RAPPORT FINAL DE PROJET

PROGRAMMATION MULTI-AGENTS AVEC MADKIT

REALISE PAR

VICTOR GROSCLAUDE ET ALICIA FLOREZ

SOUS LA DIRECTION DE

FABIEN MICHEL

ANNEE UNIVERSITAIRE 2011 – 2012

RAPPORT FINAL DE PROJET

PROGRAMMATION MULTI-AGENTS AVEC MADKIT

REALISE PAR

VICTOR GROSCLAUDE ET ALICIA FLOREZ

SOUS LA DIRECTION DE

FABIEN MICHEL

ANNEE UNIVERSITAIRE 2011 – 2012

REMERCIEMENTS

Merci à Fabien Michel, notre tuteur de projet, de nous avoir laissé la liberté dont nous avions besoin pour réaliser ce projet, mais également d'avoir été là lorsque que nous avions des questions sur le sujet complexe qu'était la programmation multi-agents.

Nous tenons également à remercier l'ensemble des professeurs de l'IUT pour les connaissances qu'ils nous ont apportées, en particulier sur les bonnes pratiques de la programmation qui nous ont permis de mener à bien ce projet.

Merci à Myriam Gelsomino, professeure de communication pour ses précieux conseils et corrections apportées au rapport.

Et pour finir Marianne Barrey et Héloïse Pialot pour leurs relectures attentives de ce document.

SOMMAIRE

TABLE DES FIGURES

GLOSSAIRE

Tous ces termes sont identifiés dans le texte par la présence d'un **astérisque** *.

Agent : entité le plus souvent autonome capable de communiquer avec un ou plusieurs autres agents en fonctions de messages.

Classe : composante de base de la programmation orientée objet, c'est une structure contenant données et méthodes à exécuter.

Game Design Document (ou GDD) : document qui présente en détail tous les aspects d'un jeu vidéo nécessaires à sa création.

Héritage : concept de la programmation objet, il permet à une classe « fille » d'hériter des méthodes et des attributs d'une classe « mère ».

Java : langage de programmation orienté objet utilisé lors du projet.

Librairie (ou bibliothèque) : ensemble de méthodes utilitaires regroupées et mises à disposition afin de pouvoir les utiliser sans avoir à les réécrire.

Système multi-agent : système composé d'un ensemble d'agents, situés dans un certain environnement et interagissant selon certaines relations.

1 INTRODUCTION

Bien que la programmation existe sous différentes formes depuis maintenant longtemps, elle ne permet malheureusement pas de répondre à tous les problèmes. C'est dans le but d'en combler un certain nombre que la programmation multi-agents* est née. Celle-ci se caractérise par la présence d'agents* dans le programme. Elle est particulièrement utile lorsqu'il s'agit de modéliser des sociétés ou une Intelligence Artificielle.

Un agent est donc une entité au moins partiellement autonome, capable d'interagir avec son environnement et d'autres agents en fonction de messages qu'on lui envoie.

Dans le cadre de notre projet, nous avons donc été initiés à la programmation multi-agents : nous devions réaliser un programme qui, en utilisant des agents, tirerait partie au maximum de leurs spécificités. Nous avons donc réfléchi, et notre choix s'est porté sur un jeu. En effet, quoi de mieux que des personnages qui agissent d'eux-mêmes et entre eux pour exploiter des agents de façon intéressante. Nous nous sommes donc lancés dans la création d'un jeu mêlant stratégie et combat, proposant à deux joueurs de faire s'affronter de petites armées.

Afin de structurer ce rapport de projet, nous avons décidé de suivre un plan classique. La première partie sera donc en rapport avec la réflexion concernant le jeu : l'analyse en elle-même et le Game Design Document*. Puis viendra la partie concernant la conception même du jeu, qui sera elle en trois parties distinctes. Enfin, nous regarderons les résultats de ces quelques mois de travail, puis pour conclure nous discuterons autour du projet final, de ses améliorations possibles ou de son avenir...

2 ANALYSE

2.1 CONTEXTE ET OBJECTIFS

Avant de commencer à parler de notre projet, nous voulions vous présenter la programmation multi-agents, la libraire MadKit* ainsi que nos objectifs.

Apparue dans les années 80, la programmation multi-agents (ou Système Multi-Agents) est très différente de la programmation objet. En effet, contrairement à cette dernière qui nous permet d'interagir de l'extérieur sur les objets, celle-ci ne permet pas de contrôler les agents créés.

En effet, ceux-ci sont le plus souvent quasiment autonomes, si ce n'est pas entièrement. On ne peut leur donner des ordres, mais l'on peut leur demander de faire une tâche par le biais d'envoie de messages, qu'ils traiteront ou non. Un Système Multi-Agents (SMA) est donc un système composé d'agents qui interagissent entre eux. Ces agents peuvent donc représenter n'importe quelle entité humaine, animale ou représenter un processus.

Notre premier objectif était l'apprentissage et la découverte de la librairie* MadKit[1] qui nous aidera dans notre programmation de notre SMA.

MadKit est une plate-forme libre de développement de systèmes multi-agents (SMA) destinée au développement et à l'exécution de SMA et plus particulièrement à ceux fondés sur des critères organisationnels (groupes et rôles).

Comme précisé précédemment, la programmation multi-agents étant complètement différente de tout ce que nous avions appris auparavant, il a fallu nous familiariser avec, à l'aide des documents et tutoriels fournis sur le site. S'en sont suivis la création de petits programmes afin de tester les différentes méthodes mises à notre disposition dans MadKit. Un des gros objectifs de ce projet était donc d'appréhender la programmation multi-agent ainsi que de progresser en Java, le langage de programmation utilisé au cours de ce projet.

Notre second objectif a été le choix puis la création d'un système multi-agents. Notre choix s'est immédiatement porté sur un jeu qui utiliserait des agents, un jeu de combat s'inspirant de La Brute[2] à la différence près qu'il n'y aurait pas un seul personnage par équipe, mais plusieurs. Chaque personnage serait représenté par un agent complètement autonome : ce sera lui qui choisira ce qu'il fera pendant sa durée de vie.

[1] http://www.madkit.org

[2] http://labrute.com

2.2 ANALYSE

Comme pour tous les projets logiciels, la première étape de la conception a été l'analyse. De plus, notre projet portant sur un jeu, il a été nécessaire de rédiger un Game Design Document. Cette première partie va donc traiter de la phase d'analyse et de la rédaction du GDD.

2.2.1 GAME DESIGN DOCUMENT

Au cours de la réalisation de notre jeu, nous avons été tout logiquement amenés à nous poser une question : que mettre dans celui-ci ? Pour répondre à cette question, les professionnels du secteur réalisent un document contenant toutes les aspects du jeu, de la charte graphique au gameplay : c'est le Game Design Document (GDD).

Voici donc une petite partie de notre GDD, qui nous a permis de concevoir un jeu à la fois intéressant pour le joueur et cohérent pour les développeurs, tout en gardant des possibilités d'évolution pour le futur.

2.2.1.1 DEFINITION DU JEU

Le choix du type et du genre de jeu dans le cadre de la programmation multi-agents n'est pas bien compliqué : ceux-ci sont conçus pour la simulation, faire un jeu de stratégie semble donc évident au premier abord. En creusant plus loin, il suffisait de faire s'affronter deux joueurs, à l'aide d'une petite armée qu'ils auraient créée pour obtenir un jeu à la fois simple et efficace.

2.2.1.2 BUT ET FIN DU JEU

Le joueur qui possède les derniers personnages en vie est le vainqueur.

2.2.1.3 CRÉATION D'UNE ÉQUIPE

Le jeu permet donc de choisir un camp à incarner ("gentil" ou "méchant"), puis de créer une équipe à l'aide des mascottes du jeu vidéo présentes dans le jeu, chaque personnage possédant divers attributs (points de vie, attaque …). Une équipe de "gentils" pourra être composée de Mario ou Sonic par exemple alors que son adversaire pourra choisir un personnage comme Bowser ou Shadow. Chaque joueur pouvant choisir cinq personnages possédant des propriétés différentes, et choisir le même personnage plusieurs fois, cela peut donner lieu à beaucoup de combats différents, et ainsi continuer à susciter l'intérêt du joueur.

2.2.1.4 SYSTÈME DE COMBAT

De ce côté là c'est aussi simple : chaque personnage possède un personnage "rival" qu'il va chercher à combattre en priorité. S'il le trouve, il engage le combat, sinon, il va se battre contre un adversaire choisi au hasard dans l'armée adverse. Une fois le combat lancé, les deux personnages engagés vont se battre jusqu'à ce que l'un des deux soit vaincu. C'est dans tout ce système de combat que les agents rentrent en jeu : soit pour la recherche d'un adversaire, soit pour le combat à l'aide d'échange de messages entre agents.

2.2.2 UML

Une fois l'aspect ludique du jeu défini dans le GDD*, il a fallu commencer à réfléchir à son utilité, ses fonctionnalités et tout ce qui entrait en compte dans sa réalisation : il a fallu réaliser une analyse.

Bien avant d'entrer dans des histoires de classes* et d'héritage*, nous avons commencé par définir les besoins de l'utilisateur, et les interactions qu'il pourrait avoir avec le logiciel.

Il est clair que dans un jeu vidéo, ces interactions sont primordiales : ce sont elles qui font le jeu, toutes sont importantes.

La première chose que peut faire un utilisateur (et même ici un joueur) est bien évidemment, une fois le jeu lancé, de le fermer à tout moment en cliquant sur la croix présente en haut de la fenêtre. Il possède également la possibilité de la réduire, ce qui met le jeu en pause tant que la fenêtre ne revient pas au premier plan.

La première chose à faire au lancement du jeu est de choisir si l'utilisateur affrontera un autre joueur humain ou une intelligence artificielle.
Une fois au cœur du jeu, un choix s'offre au joueur : il doit choisir une équipe entre les "gentils" ou les "méchants", ce qui décide des personnages qu'il peut utiliser par la suite. Il accède à un second écran qui lui permet de choisir sur quelle carte il va affronter son adversaire.

Le joueur arrive alors sur l'écran le plus important du jeu : celui de la création de son équipe. Sur cet écran, l'utilisateur va avoir plusieurs choix ; en effet, il va pouvoir ajouter et retirer des personnages à son équipe en les sélectionnant dans une liste, tant qu'il n'a pas atteint le nombre maximum. Mais le joueur a également la possibilité au moment de l'ajout d'un personnage de l'améliorer ou non par le biais de cases à cocher. Si le joueur est satisfait, il peut passer à la suite, sinon, il peut toujours revenir à l'écran précédent.

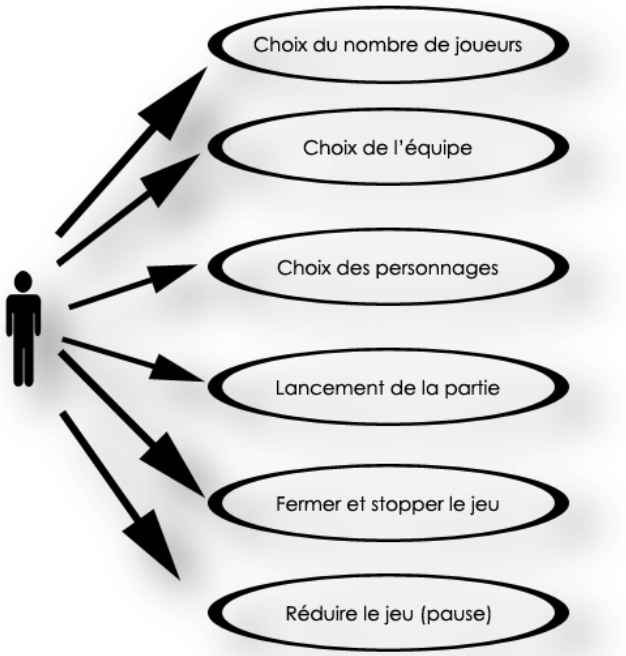

Figure 1 – Use Case du joueur

Une fois que tout a été décidé et les équipes créées et validées (par les joueurs, ou par l'ordinateur si un seul joueur physique joue), on accède à l'écran de combat. Ici, peu d'interactions s'offrent à l'utilisateur : en effet il a juste à regarder le combat afin de connaître son issue et le vainqueur du match.

L'utilisateur possède tout de même la possibilité de mettre le jeu en pause par le biais d'une pression sur la touche "echap", et bien sûr de quitter à tout moment avec la croix.

Le programme se décompose bien sûr en plusieurs classes*, la principale étant la classe Personnage, c'est elle qui contient toutes les actions des Agents. Chaque type de personnage est représenté par une classe qui lui est propre héritant* de Personnage.

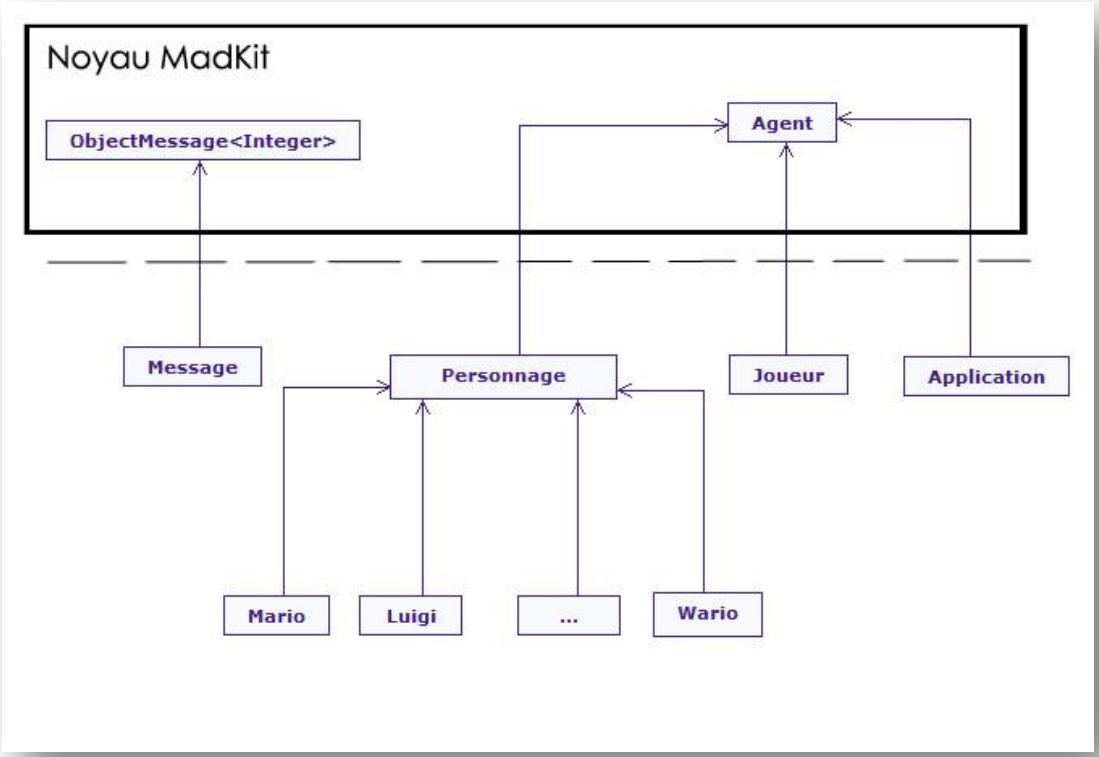

Figure 2 – Diagramme des classes de l'interface moteur

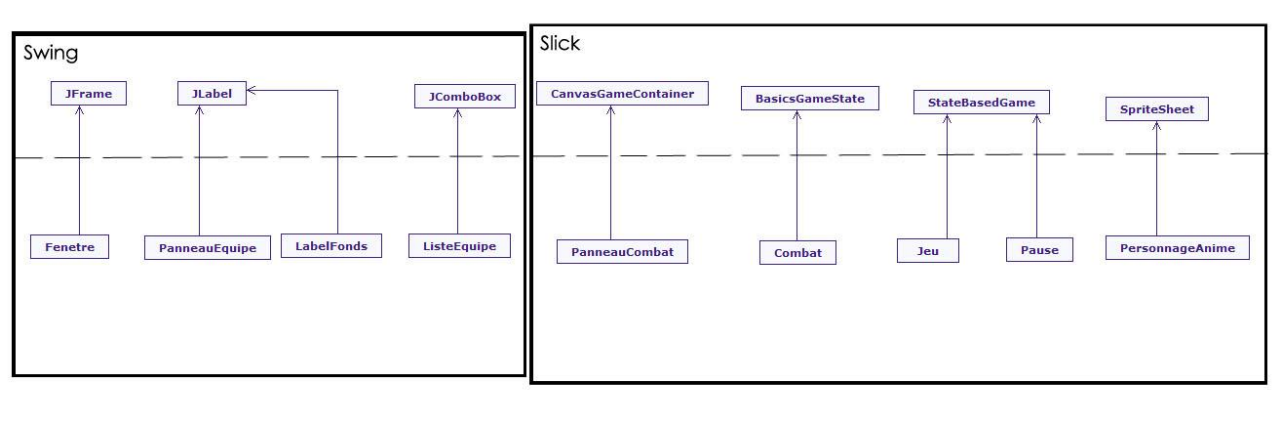

Figure 3 – Diagramme des classes de l'interface graphique

3 CONCEPTION

Voici maintenant la partie conception de notre logiciel, c'est à dire tout ce qui a trait à la programmation réalisée en Java, le langage natif de MadKit.

3.1 MADKIT

L'intitulé original de notre projet étant "Madkit", il nous a semblé logique de dédier une partie de notre rapport à cette librairie.

La librairie Madkit, qui en est aujourd'hui à sa cinquième version, est une librairie qui a pour but de permettre une utilisation d'un système multi-agents directement en Java. Celle-ci est basée sur le modèle organisationnel Agent/Groupe/Rôle, ce qui permet de facilement créer des simulations, des sociétés artificielles. Avant de rentrer en détail dans Madkit, voici la liste des principales fonctionnalités qui nous ont servies dans ce projet :

- la création d'agents artificiels et la gestion de leur cycle de vie
- une infrastructure organisée pour la communication entre les agents

Nous allons maintenant voir à quoi les agents nous ont servis dans la création de notre jeu, chacun d'entre eux possédant un Rôle particulier. Dans un premier temps, il faut savoir que nous avons utilisé un type d'agents particuliers : la classe* Agent, qui dérive d'AbstractAgent. Il faut savoir que la différence entre les deux est la suivante : un Agent est thréadé là où un AbstractAgent ne l'est pas. Nous avons choisi d'utiliser ces premiers, car dans notre application, plusieurs choses peuvent se passer au même moment, et ce, de manière automatique.

De plus, leur cycle de vie est très simple à mettre en place : une initialisation, une vie dans laquelle on place généralement une boucle infinie, et une fin.

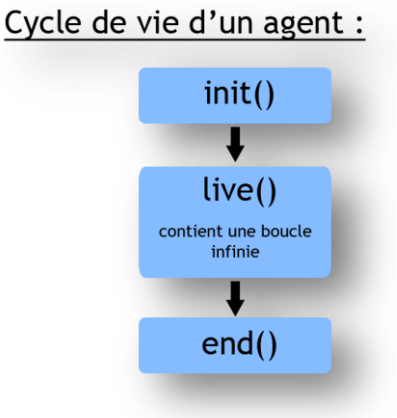

Figure 4 – Cycle de vie d'un agent

Cette dernière servant seulement dans des cas particuliers, nous n'en avons pas eu besoin. La méthode la plus importante est bien évidement "live", qui contient la vie de l'Agent, et donc les actions qu'il va effectuer tant qu'il est en vie. C'est dans celle-ci que nous avons codé toutes les actions et relations que va entreprendre un agent. Puisque l'Agent va répéter cette méthode "live", la première chose importante à savoir à propos des agents, c'est qu'ils agissent de manière autonome : on n'influe pas directement sur leur comportement.

En revanche, il existe quelque chose qui en est capable : un autre agent. En effet, les agents communiquent entre eux grâce à un service de messagerie plutôt simple. Si deux agents sont dans le même Groupe organisationnel, ils peuvent s'échanger des messages. Il existe plusieurs façon d'envoyer des messages, mais celle qui nous intéresse ici est la plus basique : SendMessage. Celle-ci permet tout simplement d'envoyer un objet de type ObjectMessage<T> (un type générique donc) à une AgentAdress. Il faut savoir que chaque Agent possède une ou plusieurs AgentAdresse en fonction de ses rôles, et que celle-ci est récupérable à l'aide de la fonction GetAgentWithRole. Une fois le message envoyé, on peut le récupérer simplement en utilisant la fonction nextMessage non bloquante ou waitNextMessage qui elle, arrête l'exécution du programme. Une fois ce message reçu, l'agent peut le traiter, et ainsi agir en conséquence.

Sachant qu'il existe d'autres méthodes plus complexes pour recevoir et envoyer des messages de façons différentes, on peut alors avoir des agents entièrement autonomes qui agissent en fonction des messages qu'ils ont reçus, et qui communiquent perpétuellement avec d'autres agents.

3.2 INTERFACE MOTEUR

Le plus difficile dans la réalisation du projet a été de choisir comment coder les personnages. En effet, représenter les personnages par des agents nous a semblé parfaitement logique, le tout était de le programmer correctement.

Les personnages sont représentés chacun par une classe* qui leur est propre, par exemple Mario possède une classe qui porte son nom, il en est de même pour Luigi, Bowser et tous les autres. Ces classes héritent* de la classe Personnage qui elle-même hérite de la classe Agent. Chaque personnage est donc représenté par un Agent appartenant au modèle ("communauté", "jeu") et a deux rôles : un avec son équipe, un autre avec son nom de personnage ; de façon à ce qu'il puisse interagir avec les bons agents par la suite. Par exemple, Mario a deux adresses ("communauté", "jeu", "gentil") et ("communaute", "jeu", "Mario"). De plus, les personnages possèdent un attribut « ennemi » dans lequel est enregistré le nom de son rival.

La méthode activate() sert ici à créer le groupe et à donner les rôles aux agents.

La méthode live() sert de guide à l'agent durant sa vie. Ici, nous lui demandons juste de chercher un adversaire, de se battre contre lui, jusqu'à ce que l'un des deux adversaires n'ait plus de point de vie et de recommencer (si l'agent est toujours en vie). Ces deux phases sont représentées par les méthodes chercher() et combattre().

```
public void live() {
	while(this.estEnVie()) { //tant que l'agent est en vie
		chercher();
		combattre();
	}
}
```

Lors de la phase de recherche, l'agent cherche un adversaire de l'équipe adverse à l'aide de getAgentWithRole(communaute, groupe, role). Afin de rendre le combat moins aléatoire, nous avons décidé de faire en sorte que chaque personnage ait un avantage sur un autre personnage. La difficulté de cette partie a donc été de trouver le moyen pour que l'agent recherche son Némésis en premier, et s'il n'en trouve pas de disponible, de rechercher tout simplement un personnage de l'équipe adverse. Nous avons commencé en pensant que chercher une fois sur deux le rival et un adversaire, mais cela n'a pas été fructueux du fait que cela était assez aléatoire.

Nous avons donc pensé à ce que l'agent recherche son ennemi numéro un pendant un certain laps de temps. Après plusieurs essais, la meilleure solution trouvée a été de capturer le temps avant l'entrée dans la boucle de recherche et de chercher le Némésis jusqu'à une certaine limite.

Avant d'entrer dans la boucle, on initialise la variable roleAChercher avec le rôle de son rival et l'on enregistre la date en millisecondes. Puis l'on test si 100 millisecondes se sont déroulées. Si ce n'est pas le cas, on ne change pas le rôle à chercher, sinon on modifie cette variable afin d'y mettre le rôle l'équipe adverse.

```
String roleAChercher = this.ennemi; //rôle de l'agent qu'il va chercher
long tempsDepartBoucle = System.currentTimeMillis(); //capture du temps

while(this.adversaire == null) {
	/*
	 * Ici, pendant les 100 premières millisecondes,
	 * on cherche this.ennemi.
	 * Et après, on cherche n'importe quel adversaire de
	  * l'équipe adverse.
	 */
	if((System.currentTimeMillis() - tempsDepartBoucle) > 100) {
		/*
		 * S'il fait partie des gentils,
		 * il cherche un adversaire de l'équipe "méchante"
		 * Et inversement
		 */
		if (this.equipe.equals("gentil"))
			roleAChercher = "mechant";
		else if (this.equipe.equals("mechant"))
			roleAChercher = "gentil";
		}
```

La recherche d'un adversaire se passe par l'utilisation de la méthode getAgentWhithRole("communaute", "jeu", roleAChercher);

Une fois la phase de recherche terminée, la méthode combattre() prend la main. La classe Message sert aux échanges entre les agents. Elle contient les dégâts et les positions des personnages afin qu'ils puissent se retrouver sur la map.

Au début de la méthode, on commence par traiter le premier "coup", ou message, envoyé à partir de la méthode chercher().

```
else {//this.coup != null //si il a reçu un coup
      //On retire la vie
      this.vie -= this.coup.getContent();
      if (!this.estEnVie()) { //si l'agent n'a plus de vie
            return; //met fin à la méthode combattre()
      }
}
```

Si on n'a pas de message, c'est à dire message == null, alors, on envoie un message, un coup, à l'adversaire, si ce dernier ne répond pas, ça veut dire qu'il est déjà engagé dans un autre combat, on stoppe donc immédiatement la méthode avec return ce qui renvoie la méthode dans le live, et donc en phase de recherche.

Si l'on reçoit une réponse, c'est que le combat peut commencer réellement : l'agent soustrait les dégâts qu'il vient de recevoir et entre dans une boucle qui contient l'échange de messages, c'est-à-dire de dégâts, et l'envoi de ses coordonnées sur la map d'un personnage à un autre. Cette boucle est vraie tant que son ennemi est en vie.

L'échange de message se déroule en deux phases : premièrement il regarde sa "boîte aux lettres" à l'aide de la méthode this.nextMessage(1000).

```
this.coup = (Message) this.waitNextMessage(1000); //regarde sa
"boite aux lettres"
```

Si la "boîte aux lettres" est vide, (this.coup == null) alors il envoie un coup à son adversaire et attend une réponse, un coup.

```
this.coup =
(Message) sendMessageAndWaitForReply(this.adversaire,
this.attaque, 1300);
```

Si ce dernier répond, alors il encaisse les dégâts. Si la boîte aux lettres n'est pas vide, l'agent reçoit les dégâts et recommence jusqu'à ce que lui ou son adversaire ne soit plus de vie.

A chaque fois que l'agent reçoit des dégâts, on lui affecte les coordonnées de son adversaire afin qu'il puisse le rejoindre graphiquement parlant.

```
this.perso.setxE(this.coup.getX());
this.perso.setyE(this.coup.getY());
```

Un test est aussi effectué à chaque envoie de message afin de savoir s'il reste de la vie à l'agent. En cas contraire, un return termine la méthode, l'arrêt de celle-ci renvoie dans la méthode live(). La boucle incluse dans cette dernière n'étant plus vrai, la méthode s'arrête et laisse place à la méthode end() qui se charge des dernières tâches que l'agent doit accomplir en fin de vie, dans notre cas cela sera juste la mort de l'agent.

3.3 INTERFACE GRAPHIQUE

Un jeu ayant beaucoup plus d'intérêt lorsqu'il n'est pas constitué de simples lignes de texte dans une console, une partie non négligeable de la réalisation du projet est la réalisation graphique. Celle-ci a été réalisée en deux parties principales.

Les premiers écrans que l'utilisateur va apercevoir ont été créés grâce à la librairie SWING intégrée à Java, afin d'avoir une interface graphique simple à mettre en place et indépendante du système d'exploitation utilisé.

La base du programme est simple : on utilise une classe Fenêtre, dérivée de JFrame, pour représenter tout le programme. Cette classe fille possède différents états : choix d'équipe, création d'équipe … Elle possède également beaucoup d'écouteurs pour les interactions avec l'utilisateur. Chaque transition vers un de ces états va vider la fenêtre, puis la remplir avec les composants nécessaires. Cette fenêtre est créée et dirigée par le programme principal.

De plus, il faut savoir que les états de création de l'équipe et de combat utilisent en fait d'autres classes dérivées.

En effet il existe une classe CreationEquipe qui hérite de JPanel, et qui contient tous les composants et leurs interactions nécessaires à la création et à la gestion d'une équipe. Une instance de cette classe est donc utilisée lors de l'état de création de l'équipe dans la Fenêtre. Elle contient alors une liste déroulante pour le choix des personnages, des boutons pour ajouter ou supprimer un personnage à l'équipe, une liste affichant l'équipe actuelle …

Voici un aperçu de la fenêtre lors de la création d'une équipe et le code correspondant au panneau :

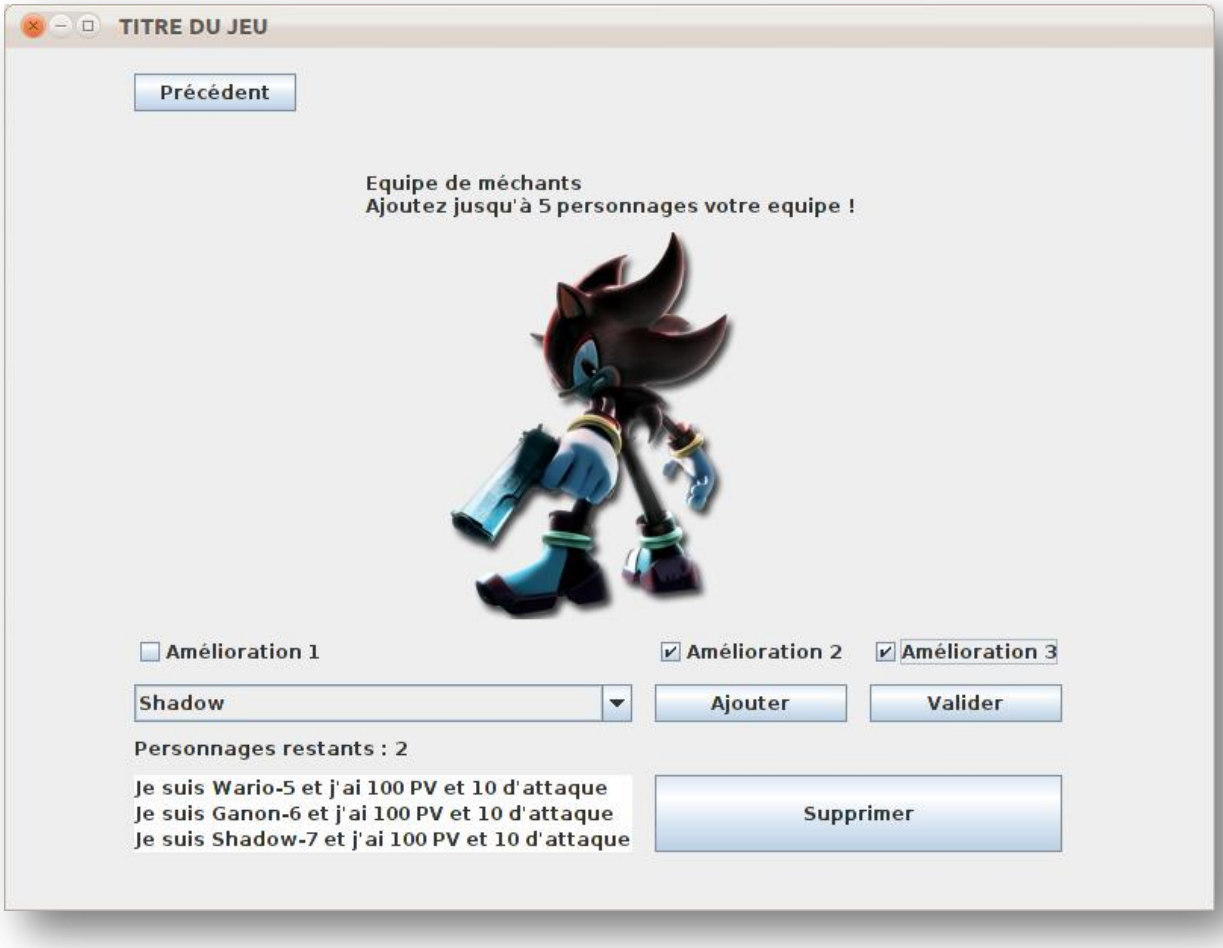

Figure 5 – Screenshot de l'écran création d'équipe

```
package com.graphique;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.ListSelectionModel;
import javax.swing.SwingConstants;

import com.madkit.Joueur;

public class PanneauEquipe extends JPanel
{
	//Panneau ajoute a la fenetre qui permet la crÃ©ation d'une
equipe
```

```
      //(les Ã©couteurs de certains boutons sont dans la classe
Fenetre)

      private Joueur j;

      private ListeEquipe lg;

      private LabelFonds l;

      private JOptionPane jop;

      private JButton ajouter = new JButton("Ajouter");
      private JButton valider = new JButton("Valider");
      private JButton supprimer = new JButton("Supprimer");

    private JCheckBox amelioration1 = new
JCheckBox("AmÃ©lioration 1");
    private JCheckBox amelioration2 = new
JCheckBox("AmÃ©lioration 2");
    private JCheckBox amelioration3 = new
JCheckBox("AmÃ©lioration 3");
    private boolean un = false;
    private boolean deux = false;
    private boolean trois = false;

    private JList equipe = new JList();

    private ActionListener ameliorationL;

    private JLabel image;
    private ActionListener lgL;

      private ActionListener ajouterL;
      private ActionListener validerL;
      private ActionListener supprimerL;

      public PanneauEquipe(Joueur j, ActionListener ecoute)
      {
            //Creation d'une contrainte pour l'utilisation du
layout afin de placer les elements
            GridBagConstraints c = new GridBagConstraints();
            this.setLayout(new GridBagLayout());
            c.fill = GridBagConstraints.BOTH;
            c.insets.bottom = 10;
            c.insets.right = 15;

            this.j = j;
            this.lg = new ListeEquipe(this.j);
            this.valider.setEnabled(false);

            //Affichage de l'image du personnage selectionne
            this.image = new JLabel(new
ImageIcon("data/listePersonnages/" + lg.getItemAt(0).toString()
+ ".png"), SwingConstants.CENTER);


            //Ajout des composant Ã  fenetre

            c.gridwidth = 3;
            c.gridy = 0;
            this.add(this.image, c);
```

```
            c.gridwidth = 1;
            c.gridy = 1;
            this.add(this.amelioration1, c);
            this.add(this.amelioration2, c);
            this.add(this.amelioration3, c);

            this.ameliorationL = new casesL();
            this.amelioration1.addActionListener(ameliorationL);
            this.amelioration2.addActionListener(ameliorationL);
            this.amelioration3.addActionListener(ameliorationL);

            c.gridy = 2;
            this.add(lg, c);
            this.add(ajouter, c);
            this.add(valider, c);

            this.lgL = new lgL();
            this.lg.addActionListener(lgL);

            this.ajouterL = new ajouterL();
            this.ajouter.addActionListener(this.ajouterL);

            this.valider.addActionListener(ecoute);

            this.l = new LabelFonds(this.j);
            c.gridy = 4;
            this.add(l, c);

            //Un seul element est selectionnable dans la liste

      equipe.setSelectionMode(ListSelectionModel.SINGLE_SELECTION
);
            c.gridy = 5;
            this.add(this.equipe, c);

            c.gridwidth = 2;
            this.add(this.supprimer, c);
            this.supprimer.hide();
            this.supprimerL = new supprimerL();
            this.supprimer.addActionListener(this.supprimerL);

            jop = new JOptionPane(); //Creation de la boite de
dialogue
      }

      class casesL implements ActionListener
      {
            //Evenement des amelioration (non utilise dans la
version finale)
            public void actionPerformed(ActionEvent event)
            {

      if(((JCheckBox)event.getSource()).getText().equals("AmÃ©lio
ration 1"))
                  {
                        un = !un;
                  }
                  else
if(((JCheckBox)event.getSource()).getText().equals("AmÃ©lioratio
n 2"))
```

```
                  {
                        deux = !deux;
                  }
                  else
if(((JCheckBox)event.getSource()).getText().equals("AmÃ©lioratio
n 3"))
                  {
                        trois = !trois;
                  }
            }
      }


      class ajouterL implements ActionListener
      {
            public void actionPerformed(ActionEvent event)
            {
                  //Mecanisme d'ajout a l'equipe, si possible ou
non
                  if(j.getFonds() >= 1)
                  {

      j.ajouterALEquipe(lg.getSelectedItem().toString(), un,
deux, trois);
                        j.modifierFonds(-1);
                        l.majLabel();

                        //remise Ã  0 des cases quand on ajoute
Ã  l'Ã©quipe
                        un = false; deux = false; trois = false;
                        amelioration1.setSelected(false);
amelioration2.setSelected(false);
amelioration3.setSelected(false);

                        equipe.setListData(j.getEquipe());

                        supprimer.setVisible(true);
                        valider.setEnabled(true);
                        if(j.getFonds() < 1)
                              ajouter.setEnabled(false);
                  }
                  else
                  {
                        jop.showMessageDialog(null, "Vous n'avez
pas assez de fonds !", "Attention",
JOptionPane.INFORMATION_MESSAGE);
                  }
            }
      }

      class supprimerL implements ActionListener
      {
            //Suppression d'un personnage de la liste
            public void actionPerformed(ActionEvent event)
            {
                  j.supprimer(equipe.getSelectedIndex());
                  j.modifierFonds(1);
                  l.majLabel();
                  equipe.setListData(j.getEquipe());
                  if(j.getEquipe().isEmpty())
                  {
```

```
                        valider.setEnabled(false);
                        supprimer.hide();
                  }
                  ajouter.setEnabled(true);
                  repaint();
            }
      }

      class lgL implements ActionListener
      {
            //Mise a jour de l'image en fonction du personnage
selectionne
            public void actionPerformed(ActionEvent event)
            {
                  image.setIcon(new
ImageIcon("data/listePersonnages/" +
lg.getSelectedItem().toString() + ".png"));
            }
      }

}
```

Une fois cet écran passé et le bouton “valider” enclenché (celui-ci n'apparaît que si la liste de personnages n’est pas vide), on arrive dans le vif du sujet : le combat.

Pour réaliser le combat, nous avons utilisé une autre librairie Java spécialisée dans le jeu : Slick. En effet, celle-ci possède beaucoup de classes et de méthodes simplifiant grandement la réalisation d’un jeu vidéo. Nous avons donc utilisé une librairie externe supplémentaire, dans le but de faciliter notre développement, pour éviter de “réinventer la roue”, ceci n’étant pas souhaitable dans un projet de cette envergure. Nous avons choisi Slick pour sa simplicité d’utilisation et sa rapidité d'exécution.

Le combat utilise donc une classe provenant de Slick : CanvasGameContainer qui est en fait un simple JPanel contenant un “Game”, et donc un jeu. Il faut savoir que nous avons ici utilisé la classe StateBasedGame (qui hérite de Game) pour réaliser le combat, ce qui permet d’avoir plusieurs état dans notre jeu, et passer de l’un à l’autre rapidement et simplement.

Notre projet en possède deux : le combat en lui-même, et un état de pause. On passe de l’un à l’autre très simplement grâce à une pression sur la touche “echap”.

Mais revenons à notre jeu. Celui-ci se base sur l’architecture classique d’un jeu créé avec Slick ; il possède trois méthodes principales : init, update et render.

Tout d’abord côté init n’est appelé qu’une seule fois lors de la création du jeu. C’est dans cette méthode qu’on va préparer la carte qui servira de décor au combat, et qu’on va commencer à initialiser et à placer la représentation graphique des Personnages. En effet, celles-ci utilisent une classe PersonnageAnime qui hérite de la classe SpriteSheet de Slick qui permet simplement de représenter et d’animer une feuille de sprite*. On a ici deux listes qui contiennent respectivement la liste des PersonnagesAnime (on utilise ici des Vectors) de chaque joueur, et qui sont créés à partir des deux objets de type Joueur qu’on a récupérés

lors de la création du jeu. De plus, on va placer tous ces personnages aux bons endroits afin que les deux équipes se fassent face et soient prêtes à s'affronter.

Les deux autres méthodes update et render vont de paire : elles sont appelées l'une après l'autre à chaque tour de boucle, et ce sont celles-ci qui vont permettre une bonne représentation graphique du jeu.

Dans update se feront tous les calculs nécessaires au programme : déplacement et animation des personnages en fonction des adversaires, vérification de la victoire … C'est également ici qu'on gère les touches du clavier, ce qui est plutôt simple puisqu'on se contentera de la touche "échap" qui permet de basculer de l'état de combat à l'état de pause, et vice versa. La partie la plus importante du update va consister dans deux boucles : chacune d'elle va parcourir l'ensemble des listes de PersonnageAnime des deux joueurs, et appeller la méthode déplacement sur ceux-ci. Cette méthode très importante permet au PersonnageAnime de se déplacer dans la direction de son adversaire, et s'il y est déjà, d'y rester. De plus, lors de chaque déplacement, on va mettre à jour différentes variables nécessaires à l'animation du personnage.

De son côté, render va se contenter d'afficher tous les éléments nécessaires au jeu en utilisant les informations modifiées par update. On va donc d'abord commencer par afficher la map en fond (grâce à la classe TiledMap de Slick), puis par parcourir une nouvelle fois les deux listes de PersonnageAnime et par appeler à chaque fois la nouvelle méthode render. En effet, la redéfinition de la méthode render de la classe Spritesheet permet d'afficher un sprite x,y de la feuille de sprite à la position x,y de l'écran.

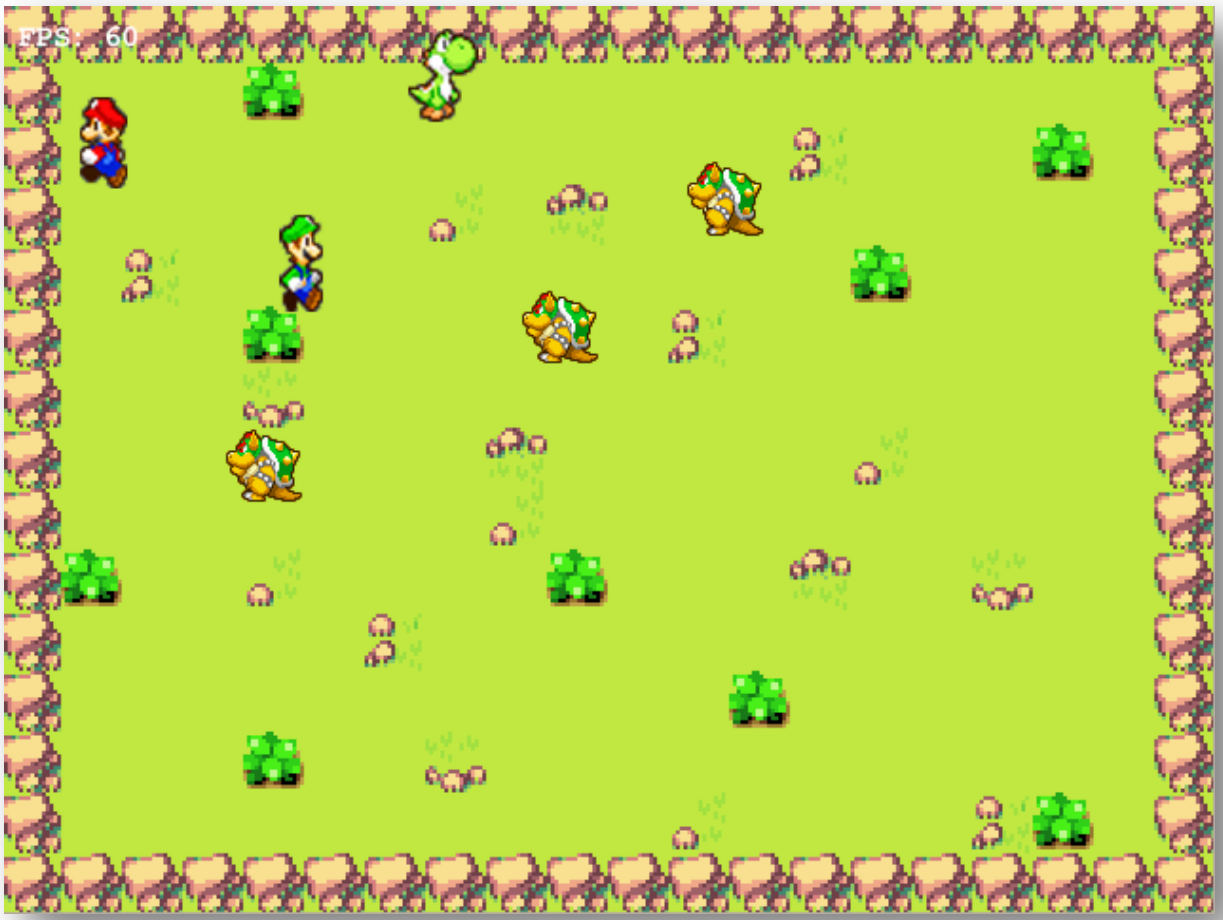

Figure 6 – Screenshot de l'écran jeu

```
package com.jeu;

import java.util.Vector;

import org.newdawn.slick.GameContainer;
import org.newdawn.slick.Graphics;
import org.newdawn.slick.Input;
import org.newdawn.slick.SlickException;
import org.newdawn.slick.state.BasicGameState;
import org.newdawn.slick.state.StateBasedGame;
import org.newdawn.slick.tiled.TiledMap;

import com.madkit.Joueur;
import com.madkit.Personnage;

public class Combat extends BasicGameState
{
	//Creation d'une state de jeu : le combat
	public static final int ID = 1;

	TiledMap map;

	private Joueur j1;
	private Vector<PersonnageAnime> p1;
```

```
	private Joueur j2;
	private Vector<PersonnageAnime> p2;

	public Combat(Joueur j1, Joueur j2)
	{
		//Affectation des joueurs depuis la fenetre
		this.j1 = j1;
		this.j2 = j2;

		this.p1 = new Vector<PersonnageAnime>();
		this.p2 = new Vector<PersonnageAnime>();
	}

	@Override
	public void init(GameContainer arg0, StateBasedGame arg1)
throws SlickException
	{
		//Initialisation du combat

		//Ajout de la carte
		this.map = new TiledMap("data/maps/zelda1.tmx");

		//Creation des personnagesAnime
		for(int i = 0; i < this.j1.getEquipe().size(); i++)
		{
			((Personnage)
this.j1.getEquipe().get(i)).creerAnim();
			this.p1.add(((Personnage)
this.j1.getEquipe().get(i)).getPersoAnim());
			this.p1.get(i).setX(10);
			this.p1.get(i).setY(i*100);
			this.p1.get(i).setySprite(1);
		}
		for(int i = 0; i < this.j2.getEquipe().size(); i++)
		{
			((Personnage)
this.j2.getEquipe().get(i)).creerAnim();
			this.p2.add(((Personnage)
this.j2.getEquipe().get(i)).getPersoAnim());
			this.p2.get(i).setX(400);
			this.p2.get(i).setY(i*100);
			this.p2.get(i).setySprite(0);
		}

		//Lancement des agents
		this.j1.activer();
		this.j2.activer();
	}

	@Override
	public void update(GameContainer arg0, StateBasedGame arg1,
int arg2) throws SlickException
	{
		Input entree;
		entree = arg0.getInput();

		//Mise a jour des coordonnees des personnages de
chaque equipe par la methode deplacement()
		for(int i = 0; i < this.p1.size(); i++)
		{
```

```
                  if(((Personnage)
this.j1.getEquipe().get(i)).getEngage())
                        this.p1.get(i).deplacement();
            }
            for(int i = 0; i < this.p2.size(); i++)
            {
                  if(((Personnage)
this.j2.getEquipe().get(i)).getEngage())
                        this.p2.get(i).deplacement();
            }

            //Passage Ã  l'Ã©tat pause si la touche echap est
pressee
            if (entree.isKeyDown(Input.KEY_ESCAPE))
                  arg1.enterState(2);
      }

      @Override
      public void render(GameContainer arg0, StateBasedGame arg1,
Graphics arg2) throws SlickException
      {
            //Rendu du jeu

            //Rendu de la carte
            this.map.render(1, 1);

            //Rendu des deux equipes de personnages
            for(int i = 0; i < this.p1.size(); i++)
            {
                  this.p1.get(i).render();
            }
            for(int i = 0; i < this.p2.size(); i++)
            {
                  this.p2.get(i).render();
            }
      }

      @Override
      public int getID()
      {
            return ID;
      }

}
```

On a maintenant un combat animé avec un agréable visuel.

4 RESULTATS

4.1 LE JEU

Au terme de ces mois de travail, nous sommes en mesure de présenter le fruit de notre labeur : un prototype et une ébauche déjà bien avancés d'un jeu vidéo, plus précisément un jeu de stratégie.

En effet, notre jeu est fonctionnel, et possède toutes les bases nécessaires à la création d'un jeu plus complet.
Le jeu permet d'ores et déjà à deux joueurs de s'affronter lors d'un combat, ou à un joueur seul d'affronter l'ordinateur. Celui-ci verra son équipe de personnages créée aléatoirement, alors que les joueurs humains pourront, par le biais de différents menus, choisir leurs personnages après avoir choisi une équipe à incarner.

Nous proposons donc déjà une interface complète qui présente toutes les fonctionnalités nécessaires à la bonne configuration du jeu.

Parlons maintenant de la fonctionnalité principale de notre logiciel : le combat. D'un point de vue programmation multi-agents, celui-ci est complet. En effet, les agents se cherchent, se trouvent en fonction de leur rôle et s'affrontent dans des combats singuliers, et ainsi de suite jusqu'à ce qu'une seule équipe subsiste et remporte la victoire.

Du point de vue de la réalisation graphique, c'est un peu différent. Pour la carte servant de fond et la présence "physique" des personnages et de leur animation, tout cela marche parfaitement de manière fluide et agréable, on a donc une vue qui présente le combat de manière classique. Même si de petits soucis subsistent au niveau de certains déplacements, dus au comportement autonome et partiellement aléatoire des Agents, le combat reste compréhensible et intéressant à regarder.

Pour conclure, nous pouvons dire que nous avons acquis des connaissances dans différents domaines, en particulier au niveau de la programmation. Dans un premier temps, nous avons appris à organiser un projet contenant beaucoup de ressources, de fichiers différents qu'il était important de bien ranger pour ne pas se perdre, en essayant d'avoir au maximum un code clair et lisible. Ensuite, ce projet nous a permis bien évidemment de découvrir la programmation multi-agents et d'en d'appréhender les bases dans le cadre de la programmation d'une simulation.

Enfin, ce projet nous a donné l'opportunité d'aller plus loin dans la programmation graphique, et ce dans deux domaines : d'abord savoir créer des interfaces utilisateurs plus complexes, composées de différents états, puis en programmation 2D pure, avec un affichage et une gestion des images comme on en retrouve dans tous les jeux vidéos.

4.2 LES PROBLEMES

Bien évidement, comme au cours de n'importe quel projet, nous avons rencontré des problèmes lors de la réalisation de notre jeu. Certains ont été rapidement résolus, d'autres le furent moins.

Il faut savoir que nous avons été rapidement contraints d'abandonner une des fonctionnalités prévue par notre GDD*. En effet, le système d'améliorations prévu était trop difficile à mettre en place, étant donné l'organisation de notre programme. Avoir une classe par type de Personnage rendait cela trop laborieux, nous avons donc laissé cette option en attente, pour la production post-rendu par exemple.

La première difficulté que nous avons rencontrée a été l'appréhension de la programmation multi-agents. En effet celle-ci bouleverse totalement les façons traditionnelles de programmer, il faut alors penser son code d'une toute autre manière, en prenant en compte les agents au centre de celle-ci. Un temps d'adaptation et de réflexion a donc été nécessaire afin de pouvoir comprendre et exploiter pleinement les agents en essayant de respecter les principes des systèmes multi-agents.

De plus, il va sans dire qu'il a été difficile d'obtenir le comportement voulu pour les agents de notre jeu, cela a demandé beaucoup de temps. Le problème principal que l'on a eu côté programmation des agents a été de faire en sorte qu'ils ne combattent pas à un contre plusieurs, en effet lors des premiers essais, les agents se retrouvaient dans des combats à un contre trois et plus à cause du fait que nous n'arrivions pas à gérer l'échange de messages et l'enregistrement de l'adresse de l'adversaire.

Ce qui a une influence directe sur l'apparence graphique du jeu : si les agents en "interne" ne fonctionnent pas correctement, cela se ressent directement sur l'aspect externe (celui que l'utilisateur aura). Il arrive donc parfois que certains combats ne soient pas totalement cohérents, avec des déplacements illogiques et peu agréables à l'œil. Mis à par ceci qui est directement lié aux agents, nous n'avons pas rencontré de gros problèmes d'un point de vue conception graphique, celle-ci étant simple mais efficace.

Bien que nous ayons rencontré plusieurs problèmes au cours de la réalisation de ce projet, leur résolution ou non nous a indéniablement apporté des connaissances supplémentaires, tant en programmation qu'en gestion d'un projet à plusieurs.

5 DISCUSSION

Portons à présent un regard critique sur ce projet, et l'ensemble du travail qui l'a composé.

Dans un premier temps, il faut savoir que malgré un planning bien organisé, il a été difficile de le suivre de près. En effet, une partie importante de la conception nous a posé problème, lors de la communication entre les agents. De plus, la tentative d'introduction à la sérialisation qui nous aurait permis d'obtenir un code plus facilement exploitable et plus clair a échoué, ce qui nous a encore légèrement retardé sur la suite du projet. Même si cela n'a pas influé au final, le développement avec Slick ayant été moins long que prévu (le choix de cette librairie pour la partie graphique a donc été pertinent).

Du point de vue de l'organisation du travail, travailler à deux sur le même projet ne nous a pas posé de problème. En effet, nous nous étions équitablement réparti le travail à faire, et nous le mettions régulièrement en commun afin de garder un déroulement constant dans la réalisation. Il a donc ensuite été facile de faire de même pour la rédaction du rapport.

Par rapport à notre objectif qui était de créer un jeu jouable et intéressant, nous pouvons maintenant dire où nous en sommes. Et nous pouvons affirmer que celui-ci est en grande partie atteint. En effet, malgré quelques soucis anecdotiques, le jeu est jouable et possède toutes les bases d'un jeu vidéo tout ce qu'il y a de plus normal, la programmation multi-agents en plus.

De surcroît, celui-ci est facilement améliorable sur différents points : on peut y ajouter du contenu (nouveaux personnages, lieux), autant de modifications plus profondes dans le gameplay qui feraient appel à nos connaissances nouvelles en termes de programmation.

Pour conclure, il est important de noter que ce projet nous a apporté des connaissances dans le domaine de la programmation multi-agents, qui jusqu'alors nous était inconnu. L'objectif d'apprentissage d'une nouvelle façon de voir les choses et de programmer est donc totalement atteint.

6 CONCLUSION

En conclusion de ces mois de travail, si nous reprenons nos objectifs initiaux, nous avons pu voir que ceux-ci ont été réalisés. L'apprentissage de la programmation multi-agents s'est bien déroulé, et nous avons un logiciel présentable à la fin de ce projet, un jeu vidéo pour être précis.

Celui-ci est bien évidemment améliorable de différentes façons, ce qui pourrait constituer une suite au projet afin d'en faire une réalisation sur la durée, en y ajoutant régulièrement des nouveautés. Mais avec nos connaissances actuelles, nous pourrions tout aussi bien partir sur un projet totalement différent qui nécessiterait une simulation quelconque que nous réaliserions à l'aide de MadKit.

Le projet nous a donc apporté beaucoup dans différents domaines, et surtout de façon bénéfique. En effet, nous avons maintenant plus de connaissances en programmation, mais nous sommes également capables de réaliser un projet à plusieurs dans un laps de temps important.

7 BIBLIOGRAPHIE ET SITOGRAPHIE

Sites Internet :

http://www.madkit.org/ pour Madkit et sa documentation

http://slick.cokeandcode.com/ pour la librairie Slick, sa documentation et son forum

http://fr.wikipedia.org/ pour ses différents articles, en particulier sur les systèmes multi-agents

http://www.developpez.com/ pour ses nombreux tutoriaux de programmation Java

http://www.oracle.com/ pour la documentation du langage Java

http://spriters-resource.com/ pour les ressources graphiques utilisées dans le jeu

RESUME

Voici le rapport réalisé en binôme dans le cadre de notre projet de seconde année à l'IUT Informatique de Montpellier portant sur la programmation multi-agents.

Au cours de ces quelques mois de travail nous avons donc réalisé successivement une analyse et un Game Design Document servant de cahier des charges, puis nous nous sommes attaqués à la programmation en Java d'un système multi-agents à l'aide de la bibliothèque Madkit et de sa représentation graphique sous forme d'un jeu vidéo de stratégie.

Ce rapport présente donc notre travail et son déroulement, et les résultats critiqués et analysés obtenus une fois la réalisation du projet terminée.

Mots clés : MadKit, Programmation Multi-Agents, Java, Jeux vidéo

SUMMARY

Here is the project report written in pairs for our project realised during our second year in the Computer Science section in the IUT of Montpellier, about multi-agent systems programming.

During these few months of work we successively did an analysis and a Game Design Document used as specifications, then we worked on concrete programming in Java, doing a multi-agent system using MadKit library and its graphical representation through a strategy video game.

This report introduces our work and its development, and the results, criticized and analyzed, obtained after the achievement of the project.

Keywords: MadKit, multi-agent programming, Java, Video Games