 18 – Diagramme de classe des classes solvers du package : Résolution

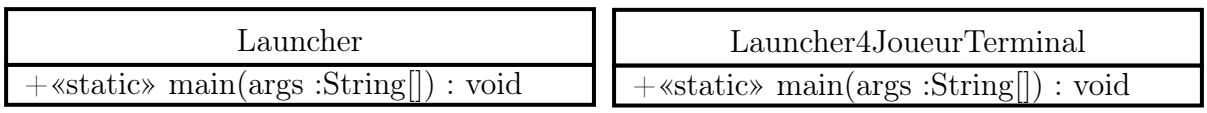


FIGURE 19 – Diagramme de classe du package : Executable