Projet intégré Macaks

MAgic Card gAme networKed Simulator

Livrable final

1 Description **2**

1.1 Avant-propos : Macaks, oui mais encore ? 2

1.2 Macaks : quoi et comment ? 2

1.3 Macaks : une rude concurrence 3

1.4 Réponse à la concurrence 5

2 Organisation du projet **8**

2.1 Module Magic 8

2.2 Module Interface 11

2.3 Module Réseau 14

2.4 Modules Noyau & Base de données 22

2.5 Module Intelligence Artificielle 25

3 Conclusions **29**

3.1 Quelques problèmes techniques 29

3.2 Une vision globale 29

3.3 Des conflits 30

3.4 Des objectifs atteints 31

3.5 Quelques chiffres pour finir. 32

1 Description

1.1 Avant-propos : Macaks, oui mais encore ?

Magic: the gathering est un jeu de cartes à collectionner. Pour la petite histoire, il a été créé voilà plus de 10 ans par la société *Wizards of The Coast* [1] et suscite encore de nos jours un engouement auprès de joueurs de tous âges. Pour donner une petite idée de l'ampleur de son succès, en mai 2005 plus de six millions de joueurs s'adonnaient à ce jeu, actuellement traduit en neuf langues différentes[2].

Chaque année sortent différentes éditions contenant chacune plus d'une centaine de cartes, ce qui nous donne au total pas moins de 8 000 cartes différentes !

Le jeu peut paraître très simple au premier abord : la plupart du temps, c'est un affrontement entre deux joueurs qui incarnent chacun un magicien. Le but est d'exterminer le magicien adverse à l'aide de cartes permettant de lancer divers sorts.

Chaque joueur a devant lui des cartes face visible qui sont des *permanents*, un tas de cartes face visible qui est son *cimetière*, un tas de cartes face cachée qui est sa *bibliothèque* ainsi que des cartes en main, connues de lui seul.

Les sorts joués par l'un et l'autre des joueurs peuvent s'attaquer à chacune de ces différentes parties du jeu, et l'on peut très bien imaginer qu'une carte du joueur A *détruise* une carte en jeu devant le joueur B, celui-ci devant alors la mettre dans son cimetière.

La condition de victoire la plus fréquente est d'abaisser le total de points de vie du joueur adverse – initialement de 20 – à 0 : pour cela on peut attaquer avec des créatures ou jouer d'autres cartes.

Les actions disponibles sont toujours sujettes à d'autres actions faites en réponse par l'un ou l'autre des joueurs : on a donc un système très simple de pile dans laquelle on *entasse* les différentes actions effectuées par les joueurs jusqu'à ce que chacun ait *cédé la priorité* (c'est-à-dire qu'il ne souhaite rien jouer dans la pile), et on dépile alors les actions des joueurs.

On voit donc que les interactions entre cartes peuvent être multiples et c'est sans compter que chaque carte peut modifier à sa guise les règles du jeu si cela est indiqué. C'est d'ailleurs ce que tout joueur de Magic appelle la règle d'or : *la carte a toujours raison.*

1.2 Macaks : quoi et comment ?

Nous avons réalisé, dans le cadre du module Projet Intégré encadré par Eddy Caron, un simulateur du jeu de cartes *Magic: the gathering*. Ce simulateur permet de jouer à plusieurs en réseau ou contre l'ordinateur, et vérifie à tout moment que les actions effectuées sont respectueuses des règles.

L'interface se veut simple, jolie et efficace, et permet la communication entre joueurs. Une réflexion sur l'interaction entre les règles et l'interface nous a conduit à automatiser le plus de choix possibles pour ne pas surcharger l'utilisateur de questions

[1] Aussi appelée couramment WoTC

[2] Source : http ://www.wizards.com/default.asp ?x=company/pr/20050509a

à tout moment – en effet lors d'une partie avec de vraies cartes, de nombreux choix sont tacites.

Avant de lancer une partie proprement dit, il s'agit de créer des serveurs, et de donner différents droits aux personnes qui y sont connectées : qui va jouer dans la partie, qui sera simple spectateur par exemple.

En plus de cela, on permet bien sûr à l'utilisateur de créer ses propres *decks* (c'est l'état initial de la bibliothèque du joueur, dans laquelle il pioche à chaque tour une carte). Quelques commandes intuitives permettent ainsi à l'utilisateur de créer facilement le deck avec lequel il va jouer.

Le simulateur utilise une base de données des cartes connues. Il n'a pas été raisonnablement possible de coder les milliers de cartes existant dans le jeu, mais le projet est extensible ; c'est-à-dire qu'il est possible d'ajouter assez facilement de nouvelles cartes à la base de données.

Pour être précis, la base de données connaît en fait toutes les cartes existantes, mais n'est pas capable de les analyser en termes de règles. Ce n'est pas en sachant que sur la carte il y a écrit « You win the game » que l'ordinateur sait qu'il doit faire gagner le joueur : il faut que cette ligne de texte soit transcrite en code. Cette transcription a été faite pour plusieurs centaines d'entre elles, permettant ainsi de nombreuses parties différentes.

Enfin, lorsqu'il le désirera, et pour un nombre un peu plus restreint de cartes, un joueur pourra tester son deck contre l'ordinateur, en fournissant à ce dernier un deck particulier à jouer. L'utilisateur lambda pourra en tirer des conclusions (finalement, telle ou telle carte n'est pas aussi puissante qu'espéré, etc.) avant de se lancer dans des parties face à un joueur humain.

1.3 Macaks : une rude concurrence

Magic: the gathering, de par sa popularité, a été décliné en plusieurs logiciels. Citons les principaux :

- **Magic : the gathering (Microprose, 1996)**
  C'est le tout premier jeu sur ordinateur commercialisé permettant de jouer contre l'ordinateur à *Magic: the gathering*.

  Le pool de cartes disponible dans ce logiciel est très restreint : en effet, malgré deux extensions rajoutant chacune de nouvelles cartes, l'éditeur a très vite arrêté les mises à jour et aucune carte postérieure à 1995 n'y est disponible. En effet devant des droits d'auteur trop importants demandés à WoTC, ils ont décidé de stopper le développement du logiciel.

  Pour la même raison, les règles utilisées sont pour la plupart obsolètes. En effet, depuis plusieurs années, *Magic: the gathering* a été bien mieux formalisé pour être rendu plus simple d'utilisation.

  L'interface de jeu est sobre mais pratique : on retrouve très facilement tout ce que l'on veut, même s'il faut avouer que les graphismes – comprenons-le, ils datent un peu – laissent fortement à désirer.

  Niveau respect des règles, l'ordinateur contrôle tout ce qui est fait lors du déroulement d'un tour.

Une intelligence artificielle a été codée qui permet à l'ordinateur de bien s'en sortir face à nous, et ce pour n'importe quel deck qu'on lui met entre les mains. Néanmoins de temps à autre on voit les limites de l'intelligence artificielle qui veut trop souvent faire des petits gains en local au lieu de voir la menace globale de chaque carte jouée par l'humain contre lequel il joue.

Niveau réseau, rien n'est proposé avant le second CD-rom d'extension. Ce CD propose de jouer via le réseau local ou sur internet (leur serveur de jeu étant dorénavant fermé).

- **Apprentice (Dragonstudios, 1996)**
  C'est le premier jeu massivement utilisé sur Internet pour jouer à *Magic: the gathering* à deux.

  L'utilisation est simple et intuitive, malheureusement l'interface ne suit pas du tout : aucun graphisme n'est intégré et c'est vraiment très pénible de ne voir quasiment que du texte pour jouer.

  De plus, l'ordinateur se contente de laisser les joueurs s'affronter comme s'ils jouaient une partie dans la « vraie vie ». C'est-à-dire qu'il ne contrôle aucune règle, il donne juste l'occasion à deux joueurs de se rencontrer sur Internet, de manipuler des cartes librement sur un plan de jeu, avec un pool de cartes quant à lui constamment mis à jour, encore maintenant.

- **MWS (Magic Work Station, Magi-Soft Development, 1999)**
  C'est le digne successeur d'Apprentice et dans le projet nous sommes beaucoup à utiliser ce logiciel. Distribué gratuitement sur Internet, il est néanmoins en version shareware et l'on peut enlever publicités et débloquer des options en payant la modique somme de 20 euros.

  L'amélioration majeure face à son prédécesseur Apprentice réside dans les graphismes de l'interface : tout est simple d'utilisation et surtout très visuel. On peut avoir accès aux images des cartes, on a également accès à tout par simple survol au lieu de devoir cliquer comme sous Apprentice. Le logiciel est vraiment très convivial.

  Par contre, ici aussi l'ordinateur se contente de laisser les joueurs jouer comme s'ils jouaient une partie dans la « vraie vie ».

  Le pool de cartes, quant à lui, est également constamment mis à jour.

- **MTGO (Magic The Gathering Online, WoTC, 2004)**
  C'est tout simplement le logiciel officiel, distribué par l'éditeur du jeu de cartes. Il est a priori gratuit, cependant pour pouvoir jouer, il faut *posséder* les cartes avec lesquelles on veut jouer. Il faut donc, comme dans la vie réelle, payer pour recevoir des cartes qui ne serviront que pour jouer via le logiciel... (à ceci près que si l'on arrive à collectionner toutes les cartes d'une édition donnée, on peut demander à recevoir cette édition en vraies cartes chez soi).

  Niveau respect des règles, interface, jouabilité, tout y est. Le logiciel gère vraiment tout ce que le joueur pourrait faire dans la vie réelle... tout ? Non. Quelques petits détails sont encore non gérés par ce logiciel. Par exemple, dans la vraie vie, il arrive qu'un des deux joueurs souhaite répéter une suite d'actions un grand nombre de fois, pour en obtenir à chaque passe un bénéfice. Dans ce cas, il montre à son adversaire comment il effectue la boucle, et il lui demande son accord pour la lancer par exemple 1 000 fois. Le logiciel ne gère pas cela, et il faudrait faire 1 000

fois à la main la suite d'actions, ce qui est vraiment impossible étant donné que les parties sont limitées dans le temps lorsque l'on joue pour des tournois...

Le jeu est axé réseau et ne gère pas du tout d'intelligence artificielle. Mais la partie réseau est pour l'instant la meilleure de toutes, elle permet en effet à d'autres personnes de regarder une partie en cours sans être l'un ni l'autre des joueurs, en simple spectateur.

Enfin le pool de cartes disponibles est l'ensemble des cartes depuis 2001, ce qui fait tout de même un paquet (environ 4 000 cartes).

L'aspect payant pour au final pas grand chose (des cartes virtuelles...) est de même assez rebutant et c'est à notre avis l'un des seuls points faibles de cette distribution.

- **Scrollrack (2005, sous licence GPL)**
  L'un des atouts majeurs de cette distribution et que le logiciel est livré avec ses sources intégrales.

  C'est également le premier projet parmi les différents cités à pouvoir tourner sous Linux et Macintosh...

  Sinon il constitue un bon compromis entre Apprentice et MWS, au sens où le seul gros apport par rapport à Apprentice est l'ajout d'images pour reconnaître les différentes cartes sur table. Sinon le reste est également assez peu *user friendly* et beaucoup moins convivial que MWS.

- **GCCG (Generic Collectible Card Game, 2001, également sous licence GPL)**
  L'ajout majeur par rapport aux autres logiciels gratuits est que, tout comme MTGO, des spectateurs peuvent intégrer n'importe quelle partie.

  Et, seul logiciel à le faire, on peut ici faire des échanges et regarder qui vend/échange quelles cartes parmi les personnes connectées.

  L'interface est très intuitive et parfaitement personnalisable, mais en aucun cas l'ordinateur ne vérifie les règles.

- **Angelfire, Thiery Schmidt, ...**
  Certains sites personnels fournissent de petites applications Javascript ayant pour but de simuler une partie de Magic. On se rend vite compte de leurs limitations, aussi bien du point de vue de la rapidité et de la jouabilité, que des règles et de la quantité de cartes gérées.

1.4 Réponse à la concurrence

On l'a vu, notre pari est donc de réunir l'ensemble des points forts de chaque logiciel permettant de jouer à *Magic: the gathering* (mis à part le côté communautaire lié aux échanges/ventes). On remarque que la plupart des atouts présentés dans les logiciels ne se retrouvent pas dans d'autres. Ainsi seul un des logiciels fournit une intelligence artificielle, seuls deux d'entre eux vérifient les règles, même si bien sûr tous comportent des éditeurs de deck et des plateaux de jeu (heureusement !).

En partant de ce constat, deux voix s'offraient à nous : améliorer un projet existant, ou bien repartir de zéro et coder notre propre système. Nous avons choisi la seconde solution qui, bien qu'elle puisse sembler un peu rebutante, s'est avérée le bon choix :

- Licence : les logiciels GPL sont, on l'a vu, moins complets et plus compliqués à utiliser que leurs homologues sharewares. Il aurait donc été peu habile de partir d'un code certes GPL mais où il faudrait réécrire toute l'interface graphique, et dans lequel il serait très difficile d'inclure nos idées sur l'intelligence artificielle et la gestion des règles car rien n'aurait été initialement prévu dans ce sens. Nous aurions donc été obligés de partir d'un shareware. Or il est évident que nous n'allions pas reprendre le code (non disponible) de logiciels payants, ce d'autant que l'impératif du projet de produire un logiciel libre va à l'encontre de tout partenariat avec les créateurs de tels logiciels.
- Réseau : il est plus commode de repartir de fonctions de bas niveau plutôt que de s'encombrer de bibliothèques préexistantes mais ne faisant pas exactement ce que l'on veut, qu'il faudrait comprendre, utiliser, et modifier, opération qui prendrait plus de temps que de coder directement ce dont on a besoin.
- Base de données : on veut un logiciel portable et non propriétaire, il était donc impossible de reprendre les formats utilisés par les logiciels existants, tous propriétaires, et dont aucun n'est devenu un standard de fait. Nous avons choisi d'utiliser le format XML dont la portabilité n'est plus à démontrer. Étant les premiers à coder la base de données *Magic: the gathering* dans le format non propriétaire qu'est XML, nous devons coder tout ce qui s'y rapporte.

Aux vues des interactions possibles avec les cartes et des différentes choses qu'on peut faire avec, il était naturel d'utiliser un langage orienté objet. Voulant un logiciel portable, nous avons opté pour la solution Java, qui, bien qu'un peu lente, nous offre une bonne base niveau interface ainsi que l'assurance d'une très forte portabilité. De plus, le Java autorise le chargement de code à la volée (intégration dynamique de classes) : si l'on code une classe par carte, on pourra charger de nouvelles cartes dynamiquement.

Si le format utilisé pour la base de données est portable (XML), il est par contre incompatible avec les autres formats utilisés par les logiciels disponibles sur Internet – notons qu'il n'existe pas de standard... C'est pourquoi nous permettons à l'utilisateur, non seulement d'importer ses fichiers de decks à partir d'autres formats, mais aussi l'exportation vers ces formats [3].

La forte popularité de logiciels tels MWS et Apprentice peut à tort faire penser que la communauté des joueurs linuxiens ou macintoshiens est délaissée. Cependant d'autres logiciels fournissent des interfaces tout aussi belles, mais sont beaucoup moins connus (nous avons tous découvert GCCG et Scrollrack au cours de ce projet). Nous essaierons de faire connaître notre logiciel auprès de la communauté pour, justement, promouvoir l'utilisation d'Internet pour jouer à *Magic: the gathering*, même si l'on ne dispose pas d'un PC sous Windows.

La tâche n'est a priori pas simple du tout, voire impossible à réaliser en moins de 3 mois. L'organisation d'un tel projet amène assez naturellement – avec ce qui a été dit plus haut – un découpage en plusieurs rubriques, dont voici pour chacune les restrictions que nous nous sommes fixé pour être à même de tenir nos objectifs.

- **Interface**
  Sur ce côté, aucune restriction. Sans une bonne interface pas de plaisir réel à jouer et l'on n'a rien laissé de côté pour cette partie.

- **Réseau**
  De ce côté non plus on ne peut pas vraiment passer de points, sinon comment jouer à plusieurs...

[3] En particulier notre logiciel est compatible avec les formats du logiciel Magic Workstation

- **Base de données**
  C'est ici que se situe la principale limitation de notre logiciel. Il serait illusoire de croire qu'en l'espace de 3 mois nous aurions eu le temps de coder plus de 8 000 cartes. Ainsi l'on s'est limité à coder l'édition la plus récente ainsi que quelques cartes très utiles, pour un total d'environ 400 cartes. Il est à noter que, par contre, notre code est muni d'une multitudes de *fonctions helper* qui automatisent la gestion de la plupart des cartes non codées. Ainsi, pour coder de nombreuses cartes, il suffit d'appeler deux ou trois fonctions helper avec les nombres inscrits sur la carte.
- **Intelligence artificielle**
  L'un des seconds *points chauds* de notre logiciel. On donnera au joueur la possibilité d'avoir une assistance lui indiquant par exemple que, attention, s'il fait ce choix, alors il y a de fortes chances qu'il perde telle ou telle carte.

  Et pour que l'ordinateur joue tout seul, il y a certaines cartes qui, bien que codées, sont difficile à analyser en terme de menace.

Néanmoins, notre logiciel se veut complet, et apporte des petites améliorations de ci, de là, que nous sommes les seuls à fournir – pour l'instant. Nous pouvons citer :

- **Boucles.** Macaks est à ce jour le seul simulateur à permettre d'enregistrer puis d'exécuter des boucles, permettant enfin au fan des decks *combo* (fondés sur une combinaison de cartes permettant de gagner en réalisant une boucle) d'utiliser un logiciel pour jouer. Le joueur trouve une suite d'actions (la boucle) qui laisse le jeu inchangé entre le début et la fin de son exécution excepté un léger gain en sa faveur (par exemple, un point de vie, du mana) au passage. Le joueur peut alors répéter ce gain autant qu'il le veut, s'il exécute la boucle suffisamment de fois.
  Avec les autres programmes, le seul moyen d'exécuter une boucle consiste à montrer la boucle à son adversaire, à se mettre d'accord avec lui, et à augmenter manuellement ses points de vie (et encore, avec MTGO, il est simplement impossible de simuler une boucle avec cet artifice puisque le logiciel vérifie les règles, il faut tout faire à la main...).
  Avec Macaks, le joueur peut enregistrer une boucle et demander au programme de l'exécuter 1 000 fois, ou plus s'il en a envie, en ayant la garantie que Macaks s'assurera que la boucle est valide ! Le moteur de jeu est suffisamment rapide pour que l'exécution soit quasi-instantanée : une boucle comportant une quinzaine d'étapes (donc de taille importante) est exécutée 1 000 fois en moins de trois secondes.
- **Chat spécifique.** La salle de chat apporte de nombreux réglages, simples à penser et à coder, mais encore fallait-il y penser. Ainsi tout utilisateur peut se voir attribuer des droits de spectateur, de joueur, de coach (pour les parties d'apprentissage), etc. On fournit aussi aux joueurs la possibilité d'échanger facilement des informations via la salle de chat, notamment d'échanger des decks sans avoir à utiliser un logiciel tiers d'échange de fichiers.
  Ce d'autant que notre salle de chat est clairement orientée Magic : elle gère bien évidemment les smileys comme toute salle de chat qui se respecte, mais en plus de cela elle permet aux joueurs d'afficher des coûts de mana à l'aide d'images au lieu d'avoir simplement un affichage texte.[4]
- **Intelligence artificielle.** Comme nous l'avons déjà mentionné, le seul logiciel à l'heure actuelle fournissant une telle approche est obsolète et présente plusieurs défauts graves.

[4] Voir le tutorial Macaks p.7 pour plus d'informations.

2 Organisation du projet

Notre groupe se compose de André, Boris, Christophe, Guillaume, Hervé, Irénée, Ivan, Jérôme, Pierre & Yann, entre lesquels nous avons divisé le travail en cinq sous-projets, déjà introduits précédemment :

- Magic : conceptualisation et codage des cartes
- Interface : graphisme et ergonomie
- Réseau : communication entre utilisateurs
- Noyau/base de données : moteur du jeu et base de données
- Intelligence artificielle : jeu jouable par la machine

2.1 Module Magic

2.1.1 Membres

- Ivan, chef de module
- Hervé
- Jérôme
- Yann

2.1.2 Consultants

- Christophe pour l'interface.
- Guillaume pour le code.

2.1.3 Description

Le but de ce module est de voir comment le jeu de cartes *Magic : the gathering* peut être interfacé sur ordinateur et comment les règles peuvent être implémentées dans le programme.

2.1.4 Objectifs

- Formaliser des règles
- Fournir aux membres du projet une « vision informatique » des règles
- Proposer des pistes pour le codage des cartes et du jeu
- Décrire une interface minimale permettant de jouer pour donner une base au module interface
- Et finalement coder les cartes.

2.1.5 Réalisations

- Présentation du jeu et ses règles aux membres du projet.
- Fournir au module interface une esquisse.
- Résolution de problème du nombre important de cartes : pour ne pas surcharger le logiciel, celui-ci ne charge que les cartes dont il a besoin pour jouer les parties. Chaque carte est ainsi représentée par une classe, et celles-ci sont fournies compilées avec le logiciel. Le logiciel charge donc dynamiquement les classes en fonction de ses besoins. La même politique a été adoptée pour les capacités les

plus répétitives, mais certaines sont chargés par défaut car difficilement contournables dans une partie de *Magic: the gathering* courante, par exemple la capacité de mana standard présente sur une grande majorité des terrains.

- Lister les primitives de base à implémenter impérativement : par exemple, pour ne citer que les plus importantes.
  - `addToManaPool(int[] mana)` (ajoute mana à la reserve de mana)
  - `isDealDamage(source, quantity)`
  - `draw(int num)`
  - `changeArea(cause, from_zone, to_zone)`
  - `lose_life(int quantity)`
  - `gain_life(int quantity)`
  - `setLife(int quantity)` (utilise les deux primitives du dessus)
  - `addPowerToughness(num, num)`
  - `tap()`
  - `unTap()`
  - `destroy(cause, boolean canBeRegenerated)` (identique à `move(cause, in_play, graveyard)` sauf que cela déclenche la régénération)
  - `chooseCard(card list, player)`
  - `counteredBy(cause)`
  - `attach(permanent)`
  - `discard()`
  - `addCounter(String counter_type, int quantity)`
  - `shuffle()`
  - Toutes les méthodes permettant d'obtenir les capacités d'une carte, en prenant en compte les modificateurs introduits par le jeu
- Lister les primitives et capacités « accessoires » qui permettent au logiciel de gérer davantage de cartes, par exemple : Bushido, Cumulative upkeep, Echo, Flanking...
- Coder la majorité des cartes de la 9^{e} édition de *Magic: the gathering*, sortie en juillet 2005.

2.1.6 Points délicats dans l'implémentation des règles

- Capacités statiques : elles permettent de recalculer les attributs des cartes à chaque action d'un joueur, ce qui se fait d'un point de vue règles par l'application de 10 couches très précises.
- Gestion des capacités déclenchées : certaines capacités se déclenchent lorsque le jeu est dans un certain état (par exemple, lorsque s'un joueur pioche une carte). La difficulté vient du fait que plusieurs capacités peuvent se déclencher simultanément, et que l'ordre d'application de ces capacités a une grande importance (surtout pour les bon joueurs), et qu'il faut donc faire choisir au bon joueur cet ordre.
- Gestion de l'arrivée en jeu des auras. Les auras sont un sous-type particulier d'enchantement qui s'attachent sur une autre carte en jeu. La subtilité vient du fait que lorsque une aura arrive en jeu après avoir été jouée normalement, elle s'attache au permanent que le joueur a ciblé en lançant le sort d'aura, alors que si l'aura arrive en jeu autrement, elle s'attache à un permanent valide quelconque au choix de son contrôleur, et ce sans cibler le dit permanent.
- Gestion des lancements alternatifs. Certaines cartes permettent de jouer d'autres cartes en modifiant la façon normale de les jouer (par exemple sans payer leur

coûts, ou en contournant les limitations normales). Nous avons ainsi introduit une méthode `playWith()`, et ainsi en récupérant les playwith possibles pour une carte donnée, le joueur choisit parmi ceux-ci celui qu'il veut utiliser.
- Nécessité de stocker des références dans les différentes classes pour représenter la connaissance que celles-ci ont de la partie.

Le codage des cartes a été géré de manière extensible – et *facilement* extensible – c'est d'ailleurs ce qui a été un grand point de travail du module. Ainsi il suffit de créer des classes Java à la main, de les compiler à l'aide des sources du projet, et d'ajouter les fichiers `.class` dans le répertoire de données. Codage rendu facile par l'ajout de nombreux helpers directement dans le noyau du projet, comme on le verra dans la section appropriée.

2.1.7 Exemple du codage d'une carte (simple)

"Enchant creature *(Target a creature as you play this card. This card comes into play attached to that creature.)* Enchanted creature has islandwalk. *(This creature is unblockable as long as defending player controls an Island.)*"

```java
/*
 * Fishliver_Oil.java
 *
 * Created on 6 décembre 2005, 13:49
 */

package magic.cards;
import magic.*;

// Fishliver Oil est une Aura,
// qui bénéficie d'une classe particulière
public class Fishliver_Oil extends Aura {

    // L'effet de Fishliver Oil s'applique dans
    // le layer 5, et n'affecte que le permanent
    // auquel elle est attachée, en lui donnant
    // la traversée des îles (Islandwalk)
    @Override
    public void effectLayer5(Permanent modified)
        {
         if (modified == attachedTo)
             modified.currentLandwalks.add(
                 new SuperAndBasicTypes(
                     (byte)0, Card.ISLAND));
    }
}
```

On remarque qu'il est inutile de préciser que Fishliver Oil enchante une créature, car c'est le cas par défaut, ni de préciser sa couleur, son coût, et autres informations déjà fournies par la base de données.

Il faut préciser que ce module a été très important pour le codage effectif de l'équipe Noyau. En effet, il est impossible de coder une seule ligne sans avoir une vision vraiment très globale de tout ce qui peut théoriquement se dérouler au cours d'une partie. Ce module a donc effectué un travail qui s'est voulu exhaustif : détermination des événements possibles à gérer, des modifications de règles à gérer... [5]

[5] Ce pour un simple but de non surcharge de calculs du serveur : en théorie toute règle est sujette

Avec cette vision globale du jeu, il a été possible de factoriser un maximum les fonctions à coder, d'avoir toujours un regard objectif et sévère sur l'application des règles. De plus, cela a permis de planifier le code ainsi que de prévoir ce qui devrait être fait. Sans ce module, les autres auraient travaillé avec des œillères.

Idées

Nous avons réfléchi à l'idée d'un parseur pour automatiser le codage des cartes. En effet, certaines cartes (les créatures essentiellement) ont un texte très court voire vide, composé de capacités à mot-clef que l'on traite en amont. C'est malheureusement le seul cas qu'on a réussi à facilement identifier, et créer un parseur reconnaissant vraiment un nombre substantiel de cartes nous a paru peu réalisable : il s'agirait effectivement quasiment de faire un parseur pouvant reconnaître la langue anglaise (certes, pas tout à fait, néanmoins l'étendue des effets des cartes à Magic est telle que les mots utilisés sont vraiment légion).

Nous avons donc décidé de ne pas poursuivre cette voie qui peut néanmoins – avec beaucoup de temps et à la condition sine qua none que les textes en entrée n'aient aucune faute de typographie – être envisageable plus tard au lieu de coder les 8 000 cartes à la main !

2.2 Module Interface

2.2.1 Membres

- Christophe, chef de module
- Boris
- Guillaume
- Hervé
- Ivan
- Jérôme

2.2.2 Consultants

- Yann : automatisation et intuition des actions

2.2.3 Description

Le but de ce module est simple :
- Créer une interface pour le jeu, jolie, simple d'utilisation et modifiable selon les goûts de chacun.
- Créer une interface pour consulter/modifier la base des cartes gérées.
- Créer une interface agréable pour discuter et proposer des parties.
- Lier le tout avec des fenêtres utiles : fenêtre d'accueil, fenêtre de connexion au serveur, fenêtre des informations sur le projet.

L'accent a été mis surtout sur la simplicité d'utilisation. Il faut que n'importe quelle personne connaissant *Magic: the gathering* puisse utiliser notre projet. Ensuite est venu l'aspect ergonomie. Il nous a paru important que chaque utilisateur puisse modifier un minimum l'interface pour l'adapter à ses besoins/habitudes/goûts. Enfin,

à changements, mais si en pratique aucune carte ne change tel ou tel point ce n'est pas la peine de rajouter des tests pour rien !

nous avons dû penser à comment rallier l'utile à agréable. Cela était primordial pour l'interface spéciale au jeu.

2.2.4 Critères de réussite

1. Une interface pour la base de cartes suffisamment intuitive pour que tous les joueurs de *Magic: the gathering* comprennent comment ça marche.
2. Une interface de jeu simple d'utilisation.
3. La possibilité de modifier les interfaces.
4. Intégrer des sons, enrichir le jeu.

2.2.5 Travail accompli

Pour l'interface de la base de donnée :

- Le projet dispose d'une interface permettant des recherches rapides dans la base de donnée et un panneau affichant le deck qu'on est en train de construire/modifier. On peut ainsi créer son deck en toute simplicité.
- On peut ensuite exporter le deck ainsi construit au format XML ou dans un format de deck utilisés par quelques autres programmes. Nous gérons aussi l'import de ces formats pour les gens qui désirent conserver des decks déjà faits à l'aide d'autres logiciels.

Pour l'interface de jeu :

- Nous avons décidé de mettre chaque partie ouverte par l'utilisateur dans son propre onglet. De cette façon, l'utilisateur peut rapidement passer d'une partie à une autre et suivre (voire jouer) plusieurs parties en simultané.
- À l'intérieur de cet onglet, on dessine à l'aide de Java 2D la zone de jeu et les cartes de la main du joueur.
- Les cartes jouées peuvent être engagées comme dans une vraie partie, elles apparaissent alors pivotées de 90 degrés, texte compris.
- On permet au joueur de déplacer les cartes de la zone de jeu par glisser-déplacer. Les capacités graphiques de Java et le *double-buffering* permettent de mettre à jour la position des cartes en temps réel (elles se déplacent en même temps que le curseur de la souris) avec affichage sans scintillement ni clignotement.
- Pour plus de rapidité, les cartes attachées sont pré-gérées par le client. Dans la vraie vie, les cartes attachées sont des cartes que l'on place sous d'autres, légèrement décalées vers le haut et la droite, et qui forment un bloc. Elles peuvent former un arbre d'attachements (plusieurs cartes peuvent être attachées à une même carte) que, pour imiter le jeu réel, on « linéarise » en une pile. Dans l'interface, il faut d'une part calculer la position de chaque carte dans la pile, et d'autre part déplacer l'ensemble de la pile lorsque le joueur déplace une carte y appartenant. Cette opération est validée par le serveur lorsqu'un synchronise la position des cartes entre les deux joueurs, mais le client précalcule la position des cartes (figure 1) lors du glisser-déplacer, en temps réel.

Pour l'interface "client" :

- Nous avons introduit – en plus d'un chat principal – un chat au sein de chaque partie. Il a fallu en gérer l'affichage en introduisant une zone de chat dans chaque onglet de partie.
- Pour rendre la discussion pratique et agréable à lire, nous avons ajouté quelques smileys, dont certains représentant les couleurs de cartes du jeu.

Les autres fenêtres qui s'ajoutent au reste :

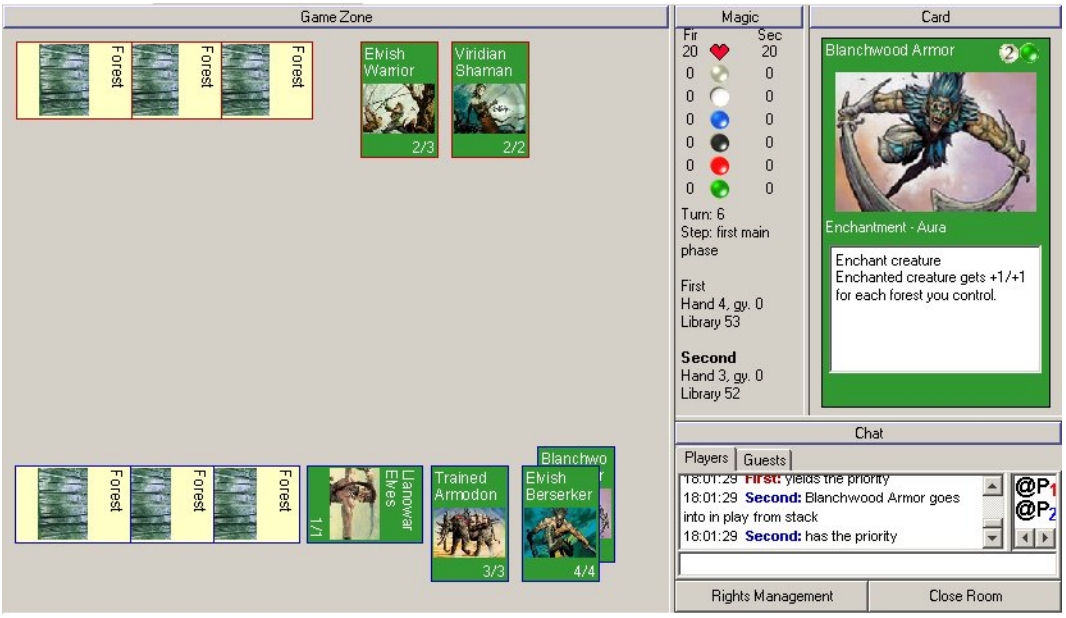


FIG. 1 – Une Blanchwood Armor attachée à un Elvish Berserker, en bas à droite.

- Une fenêtre principale s'affiche au lancement du programme, permettant d'accéder soit à la partie jeu, soit à la partie base de données.
- Une fenêtre de connexion au serveur apparaît dès qu'on veut rejoindre la zone de jeu. Elle permet de soit créer un serveur sur sa machine, soit se connecter à un serveur existant.
- Une fenêtre donnant des informations générales sur le projet est accessible en cliquant sur le bouton "About" de la fenêtre principale.

Toutes ces fonctionnalités sont décrites en détail dans le tutoriel, qui fournit en plus de cela des captures d'écran montrant les différentes zones de jeu ainsi que leur utilité.

2.2.6 Idées

Nous avons réfléchi à l'ajout d'effets sonores dans le projet, ainsi qu'à l'ajout d'effets graphiques (notamment pour la sélection des cartes). Cela sera intégré dans un futur proche à notre projet.

Nous avons d'ailleurs déjà codé toute la structure bas niveau permettant de jouer des sons aux moments propices, mais l'interfaçage avec le serveur de jeu n'a pas pu être conclu et il est resté difficile de savoir, lors de la réception de l'état du jeu par le client, s'il fallait lancer la lecture d'un son ou pas.

Quant aux graphismes, on s'est voulu novateurs sur plusieurs points :

- **Personnalisation de l'interface.** On peut déplacer les panneaux de l'interface par glisser-déplacer de leur titre (figure 2). Cela permet au joueur d'adapter l'interface à ses besoins et à la taille de son écran.
- **Affichage dynamique des caractéristiques des cartes.** Les caractéristiques des cartes sont régulièrement modifiées par le jeu : force, endurance, type de créature, etc. peuvent être affectées par des sorts. Alors que les autres logiciels affichent une photo statique des cartes, ou bien ne vérifient pas les règles donc ne connaissent pas les caractéristiques courantes des cartes, MACAKS affiche en

permanence les caractéristiques actuelles des cartes : si l'on joue un sort qui augmente la force d'une créature, le retour visuel est immédiat.

- **Interface adaptée aux questions posées par les cartes.** Les cartes peuvent poser des questions portant sur divers types d'objets. Cela peut aller de « choisissez une carte à défausser » à « assigner des dégâts aux créatures adversaires » en passant par « choisissez un entier X compris entre 1 et 5 ». Le module interface a dû communiquer avec le module Magic pour concevoir des interfaces adaptées au type de chacune des questions, et avec le module Noyau (cf. page 20) pour gérer la réception des questions et afficher la boîte de dialogue qui correspond au type de la question. De plus, l'interface ne se contente pas d'afficher des choix et de les envoyer au serveur, mais elle les valide côté client, par exemple en vérifiant que le coût de mana à payer peut effectivement être payé et en grisant le bouton *OK* si ce n'est pas le cas.

Une optimisation, et pas des moindres, a été effectuée au cours du projet (et ce en parallèle avec Magic-Ville, cf. la partie du module Noyau). En effet il n'est pas rare sur les cartes, de voir apparaître divers symboles, qui sont en pratique des petites images sur les cartes, mais qui n'apparaissent souvent qu'avec du texte dans la plupart des bases de données sur Internet. Après un travail de chercher/remplacer en trouvant une grammaire adaptée pour mettre à jour les textes, nous avons réussi a convertir les différents T, 2, W, U, B, R, G, etc. par leurs symboles correspondant. Cela se traduit par l'ajout de balises `%` insérées au bon endroit, et la réutilisation du code de chat pour la gestion des images.

2.3 Module Réseau

2.3.1 Membres

- Guillaume, chef de module
- André
- Hervé
- Pierre

2.3.2 Description

L'un des objectifs majeurs du projet est de permettre de jouer à distance. Le module réseau doit donc remplir plusieurs objectifs :

- fournir une couche de communication bas niveau au module Noyau, afin de véhiculer les messages relatifs au jeu ;
- proposer un système de chat similaire à IRC, pour que les joueurs puissent discuter avant de jouer une partie – et pendant. Le système de chat sert également à informer les joueurs du déroulement de la partie, à l'aide de messages de log diffusés.

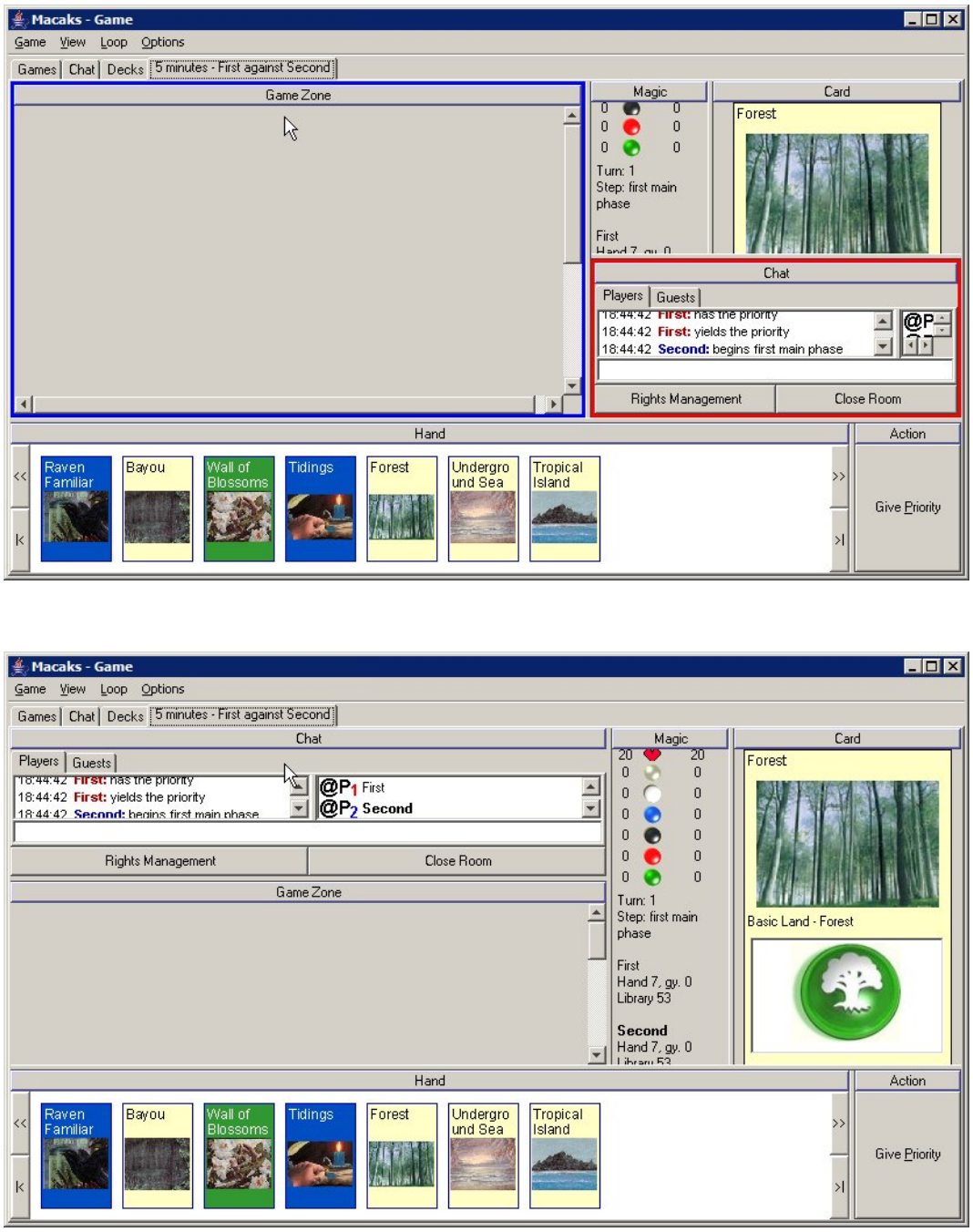


FIG. 2 – Personnalisation de l'interface par glisser-déplacer des zones. En haut, avant ; en bas, après.

2.3.3 Couche de communication

Le jeu suit une architecture de type client/serveur : le serveur offre des salles de discussion à l'aide d'objets `Room`, fait tourner le noyau du jeu sur différentes parties en maintenant un objet `GameServer` pour chaque `Room` sur laquelle une partie est en cours. Cette architecture impose donc un fonctionnement à base de messages, puisque clients et serveurs tournent de manière non synchrone.

Le code comporte deux classes abstraites : `PipeServer`, qui crée un serveur attendant que des clients se connectent, et `PipeClient`, qui est instanciée soit du côté serveur lorsqu'un client se connecte au serveur (c'est alors le canal permettant de communiquer avec le client), soit du côté client pour établir un lien de communication avec le serveur. Il faut bien comprendre que, exactement pareil qu'avec des sockets, pour un unique lien de communication client-serveur on crée *deux* objets `PipeClient`, un de chaque côté du lien.

Chaque objet `PipeClient` dispose d'une poignée de méthodes classiques (figure 3, page 17) : `connect()`, `send(message)`, `close()` et de deux *listeners*, c'est-à-dire des interfaces dont le `PipeClient` appelle des méthodes lorsque certains événements surviennent : connexion réussi, lecture d'un message, fermeture de la communication par le correspondant. Noter la différence fondamentale entre ce schéma et celui des sockets classiques : il n'y a pas de méthode `read()`, puisque l'on fonctionne par messages et par *listeners* prévenus de manière asynchrone. Par contre la séquentialité est respectée : les messages sont lus dans l'ordre de leur envoi, et on n'appelle pas les *listeners* plusieurs fois en même temps : cette contrainte simplifie la gestion des messages par le reste du programme, notamment la partie codée par le module Noyau.

Noter également que la méthode `onPipeMessage(Message msg)` du *listener* `MessageListener` est mise en commentaire, car ne doit pas être nécessairement implémentée avec ce prototype. Voir la section suivante pour plus de détails.

La couche de communication se trouve donc responsable du transfert de messages entre le serveur et ses différents clients. Afin d'offrir un niveau de sécurité maximal, les clients ne peuvent pas communiquer entre eux directement : ils doivent passer par le serveur, qui filtre les messages.

On utilise le système de communication asynchrone de Java, afin de ne pas multiplier les threads dans le serveur. Un unique thread par machine se charge de rediriger les messages qu'il reçoit au thread de l'interface graphique (pour le client) ou au thread de chaque `GameServer` (pour le serveur). Cela nous a amené à utiliser la bibliothèque `java.nio.channels`.

Malheureusement cette bibliothèque semble être boguée sous les systèmes Unix[6], d'où des soucis pour lancer le jeu sous Linux. Nous avons alors choisi de développer une bibliothèque interne pour simuler un jeu local, afin d'offrir aux utilisateurs de Linux la possibilité de tester le jeu sans plantages et dysfonctionnements aléatoires. Ce pseudo-réseau permet également de tester le jeu plus rapidement, car il ne fait pas appel aux lourdes couches réseau et au système événementiel de `java.nio.channels`.

Concrètement, Macaks contient deux implémentations de la couche de communication :

[6] http ://bugs.sun.com/bugdatabase/view_bug.do ?bug_id=5056395 illustre le genre de problème rencontré.

```java
/*
 * Pipe.java
 *
 * Created on 24 octobre 2005, 20:44
 */

package network;
import java.io.IOException;
import tools.Introspection;

public abstract class PipeClient {

    public interface ConnectListener {

        /**
         * Called when the pipe successfully connects,
         * or when a connection error occurs.
         */
        public void onPipeConnect(boolean success);
    }


    public interface MessageListener {

        /**
         * Called when a message is received.
         */
        // public void onPipeMessage(Message msg);

         /**
          * Called when the pipe is closed.
          * May be when a fatal error occurs.
          */
        public void onPipeClosed();
    }


    public void setConnectListener(ConnectListener listener);
    public void setMessageListener(MessageListener listener);


    public abstract void connect(String hostName) throws
       IOException;


    public abstract boolean send(Message msg);

    public final synchronized void close();
}
```

FIG. 3 – La classe de base d'une interface de communication événementielle, à base de messages.

- `ManagerChannel` : utilise `java.nio.channels` pour créer des clients et des serveurs sur le réseau ;
- `ManagerLocal` : utilise un système de FIFO en mémoire pour simuler une couche réseau. Ce système est local dans le sens où les liens créés ne peuvent pas « sortir » de l'application : on ne peut pas communiquer entre deux ordinateurs distants.

Chaque manager définit ses propres classes héritant respectant de `PipeServer` et `PipeClient`, ce qui permet aux deux couches de communication d'utiliser la même interface et de se comporter exactement pareil (excepté bien évidemment que `ManagerLocal` est local à chaque application lancée), ce qui permet de basculer de l'une à l'autre très facilement : l'option de ligne de commande `--pseudo-network` passe en mode `ManagerLocal`.

Réaliser la couche réseau bas niveau a été effectué avec succès sans rencontrer d'autre problème à part le bug de la JVM sous Linux. Dans un premier temps nous avons programmé la couche réseau uniquement, puis nous avons réalisé la couche pseudo-réseau en reprenant l'architecture de classes de la couche réseau déjà réalisée, afin de minimiser les changements à apporter ailleurs dans le code (lesquels changements se sont bornés à choisir au démarrage de l'application quel système on souhaite utiliser).

2.3.4 Système de discussion, échange de messages

Le système IRC doit fournir :
- un système de salles, chaque salle correspondant à une partie en cours ;
- des droits pour les joueurs dans chaque salle, afin de permettre des schémas comme « tel observateur coache tel joueur » (assistance au jeu) ou bien « tous les joueurs connaissent tout sur le jeu » (partie pédagogique) ou encore « invités interdits » (partie privée) ;
- un système d'écriture de messages et d'envoi de decks en privé : la communication passe par le serveur mais les autres clients n'en sont pas informés. Cela correspond respectivement aux *private messages* et à l'échange de fichiers du protocole IRC.

Concevoir et implémenter la gestion des salles de discussion et de l'échange de messages a été fait facilement puisque l'on disposait du modèle théorique d'IRC et d'une couche réseau bas niveau efficace et facile à déboguer : NetBeans, son débogueur, et sa fonction de recompilation à la volée se sont avérés très utiles pour traquer les bugs de synchronisation. Le mécanisme d'introspection de Java a permis de grandement facilité l'interception des messages. À chaque message correspond une classe de type donnée ; chaque objet qui veut intercepter certains messages code, pour *chaque* message, une fonction `onMessage(MessageType msg)` où `MessageType` est le type du message que l'on veut intercepter. Lorsqu'un message arrive, le gestionnaire appelle `listener.callMethod("onMessage", theMessage);`, ce qui exécute automatiquement la fonction `onMessage(MessageType msg)` dont le `MessageType` correspond à celui de `theMessage`. Si aucune fonction n'existe, une exception est lancée et récupérée proprement. Cela évite la lourdeur du système des interfaces, qui oblige chaque *listener* à implémenter toutes les fonctions d'événement, y compris celles pour lesquelles on ne veut rien faire (code vide).

En plus de ces impératifs, il serait intéressant d'améliorer la gestion du chat en proposant un mécanisme de migration automatique du serveur. Le serveur serait mobile, et lorsque le joueur qui l'héberge quitte le réseau, le serveur migrerait automatiquement sur un autre membre. Dans un premier temps, on ignorera les problèmes

de défaillance matérielle en supposant que le joueur serveur quitte toujours « poliment », i.e. en prévenant le reste du réseau. On s'attachera ensuite à concevoir un protocole plus robuste aux défaillances matérielles.

2.3.5 Migration du serveur

Nous avons commencé à étudier le problème, qui est théoriquement faisable et relativement simple si l'on suppose que le serveur prévient les autres clients avant de se déconnecter, et leur laisse le temps de reprendre le flambeau. Le principe consiste à envoyer tout l'état du jeu à un autre client, d'avertir les autres clients de l'IP du futur nouveau serveur, qui pourront alors se reconnecter.

Bien que simple d'un point de vue théorique, cette fonctionnalité demande beaucoup de codage et nous avons préféré ne pas retarder le planning sur des points plus vitaux concernant l'IA ou le module noyau, qui demandaient que des personnes du module réseau intègrent la couche de communication aux autres modules.

2.3.6 Liaison avec les autres modules

Le module réseau fournit des outils codés « sur mesures » pour les autres modules, notamment Interface et Noyau. Plusieurs mécanismes ont été mis en place sur le moteur du jeu pour satisfaire différentes demandes.

Envoi de l'état du jeu

Problème. Certains traitements du jeu sont longs, les boucles notamment, il ne faut surtout pas bloquer la gestion des salles de chat pendant que le jeu tourne. De plus, plusieurs jeux sont susceptibles de tourner en même temps sur le serveur.

Solution retenue. Nous avons donc choisi de faire tourner chaque le moteur de jeu de chaque partie sur un thread séparé du serveur.

Réception des événements

Problème. Le moteur du jeu a besoin d'être prévenu lorsqu'un événement survient sans faire attente active : `while (true)` est à proscrire ! Les événements sont du type « tel joueur a passé la priorité » ou « tel joueur joue une carte ».

D'autre part, l'objet `LinkedBlockingQueue` (file bloquante) de Java permet de maintenir une file d'événements *thread-safe* bloquante : lorsque le thread veut récupérer un élément de la file et que cette dernière est vide, il est mis en attente jusqu'à ce qu'un autre thread place un élément dans la file. Ainsi, le thread du serveur peut se mettre en attente passive (sans) pour attendre que le thread réseau lui donne un la réponse à la question qu'il a posée. On utilise une file et non une unique variable globale pour ne pas perdre de messages si plusieurs sont envoyés en même temps.

L'état du jeu a besoin d'être envoyé régulièrement aux clients pour que l'interface graphique de chacun d'eux se mette à jour. Il faut éviter de le faire trop souvent, pour ne pas saturer le réseau ni rafraîchir continuellement l'interface

au point de la rendre inutilisable, mais il est vital que l'état du jeu soit à jour au moment où le serveur pose une question à un joueur, car la fenêtre qui permet de répondre à la question fait appel à l'état du jeu pour afficher les cartes, et il faut que les informations soient à jour à ce moment-là.

Solution retenue. pour chaque partie en cours, un thread supplémentaire est créé pour envoyer l'état du jeu aux clients en imposant certains délais entre les envois, à l'aide de la fonction `sleep()`. Le mot clé `synchronized` allié aux fonctions `wait()` et `notify()` de Java permettent de mettre en place respectivement une section critique et un sémaphore : lorsque le thread de mise à jour a fini d'envoyer l'état du jeu, il fait une pause avec `sleep()` et se met en attente avec `wait()`. Plusieurs sections critiques permettent de s'assurer que les variables globales indiquant que l'état du jeu a changé ne soient pas modifiées en même temps ; ces variables sont `isDirty` pour l'état du jeu complet, et `isDirtyLight` pour l'état du jeu restreint, i.e. ne comportant que des informations globales du genre « qui a la priorité ? ».

Le thread du serveur réveille le thread d'envoi de l'état du jeu avec `notify()` lorsque l'état du jeu a changé et qu'il a besoin d'être envoyé aux clients. En pseudo-code, cela donne le code de la figure 4 (page 21).

Questions/réponses

Problème. Au cours d'une partie, le serveur peut avoir besoin de poser des questions à un joueur. Une question est bloquante : le serveur doit attendre que le joueur réponde avant de poursuivre. Les questions sont variées : « choisissez une carte parmi celles-ci », « répondez à une question par oui ou par non », « répartissez vos dégâts parmi ces créatures », etc., et la nature des réponses dépend des questions.

Solution retenue. Nous avons choisi de créer deux classes de base `Ask` et `Result`, respectivement pour les questions et les réponses. Chaque couple question/réponse particulier possède ses propres classes dérivées de `Ask` et `Result`.

Chaque question possède un identifiant numérique qui doit être redonné par la réponse, ce qui permet au serveur de s'assurer que le client répond bien à la bonne question. De plus, chaque réponse possède une fonction de validation qui, si on lui donne une question du type correspondant à la réponse, vérifie que la réponse est valide pour la question. Cela inclue deux choses : premièrement, il vérifie que la réponse correspond à la classe attendue : une question `AskPayMana` exige une réponse de type `ResultPayMana`. Deuxièmement, la réponse est vérifiée en détail par une fonction `validate()` : pas de valeurs nulles, tableaux de la bonne taille, valeurs numériques dans leur intervalle de validité, etc. La fonction `validate()` assure que même un client malhonnête ne pourra pas faire planter le serveur en envoyant des messages fabriqués de toutes pièces.

Côté client, on crée une fonction `onGameMessage` par type de question, et on affiche la boîte de dialogue spécifique permettant de répondre à la question. Dans le code de la boîte de dialogue, on utilise la fonction `validate()` pour griser le bouton *OK* si l'utilisateur a effectué un choix invalide.

```java
/**
 * Send the current game state to all clients.
 * and only if the game state is dirty.
 * In practice, we notify the game state sending thread
 * that it must do it.
 */
public void sendGameState() {
    if (!threadSendStateThread.isAlive())
        // Launch the thread
        threadSendStateThread.start();
    else{
        // Or wake him up if it is already running
        synchronized (threadSendStateThread) {
            threadSendStateThread.notify();
        }
    }
}

/**
 * Entry point of the game state sending thread.
 */
private void sendGameStateThread() {
    try{
        for (;;) {
            // The function which sends the game state
            doSendGameState();

            // Wait a little
            // During loops, we wait much more
            threadSendStateThread.sleep((isLoopRunning())
                ? DELAY_SENDGAMESTATE_LOOP :
                    DELAY_SENDGAMESTATE_DEFAULT);

            // If the game has not been changed
            // when we were sending it,
            // wait until sendGameState() is called
            synchronized (threadSendStateThread) {
                if (!isDirty && !isDirtyLight)
                    threadSendStateThread.wait();
            }
        }
    }catch (InterruptedException e) {
        // Happens at the end of the game
    }
}
```

FIG. 4 – Code de gestion de l'envoi du jeu, avec synchronisation entre le thread du jeu et le thread dédié à l'envoi du jeu.

2.4 Modules Noyau & Base de données

2.4.1 Membres

- Yann, chef de module
- André
- Guillaume
- Irénée

2.4.2 Consultants

- Christophe : base de données
- Ivan : points de règle
- Pierre : formalisations

2.4.3 Sous-modules

- Base de données
- Implantation du jeu
- Linkage correct

Nous ne nous sommes pas divisés entre ces différents sous-projets, puisqu'ils se sont étalés dans le temps. La base de données nous a occupés au début, et le linkage a été fait en parallèle de l'implantation du jeu, pour les différents tests.

2.4.4 Description base de données

On travaille sur du XML, ce qui donne, pour la carte Fishliver Oil que voici :


```
<?xml version="1.0" encoding="ISO-8859-1"?>
<macaks_database>
  ...
  <card>
    <name>Fishliver Oil</name>
    <cost>1U</cost>
    <type>Enchantment - Aura</type>
    <text>Enchant creature (Target a creature as you play
this card. This card comes into play attached to that
creature.)
    Enchanted creature has islandwalk. (This creature is
unblockable as long as defending player controls an
Island.)</text>
    <power></power>
    <tough></tough>
    <expansions nb="3">
      <exp edition="ara" serial="13">Anson Maddocks</exp>
      <exp edition="chr" serial="20">Anson Maddocks</exp>
      <exp edition="9th" serial="77">Ralph Horsley</exp>
    </expansions>
  </card>
  ...
</macaks_database>
```

C'est donc une liste de cartes.

La base de données a été créée à partir de données récupérées sur Internet. Pour être précis, nous utilisons assez souvent un site francophone parlant de ce jeu de cartes, Magic-Ville [7]. Connaissant ce site depuis quasiment ses débuts, il se trouve que nous effectuons un travail assez soutenu pour la mise à jour de ce site, et avec l'aide de Jean-Marc Blanchard, le webmaster du site, nous avons récupéré l'intégralité des textes des cartes. Pour cela, il a fallu écrire un petit script qui récupère les données nécessaires dans la base MySQL et les convertit dans le format XML voulu.

On peut au passage remarquer que les textes sont à jour. Notons-le, les textes des cartes sont amenés à varier au cours du temps, et il arrive que WoTC change les textes des cartes, pour diverses raisons (simplifications, harmonisations, tout simplement changement d'un texte un peu bogué...). Et donc, il a fallu sur Magic-Ville, avant de reprendre toutes les données, mettre à jour ces cartes « à la main », en comparant avec les textes officiels sur la page de WoTC (effectivement, nous nous voyions mal demander à WoTC de nous envoyer directement leur base de données !). D'une pierre deux coups donc.

Une fois cette base de données prête à l'emploi, il s'agit de pouvoir construire des decks à partir de ces cartes, et pour cela il faut créer des fonctionnalités comme le filtrage, la recherche. Cette partie nous a occupés pendant les trois premières semaines (nous avions de toute façon besoin de l'avancement du module Magic pour commencer à nous occuper du Noyau proprement dit, ainsi le pipeline a été optimal).

On peut ainsi trier les cartes et les rechercher comme on aurait l'habitude de le faire nous-mêmes : par édition, par couleur, par type, par force, par nom, etc. Des fonctionnalités de tri ont été implémentées qui ne sont pas visibles dans l'interface mais qui peuvent servir dans de prochaines mises à jour de notre logiciel, notamment pour filtrer par édition.

Notons que le format des decks est le même pour l'éditeur et le jeu en lui même (c'est également un format XML). On ne stocke dans le format des decks en fait que les noms des cartes et leur quantité présente. Lors du chargement du deck dans une partie, le logiciel charge les classes qui ont le bon nom.

Comme on le disait, notre logiciel est extensible, on peut vraiment facilement ajouter des cartes dans cette base de données (en respectant simplement le codage XML), mais il faut bien faire attention à ne pas faire de fautes de frappe dans l'écriture des noms de cartes, puisque comme on vient de le dire, les noms de classes des cartes d'un point de vue règles doivent correspondre aux noms des cartes dans la base de données.

Dans cette partie on l'a vu le travail a essentiellement été un travail de longue haleine, fastidieux même, mais aucune grande difficulté ne nous est apparue.

2.4.5 Implantation du jeu

Il a fallu tout d'abord attendre la création des premiers squelettes de fonctions par le module Magic, ainsi que la structure globale des héritages.

Une fois cela fait, nous avons pu coder proprement dit chacune de ces fonctions, en faisant à chaque fois bien attention à respecter toutes les règles. Si les algorithmes

[7] http ://www.magic-ville.com/fr

sont sûrement optimisables, en tout cas ils correspondent à l'application « à la lettre » des règles de *Magic: the gathering*, qui prévoient de recalculer toutes les données de jeu à tout moment dès que quelque chose a pu les modifier. Cela peut apparaître au premier abord vraiment peu subtil, en pratique vu le très faible nombre de cartes posées sur le plateau de jeu, il n'y a aucun problème. En pratique, l'exécution est instantanée, et même les boucles (qui exécutent une série d'actions un très grand nombre de fois) tournent rapidement, à raison de plusieurs milliers d'actions par seconde, donc autant de d'application des algorithmes.

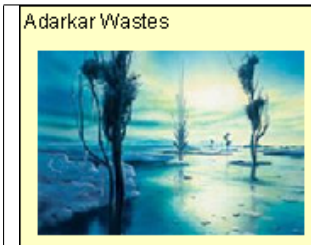
Interdire au joueur de faire des actions illégales a été pensé en amont, tout simplement en ne proposant au joueur que ce qu'il a le droit de faire à un instant donné. Pas de conflits de messages qui arrivent en même temps de deux joueurs ou quoi que ce soit niveau réseau, chaque joueur joue lorsque c'est à lui de jouer, et ne peut demander à faire que des actions légales. Tout est vérifié en amont et en aval, pour éviter que le joueur arrive néanmoins à tricher en renvoyant des valeurs biaisées au programme via quelconque changement de code.

La plupart des problèmes liés à cette partie sont survenus au moment où il fallait rajouter la gestion de cas spéciaux, ainsi que l'interfaçage avec les autres modules, comme on le verra dans la prochaine section. Effectivement certains cas spéciaux ne peuvent pas être gérés par les cartes elles-mêmes, pour factoriser le code et également pour certaines contraintes générales sur l'état du jeu difficilement accessibles par la carte. Et, lorsqu'il s'agit de les coder, il faut faire en sorte de rester toujours le plus général possible pour traiter – par avance – des cas qui pourraient survenir dans des cartes qui n'existent pas encore. La plupart des cas avait ainsi été prévue à l'avance par le module Magic, mais un ou deux points nous ont demandé de revoir plusieurs points de code.

2.4.6 Travail en commun avec d'autres modules

De fait, appliquer les règles « à la lettre » donnerait quelque chose d'injouable pour les joueurs : il faudrait très souvent céder la priorité, ce qui se fait de manière naturelle dans une partie de Magic dans la vraie vie. Ainsi, il est de notre devoir de proposer des options au joueur pour automatiser certains choix qui, pour la plupart, feraient perdre du temps pour rien à l'utilisateur en l'obligeant à cliquer pour pas grand chose.

Nous avons donc travaillé avec le module interface pour avoir des options intuitives correspondant aux situations de jeu réelles, pour éviter cette lourdeur.



Toujours dans le cadre du graphisme, une optimisation, et pas des moindres, a été effectuée au cours du projet (et ce en parallèle avec Magic-Ville). En effet il n'est pas rare sur les cartes, de voir apparaître divers symboles, qui sont en pratique des petites images sur les cartes, mais qui n'apparaissaient avant qu'avec du texte. Après un travail de chercher/remplacer en trouvant une grammaire adaptée pour mettre à jour les textes, nous avons réussi a convertir les différents T, 2, W, U, B, R, G, etc. par leurs symboles correspondant. Cela se traduit par l'ajout de balises `%` insérées au bon endroit, et la réutilisation du code de chat pour la gestion des images.

Au niveau des points chauds de cette partie, il y a eu l'héritage. Pensé au début pour coller aux règles du jeu, il s'est avéré que cet héritage n'était pas du tout commode pour l'interfaçage avec le réseau, qui avait besoin d'avoir deux classes tout à fait différentes pour chaque objet : la classe client, et la classe serveur (le serveur gérant seul les règles, et le client n'ayant accès qu'à du texte et à quelques caractéristiques). Après de nombreux commits changeant la structure des héritages assez brutalement, nous sommes arrivés à une gestion parfaitement correcte des objets à la fois du point de vue des règles et du point de vue du réseau. Si l'on avait eu l'héritage multiple, bien sûr, il en aurait été autrement...

Mais également, et le choix des personnes dans le module y est pour quelque chose, il a fallu gérer l'interaction entre les messages envoyés au client et l'intelligence artificielle (qui est considérée comme un client). Ainsi nous avons codé de manière intelligente les chaînes de caractères que l'on envoie au client pour que l'IA puisse les analyser et comprendre tous les messages qu'on lui envoie.

2.4.7 Dans le futur

Il est parfaitement envisageable, lors de la création de nouvelles cartes, de créer de nouveaux helpers pour automatiser les actions qui deviendront habituelles avec les nouvelles cartes créées, pour rendre aisée la création de cartes.

2.5 Module Intelligence Artificielle

En plus de créer un simulateur, notre ambition était de créer une Intelligence Artificielle pour Magic intégralement autonome. Il est en effet plus agréable d'avoir la possibilité de jouer tout seul car il n'est pas toujours facile de trouver un adversaire humain. La complexité du jeu fait qu'il est très difficile de créer une telle IA et les rares tentatives précédentes de créer à une IA ont abouti à des échecs. Néanmoins nous pensons avoir réalisé une IA de qualité qui suffira amplement pour n'importe quel joueur voulant tester ses decks ou juste jouer à tout moment. Parmi les nombreuses qualités de notre IA, nous pouvons souligner :

- Une gestion irréprochable des combats
- Un système dynamique de valeurs des cartes
- La gestion des effets compliqués

- Un choix intelligent au niveau des cartes jouées
- Une réactivité élevée au choix de l'adversaire

2.5.1 Combats

Les combats sont l'une des seules parties du jeu qu'on peut prévoir à l'avance. Même si différents effets peuvent modifier la nature d'un combat, la façon dont les dégâts sont assignés reste en général la même. On peut donc se servir de la puissance de calcul de l'ordinateur pour faire un système de combat efficace.

Localement, au niveau d'une créature bloquée par d'autres créatures, on teste toutes les assignations de dégâts d'une part valides et d'autre part intéressantes et on vérifie en fonction de la valeur des créatures l'assignation qui maximisera les pertes adverses.

L'écart des pertes de l'adversaire entre deux assignations de dégâts différentes est parfois faible et il est aisé de récupérer la valeur d'une assignation en fonction d'une autre. Ainsi, même si l'on teste toutes les possibilités intéressantes, on ne recalcule pas tout à chaque fois et on a réussi à créer un algorithme efficace même pour un grand nombre de créatures.

Ensuite, il est aisé pour l'IA de trouver la façon de bloquer optimale lorsqu'elle connaît les créatures attaquantes. On a testé toutes les solutions valides en optimisant de la même façon que la dernière fois. Ceci nous permet de déterminer la meilleure façon d'attaquer en regardant pour chaque assignation d'attaquants laquelle est la plus dévastatrice pour l'adversaire.

Parmi les optimisations supplémentaires, nous avons séparé les créatures ne pouvant être bloquées que par un type particulier de créature. Par exemple, les créatures avec le vol sont traitées séparément ce qui réduit le nombre de créatures qu'il faut gérer en une fois. Malgré toutes ces optimisations, l'algorithme reste exponentiel et met trop de temps pour les grandes valeurs. Typiquement, quand il y plus de 8 créatures attaquantes et 8 créatures bloquantes (ce qui en pratique arrive très rarement), l'algorithme met plus d'une seconde pour renvoyer un résultat.

Pour ne pas que l'IA soit trop lente, on utilise des heuristiques naturelles pour prendre une décision. Quand le nombre de créatures est élevé, soit attaquer avec toutes les créatures tue l'adversaire, soit c'est du suicide. Dans ces situations, seules les créatures vraiment plus fortes que les autres ou les créatures qui ne peuvent être bloquées peuvent attaquer sans risque. Notre heuristique trouve donc individuellement les créatures qui peuvent attaquer. Les résultats de cette heuristique sont excellents car c'est la façon intuitive de jouer pour un joueur. Il n'y a que quelques rares cas ou cette heuristique n'est pas optimale. C'est pourquoi on utilise la méthode exhaustive quand on le peut, même si l'amélioration est faible.

En plus, notre IA gère les effets qu'on peut jouer pendant les combats pour améliorer certaines créatures de la même façon. Néanmoins, la réalisation de ces algorithmes est beaucoup plus délicate car on ne peut pas prévoir tous les effets qu'on pourra jouer et il difficile de voir quand on peut jouer en effet qui pourrait influencer le combat car ces effets sont en très grande quantité donc on ne peut les lister en amont.

2.5.2 Valeur des cartes

La majeure partie de l'IA repose la valeur des cartes qui lui permet d'évaluer toutes les situations. Comme on l'a dit précédemment, il est très difficile de comprendre ce que font les cartes. Néanmoins, on va pouvoir dire qu'une carte nous aide ou nous dérange en lui donnant une certaine valeur. Tout d'abord, nous devons aider l'IA en donnant à chaque carte une valeur de base. Nous donnons cette valeur que pour les cartes difficiles à calculer par exemple les enchantements avec un texte compliqué que l'IA ne peut pas comprendre. Ainsi, on pourra dire à l'IA si cette carte est vraiment dangereuse ou pas. L'IA peut aussi donner une valeur par défaut à une grande partie des cartes. Par exemple, l'IA donne une valeur par défaut pour les créatures en fonction de leur force/endurance, capacités et coût. Ceci donne un bon aperçu de la valeur des cartes mais cette valeur n'est pas contextuelle, c'est-à-dire qu'elle ne dépend pas de l'état du jeu. Il faut donc rajouter à ces valeurs statiques des valeurs dynamiques.

L'utilité des valeurs dynamiques part d'un constat de base : une créature 2/2 est utile lorsque l'adversaire ne peut pas bloquer mais l'est beaucoup moins lorsque l'adversaire contrôle une créature 5/5. A chaque fois qu'un sort ou une capacité est jouée, ces valeurs sont mises à jour. Les valeurs des cartes en main sont également modifiées. Par exemple, on à un Disenchant (qui détruit un artefact ou un enchantement) dans la main qu'on n'a pas pu jouer et l'adversaire à un artefact très dangereux en jeu. Cet adversaire nous force à nous défausser d'une carte (c'est à dire supprimer une carte de la main). Ce Disenchant pourrait avoir la valeur statique la plus faible des cartes en main mais en l'occurrence, on souhaite garder cette carte pour détruire la menace adverse. On dit donc que la valeur de la carte en main est la même que lorsqu'on va la jouer en détruisant ce gros artefact ce qui lui donne une valeur très élevée. À partir du contexte, on a pu sauver la carte qui nous intéressait grâce aux valeurs dynamiques.

2.5.3 La gestion des effets compliqués

Pour les effets fréquents, les valeurs dynamiques sont automatiquement. Néanmoins, à chaque nouvelle extension, de nouvelles cartes sont créées. Pour les cartes vraiment compliquées, il faut dire à la main comment les valeurs dynamiques sont calculées. Notre IA peut donc efficacement jouer avec toute carte qui a été prétraitée, ce qui se fait rapidement. Un travail d'au plus une heure sur chaque nouvelle extension (c'est-à-dire environ tous les 4 mois) permettra de jouer avec toutes les nouvelles cartes. On pourra donc mettre à jour notre IA pour qu'elle gère toutes les cartes.

2.5.4 Réactivité face aux cartes de l'adversaire

L'IA doit aussi s'adapter aux cartes de l'adversaire. C'est probablement le plus difficile car on ne connaît pas les cartes que l'adversaire a en main ou dans son deck. Il est donc quasiment impossible de prévoir ce que l'adversaire va jouer et il faut prévoir une réponse aux effets de l'adversaire quand ceux-ci sont joués. Tout d'abord, il est comme d'habitude difficile de comprendre l'effet joué par l'adversaire.

De plus, lorsqu'un effet est en train d'être traité, on ne peut plus rien faire pour le contrer. Il faut donc réagir juste avec le texte de l'effet avant de savoir ce qu'il fait modifie vraiment dans le jeu. Pour pouvoir néanmoins répondre aux effets de

l'adversaire, certaines cartes sont réactives, c'est à dire que si certains effets sont joués et qu'elles ciblent certaines cartes, celles-ci (ou d'autres) activent l'une de leurs capacités en réponse. Les valeurs des cartes dans la main sont également différentes si elles sont joués en réponse de certains effets particuliers.

2.5.5 Choix des cartes jouées, mémoire et anticipation

Pour choisir quelle carte l'IA doit jouer, elle se base essentiellement sur la valeur des cartes. Néanmoins, on peut aussi se garder une carte pour plus tard même si elle peut nous être bénéfique maintenant. Par exemple, si on utilise une carte qui détruit une créature faible, on est gagnant (l'adversaire a une créature de moins) mais on ne pourra plus utiliser cette carte plus tard alors qu'elle aurait pu nous être plus utile. Il y a donc un seuil de satisfaction qui gère à partir de quel gain il vaut mieux jouer une carte. De même que précédemment, ce seuil est variable en fonction des données du jeu.

Ce seuil pourrait également dépendre du deck joué par l'adversaire. L'IA ne connaît pas toutes les cartes de l'adversaire (ce serait tricher).

Néanmoins, si l'IA joue plusieurs fois contre le même adversaire, il pourrait se souvenir des cartes que celui joue et faire varier les valeurs en fonction. On prévoit donc dans une version ultérieure de doter l'IA d'une mémoire qui évaluera les cartes en fonction du deck de l'adversaire. Plus précisément, l'adversaire aura pour chaque deck qu'il rencontre des paramètres qu'il modifiera en fonction des parties qu'il a joué précédemment contre ce même deck.

2.5.6 Travail futur

En plus de gérer une mémoire à notre IA, nous comptons réaliser toute une capacité d'apprentissage : l'IA pourra jouer contre une autre IA des milliers de parties et modifier certains paramètres en fonction du résultat final de la partie. Les réglages qui offrent le plus de victoires seront conservés. Il sera également possible de donner à l'IA des comportements particuliers, par exemple « timide » ou « agressif ». Ainsi l'IA pourra s'adapter au deck qu'elle joue et utiliser de façon plus naturelle les cartes de son deck. Par exemple, dans un jeu contrôle, les créatures en début de partie sont utilisées uniquement pour se défendre. L'IA ne va donc pas utiliser ces cartes pour attaquer l'adversaire.

De plus, on souhaite réaliser une interface d'aide pour le joueur qui utiliserait l'IA. Lorsqu'un joueur ne sait pas quoi joueur, il pourra demander conseil à l'IA qui lui répondra ce qu'elle aurait fait à sa place.

2.5.7 En pratique

En résumé, notre IA peut jouer plus que convenablement dans de très nombreuses situations. Nous avons joué avec plusieurs types de deck. Le succès a été quasi-immédiat avec les decks contenant beaucoup de créatures. En effet, il s'agit essentiellement dans ces decks de gérer les combats. L'évaluation dynamique a grandement amélioré l'efficacité de notre IA.

Même avec des decks plus techniques, l'IA arrive à de bons résultats même s'il lui manque parfois une vision à très long terme du jeu. De plus, nous n'avons pas trouvé

dans notre IA les défauts qui existent dans s'autres. Par exemple, nous avons vu dans des programmes commercialisés des IA qui engageaient des terrains sans rien faire par la suite ou alors qui détruisaient des cartes pas du tout menaçantes alors qu'il y avait clairement des menaces plus grandes. Notre IA n'a pas ce genre de défauts et après un grand nombre de parties, nous pensons que l'IA remplit parfaitement son rôle, c'est à dire offrir à un joueur humain un adversaire intelligent avec qui on peut tester ses decks ou juste jouer pour le plaisir.

3 Conclusions

3.1 Quelques problèmes techniques

Si l'on peut croire que tout s'est passé pour le mieux dans le codage de ce logiciel, malheureusement il n'en a pas été le cas. Des gros problèmes liés au CVS (problème de droits, de logs) ont interdit à plusieurs personnes l'accès au fichier pendant une grande partie du début du projet, les empêchant par la même de commiter, et donc de participer activement.

De même, deux coupures importantes du réseau (de plusieurs jours chacune) ont donné lieu à des merges vraiment pas triviaux à résoudre lors des modifications de fichiers par deux membres d'un même module. Cela nous a fait perdre un temps vraiment précieux, et sans ces problèmes nous aurions pu nous concentrer sur un point qui malheureusement n'apparaît pas dans cette version : la migration de serveur.

3.2 Une vision globale

Pour la plupart d'entre nous, mis à part le projet POOGL à trois que nous avions tous fait l'an dernier, c'était notre premier projet à plusieurs. Il est apparu premièrement que la division du travail en sous-modules a été peu respectée en pratique : on l'a vu, chaque module a des liens très forts avec tous les autres, et il a fallu une véritable cohésion interne au projet pour aboutir à la version finale. Non seulement la division du travail en sous-projets est intervenue beaucoup trop tôt dans le développement (dès la présentation initiale du projet), ce qui a empêché de se répartir le travail en connaissance de cause puisque personne ne pouvait avoir de vision d'ensemble claire du projet, mais en plus il a fallu sans cesse que les membres du projet sortent du cadre restreint de leur module pour faciliter le travail des membres d'autres modules.

Effectivement, s'il n'est pas du ressort du module interface de séparer les cartes en différentes zones et de traiter leurs effets séparément, il faut tout de même qu'il comprenne ce qu'il code pour afficher des menus les plus intuitifs possibles par rapport à ce que l'utilisateur attend d'un tel logiciel. De même il a besoin de comprendre la façon d'utiliser les messages du module réseau pour envoyer les bons popups au bon moment. L'utilisation de classes bien nommées et d'un code propre et commenté a beaucoup aidé à passer le moins de temps possible au moment du linkage et du débogage du projet (choses qui se sont d'ailleurs faites naturellement tout au long du projet et non pas simplement à la fin).

3.3 Des conflits

Il est ressorti clairement du projet que tout le monde n'est pas à même de coder efficacement, ce qui était bien sûr prévisible. Ainsi pour optimiser au mieux les capacités de chacun, il a fallu là encore une vision globale du projet. Les objectifs ont été clairs et bien fixés dès le départ, ainsi il a été possible de bien coder, mais certaines erreurs de visée ont conduit à quelques déboires, naturellement. Ce d'autant que les échéances ont été vraiment nombreuses, fractionnant d'autant le travail de longue haleine en *rush* à court terme.

Le fait d'avoir une démo bizarrement planifiée aux deux tiers seulement du projet nous a forcés à colmater de rustines des points de règle non encore interfacés, et à programmer en désespoir de cause une interface pour avoir quelque chose à montrer, ce qui a conduit à une prestation assez peu probante au niveau du moteur même du jeu, alors qu'un gros travail de fond avait pourtant été accompli. Temps une fois de plus précieux que nous avons perdu au lieu de linker proprement et calmement comme pendant les vacances de Noël et début janvier. Au final, nous regrettons amèrement de ne pas pouvoir organiser de démo à la fin du projet, pour être à même de présenter le travail réalisé pendant le dernier mois du planning.

Enfin, le travail à 10 n'est une chose pas triviale à gérer pour l'ensemble de l'équipe. Pour que chaque membre du projet garde à l'esprit une idée claire sur l'ensemble du projet, et ainsi soit pleinement efficace (il faut à tout prix éviter d'avoir à repasser derrière du travail déjà fait), il a fallu parfois perdre du temps sur des points mineurs pour des conflits d'idées ou des incompréhensions. C'est toute la différence avec un projet dans lequel un leader a une idée précise du résultat et des étapes du développement, est à même de dire comment faire les choses en cas de soucis, et aux décisions duquel on doit se plier. La plupart du temps les avis divergents n'étaient pas inintéressants mais néanmoins peu utiles compte tenu du temps limité qui nous était imparti. De plus ce sont ceux qui avaient la vision la plus claire des événements à venir qui obtenaient l'accord final des autres membres du projet, mais après de dures batailles très consommatrices en temps et génératrices d'énervements inutiles. En bref, pour mener à bien un projet aussi « unitaire » (dans le sens où un découpage en modules est artificiel aux vues de l'imbrication des modules), avec un effectif aussi important, et dans des délais aussi courts, nous aurions dû adopter d'entrée de jeu une organisation nettement plus hiérarchisée.

Nous avons regretté que les exigences scolaires ou personnelles de chacun réduisent comme peau de chagrin le temps dont nous disposions pour nous réunir à 10. À l'inverse d'un projet logiciel de la « vraie vie », il était impossible à *tous* les membres de se rencontrer à heures fixes pendant des plages horaires de plus d'une heure ou deux, ce qui a décuplé les phases de communication du projet, puisque chaque membre devait répéter de nombreuses fois les explications *détaillées* sur son travail : scénario impensable dans un contexte professionnel normal.

À de nombreuses reprises, il s'est avéré beaucoup plus efficace à certains de réécrire certaines parties du code à leur manière – réécritures sources d'encore bien des frictions. . .– plutôt que de parvenir à trouver le temps pour en parler avec la personne concernée et comprendre comment l'utiliser ! De même, certaines personnes n'avaient pas suffisamment de recul par rapport à l'ensemble au projet pour programmer du code bien adapté au projet, ce qui a donné lieu à de nombreuses retouches et corrections qui auraient facilement pu être évitées si, au début du projet, on avait pu discuter de l'architecture du logiciel au lieu de démarrer d'emblée avec un découpage

en sous-modules très rigide.

Pour finir, par manque de temps et de créneaux horaires communs, les discussions où chacun peut donner son point de vue sur des points importants d'architecture du code n'ont pas pu être mises en place à grande échelle, sachant que le temps du lundi matin étant très insuffisant et beaucoup plus réservé à une activité de compte rendu qu'à une véritable avancée du projet.

3.4 Des objectifs atteints

Pour finir, il est bon de souligner que quels qu'aient été les différends au sein de ce projet, quelles qu'aient été les difficultés auxquelles nous avons été soumis, nous avons rendu un logiciel qui fonctionne, conforme aux objectifs pourtant ambitieux :

- créer une interface graphique digne des programmes utilisés en ce moment sur Internet pour jouer ;
- vérifier les règles à tout moment ;
- proposer une intelligence artificielle pour jouer ;
- proposer une architecture réseau adaptée permettant aux joueurs d'échanger des données de jeu.

Chacun de ces objectifs a visé à une simplicité d'utilisation du jeu et à une vulgarisation de l'utilisation de l'outil informatique. En effet si l'utilisation de l'ordinateur peut s'avérer rebutante dans un premier abord (nécessité de devoir revenir sur des accords tacites dans la vraie vie par exemple), l'ordinateur apporte néanmoins de grandes simplifications. Nous avons rédigé dans ce but un tutoriel destiné aux joueurs de *Magic: the gathering* qui ne sont pas particulièrement experts en maniement d'un ordinateur.

Nous avons apporté un soin particulier à la prévision d'une maintenance soutenue du programme. De nombreux points de règles compliqués ont été prévus malgré le fait qu'aucune carte codée ne les utilise. Nous gérons ainsi les 15 événements différents (liste à ce jour exhaustive) pouvant se produire dans *Magic: the gathering* (un joueur pioche ou se défausse, perd ou gagne des points de vie, un permanent est engagé, attaque, etc.) même si seulement 32 cartes les utilisent (Howling Mine, Soul Warden par exemple). Mais nous avons l'assurance qu'il n'y a plus qu'à coder dans chaque carte la capacité liée à cet événement, qui sera géré par le noyau dur du code quelle que soit la manière dont cet événement se produit. Nous facilitons la création de cartes avec un coût en X manas, même si seulement 5 d'entre elles sont codées. Les jetons – qui sont des créatures spéciales arrivant par l'intermédiaire d'autres cartes et n'ayant ainsi aucun lien dans la base de données en tant que tels – sont gérés en créant à la volée toutes les caractéristiques dont ils ont besoin. À l'heure actuelle seules 2 cartes de notre base en ont besoin, mais leur création est automatisée.

Le seul point qui nous reste à formaliser est l'utilisation de manas spéciaux (par exemple, les manas de guilde, apparus il y a 3 mois – après le début de notre projet) utilisés dans une trentaine de cartes au total.

De même, nous avons prévu le codage de certaines familles de cartes en réalisant des *fonctions helper* génériques qui, à ce jour, ne sont pas utilisés suffisamment pour justifier leur présence.

Notre projet se base sur un jeu existant. Il va de soi que l'utilisation des logos et autres images associées à ce jeu de cartes est limitée par un copyright. Ainsi, si mise

en ligne du logiciel il y a, il faudra bien voir à ne pas fournir la plupart des petits plus que l'ont fournit en ce moment dans ce logiciel.

3.5 Quelques chiffres pour finir...

Notre base de données contient 7 954 cartes, ce qui correspond à l'intégralité des cartes éditées depuis le début de Magic. Ainsi, même si nous n'avons codé que 312 cartes parmi ces dernières, nous sommes capables de toutes les afficher dans notre éditeur de decks.

Éditeur qui nous a d'ailleurs servi à créer pas moins de 6 decks complet contenant un échantillon parfaitement représentatif des différents decks joués en ce moment. L'un de ces decks, Aluren, a d'ailleurs gagné des tournois majeurs cette année, et, tout joueur expérimenté vous le dira, compliqué à gérer. C'est d'ailleurs grâce à ce deck que nous avons pu tester de façon précise notre algorithme de boucle : nous avons créé une boucle de 14 actions différentes nous permettant de gagner la partie un tour après l'avoir jouée, boucle qui a été exécutée 10 000 fois par nos ordinateurs personnels en 1 minute 30 secondes.

Si le nombre de cartes peut sembler restreint, en revanche le nombre de cas que nous avons traité est énorme : nous avons codé en dur les 48 capacités à mot-clef de Magic, ce qui automatise à chaque fois un nombre considérables de cartes. Nous avons ainsi privilégié le travail de conception (faciliter le codage des cartes) au travail quantitatif (coder le plus de cartes possibles, à la chaîne).

Un joueur occasionnel peut donc d'ores et déjà tester plusieurs decks récents, et quant au joueur expérimenté qui a besoin de cartes spécifiques, il ne nous reste plus qu'un travail long – mais rendu très facile – pour coder ces cartes spécifiques !