JEU DU SOLITAIRE

Projet de développement logiciel

Gustavo **CIOTTO PINTON**
Ignacio **GARCIA ATANCE GARCIA**

Séquence 5 - Promotion 2013-2015

Octobre 2014

Table des matières

1 Introduction

Le principal objectif du projet de conception logiciel est le développement des outils qui permettent le bon déroulement d'un jeu du solitaire. Afin d'y bien arriver, on considère toujours les connaissances apprises dans le cours de première année, notamment les « Fondements de l'Informatique », « Génie Logiciel » et « Modèles de programmation ».

En ce qui concerne au jeu du solitaire, il consiste à un plateau composé des trous qui peuvent être occupés par des pions ou pas. L'objectif est alors d'éliminer le maximum des pièces possibles. Une pièce est enlevée du jeu lorsqu'une autre pièce la « mange », c'est-à-dire, la saute par dessus.

Notre programme doit alors être capable de réaliser plusieurs fonctions afin de garantir l'efficience et la performance d'un tel jeu. En effet, l'application construite permet à l'utilisateur réaliser de tâches allant de la manipulation des fichiers texte (réalisant ainsi la lecture de l'état du plateau et aussi l'écriture de la liste des coups qui ont été faits), l'affichage graphique à travers du paquetage *Swing*, fourni par Java, jusqu'à l'implémentation d'algorithmes d'intelligence artificielle qui puissent calculer la meilleure liste de coups possible.

Afin d'attendre ces besoins, on a bien considéré l'étape de modélisation UML et certains de ses diagrammes. On remarque l'utilisation des outils fournis par le logiciel Eclipse comme le *plug-in* Papyrus, avec lequel nous avons construit les diagrammes de classes et de cas d'utilisation, et aussi l'importance du système de contrôle des version, SVN, qui nous a bien permis de gérer plus facilement nos nombreuses modifications.

2 Modélisation du programme - UML

La construction d'un modèle permet au programmeur d'identifier les principaux besoins et les possibles incohérences dans la structure de son application avant même de l'étape de codage, ce qui peut éviter des grands changements à l'implémentation et, par conséquent, des gros problèmes.

2.1 Diagramme de cas d'utilisation

La première étape de la modélisation a consisté à la construction d'un diagramme de cas d'utilisation, dont principal but est de montrer les fonctionnalités que notre programme devra fournir. Le résultat de notre première conception peut être trouvé dans la figure 1, où on remarque la présence d'un seul acteur, le joueur, et de quatre cas principaux :

— **Choisir le plateau** : le joueur pourra choisir un plateau quelconque, mais pour le faire, il doit obligatoirement choisir un fichier d'origine. C'est pour cette raison qu'on utilise une relation de « include » entre le cas d'utilisation « Choisir Plateau » et le cas « Lire fichier ».

— **Sauvegarder résultat** : dans ce cas le joueur peut choisir entre sauvegarder l'état du plateau ou sauvegarder la liste des coups réalisés. Donc, on utilise des relations « extend » entre les respectifs cas. De même façon, le joueur devra obligatoirement choisir un fichier à être écrit.

— **Calculer prochain coup** : le programme devra fournir le prochain coup à être réalisé et conséquemment une stratégie doit être choisit par le joueur. L'application devra également calculer et fournir tous les coups possibles à chaque étape.

— **Obtenir plateau final** : ce dernier cas d'utilisation consiste à calculer l'état final du plateau et de l'afficher selon la stratégie choisie par le joueur.

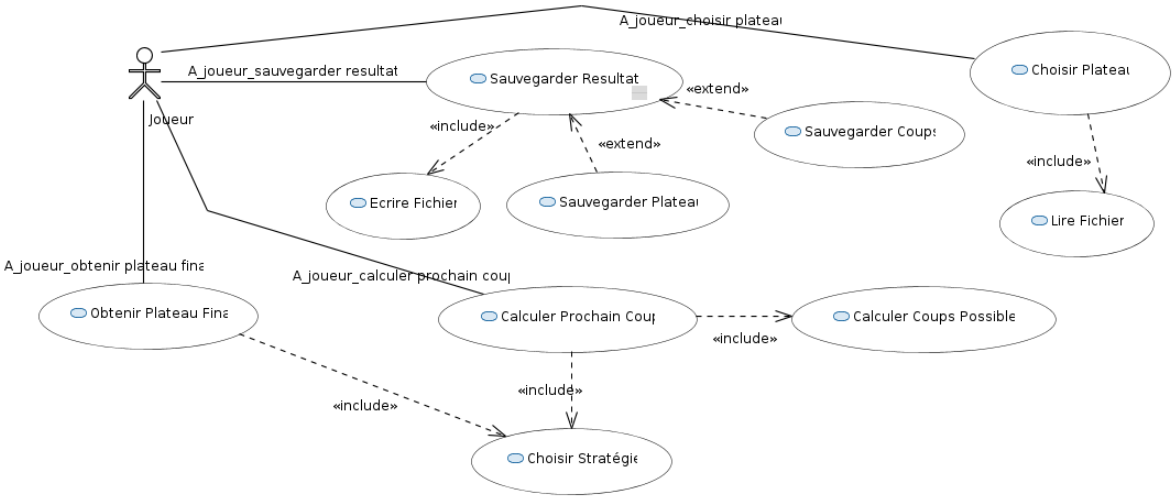

FIGURE 1 – Diagramme de cas d'utilisation.

2.2 Diagramme de classes

Une fois tous les besoins identifiés, on peut penser aux détails, c'est-à-dire, leurs méthodes et variables, qui concernent les classes qui feront partie de notre application. Tout d'abord, nous avons considéré de diviser nos classes selon leurs caractéristiques et les placer dans trois groupes, étant ainsi :

— **Entités** : les classes placées dans ce groupe répresentent les données et encapsulent les données utilisées. Ces classes seront naturellement utilisées dans tous les autres groupes de l'application. On cite notamment les classes *Plateau* et *Coup* qui contiennent respectivement les données nécessaires à la répresentation d'un plateau et d'un coup.

— **Contrôle** : ce groupe contient les classes responsables pour contrôler le déroulement du jeu. Elles sont ainsi capables de contrôler les interfaces, de décider les prochains coups et contiennent les variables qui décrivent l'état actuel du jeu.

— **Interfaces** : les dernières classes sont engagées à l'interface entre l'utilisateur final, soit à travers du terminal, soit par fênetres. On place également dans cette classification, la classe dédiée à la manipulation des fichiers.

Le résultat est finalement montré dans figure 6 de la section **Annexes**.

2.2.1 Les classes « Entités »

Comme abordé précédemment, les classes qui appartiennent à ce groupe possèdent les informations importantes qui seront utilisées par toutes les autres structures de l'application. En d'autres mots, ce sont des données fondamentales à d'autres classes et aux algorithmes d'intelligence artificielle. La figure 2 montre en détail les components de ce paquet.

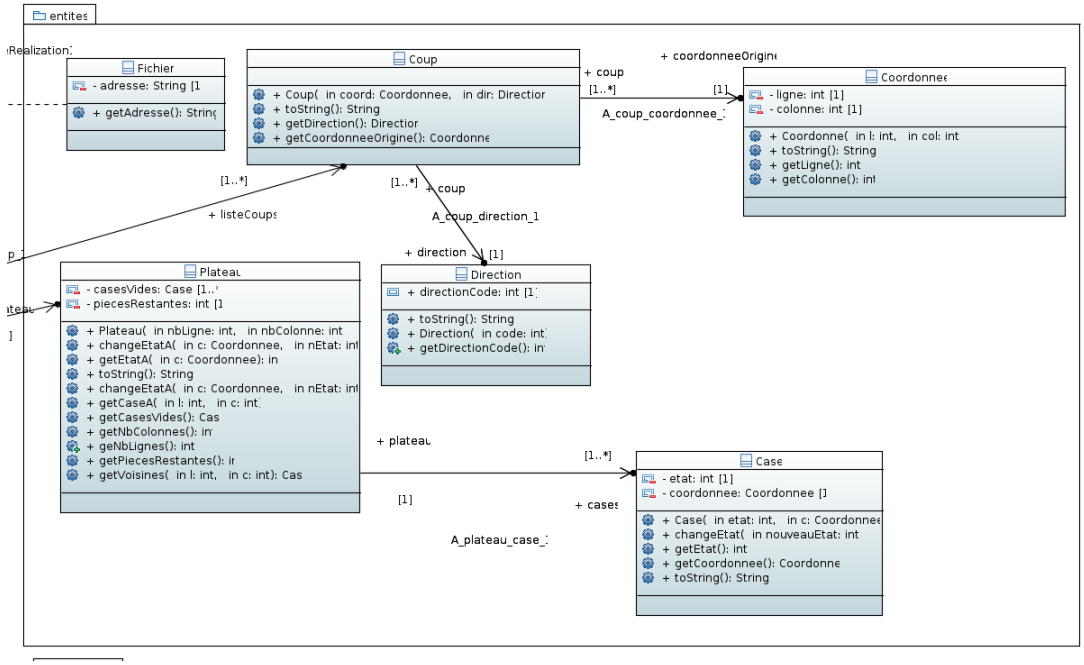

FIGURE 2 – Paquetage « entités » du diagramme de classes.

On peut notamment citer :

— La classe **Plateau** contient les informations qui décrivent complétement un plateau de taille et nombre de pions quelconque. Elle garde aussi une liste avec les cases actuellement vides et le nombre de pièces restantes. C'est à un objet de cette classe gérer les changements d'état des ses cases, gardées sur une matrice d'objets de type Case.

— Chaque objet appartenant à classe **Case** a un attribut qui décrit son état actuel et sa coordonnée vis-à-vis au plateau. A principe, on a rejetté l'idée d'avoir une telle classe, puisqu'on pourrait implémenter

une matrice des entiers directement, mais après réflechir un peu, on a décidé de la maintenir, vu qu'elle rendrait plus facile le codage des algorithmes et un possible changement de la structure de données (par exemple, si on voulait remplacer une liste au lieu de la matrice).

— La classe **Coup** répresente un mouvement et possède une direction et une coordonnée d'origine.

— La classe **Direction** possède un attribut entiers qui peut avoir seulement quatre valeurs possibles (haut, bas, droite et gauche). Sa concéption a été justifiée par la facilité d'écrire de méthodes comme *toString* et de constantes qui répresentent les directions.

— La classe **Coordonnee** répresente les valeurs d'une ligne et d'une colonne par rapport à la matrice qui forme le plateau.

— La classe **Fichier** implémente l'interface *IMilieu* du paquet *controle* (*cf.* section 2.2.2) et répresente les données d'un fichier. On remarque que cette construction a été choisie pour garantir sa généralité, c'est-à-dire, si on veut étendre notre programme pour lire le plateau d'un serveur par exemple, ce dernier paquet reste inchangeable.

2.2.2 Les classes « Contrôle »

Toutes les opérations qui concernent la gestion des données et des mouvements sont réalisées par les classes dans ce paquet, selon figure 3. On observe la présence de trois interfaces qui garantissent sa généralité : une fois que l'interface graphique ou la façon de lire le plateau changent, aucune modification sera nécessaire à ce paquet. Dans notre cas, nous avons implémente deux façons d'affichage des résultats (une via terminal et autre via des fênetres), mais toutes les deux utilisent la même classe *ControleurJeu*.

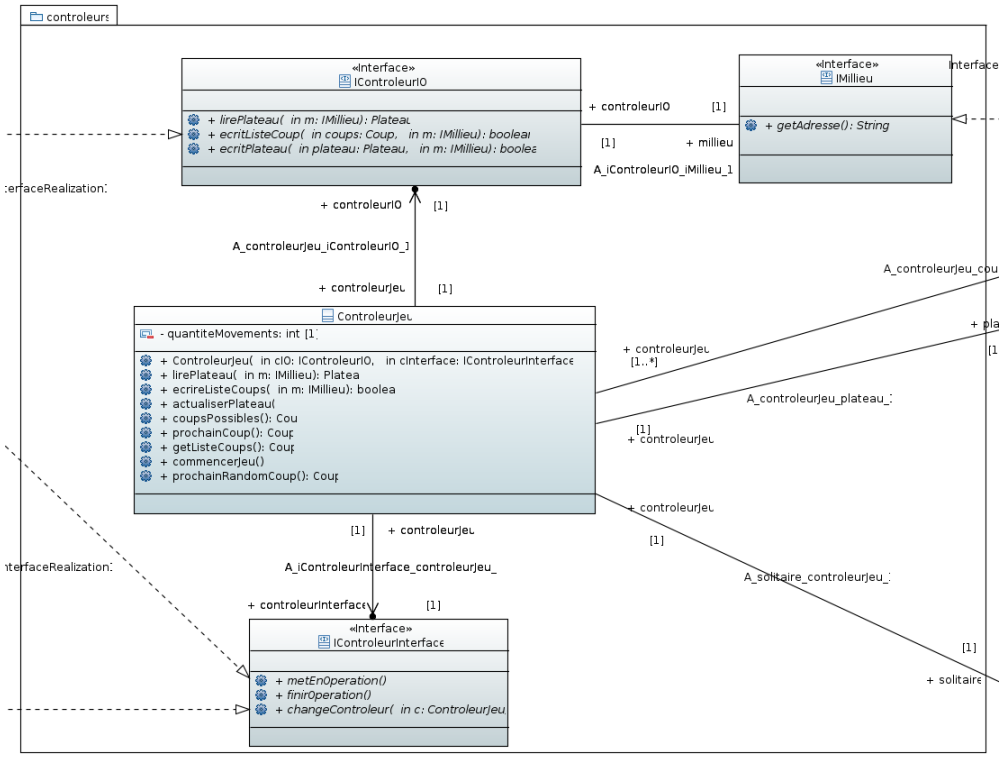

FIGURE 3 – Paquetage « controle » du diagramme de classes.

En ce qui concerne les fonctionnalités de chaque classe individuellement, on observe que :

— le classe **ControleurJeu** centralise le contrôle du jeu et, conséquemment, son déroulement. C'est à elle aussi d'assigner aux contrôleurs restants leurs respectives tâches, comme par exemple, envoyer au contrôleur d'entrée et sortie de messages de lecture ou écriture. Compte tenu de sa nature, cette classe utilise plusieurs attributs qui répresentent l'état du jeu, notamment la liste de coups réalisés, un plateau et la quantité de mouvements.

— l'interface **IControleurInterface** défine les opérations qu'une classe doit fournir si elle est destinée à l'affichage des données.

— l'interface **IControleurIO** définera ce qu'une classe responsable pour contrôler les opérations d'entrée et sortie devra fournir. Elle fait l'usage également d'une autre interface, **IMilieu**, qui répresente les caractéristiques d'un milieu quelconque.

2.2.3 Les classes « Interface »

L'affichage, soit via le terminal soit via l'interface graphique, et la communication à travers de fichiers sont réalisés par les classes de ce paquet. La figure numéro 4 montre sa modélisation.

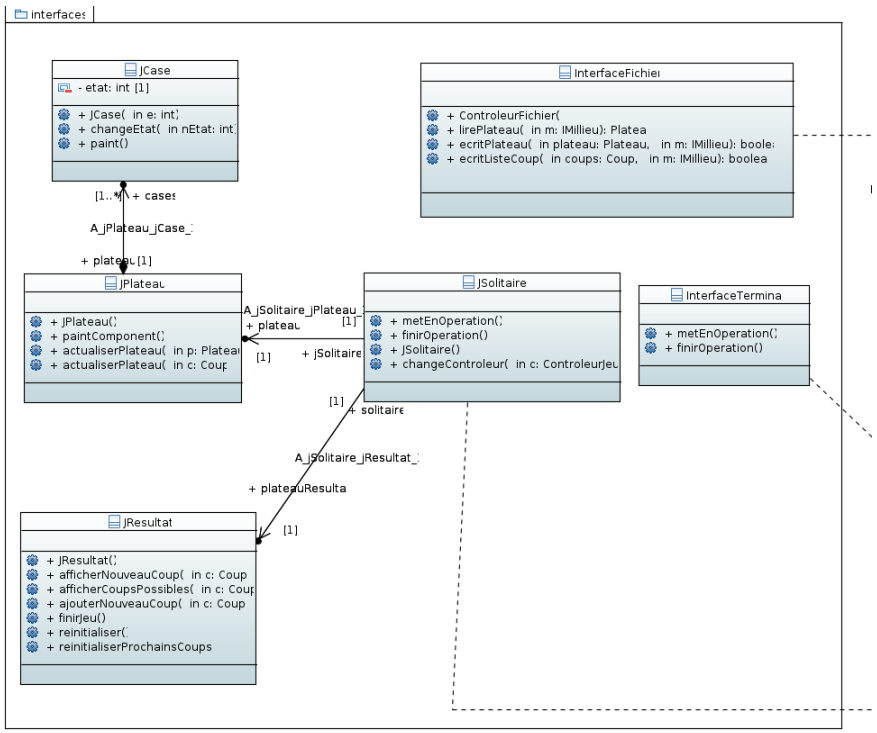

FIGURE 4 – Paquetage « interface » du diagramme de classes.

On remarque que :

— la classe **interfaceTerminal** est responsable d'afficher les plateaux et la liste des coups possibles en les écrivant en format texte sur le terminal. Donc, les plateaux seront affichés de la même façon qu'on les représente dans les fichiers .

— a classe **interfaceFichier** s'occupe des opérations sur les fichiers comme l'écriture et la lecture. Elle implémente aussi l'interface *IControleurInterface* du paquet *Contrôle*.

— L'affichage graphique est fait par la classe **JSolitaire** et ses components. Elle est composée par deux parties principales : le plateau où les pions seront représentés et un tableau de résultats. Cette classe hérite *JFrame* de Java qui implémente une fenêtre basique.

— la classe **JPlateau** est responsable pour gérer l'affichage des pions. Elle est composée par une matrice de **JCase** qui représente une seule position du plateau. Ces deux classes héritent de *JPanel* fourni par Java et ont leurs méthodes paint redéfinies pour qu'elles puissent bien représenter l'état du plateau.

— **JResultat** est la classe chargé d'afficher les résultats. Elle fournit une liste de stratégies possibles et deux boites de texte où seront affichés la liste de coups déjà réalisés et la liste de coups de coups que sont possibles à chaque moment. Il y a aussi deux boutons, un pour réaliser un seul coup et l'autre pour finir le jeu selon la stratégie choisie.

3 Le codage

3.1 L'implémentation du paquet « Entités »

Les classes de ce paquet en général ont d'implémentation simple puisque leurs méthodes ne font que retourner l'état actuel de leurs attributs. Par conséquent, les méthodes des objets des classes comme **Coordonnee**, **Direction**, **Fichier** et **Case** ne fournissent que les *getters* et les méthodes d'écriture *toString*. Par contre, les classes **Plateau** et **Coup** possèdent des fonctionnalités plus importantes vis-à-vis au déroulement du programme. Par exemple, on trouve dans ces respectives classes les fonctionnalités suivantes :

— Un objet du type **Plateau** est capable de fournir des méthodes qui calculent toutes les cases actuellement vides, changent l'état d'une position quelconque et également retournent tous les voisins d'une coordonnée. Ces méthodes sont très utilisées par le contrôleur lors du calcul de chaque coup du jeu.

— Les méthodes appartenant à la classe **Coup** s'occupent plutôt de rétourner les coordonnées des cases d'origine, cible et l'éliminée et aussi la diréction du saute.

3.2 L'implémentation du paquet « Contrôle »

Ce paquet ne contient que trois components : deux interfaces et une classe, la plus importante de tout le programme, puisqu'elle doit gérer le fonctionnement de tous les éléments de l'application. Les interfaces ne font que fournir des méthodes qu'une classe doit implémenter pour être utilisée. Ainsi, toute classe qui a comme but afficher le plateau et les résultats doit avoir des méthodes qui permettent une communication entre elle-même et le contrôleur.

En ce qui concerne les outils fournis par **ControleurJeu**, on remarque :

— Le calcul d'un prochain coup selon une stratégie les entre neuf implémentées (voir section 3.2.1).

— Gérer les contrôleurs d'affichage et d'éntrée/sortie.

— Garder et maintenir l'état du jeu toujours consistent.

3.2.1 Stratégies de résolution

On divise les neuf stratégies développées en trois groupes, étant eux :

— Stratégies s'appuyant sur des critères locaux : dans ce groupe, on trouve des stratégies qui utilisent de caractéristiques par rapport à l'état actuel du plateau. En d'autre mots, elles n'essayent pas de trouver un parcours optimal entre plusieurs. Les critères jugés les plus importants ont été alors :

  — Nombre de voisins de la case finale : on choisit le couple dont réalisation donne une case avec le nombre le plus grand de voisins autour de soi. De cette façon, on préfère toujours les coups qui auront plus de possibilités d'avoir d'autres coups dans l'avenir. L'algorithme qui considère le nombre le plus petit a été fait également, mais seulement pour tester les différentes solutions.

- Distance par rapport au dernier coup réalisé : l'idée est de prendre un coup le plus distant possible du coup qui vient d'être fait. De cette façon, on espère avoir un équilibre de la quantité des pièces partout le plateau. On a également écrit l'algorithme qui fait l'opposé, c'est-à-dire, prefère les coups les plus proches et d'avoir les coups concentrés dans une partie du plateau.

- Distance du coup par rapport au centre : l'idée de cet algorithme est de prendre les coups qui, si réalisés, vont approcher les pièces du centre du plateau.

— Stratégies s'appuyant sur des critères globaux : les algorithmes appartenant à ce groupe vérifient plusieurs solutions possibles et les comparent.

- Chercher la solution optimale (*backtracking*) : la base de cet algorithme est de explorer récursivement tout les coups possibles du jeu, afin de trouver la solution contenant le maximum de coups. Évidémment, cet algorithme devient impraticable pour des plateau grands à cause de sa haute complexité.

- La technique *Branch and Bound* : l'algorithme consiste à chercher la liste de coups la plus grande possible en se baseant sur les tailles des sous-arbres que chaque coup possible peut avoir à chaque niveau d'itération. Par exemple, si à un instant donné, on a trois coups possibles, l'algorithme, pour chaque coup, va choisir et calculer la taille d'une des leurs sous-arbres et, une fois fini, il va les comparer. La sous-arbre choisie est alors la plus longue. On remarque encore qu'on a décidé de construire aléatoirement chaqu'une des sous-arbres.

— Autres stratégies : on a encore implementé certains algorithmes qui utilisent les méthodes décrites précédemment de façon mélangée. On cite :

- Mélanger l'algorithme de *Branch And Bound* avec la recherche de la solution optimale : on commence la recherche par la méthode de *Branch And Bound* et, lorsqu'on arrive à une déterminée quantité de pièces restantes, on finit avec l'algorithme qui compare toutes les solutions possibles.

- Réalisation de *Branch and Bound* plusieurs fois : on éxécute cet algorithme plusieurs fois et prend la liste de coups la plus grande entre les résultats.

- Réalisation de *Branch and Bound* avec des niveaux de profondeur : différemment de l'algorithme précédent, on parcourt chaque sous-arbre, qui répresente essentiellement un nouveau plateau, et après être arrivé à la profondeur choisie, on calcule le coups suivants avec l'un des nos algorithmes déjà présentés. Le choix de la profondeur dépend évidemment de la taille du plateau original puisque le temps de calcul est directement relié à la quantité de coups possibles à chaque niveau, ce qui augmente dans les cas des grands plateaux.

Après quelques essais, on a conclu que la dernière stratégie présente les meilleurs résultats.

3.3 L'implémentation du paquet « Interface »

Les dernières classes à être codées sont les classes qui vont effetivement afficher les résultats et écrire/lire des fichiers.

3.3.1 L'interface graphique

L'interface graphique a été conçue entièrement à partir des bibliotèques *swing* et *awt* fournies par *Java*, plus précisement des classes **JFrame** et **JPanel**, qui servent de *super classe* à certaines de nos classes, comme **JSolitaire** et **JPlateau/JResultat** respectivement. Le résultat se trouve affiché par la figure 5.

Outre son héritage, la classe JSolitarie implémente également l'interface **ActionListener**, qui lui permet de savoir et traiter l'événement généré par un possible click sur un des boutons. Cette classe est composée

par un plateau - un objet **JPlateau** - et un panneau de résultats - du type **JResultat**. Ces deux components sont placés sur l'espace disponible de fênetre selon les directives d'un *layout manager*. Dans ce cas, on a utilisé un simple **BorderLayout** qui n'impose pas de grandes difficultés de configuration mais, par compensation, n'offre pas un contrôle très précis des ses components.

La classe **JPlateau** contient une matrice de components **JCase**, qui héritent également de **JPanel**. Cette matrice est évidemment une réflexion de la matrice contenue dans un objet du type **ControleurJeu**. En ce qui concerne le *layout manager*, appelé **GridLayout**, on remarque qu'il joue un rôle fondamental puisque c'est lui le responsable pour diviser et mettre l'écran en forme matricielle. En conséquence, nous sommes épargnés de certains calculs chaque fois qu'on doit modifier les dimensions du plateau. Le dernière caractéristique importante consiste à modifier la méthode *paintComponent* pour qu'elle puisse appeler l'opération *paint* de chaque objet **JCase**. Cette dernière ne fait que dessiner un cercle peint en fonction de l'état.

Le dernier remarque consiste à noter que, à principe, on n'aurait pas besoin de construire une nouvelle classe pour répresenter une case. On pourrait n'avoir qu'une seule matrice d'entiers pour garder l'état. Cependant, cette configuration présente un problème, puisqu'on ne serait pas capable d'appeler la méthode *paint* lors d'une modification. En conséquence, on serait obligé à calculer « à la main »toutes les coordonnées à chaque fois.

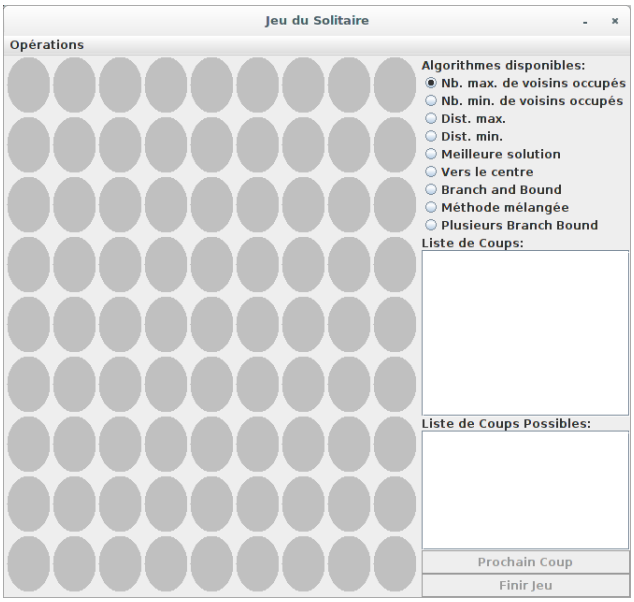

FIGURE 5 – Interface graphique.

3.3.2 L'interface d'entrée et sortie

Afin de lire et écrire sur les fichiers on a utilisé les classes **BufferedReader** et **FileWriter** respectivement. Toutes les deux peuvent lancer une exception si le fichier qu'on essaie de lire ou écrite n'existe pas. Dans ce cas, on a choisi de les traiter à niveau de la classe **JSolitaire**, qui peut, à son tour, afficher une boîte d'information en communiquant l'erreur.

4 Annexes

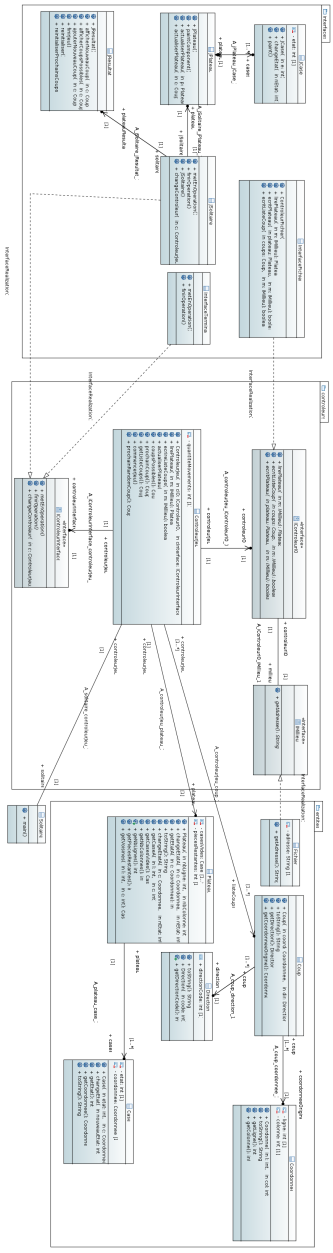

FIGURE 6 – Diagramme de classes.