

Sylvain Archenault
Yves Houpert



PROJET INFORMATIQUE :

Langage Java : Jeu De Dames en Java



Projet GM3
Mai 2005

Chapitre 1

INTRODUCTION

Le projet qui nous a été confié est de réaliser un jeu de dames en Java. Le but de ce projet est d'utiliser habilement les concepts de la programmation objet, c'est à dire l'héritage entre les classes, la gestion des interfaces ou des classes abstraites , etc...Nous avons déjà réalisé un jeu de dames en langage C au semestre dernier et il a été donc très intéressant de pouvoir remarquer les différences de programmation entre une programmation séquentielle et une programmation objet.

Pour bien répondre au problème, nous allons d'abord présenter **les différentes classes** qui été nécessaires à la réalisation du jeu de dames .

Dans une autre partie, nous parlerons plus précisément du **polymorphisme** propre aux langages objets. Nous le verrons ici plus particulièrement en action avec le Java.

Enfin,nous exposerons **les différents problèmes** que nous avons rencontré lors de la programmation et **comment nous y avons remédié.**

Chapitre 2

LES CLASSES

Les classes sont la base de la programmation objet . Nous avons eu besoin d'utiliser plusieurs classes dans ce projet avec par exemple des relations d'héritage entre elles (nous verrons ceci plus précisément dans la partie 3). Nous allons tout d'abord exposer les différentes classes créées pour la conception du jeu. Ensuite, nous illustrerons nos explications par un graphisme réalisé à l'aide du logiciel Umbrello pour bien comprendre la composition des classes ainsi que leurs différentes relations.

2.1 Liste de toutes les classes créées

- **la classe Arbitre :**

La classe Arbitre est la classe qui contrôle le déroulement du jeu. C'est l'Arbitre qui vérifie que la partie n'est pas terminée, si un coup est valable. Il détermine aussi les coups obligatoires, etc..

- **la classe Plateau :**

La classe Plateau décrit le plateau du jeu de dames.

- **la classe Case :**

La classe Case est une classe abstraite décrivant une case du plateau du jeu de dames. Cette classe hérite de la classe JComponent puisqu'on souhaite l'ajouter à un JPanel. On va donc redéfinir les méthodes d'affichages. On implémentera également l'interface *MouseListener* car on veut gérer les cliques et les déplacements de la souris.

- **la classe CaseBlanche :**

La classe CaseBlanche décrit une case blanche du jeu de dames. Cette case est bien sûr *inactive* car les pions ne se déplacent que sur les cases noires.

- **la classe CaseNoire :**

La classe CaseNoire décrit une case noire du jeu. Contrairement aux cases blanches, elles sont *actives*.

- **la classe Rafle :**

La classe Rafle est une des classes importantes du jeu. En effet, elle permet de calculer la rafle maximale et de la signaler au joueur. Expliquons ceci plus en détails :

La classe Rafle symbolise un coup d'une rafle effectuée au jeu de dames. Une rafle au jeu de dames est donc symbolisée par une liste de Rafle. Chaque élément (objet) contient la case du début du coup, la case prise (c'est à dire la case du dernier pion pris) du coup précédent et un vecteur contenant la liste des cases suivantes possibles. En effet, il se peut que pour une case donnée, on ait la possibilité d'obtenir plusieurs autres coups possibles.

- **La classe HistoriqueCoup :**

Cette classe sert bien sûr à gérer l'historique des coups.

- **la classe Piece :**

La classe Piece est une autre classe abstraite. Elle symbolise une pièce du jeu. En l'occurrence ici, il s'agira soit d'un pion, soit d'une dame.

– **la classe Pion :**

La classe Pion symbolise un pion du jeu de dames.

– **la classe Dames :**

La classe Dame représente une dame. Elle est un peu plus complexe que la classe pion car le déplacement d'une dame est plus difficile à étudier que celui d'un pion.

– **la classe Joueur :**

A nouveau une classe abstraite. Elle décrit un joueur du jeu. Ce joueur peut aussi bien être un humain que l'ordinateur.

– **la classe Humain :**

La classe Humain décrit un joueur du type Humain.

– **la classe Ordinateur :**

La classe Ordinateur représente bien sûr un joueur contrôlé par l'ordinateur pendant la partie.

– **la classe JeuDeDamesWindow :**

Cette classe hérite de la classe JComponent comme la classe Case vu précédemment. Elle gère la fenêtre du jeu de dames et l'affichage des composants de la fenêtre.

– **la classe NouvellePartieWindow :**

Comme son nom l'indique, la classe NouvellePartie décrit la fenêtre permettant de commencer une nouvelle partie.

– **la classe APropos :**

C'est la classe qui décrit la fenêtre APropos. Cette fenêtre donne des renseignements sur le jeu ; ses concepteurs par exemple ou d'autres indications.

Nous allons maintenant illustrer toutes ces informations par un diagramme de classes qui rendra les explications de ce paragraphe plus claires.

2.2 Les classes représentées par un graphisme UML

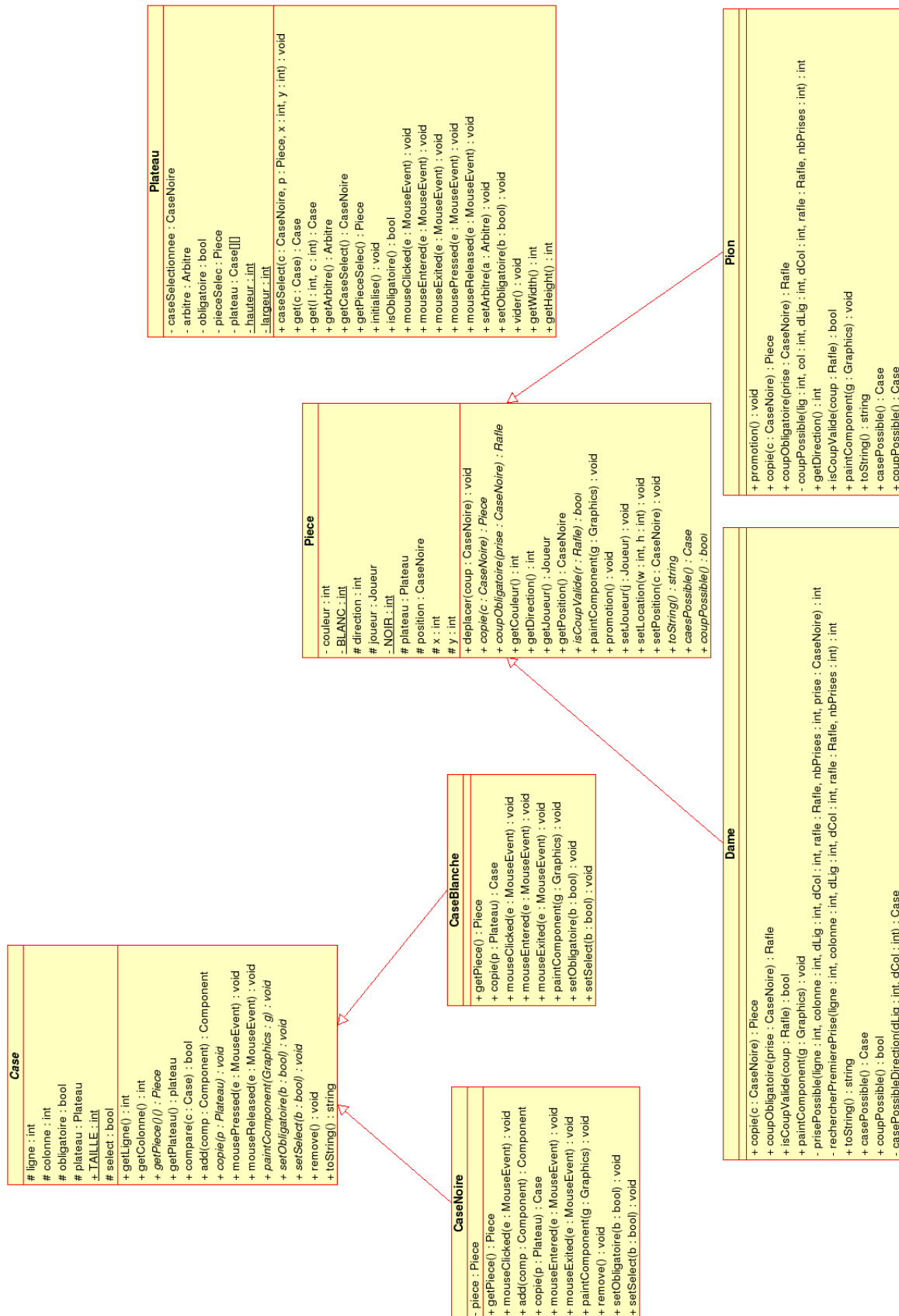
Cette idée d'utiliser *le logiciel Umbrello* nous a permis d'obtenir *une programmation très structurée*. En effet, nous avons eu la possibilité d'illustrer toutes les classes avec leurs attributs et leurs méthodes. Nous avons pu facilement remarquer les différents liens que nous devons instaurer entre les classes .

2.2.1 Le diagramme des classes

Nous avons également *réaliser des scénarios représentant les différentes possibilités de jeu* que nous offrions par exemple la gestion d'une raffle. Cette pratique nous a permis de trouver les différentes méthodes de chaque classe d'une manière optimale et très structurée. Pour plus d'informations sur les classes, ainsi que leurs méthodes et attributs, nous avons généré une documentation à l'aide de javadoc ; elle est disponible sur :

<http://gm.insa-rouen.fr/~archenas/projets/JeuDeDames/Doc/>.

II. LES CLASSES : Les classes représentées par un graphisme UML



II. LES CLASSES : Les classes représentées par un graphisme UML



JeuDeDamesWindow	NouvellePartieWindow
- arbitre : Arbitre - constraints : GridBagConstraints - hauteur : float - j1Abandon : JButton - j1Annul : JButton - j1Nul : JButton - j2Abandon : JButton - j2Annul : JButton - j2Nul : JButton - joueur1 : JLabel - joueur2 : JLabel - largeur : float - pane : JPanel - plateau : Plateau - posX : float - posY : float + aPropos() : void + chargerPartie() : void + effacePane() : void + getJ1Label() : JLabel + getJ1Nom() : string + getJ2Label() : JLabel + getJ2Nom() : string + main(args : String[]) : void + nouvellePartie() : void + nouvellePartieCree() : void + quitter() : void + sauverPartie() : void + setJ1Nom(nom : string) : void + setJ2Nom(nom : string) : void	- annulButton : JButton - arbitre : Arbitre - height : int - j1Hmain : JRadioButton - j1Nom : JTextField - j1Ordianteur : JRadioButton - j1Type : ButtonGroup - j2Hmain : JRadioButton - j2Nom : JTextField - j2Ordinateur : JRadioButton - j2Type : ButtonGroup - joueur1 : JLabel - joueur2 : JLabel - JPanel : JPanel - nomJ1 : JLabel - nomJ2 : JLabel - okButton : JButton - parent : JeuDeDamesWindow - width : int - creeNouvellePartie() : bool - getAnnulButton() : JButton - getJ1Hmain() : JRadioButton - getJ1Nom() : JTextField - getJ1Ordinateur() : JRadioButton - getJ1Type() : ButtonGroup - getJ2Hmain() : JRadioButton - getJ2Nom() : JTextField - getJ2Ordinateur() : JRadioButton - getJ2Type() : ButtonGroup - getJPanel() : JPanel - getOkButton() : JButton - initialize() : void - nomCorrect(nom : string) : bool

Chapitre 3

UTILISATION DU POLYMORPHISME

Le projet a été réalisé de manière à ce que les atouts de la programmation objet soient utilisés habilement et mis en valeurs. Nous allons voir dans cette partie comment nous nous en sommes servis et quels sont leurs caractéristiques au niveau du jeu de dames.

3.1 L'héritage dans le jeu de dames

On sait que l'héritage permet la construction et l'extension par affinage des applications. Dans ce projet, nous avons trois grands exemples d'héritage :

3.1.1 Les classes *CaseNoire* et *CaseBlanche* qui héritent de la classe *Case*

Il nous est apparu évident d'utiliser l'héritage pour gérer les cases du plateau. En effet, les cases noires et les cases blanches se comportent différemment, tout en ayant des caractéristiques communes. Le plateau du jeu de dames possède à la fois des cases blanches et noires. Nous avons donc décidé de créer une classe abstraite *Case* dont dérivent les classes *CaseNoire* et *CaseBlanche*. De ce fait, *Plateau* est simplement un tableau de *Case*. Le polymorphisme nous permet donc de spécifier un comportement différent pour les cases blanches et les cases noires de manière très simple. Illustrons cela.

A l'initialisation du plateau, nous attribuons à chaque élément du tableau une *CaseBlanche* ou une *CaseNoire* suivant sa position. C'est ici le seul moment dans le programme où nous allons différencier les cases par leur position. En effet, dans la suite de l'exécution, c'est le polymorphisme qui se chargera de différencier les cases blanches des cases noires. Prenons l'exemple de l'affichage des cases. Nous avons défini dans la classe *Case* la méthode abstraite *paintComponent()* qui est appelée par Java lorsque la case doit être affichée ou bien repeinte. Dans les classes dérivées *CaseNoire* et *CaseBlanche* nous avons donc redéfini cette méthode pour que l'affichage des cases soit différent. Dans *paintComponent()* de la classe *CaseNoire*, nous dessinons un carré noir, alors que dans *CaseBlanche* nous dessinons un carré blanc. Nous avons donc réussi à afficher le damier d'une manière très facile grâce au polymorphisme.

Il est important de remarquer que la classe *Case* peut être réutilisée dans la réalisation d'un autre jeu puisque celle-ci reste très générale et offre les méthodes de bases pour la gestion des cases d'un jeu. Nous pourrions, par exemple la réutiliser dans un jeu d'échec ou d'othello.

3.1.2 Les classes Humain et Ordinateur qui héritent de la classe Joueur

Nous avons vu dans le précédent chapitre que c'est une instance de la classe arbitre qui contrôle le jeu et l'interaction entre les joueurs. Nous avons décidé là aussi, d'utiliser le polymorphisme pour la modélisation des joueurs. Nous avons donc créé une classe abstraite Joueur qui décrit d'une manière générale un joueur du jeu de dames. Nous avons ensuite créé deux classes dérivées Ordinateur et Humain, celles-ci plus spécifiques. L'application du polymorphisme dans ce cas se traduit par la méthode *jouer()* qui permet au Joueur de jouer. En effet, s'il s'agit d'un humain ou d'un joueur contrôlé par l'ordinateur, la méthode doit être différente. Nous avons donc créé une méthode abstraite *jouer()* dans la classe Joueur, que nous avons défini différemment dans les classes Humain et Ordinateur. De cette façon, l'arbitre ne se préoccupe pas si les joueurs sont des humains ou des joueurs contrôlés par l'ordinateur.

D'un autre côté, la méthode gérant la création des pièces est commune aux deux types de joueurs. Nous avons donc créé une méthode *debut()* qui crée les pièces du joueur.

Ceci apporte donc une grande facilité de programmation, ainsi qu'une meilleure clarté.

De la même manière que pour la classe Case, nous avons essayé de rendre la classe Joueur la plus portable possible, c'est à dire qu'on ait la possibilité de la réutiliser dans d'autres programmes.

3.1.3 Les classes Pions et Dames qui héritent toutes les deux de la classe Piece

Ce choix semble donc également très pertinent pour la gestion des pièces. En effet, nous pouvons utiliser les mêmes méthodes pour par exemple retourner la couleur de la pièce ou sa position. Ces méthodes sont tout à fait indépendantes des pions ou des dames. Nous n'avons pas besoin d'utiliser de boucle if comme en programmation séquentielle car nous pouvons généraliser les méthodes en utilisant la classe abstraite Piece. Là où la programmation objet prend tout son sens, est lorsque nous souhaitons calculer les coups obligatoires, ou encore vérifier qu'un coup est valide. En effet, en programmation séquentielle, nous devons différencier les cas des dames ou des pions, ici nous avons simplement à redéfinir la méthode abstraite de la classe Piece dans les sous-classes Pions et Dames pour avoir un traitement particulier. Illustrons ceci par deux exemples pertinents :

- Vérification de la validité d'un coup *isCoupValide()*

Lorsque le joueur veut effectuer un coup, il clique sur la case de la pièce qu'il souhaite déplacer, puis clique sur la case d'arrivée du coup. Afin de vérifier si le coup est possible, nous avons implémenté la méthode *isCoupValide()*. C'est une méthode abstraite dans la classe Piece, que nous avons définie dans les classes Pions et Dames. Il est évident que les règles de déplacements diffèrent pour les pions et les dames. Nous l'avons donc programmée d'une manière différente dans chaque classe. Mais, lorsque nous vérifions si un coup est valide nous appelons la méthode *isCoupValide()* de la classe Piece, c'est ensuite Java qui choisit d'appeler la méthode de la classe Pion ou de la classe Dame suivant le cas. Ceci nous permet donc de gérer les pions et les dames du jeu de la même manière.

- Calcul du coup obligatoire :

Lors que l'on souhaite calculer le coup obligatoire pour un joueur, nous parcourons la liste des pièces qu'il possède. Il est important de noter que la liste des pièces du joueur est stockée dans un Vecteur. Pour chaque pièce, nous appelons la méthode *coupObligatoire()*, qui renvoie le coup obligatoire pour la pièce en question si il existe. Il est bien évident que le coup obligatoire est différent pour les pions et les dames. Mais grâce au polymorphisme, nous ne faisons aucune différence. Le polymorphisme apporte donc ici une très grande simplicité de programmation.

Comme pour les deux classes abstraites précédentes, la classe Piece est réutilisable pour la modélisation d'autres pièces.

Nous avons présenté ici que quelques exemples d'héritage et de polymorphisme, il y a bien évidemment d'autres cas où le polymorphisme intervient.

3.2 Les interfaces dans le jeu de dames

Dans le projet nous n'avons pas programmé d'interfaces. Mais nous avons tout de même utilisé des interfaces fournies par la langage Java, il s'agit principalement de l'interface `MouseListener`. Cette interface permet de gérer les événements de la souris. Nous avons implémenté cette interface dans la classe `CaseNoire`. En effet nous voulons intercepter les clics de la souris sur la case, en effet c'est par ce biais que le joueur peut sélectionner une pièce. Ensuite nous voulons également connaître quand la souris entre ou sort d'une case pour pouvoir mettre en sur-brillance cette case. En effet, un fois que le joueur a sélectionné une case, c'est-à-dire une pièce, dès que la souris entre dans une autre case, alors on la met en sur-brillance si le coup est réalisable.

Chapitre 4

LES DIFFICULTÉS RENCONTRES LORS DU PROJET

4.1 La raffle maximale

Une des difficultés majeures du projet a été de calculer les coups obligatoires. Rappelons brièvement les règles du jeu de dames. La prise prioritaire est celle où il y a le plus de pions à prendre (la dame compte pour un pion), Le joueur doit obligatoire effectuer la prise prioritaire. Il se peut qu'il existe plusieurs prises prioritaires possibles.

Pour calculer les coups obligatoires, nous calculons pour chaque pièce le coup prioritaire, et ne retenons à la fin que les coups qui permettent de prendre le maximum de pions. Pour modéliser une raffle, nous avons choisis d'écrire une classe *Raffle* ayant les attributs suivant :

- (CaseNoire) *caseDebut* : C'est la case du début de la raffle.
- (CaseNoire) *casePrise* : C'est la case prise au coup précédent.
- (Vector) *casesSuivantes* : C'est la liste des Raffles pouvant ensuite être effectuées.

La classe possède d'autre attributs, mais nous nous intéressons ici qu'au principaux.

Une raffle est alors modélisée par une suite de *Raffle*. Illustrons ces propos par un exemple simple. Si nous voulons symboliser le coup A,1 - C,3 avec un pion pris en B,2, nous allons donc avoir un premier objet *r1* de type *Raffle* qui aura comme attribut (*caseDebut* : A1, *CasePrise* : null, *casesSuivantes* : *r2*). L'objet *r2* est également du type *Raffle* et a pour attribut (*caseDebut* : C3, *casePrise* : B2, *casesSuivantes* : null).

4.2 Les problèmes liés à la partie graphique du projet

4.2.1 Déplacement des pièces

Au début du projet, nous souhaitions déplacer les pièces du jeu par un principe de glisser-déposer. Mais nous n'y sommes pas parvenus. En effet la méthode *MouseDragged()* de l'interface *MouseMotionListener*, ne nous paraissait pas bien adaptée à notre problème. Il aurait fallu redéfinir le glisser-déposer pour la pièce. Nous n'avons pas eu le temps de nous pencher sur ce problème.

Ensuite nous avons eu l'idée de déplacer la pièce de manière à ce qu'elle suive la souris. C'est-à-dire que lorsque le joueur clique sur une pièce, la pièce suit le mouvement de la souris (*mouseMove()*). Puis lorsque l'utilisateur clique à nouveau, nous gérons le coup effectué. Mais ceci pose problème, en effet, comme la pièce se trouve constamment sous la souris, c'est la pièce qui reçoit l'événement *mouseClicked()*. Or, dans le programme, c'est la case qui est censée gérer les cliques de souris. Une astuce possible pour résoudre le problème serait de changer le curseur de la souris, c'est à dire d'attribuer à la souris la forme de la pièce.

4.2.2 Redimensionnement de la fenêtre

La fenêtre du Jeu de Dames ne réagit pas bien au redimensionnement. En effet, le plateau reste toujours à la même position alors qu'il serait mieux qu'il se déplace au centre de la fenêtre.

Ensuite la taille des cases du plateau est toujours la même quelque soit la résolution de l'écran, il serait plus agréable que celle-ci s'adapte à la résolution de l'écran.

Chapitre 5

CONCLUSION

Ce jeu de dame en langage Java fut très intéressant à concevoir et à programmer.

Ce projet nous a permis de bien comprendre les qualités de la programmation objet. Nous avons essayé d'utiliser au mieux ses caractéristiques tels que l'héritage ou les classes abstraites.

Ces propriétés de la programmation sont très pratiques et permettent une bien meilleure conception et réalisation du projet. Nous avons pu remarqué ces qualités car nous avons déjà réalisé un jeu de dames en C. La programmation du jeu de dames en Java, nous a donc permis de nous rendre compte de la différence entre ces deux modes de programmation. Il nous est apparu bien plus facile de concevoir le programme en Java qu'en C. De plus, nous sommes rendus compte que ***les classes abstraites pourront être facilement réutilisable dans d'autres projets***, chose qui serait beaucoup plus difficile avec les fonctions écrites en C.

Au début du projet, nous avons trouvé les différentes classes et méthodes à créer grâce à l'utilisation de scénarios et de diagrammes de classes. Cette démarche permet d'avoir une réflexion structurée tout au long de la programmation.

Au niveau des améliorations possibles du projet, il serait très intéressant de pouvoir mettre le jeu en réseau. Il serait également plus agréable que l'interface graphique soit plus chaleureuse. Pour le confort du jeu, il serait intéressant d'améliorer le jeu de l'ordinateur.