RAPPORT DE PROJET

DOMINIA

Interface Web pour le jeu de cartes Dominion

Réalisé par

Manuel CHATAIGNER

Stéphane WOUTERS

Sous la direction de

Victor POUPET

Pour l'obtention du DUT Informatique

Année universitaire 2012 - 2013

REMERCIEMENTS

Nous remercions premièrement notre tuteur Victor POUPET pour nous avoir proposé un sujet complet, intéressant et comprenant divers aspects du développement tels que le moteur de jeu, une interface web, la communication client-serveur ou encore la gestion d'un ensemble de cartes. Nous le remercions également de nous avoir expliqué le but à atteindre et les bases d'un projet bien construit, de nous avoir bien conseillé et aiguillé quand nous étions confrontés à un problème quel qu'il soit, tout en nous ayant toujours laissé libre de faire nos propres choix de conception et de solutions.

Nous tenons également à remercier l'ensemble des personnes s'étant prêtées au jeu et ayant effectué les tests avec nous afin d'atteindre le nombre maximum de quatre joueurs sur une même partie, ainsi que pour le lancement de plusieurs parties différentes simultanément.

SOMMAIRE

TABLE DES FIGURES

GLOSSAIRE

Les termes définis dans ce glossaire sont identifiables dans le corps du texte au moyen d'un astérisque (*).

Deck : Pile de cartes personnelle au joueur. Dans le jeu Dominion, il est formé au fil de la partie.

Défausse : Pile de cartes dans laquelle le joueur ajoute les cartes qu'il ne veut plus garder.

HTTP : Protocole de communication web. Il est possible d'émettre des requêtes HTTP en JavaScript grâce à l'Ajax.

jQuery : Bibliothèque qui a pour but de simplifier des commandes communes de JavaScript

DOM (Document Object Model) : recommandation du W3C permettant à des programmes informatiques et à des scripts d'accéder ou de mettre à jour le contenu, la structure ou le style de documents XML. Le document peut ensuite être traité et les résultats de ces traitements peuvent être réincorporés dans le document tel qu'il sera présenté.

INTRODUCTION

Dominion est un jeu de carte de stratégie où chaque joueur forme son propre Deck* au cours de la partie, le but du jeu est d'évoluer plus vite que ses adversaires.

Peu de versions virtuelles sont proposées, car il est difficile de donner l'impression physique des jeux de cartes. De plus, tous les jeux de cartes n'ont pas des règles imaginées pour l'implémentation informatique. Par exemple, certains jeux possèdent plusieurs extensions, chacune ayant ses propres impacts sur les règles de bases. Dominion a déjà été virtualisé à plusieurs reprises mais ces versions sont soit incomplètes, soit trop éloignées de l'aspect réel des jeux de cartes.

Le but de notre projet est donc d'implémenter le jeu de cartes Dominion en ligne. L'ensemble des actions possibles étant trop important, nous allons nous concentrer sur les fonctionnalités de base afin d'obtenir un moteur de jeu fonctionnel et modulable, agrémenté d'animations, afin de le rapprocher au maximum du jeu réel et de retirer l'aspect instantané de l'informatique.

Dans ce rapport il sera d'abord listé les besoins du projet dans un cahier des charges ainsi qu'une analyse du sujet. Ensuite nous détaillerons d'un point de vu technique ce qui a été conçu entre architecture client serveur, moteur de jeu et interface. Un bilan sera fait sur le résultat obtenu, et ce qui est possible d'améliorer. Viendra ensuite un manuel d'installation et d'utilisation de notre production. Pour finir, notre démarche sera détaillée dans les étapes clés de la conception, entre découverte des technologies, réflexions et méthodes de travail.

1. CAHIER DES CHARGES

1.1. ANALYSE DU SUJET ET DE SON CONTEXTE

Il existe peu de versions informatisées de Dominion jouables en ligne : celles-ci sont soit incomplètes, soit peu représentatives d'une partie réelle.

Par exemple, Isotropic a implanté une première version de Dominion en ligne, par navigateur. Cette version est complète, elle implémente la totalité des règles et toutes les cartes sont fonctionnelles. Cependant, elle apporte une interface éloignée de la réalité.

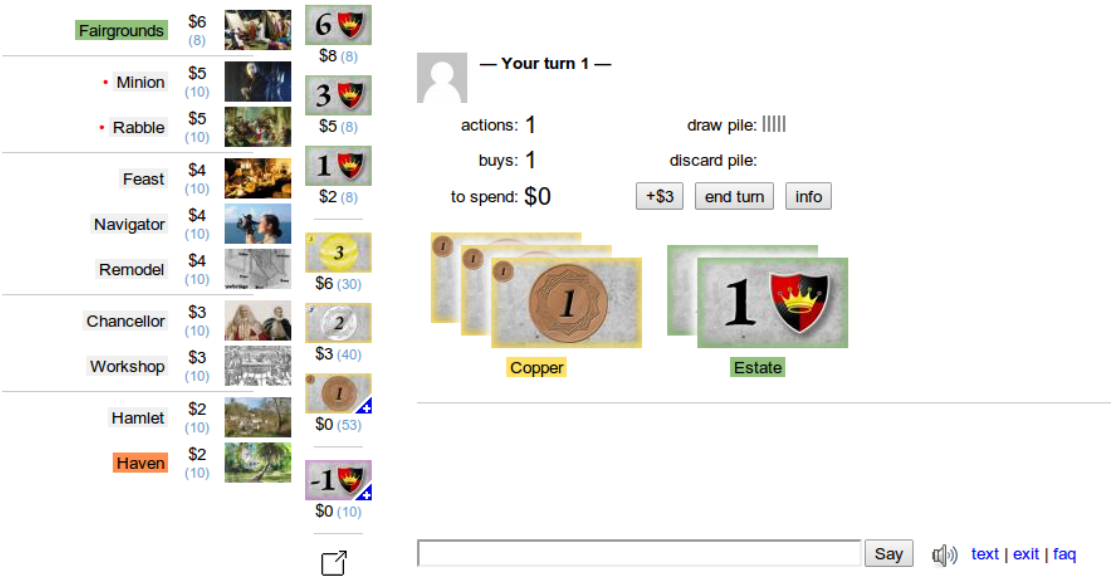


FIGURE 1 - INTERFACE ISOTROPIC

En effet, on voit que l'interface va au plus simple : il y a peu voir pas de graphismes et seul l'aspect pratique est mis en avant : les actions sont instantanées et sans animations, on est loin de l'aspect physique des cartes et proche de l'informatique : l'interface a été trop informatisée et seuls les éléments nécessaires ont été présentés.

Par exemple, bien que Dominion soit un jeu multi-joueurs, l'ascension se fait de façon solitaire. De ce fait, Isotropic a complètement masqué la présence des adversaires alors qu'ils ont une place importante dans le cas d'un jeu de cartes.

Ensuite, une version officielle a été mise en place. Cependant elle n'est pas fonctionnelle et pas assez aboutie. Elle n'a donc pas été finalisée.

1.2. ANALYSE DES BESOINS FONCTIONNELS

Dominia doit être une version virtuelle de Dominion en ligne, jouable par navigateur et à plusieurs. Chaque joueur doit pouvoir créer une partie, rejoindre ses amis ou les inviter. Plusieurs parties doivent pouvoir être lancées simultanément.

Une interface proche du plateau de jeu réel de Dominion doit être présentée, ainsi que des animations qui permettront de comprendre les mouvements et de suivre facilement l'avancement de la partie.

Voici un exemple de disposition des piles :

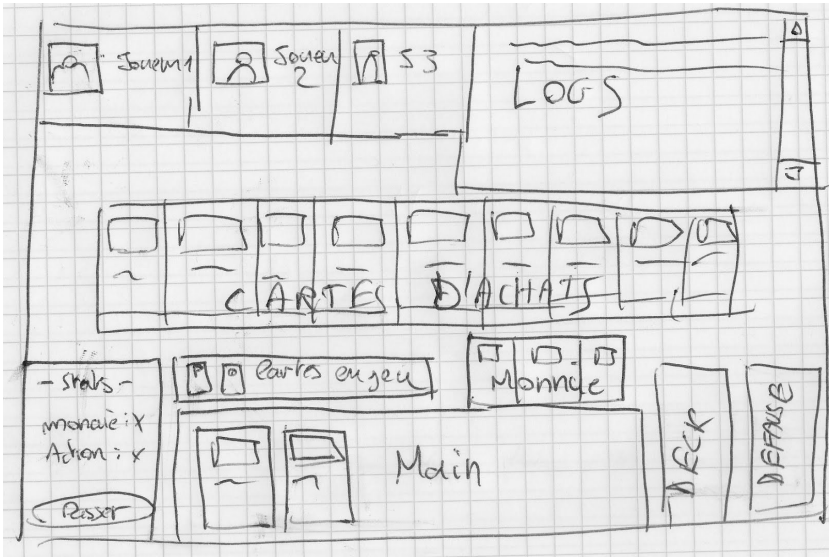


FIGURE 2 - INTERFACE PAPIER

Il faut d'abord comprendre avant tout le déroulement d'une partie du jeu Dominion et en particulier le déroulement d'un tour de jeu. Voici donc un diagramme de séquence représentant les différentes étapes du tour d'un joueur (annexe 2).

Le tour d'un joueur est décomposé en trois phases qui doivent être effectuées dans un ordre précis : Au début du tour, une action et un achat sont possibles. Ces nombres seront ensuite modifiés pendant le tour ou par les autres joueurs.

Le joueur peut donc jouer des cartes actions, dépenser ses cartes de monnaie, acheter des cartes, lancer des attaques durant son tour de jeu, cependant ces évènements doivent être exécutés dans un ordre précis :

- Phase Action durant laquelle le joueur actif joue une ou plusieurs cartes action
- Phase Monnaie, le joueur joue ses cartes monnaie
- Phase Achat où il achète une ou plusieurs cartes parmi celles disponibles

1.3. ANALYSE DES BESOINS NON-FONCTIONNELS

L'ensemble des cartes de Dominion étant trop important, elles ne seront pas toutes implantées. Cependant, l'objectif sera de réaliser un moteur de jeu modulable, ce qui permettra d'ajouter par la suite d'autres règles et effets facilement.

Le projet n'est pas commercial et donc les images des cartes officielles seront utilisées.

Le jeu se doit d'être sécurisé. Le client étant développé en JavaScript, le serveur doit vérifier la totalité des requêtes envoyées. Le client sera compatible avec tous les navigateurs récents. (Internet explorer supérieur à la version 6). Le jeu doit apporter les performances nécessaires à une communication par Internet fluide.

2. RAPPORT TECHNIQUE

2.1. CONCEPTION

2.1.1. ARCHITECTURE CLIENT SERVEUR

2.1.1.1. BASE COMMUNICATION CLIENT SERVEUR

Pour les besoins du projet, il faut que le client puisse communiquer en temps réel avec tous les autres joueurs. Le client ne peut communiquer qu'à travers des requêtes HTTP*, de ce fait le serveur Java utilise le protocole HTTP*. Une solution de communication synchrone est conçue pour cet échange.

Un unique serveur a été développé. Celui-ci reçoit la totalité des requêtes HTTP* réalisées par tous les clients connectés. Dès qu'un client charge l'interface de jeu, il est immédiatement connecté à ce serveur.

Nous avons donc intégré un serveur simplifié HTTP* à notre serveur Java. Celui-ci ne gère que des requetés de type GET. Le client passe ses messages uniquement par l'url, le serveur répond en écrivant sa réponse dans le corps du fichier envoyé. Par exemple, si le serveur tourne sur l'IP 127.0.0.1 avec le port 6020, pour envoyer le message « test » au serveur, le client fait la requête suivante : *« http://127.0.0.1:6020/test ».* S'il y a plusieurs composants pour le message, on utilise le séparateur « : ». Le client récupère la réponse du serveur dans le contenu de la page reçu.

2.1.1.2. CONNEXION AU SERVEUR

Avant d'échanger des messages avec le serveur, le client doit générer sa connexion. Le client envoie donc une première requête « http://127.0.0.1:6020/_:NEW:[pseudo]». Le client spécifie le pseudo qu'il souhaite. Le serveur reçoit ce message, il sait qu'il doit générer une clé unique pour le client. Par exemple : K15479. Il répond à la requête client en retournant cette clé.

Le client va ensuite devoir ajouter cette clé à chacun de ses messages futurs pour s'identifier, avec la syntaxe *http://127.0.0.1:6020/_:K15479:packet*

A partir de là, il existe deux façons de communiquer :

- Pour recevoir un message, le client envoie le message *http://127.0.0.1:6020/_:K15479:R*. Le fonctionnement de réponse sera expliqué dans la partie suivante. Le contenu de la réponse peut correspondre au résultat de notre action, de l'état de la partie, etc.…
- Pour envoyer un message, le client utilise la syntaxe suivante : *http://127.0.0.1:6020/_:K15479:S:message*. Le serveur répondra directement par « OK » quel que soit le message envoyé.

2.1.1.3. UN ECHANGE SYNCHRONE

Nous avons mis en place notre propre solution d'échange synchrone entre le client et le serveur : Le serveur peut envoyer quand il le souhaite des informations au client.

La requête *http://127.0.0.1:6020/_:K15479:R* est en réalité une requête d'attente. Lorsque le serveur reçoit cette requête, il identifie le client, regarde s'il y a un message à lui envoyer, répond au client par ce message si c'est le cas. S'il n'y a pas de message à envoyer, la requête est mise en attente jusqu'à ce que ce soit le cas.

Le client doit donc charger constamment cette requête de réception. Dès qu'il reçoit quelque chose, il doit relancer cette requête. De cette façon, le serveur peut envoyer quand il le souhaite un message au client.

Voici un petit aperçu algorithmique de notre solution, à lancer côte client :

TABLEAU 1 - BOUCLE DE RECEPTION

```
while (true)
{
       // Ici, fonction bloquante tant que le serveur ne finalise pas la requête
       paquet = get(« http://127.0.0.1:6020/_:K15479:R »);
       traiterPaquet(paquet);
}
```

2.1.1.4. LA DECONNEXION RECONNEXION

Pour des questions de fiabilité, il faut que le client puisse se déconnecter puis se reconnecter à la partie. En cas de perte de connexion internet, ou de bug de l'interface.

Pour cela, le pseudo est utilisé pour retrouver la clé du client. Au moment de la réception du paquet « NEW », le serveur regarde si ce pseudo n'existe pas déjà dans les anciennes clés. Si c'est le cas, il retourne la clé du client et celui-ci retrouve l'état exact de la partie dans laquelle il était.

De cette façon, le joueur peut même changer de pc au milieu d'une partie.

Pour des questions de sécurité, on pourrait facilement ajouter un mot de passe en plus du pseudo pour que n'importe qui ne puisse pas accéder à la clé privée de n'importe qui.

2.1.1.5. SYNCHRONISATION DE LA PARTIE

Il existe plusieurs types de paquet qui font évoluer l'état de la partie, et cela est valable pour le serveur et le client.

Parmi les paquets envoyés par le serveur, on peut recevoir des informations sur l'état de la partie, les logs, les choix, etc.

Voici un tableau récapitulatif des informations envoyées par le serveur :

E : [XML]	Contient toutes les informations sur l'état de la partie sous la forme d'un arbre XML. Il peut contenir la liste complète des logs (en cas de première connexion à l'interface). Il est envoyé à chaque modification de l'état de la partie, du joueur ou des

	adversaires
C:[typeChoix]: **[informations]**	Informe le joueur qu'il doit faire un choix. Souvent envoyé après avoir joué une carte action. Le type de choix est numéroté de 1 à n. Quelques exemples : • Type 1 : Choisir une carte parmi la liste donnée • Type 2 : Choix une ou plusieurs cartes parmi la liste donnée • Type 3 : Choix oui/non avec question en paramètre
Z :infos partie	Contient la liste des parties actuelles, créées ou en cours de jeu.
L –contenu	Contient une ligne de log

Le joueur informe ces actions par les paquets suivants :

B-nCarte	Prévient le serveur d'une tentative d'achat de la pile kingdom « n »
A-idCarte	Prévient le serveur d'une tentative de jouer une carte de la main.
C-[infos]	Retourne des informations sur un choix réalisé
P	Passe le tour du joueur actuel

Paquets de gestion des parties, envoyés par le client :

N-nbrJoueur	Crée une partie avec le nombre de joueurs demandés
Z	Demande la liste des parties
J-nPartie	Demande à rejoindre une partie

Tous les paquets peuvent être envoyés par le client n'importe quand, ces actions sont donc toutes vérifiées par le serveur. Si le joueur clique sur une carte de sa main alors que ce n'est pas son tour de jeu, le paquet est de toute façon envoyé. Par contre il sera absorbé par le serveur et n'aura pas d'impact sur l'état futur de la partie. Le client ne se préoccupe donc pas du tout des tests.

2.1.1.6. GESTION DES PARTIES

Le serveur peut gérer plusieurs parties en même temps. Une interface pour créer des parties et en rejoindre a donc aussi été mise en place.

Lors de sa connexion, le joueur reçoit directement la liste des parties, avec les informations suivantes :

- Nombre de joueurs maxima
- Nombre de joueurs actuel
- Si le joueur est déjà dans la partie (dans le cas d'une déconnexion reconnexion du joueur)

N'importe quel joueur peut créer une partie lorsqu'il le souhaite. Le serveur génère alors une nouvelle partie avec le nombre de joueurs indiqué, vide de base, avec les pseudo suivants : joueur1, joueur2, joueur3… Ces joueurs sont dits « libres », ils ne sont pas liés à un client. La partie est directement lancée. Ensuite la partie apparaît dans la liste des parties. Les joueurs peuvent rejoindre les places libres.

Au niveau conceptuel, un système « joueur, contrôleur » a été mis en place. Lorsqu'une partie à 4 joueurs est générée, quatre objets « joueurs » sont créés. Une classe « joueur » possède un pointeur vers un objet « contrôleur », qui pointe vers rien au départ. La classe « joueur » contient les informations sur le joueur au sens de la partie : Sa main, son Deck*, son tour de jeu, etc. Le contrôleur représente plus le joueur au sens physique, celui qui contrôle le joueur.

Lorsqu'un client rejoint une partie, le contrôleur du client est relié au joueur, le joueur n'est donc plus « libre ». L'ancien pseudo provisoire « joueurX » est replacé par le pseudo du client.

Un contrôleur ne peut pas contrôler plusieurs joueurs (donc jouer plusieurs parties en même temps), et un joueur ne peut pas avoir plusieurs contrôleurs.

2.1.2. CONCEPTION MOTEUR DE JEU

2.1.2.1. FONCTIONNALITES DE BASE

Parmi ce qui a été développé au cours du projet, le serveur Java lui s'occupe de la totalité des joueurs connectés ainsi que des parties jouées. C'est donc à celui-ci d'implémenter la totalité des composantes et des règles du jeu.

Il y a plusieurs éléments qui composent le jeu de cartes, le plus important d'entre eux est le joueur. Une classe est prévue pour contenir toutes les informations sur chacun d'eux. Elle possède les informations de base telles que le nom du joueur et la partie dans laquelle il se situe, ainsi que tout son jeu, composé de l'ensemble des cartes.

Pour stocker les cartes, une classe « Pile » est créée. Cette classe comporte également les méthodes permettant d'effectuer les actions nécessaires au joueur sur ces piles, comme piocher la première carte ou ajouter des cartes en fin de pile. Le joueur possède donc plusieurs piles, parmi elles on trouve le Deck*, la défausse*, sa main ou encore les cartes posées sur le plateau.

L'ensemble des joueurs est organisé en parties, comprenant les joueurs qui la composent et les piles de cartes communes à la partie comme les piles d'achats. Un attribut contient le numéro du joueur actif. C'est avec celui-ci que la liste des joueurs est parcourue en boucle.

Afin de communiquer avec le client par navigateur, chaque joueur possède un contrôleur. C'est celui-ci qui est chargé des échanges client-serveur effectués via le protocole HTTP*. Il envoie au client les demandes de choix et récupère les actions du joueur.

2.1.2.2. GESTION DES CARTES

Un des aspects les plus importants du serveur est la gestion de l'ensemble des cartes du jeu. Celui-ci possède des cartes de base présentes en petit nombre mais également des cartes appelées action et dont chacune possède ses propres effets.

Il y existe donc une classe abstraite "Carte" qui contient les propriétés communes à chaque carte (annexe 3).

Le projet possède aussi une classe pour chaque carte du jeu, qui hérite directement de la classe abstraite et redéfinit les méthodes pour chaque propriété. Celles de base comprennent l'identifiant de la carte, son nom et son prix. Concernant ce dernier point, il y a deux méthodes : celle correspondant au prix indiqué sur la carte et celle pouvant être affecté en jeu par des cartes jouées.

Les cartes sont divisées en trois principales catégories :

- les cartes action
- les cartes trésor, qui ajoutent de l'argent
- les cartes victoire, qui attribuent des points victoire

Il existe également d'autres types de carte. Malédiction, qui est un type de victoire retirant des points, le type attaque correspondant à une action s'appliquant aux autres joueurs ainsi que le type réaction, action pouvant faire effet lorsque le joueur reçoit une attaque.

Une carte peut posséder plusieurs types, par exemple une carte action-attaque possède une action qui est effectuée lorsque le joueur utilise la carte et une attaque lancée à chacun des joueurs.

Une méthode retourne donc le ou les types que possède une carte.

Aux trois types de cartes principaux correspondent des méthodes du même nom, de même que pour les catégories attaque et réaction. Par défaut dans la classe mère, les méthodes action, attaque et réaction sont vides, quant à elles les méthodes trésor et victoire retournent 0.

Ces méthodes sont directement surchargées dans chaque classe en fonction du ou des types de la carte. De ce fait les cartes victoires et trésors sont facilement créées, elles surchargent la méthode correspondante et retournent la nouvelle valeur.

2.1.2.2.1. CARTES ACTIONS

Au contraire des cartes victoires et trésors, les cartes actions sont en grand nombre dans le jeu de cartes et chacune d'elle est unique. Les différentes possibilités que proposent ces cartes varient et ces cartes se doivent d'ajouter un bon nombre de fonctionnalités supplémentaires au serveur. Il existe donc différents stades de création suite aux fonctionnalités imposées par ces cartes, qui ajoutent augmentent le nombre de possibilités du joueur, lui demandent un choix ou font intervenir les autres joueurs.

2.1.2.2.2. CARTES BASIQUES

Parmi l'ensemble des cartes on trouve tout d'abord les actions augmentant ses capacités de jeu en ajoutant des achats, actions, argent ou encore des cartes. Ces cartes action font uniquement appel à des méthodes "setters" dont le rôle est de modifier le

contenu des attributs ou bien de faire piocher une carte au joueur. Voici un exemple de l'une d'entre elles (figure suivante).

```
public void action(Player p)
{
        p.drawNCard(1);     // +1 carte
        p.addActions(1);    // +1 action
        p.addBuys(1);       // +1 achat
        p.addMoney(1);      // +1 pièce
}
```

FIGURE N°4 : METHODE ACTION SURCHARGEE DE LA CARTE MARKET.

Ici la carte ajoute action, achat, argent et fait piocher une carte au joueur. Un simple appel aux méthodes qui modifient les attributs ou alors fait piocher une carte suffit.

2.1.2.2.3. CARTES CHOIX

Des cartes plus difficiles à implémenter sont les cartes demandant au joueur de faire un choix. Celui-ci peut se présenter de différentes façons, comme choisir une carte parmi une liste ou une simple question oui/non.

Ces cartes font directement appel aux contrôleurs évoqués plus haut et donc à la communication client-serveur, afin de recevoir une réponse du joueur depuis le client Javascript. Le contrôleur lui envoie en premier lieu le type de choix à effectuer, parmi les différentes possibilités :

- Choisir une carte dans une pile donnée
- Choisir si le joueur le souhaite, une ou plusieurs cartes d'une pile donnée
- Choisir une pile parmi celles communes aux joueurs
- Réponse à une question oui/non

La méthode action de la carte doit ensuite traiter la réponse envoyée par le client et effectuer les actions en jeu en conséquence. La figure suivante présente une carte nécessitant deux types de choix différents. Le joueur doit choisir une carte trésor parmi celles en main s'il en possède au moins une, la jeter et choisir une nouvelle carte trésor coûtant trois de plus parmi les piles disponibles.

```
public void action(Player p)
{
        ArrayList<Card> treasures = new ArrayList<Card>();

      // Pour chaque carte en main, on regarde si elle contient le
       // type trésor.
       for(Card c : p.getMain().cards())
      {
             // Si oui on l'ajoute à la liste.
             if(c.getTypes().contains(TYPE_TREASURE))
                    treasures.add(c);
      }

       // On demande au joueur de choisir une carte parmi ses trésors.
       treasures = p.getControleur().needChooseCards(treasures, 1,
             "Choisissez une carte trésor à trash");

       // Si on a récupéré son choix.
      if(!treasures.isEmpty())
```

```
	{
		// On récupère depuis sa main la carte choisie et on la jette.
		Card c = p.getMain().draw(treasures.get(0).getId());
		p.trashCard(c);

		// On lui demande de choisir une carte coûtant trois de plus,
		// et on l'ajoute à sa main.
		p.getMain().add(
			p.getControleur().needChooseTreasureCard(
			c.getFixedPrice() + 3,
			"Choisissez une carte trésor coutant "
				+ (c.getFixedPrice() + 3) + " max.")
		);
	}
}
```

FIGURE N°5 : METHODE ACTION SURCHARGEE DE LA CARTE MINE AVEC DEMANDE DE CHOIX.

2.1.2.2.4. CARTES INTERACTIONS

Un troisième type d'action correspond aux interactions entre joueurs. Lorsqu'un joueur joue une carte de ce type, le serveur doit alors, pour tous les joueurs de la partie, effectuer les indications de la carte jouée.

Toutes les cartes faisant intervenir les autres joueurs ne comportent pas toutes le type attaque, et dans ce cas l'action sur les autres joueurs est effectuée dans la méthode action de la carte.

Mais ce type de carte est dans la plupart des cas considéré comme une carte attaque, et ici en revanche, l'attaque est lancée via une méthode prévue à cet effet. Il est possible lorsqu'un joueur reçoit une attaque, de jouer une carte comportant le type "réaction" prévue à cet effet.

La méthode de lancement d'attaque prend donc en paramètre la carte et exécute sa méthode attaque pour chacun des joueurs en envoyant en paramètre le lanceur et la cible, ainsi qu'en vérifiant si le joueur ciblé possède une carte réaction et souhaite la jouer (annexe 4).

La méthode demande à chaque joueur s'il possède et souhaite jouer une carte réaction. Ce test est donc effectué juste avant le lancement d'une attaque sur un joueur. Le serveur pose la question au joueur, si oui ou non il possède et souhaite jouer une carte réaction, lorsque celui-ci est ciblé. Si oui alors le serveur va vérifier s'il possède effectivement ce type de carte et va lui demander laquelle il souhaite jouer. La carte réaction est ensuite exécutée, suivie par l'attaque lancée, sauf cas particulier comme celui de la carte "Douves" dont l'effet est d'annuler l'attaque. Le cas est ici directement traité dans la méthode présentée.

Dans le cas où l'attaque demande un choix au joueur cible, le contrôleur de chaque joueur sera appelé à tour de rôle de la même manière que précédemment. Il s'agit ici du seul cas où les joueurs sont actifs quand ce n'est pas leur tour de jeu.

2.1.3. CONCEPTION INTERFACE

L'interface est l'affichage visuel d'état de la partie pour le joueur, elle lui permet aussi de déclencher ses actions. Le côté visuel est développé en HTML, css3, JavaScript et jQuery*. La communication avec le serveur est réalisée grâce à la technologique « Ajax » apportée par le JavaScript.

L'interface n'a aucune faculté « intelligente », elle ne fait aucun test et aucune vérification. Elle ne vérifie pas si le joueur peut jouer une carte alors que ce n'est pas son tour. Chaque clique sur une carte envoie le paquet correspondant au serveur. L'interface est rafraîchi à chaque fois que l'état de la partie est reçu.

Cependant elle dispose tout de même de plusieurs animations et n'est pas qu'un rafraichissement complet de l'html. Par exemple pour la main, lorsque l'état de la partie est reçu, un algorithme de comparaison entre l'ancienne main et la nouvelle calcule les cartes à retirer et à ajouter dans l'ancienne main, pour pouvoir jouer des animations.

Il y a donc un côté réflexion sur l'interface mais elle ne s'appuie pas uniquement sur la différence entre les états : Si le nombre de cartes dans une pile d'achat est modifié, ce nombre clignote pour informer le joueur qu'il y a eu un changement.

Ces animations permettent ainsi de suivre l'état de la partie, les cartes se déplacent, apparaissent, les chiffres clignotent, etc. On n'est pas perdu et on peut voir en temps réel les actions de nos adversaires.

Le fond de l'interface a été désigné sur Photoshop, ensuite sont superposées les images des cartes officielles et les chiffres correspondants.

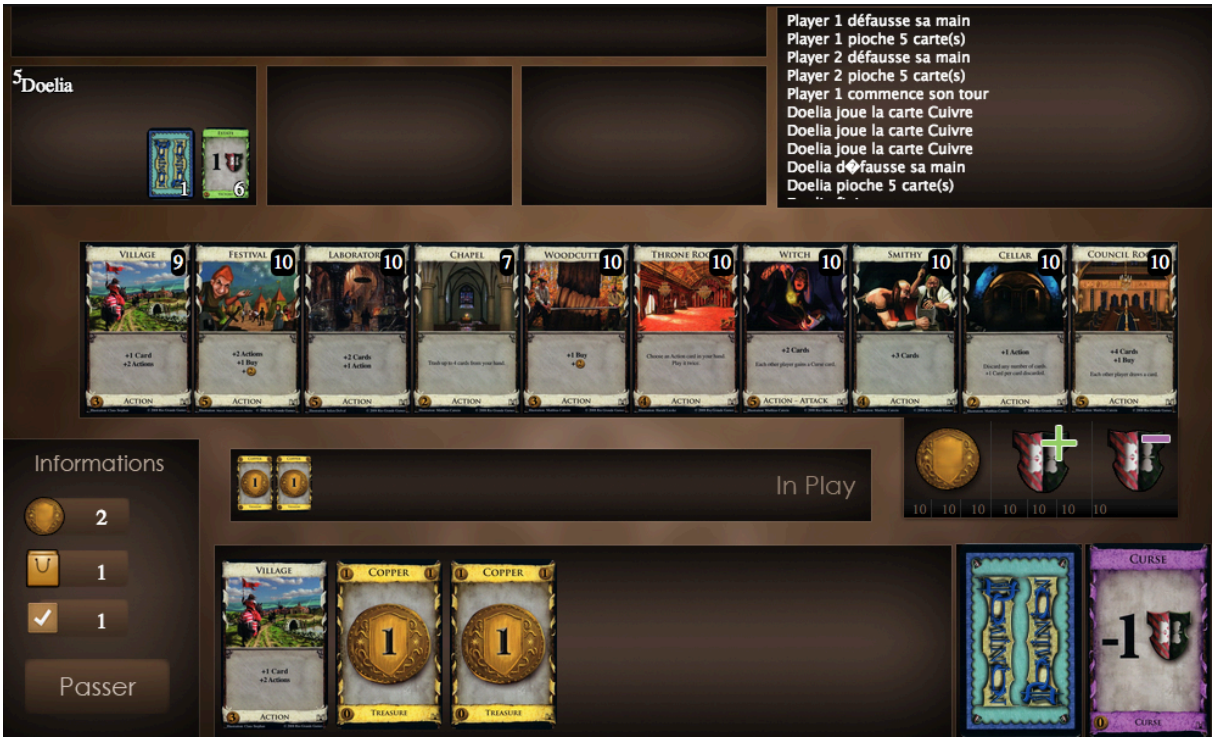


FIGURE 3 - INTERFACE JOUABLE

Un « clic droit » est possible sur n'importe quelle carte de l'interface pour effectuer un zoom sur celle-ci. Un log complet de toutes les actions de tous les joueurs de la partie est disponible et est mis à jour à chaque nouvelle ligne ajoutée.

Les piles centrales ont été clairement affichées en comparaison avec les actions de monnaie et de victoires, qui elles sont toujours les mêmes. Ces dernières sont réduites mais affichées au survol de la souris.

2.2. RESULTAT

Au terme du projet, le jeu de cartes est fonctionnel, une partie peut être jouée entièrement avec les cartes présentes en jeu. Le serveur Java gère la totalité des actions du jeu et effectue tous les calculs nécessaires, le client par navigateur ne s'occupe que de l'affichage. Et le joueur peut suivre l'état de la partie en temps réel. Cette dépendance permet l'évolution du moteur de jeu sans se soucier du client.

L'ensemble des cartes du jeu de base a été développé entièrement et est fonctionnel. Une partie peut donc être jouée comme si le jeu avait été acheté sans extensions.

L'interface apporte toutes les interactions qui existent dans une partie réelle. Elle est suffisamment animée pour pouvoir suivre la partie et rester au courant de nos actions et celles de nos adversaires, comme souhaité au départ afin de se rapprocher au mieux du jeu réel.

Et comme prévu dans le cahier des charges et demandé par notre tuteur, le moteur de jeu a été développé, grâce à une programmation objet, de façon modulable. Il sera donc facile d'ajouter des fonctionnalités ou d'implémenter d'autres cartes.

2.3. PERSPECTIVES DE DEVELOPPEMENT

Par la suite, le jeu peut encore être amélioré. En effet côté serveur, les cartes des extensions du jeu peuvent être ajoutées.

Le moteur a été organisé de façon modulable et extensible, de manière à permettre un ajout facile des autres cartes et des fonctionnalités requises par celles-ci. En effet si par exemple une extension considère un des aspects des cartes de manière différente, un ajout dans la classe mère « Carte » modifie instantanément l'ensemble des cartes du jeu. De même les propriétés d'un joueur ou de la partie peuvent facilement être modifiées.

L'interface est indépendante du moteur de jeu, une modification faite sur le moteur n'a pas besoin d'être réalisée sur le client. On peut très facilement programmer une nouvelle interface sur les bases de l'ancienne.

3. MANUEL D'UTILISATION

3.1. MANUEL D'INSTALLATION

Pour jouer à Dominia, il suffit d'accéder au client qui est accessible facilement sur internet, puisqu'il ne s'agit qu'une page web.

Cependant, il sera ici expliqué une installation complète des modules (lancement serveur, configuration client/serveur), même si seul l'administrateur effectuera cette tâche.

3.1.1. CONFIGURATION SERVEUR

Un simple « .jar » est à exécuter via la commande « *java -jar serveurDominia.jar* ». Par défaut le serveur est lancé en écoute sur toutes les IP de la machine, sur le port 6020. Il est possible de forcer l'adresse ip et de modifier le port via un fichier config.txt à disposer dans le même répertoire que celui du fichier jar avec la syntaxe suivante :

```
host_ip=192.168.1.92
port=6020
```

Une fois lancé, le serveur accepte les connexions et les joueurs peuvent s'y connecter, lancer des parties et les rejoindre.

Un dossier "log" sera crée dans le même répertoire que le fichier .jar, dans lequel sera écrit tous les logs des parties en cours et des évènements.

3.1.2. CONFIGURATION CLIENT

Il faut modifier les sources d'un fichier JavaScript pour spécifier l'adresse IP et le port de connexion. Il s'agit du fichier js/main.js. On peut modifier les deux premières lignes :

```
var host = 'localhost';
var port = 6020;
```

On partage ensuite aux utilisateurs l'adresse de l'emplacement du client html, par défaut : template.html. Il est possible de renommer le nom de ce fichier, en index.php par exemple.

3.2. MANUEL POUR L'UTILISATEUR

3.2.1. REGLES DE BASE DE DOMINION

Dominion est un jeu de cartes où chaque joueur forme au fil de la partie son propre jeu, appelé « Deck* », en achetant les cartes qu'il souhaite posséder et utiliser.

Les cartes se divisent en 3 catégories :

- les cartes monnaie
- les cartes victoire
- les cartes action

Chaque joueur commence avec un minimum de cartes, et doit acheter au choix des cartes victoires (attribuant des points), des actions et/ou de l'argent (pour acheter d'autres cartes). Les cartes à acheter se présentent en piles, chacune d'elle contient plusieurs exemplaires de la même carte.

La partie se termine lorsque 3 piles ont été vidées, le gagnant est celui comptabilisant le plus de points victoires dans son jeu.

Au départ chaque joueur commence avec 5 cartes monnaie valant chacune 1. Chaque tour de jeu se décompose en 3 phases exécutées dans cet ordre :

- Phase Action durant laquelle le joueur actif joue une ou plusieurs cartes action
- Phase Monnaie, le joueur joue ses cartes monnaie
- Phase Achat où il achète une ou plusieurs cartes parmi celles disponibles

Ces actions sont limitées car chaque joueur ne possède de base qu'une action jouable par tour et qu'un achat. Ces chiffres peuvent être affectés par des cartes actions qui augmentent les propriétés du joueur.

Les cartes achetées vont à la défausse*, ainsi que la main du joueur à la fin du tour. Il pioche ensuite 5 nouvelles cartes pour le tour suivant. Une fois le Deck* vide on retourne la défausse* afin qu'elle puisse le reformer. Les cartes sont donc en perpétuelle rotation au cours du jeu.

La stratégie est donc basée sur la formation de son propre Deck* qui doit, à la fois contenir des cartes intéressantes, et à la fois ne pas être trop rempli pour que les cartes achetées puissent rapidement revenir dans le jeu. Quand on achète une carte on prévoit donc qu'elle ne pourra être utilisée que plus tard dans la partie.

3.2.2. PRESENTATION DE L'INTERFACE

Une fois la page html chargée, on entre simplement son pseudo pour accéder à la liste des parties :

FIGURE 4 - LISTE DES PARTIES

On peut rejoindre une partie où il reste de la place. Les boutons jaunes indiquent les parties avec des places libres. S'il n'y a pas de bouton, c'est que la partie est pleine. Si le bouton est rouge, c'est que vous jouez déjà sur cette partie (en cas de déconnexion / reconnexion).

Si toutes les parties sont pleines, vous pouvez en créer une grâce à la partie de droite, de un à quatre joueurs. (Vous pouvez apprendre à jouer en créant une partie solitaire).

Pour ce qui est de l'interface de jeu, les piles d'achat sont au centre. Le nombre en haut à droite désigne le nombre de cartes restantes dans la pile.

La zone "in play" présente les cartes qui sont en cours de jeu, par exemple si vous jouez une carte monnaie, celle-ci sera dans l'état "in play" le temps de votre tour de jeu.

A gauche, la partie "Informations". Elle indique de haut en bas votre monnaie restante, le nombre d'achats possible et le nombre d'actions restantes. Le bouton "Passer" vous permet de passer votre tour. Le temps n'est pas limité pendant les tours de jeu.

Vous trouvez en haut la liste de vos adversaires, avec pour chacun leur Deck* et leur pile de défausse*. Le petit nombre sur chaque pile indique le nombre de cartes restantes. Le joueur actif est en surbrillance.

4. RAPPORT D'ACTIVITE

4.1. LES DEBUTS DU PROJET

Lors de la première séance avec notre tuteur, celui-ci nous a présenté le jeu Dominion que nous ne connaissions pas. Il nous a expliqué les particularités de ce jeu, ses règles ainsi que les différentes possibilités de le concevoir tout au long du projet.

Il nous a également exposé la situation actuelle concernant les versions informatisées du jeu qui existaient et nous a fait parvenir les sites comprenant la documentation nécessaire ainsi que le lien vers un site ayant informatisé le jeu afin que nous puissions y jouer pour le comprendre dans son intégralité.

A la suite de parties jouées sur ce site, le jeu nous est apparu plus clair, et nous avons pu commencer la phase d'analyse.

Celle-ci a débuté par la conception d'un diagramme de classes (annexe 1) mettant en avant les différents éléments du jeu et leur organisation au sein du programme à réaliser. Ce diagramme a été conçu à deux, chacun apportant les éléments retenus de la pratique du jeu. Nous sommes donc arrivés à un résultat regroupant l'intégralité des éléments du jeu.

Une fois ce diagramme terminé, il a été exposé à notre tuteur lors de notre second rendez-vous pour une vérification afin de déterminer si les règles avaient été bien comprises et si les éléments étaient correctement agencés ensemble. Quelques légères modifications ont été apportées suite à la discussion avec M. Poupet. Le diagramme a ensuite été validé pour pouvoir commencer le développement en suivant ce modèle de conception.

4.2. CONCEPTION MOTEUR DE JEU

Pour développer ensemble le projet, nous devions trouver un outil collaboratif adapté. Nous avons donc dès le départ mis en place un serveur Svn qui nous permettrait tout au long du développement le partage du code stocké en ligne. Une fois celui-ci fonctionnel nous avons pu l'utiliser pour commencer la création du projet.

La phase de développement a débuté par la création du moteur de jeu Java. C'est celui-ci qui permettrait par la suite de gérer l'intégralité des parties et des joueurs. Voilà

pourquoi pour commencer nous nous devions d'avoir un jeu de base fonctionnel sans pour le moment s'occuper des fonctionnalités des cartes. Cette version simplifiée devait contenir l'ensemble des actions indispensables au déroulement du jeu telles que la distribution des cartes en jeu, le passage des tours de joueur en joueur ou encore la gestion des cartes en main et en jeu.

Pour ces premiers essais, notre tuteur nous avait recommandé de débuter en affichant l'état du jeu sur une interface console afin de laisser de côté dans un premier temps l'aspect graphique.

Nous avons donc débuté ensemble sur la création des classes prévues suite à la conception du diagramme de classe.

Lors de cette première phase de développement, nous travaillions toujours ensemble, en se répartissant les classes à implémenter, afin d'effectuer au mieux les liens entre chaque objet et pouvoir discuter sur les points encore incertains, toutes les situations de jeu n'étant pas toutes apparues lors des parties effectuées. Nous avons au cours de ces séances, mis en place les objets correspondant à la partie, aux joueurs, aux piles et enfin aux cartes elles-mêmes, sans se préoccuper des effets de chacune des cartes.

Cette phase a duré plusieurs semaines, et pour cause nous devions mettre en place la plus grande partie des fonctionnalités de jeu mis à part les effets apportés par les cartes. Nous allions régulièrement voir notre tuteur pour lui faire part de l'avancement du serveur. Celui-ci a bien sûr su nous conseiller sans pour autant nous imposer une méthode.

Cette partie n'a pas comporté que du développement, en effet une fois l'avancement de plus en plus important, il nous fallait éclaircir des points détaillés imposés par les règles du jeu. Nous avons conçu d'autres diagrammes pendant cette période, tel que le diagramme de séquence du déroulement d'un tour d'un joueur (annexe 2), un ordre précis de différentes phases de jeu étant imposé selon les actions du joueur.

Lorsque le moteur de jeu est devenu fonctionnel, il nous a fallu réfléchir sur le client par navigateur et donc aborder le sujet de la communication client-serveur.

4.3. CONCEPTION ARCHITECTURE CLIENT-SERVEUR

4.3.1. REFLEXIONS SUR L'ECHANGE

Le client étant développé en html/css/JavaScript, nous devions utiliser l'Ajax pour la communication. Le premier serveur qui recevrait toutes les requêtes devrait donc être un serveur HTTP*.

Au départ nous pensions utiliser un intermédiaire (Serveur Apache) qui recevrait l'ensemble des requêtes des joueurs.

Chaque partie aurait été lancée indépendamment dans un processus séparé, et le serveur apache aurait envoyé les requêtes à la bonne partie.

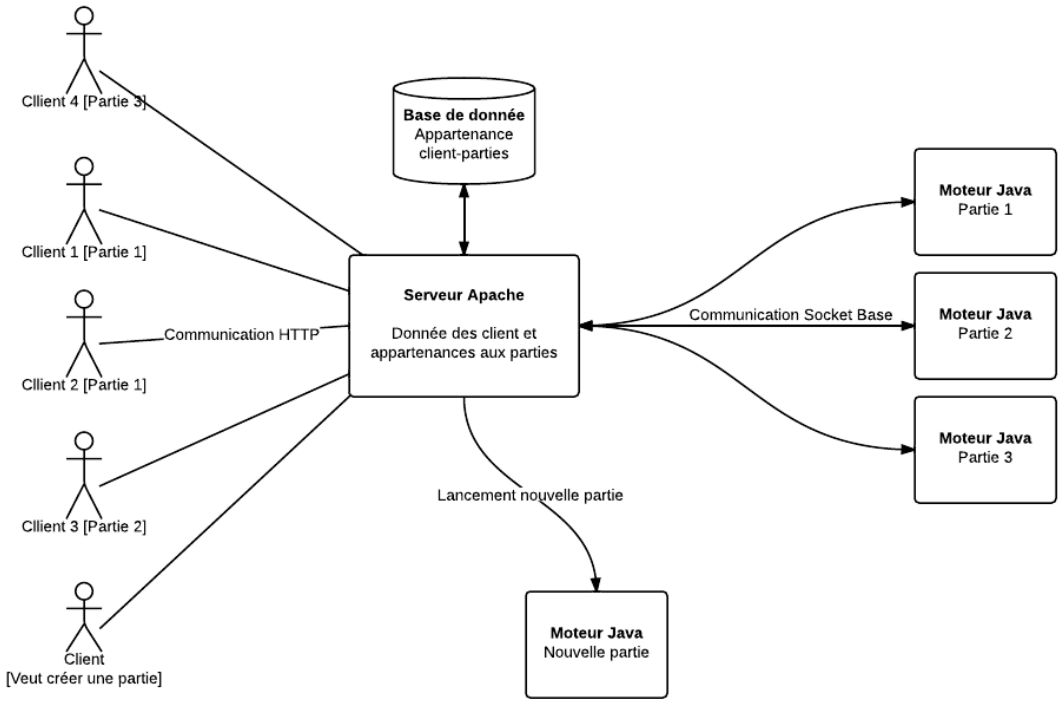


FIGURE 5 - SERVEUR INTERMEDIAIRE

Mais cela aurait engendré plusieurs problèmes : La gestion des processus, l'implémentation d'une base de donnée pour savoir quel joueur appartient à quelle partie, etc.

Après quelques recherches sur le fonctionnement d'un serveur HTTP*, nous avons décidé d'insérer notre propre serveur HTTP* dans le moteur Java.

Nous avons donc utilisé seule application Java qui contient l'ensemble des parties, le serveur HTTP*, et donc les informations sur chacun des clients. Un unique serveur sera utilisé pour la gestion de la totalité des clients, et de toutes les parties en simultané. Les clients se connecteront tous au serveur et le serveur enverra les données nécessaires à chaque joueur.

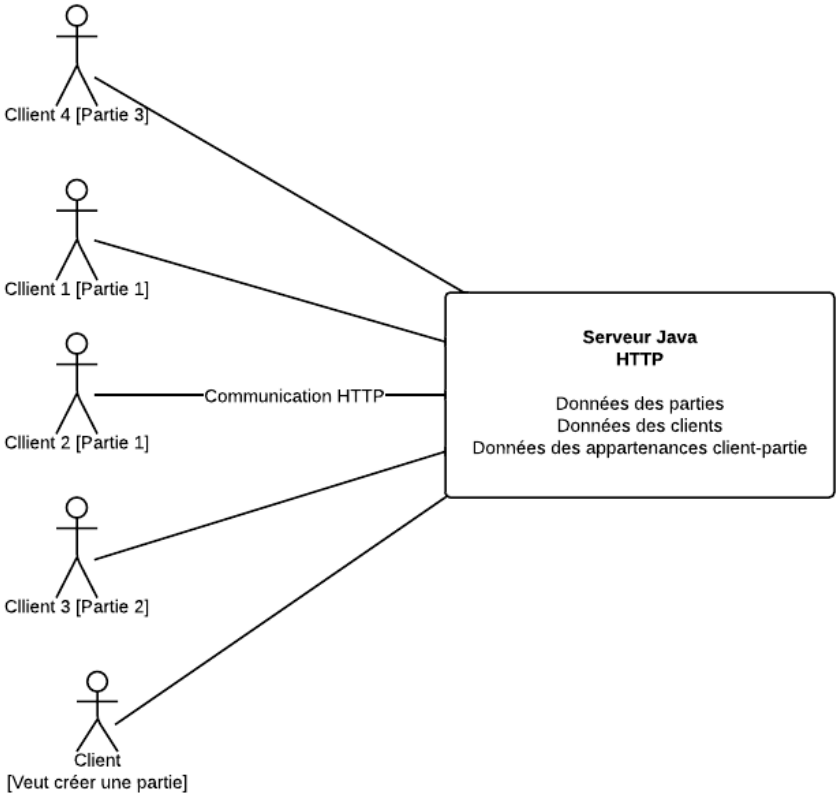


FIGURE 6 - UN UNIQUE SERVEUR JAVA

4.3.2. CREATION SOLUTION SYNCHRONE

Il était nécessaire de développer un échange de communication synchrone, dans le but d'échanger des messages de façon synchronisée. Par exemple le joueur fait une action, et reçoit immédiatement les messages nécessaires, répond en fonction, etc. Bref, il y a un échange.

Le synchrone est très rarement vu dans le monde du web. On ne peut de base pas envoyer des messages à un client sous écoute « bloquante ». La base du web, c'est le client qui demande des informations au serveur, le serveur répond.

Il n'est pas possible de faire les choses dans le sens inverse, le serveur ne peut pas stimuler le client. Il n'existe pas de connexion instantanée et durable stable pour le moment entre le serveur et le navigateur qui proposeraient des systèmes d'écoute. Il existe les Web Socket, objet du web récent. Mais nous nous sommes rapidement aperçu qu'il s'agissait de quelque chose d'assez expérimental.

Nous avons donc réalisé un système de requêtes HTTP* qui se mettraient « indéfiniment » en attente : un serveur apache peut simuler un temps d'attente sur une requête. Par exemple, lorsqu'on appelle un page web, le serveur peut ne pas nous répondre

instantanément, et notre navigateur attend cette page, provoquant une image de « chargement ».

Nous avons eu l'idée de créer notre propre serveur Apache qui jouera sur ces temps de réponses. Le client appelle une page sur notre serveur, il est mis en attente. Pendant ce temps aucune ressource réseau n'est utilisée. Le serveur peut alors envoyer sa réponse à tout moment, et le client la reçoit.

Le principe est de jouer sur ces requêtes HTTP*, pour que l'on puisse envoyer des informations à tout moment au client. Côté client, le but est que dès qu'il reçoit une réponse, il relance à nouveau une requête. Il y aura donc une tache en JavaScript qui s'occupera uniquement de la réception des données.

On peut alors développer côté serveur tout un système de buffer, de test de connexion et de réponses, etc. On peut envoyer des chaines de paquets, et le client les recevra un par un.

Une fois correctement implémenté, on retrouve une solution proche d'une connexion directe avec des sockets. Tout simplement avec une méthode client.envoyer() et une fonction client.onMessage().

4.3.3. LE PROBLÈME DES CHOIX

Si la plupart des paquets sont des causes à effets instantanés (du type : Je joue une carte, elle s'actionne, je reçois le nouvel état de la partie), ce n'est pas le cas des choix. Avec les choix la synchronisation devient plus difficile, il faut attendre la réponse du joueur sans bloquer le déroulement de partie.

Au départ nous n'avions pas vraiment pensé à cela : Le joueur recevait le paquet du choix, et s'il ne répondait pas, aucune autre action n'était possible. De plus il pouvait y avoir plusieurs choix en même temps, de joueurs différents. Et si le joueur quittait la partie et revenait, le choix n'était plus possible puisque le paquet avait déjà été envoyé.

Tout un système de sémaphore et de threads a été mis en place pour pallier à ce problème. Maintenant, une pile de choix est ajoutée à chaque joueur. Une sémaphore pour la réponse du choix est mise en place dans l'exécution de la carte : l'action d'une carte est exécutée dans le thread de traitement du paquet envoyé par le joueur. D'autres paquets sont donc possibles, même si le choix n'a pas été réalisé. Lorsque le choix sera envoyé par le client, le sémaphore sera libéré et la carte continuera son exécution.

4.4. CONCEPTION DE L'INTERFACE

4.4.1. UNE PREMIERE VERSION POUR LES TESTS

Au départ une première version de l'interface a été développée pour les tests. En effet dans ce type de projet, il faut rapidement un affichage visuel pour tester le moteur de jeu : Une interface de « debug » munie d'une console.

La console permet d'envoyer un paquet n'importe quand. Il fallait entrer les paquets pour rejoindre une partie manuellement, étant donné que l'interface pour lister les parties n'existait pas.

Nous avons développé 80% du moteur avec cette interface. Celle-ci était en JavaScript uniquement, et nous a permis de découvrir la lourdeur de cette première version au niveau des algorithmes utilisés.

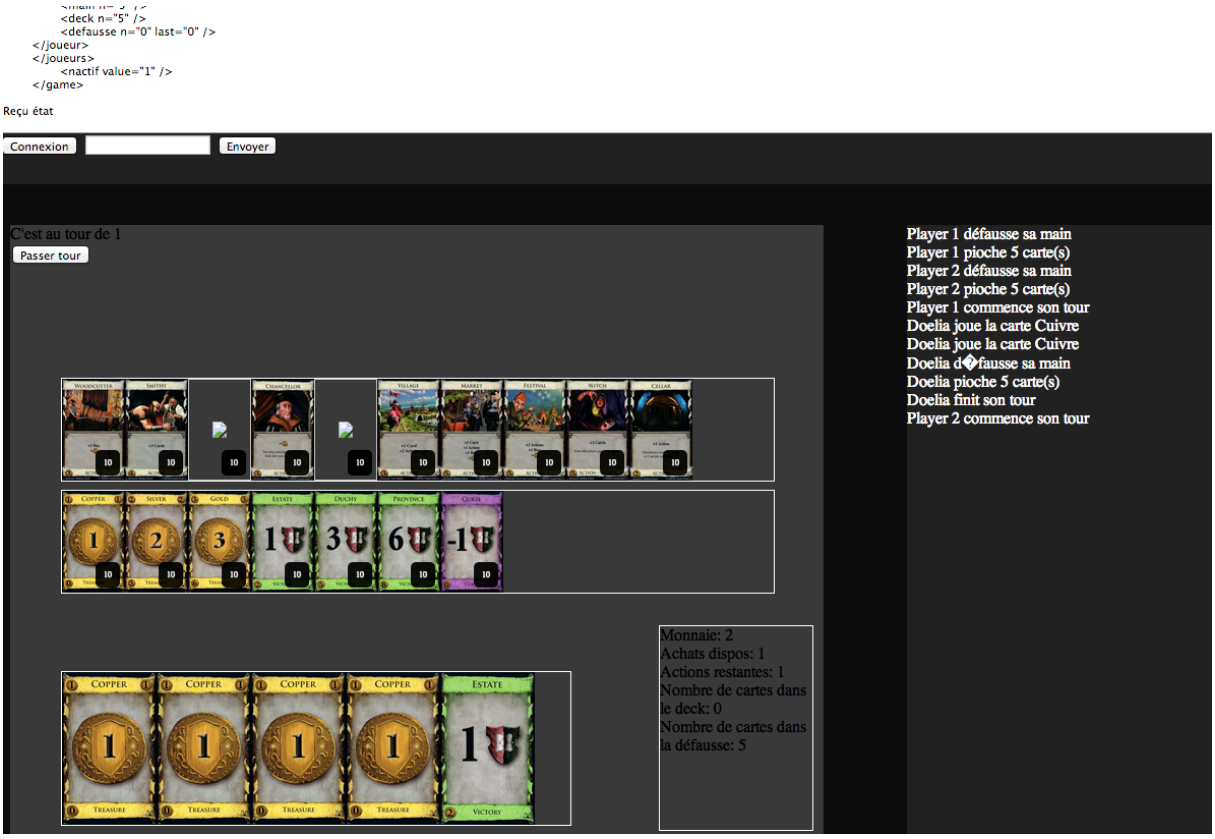


FIGURE 7 - INTERFACE VERSION 1

4.4.2. TRAITEMENT DE L'ETAT DE LA PARTIE

L'état de la partie est reçu sous forme XML (annexe 5), et peut être reçu plus d'une fois par seconde. Il faut donc une méthode de rafraichissement rapide et optimisée. Heureusement le JavaScript possède des méthodes efficaces pour le parcours d'arbre XML, car l'html est avant tout du XML. Voici un extrait de l'algorithme final du parcours du DOM* :

```
xml = StringtoXML(packet);
var racine = xml.documentElement;

for (var i in racine.childNodes)
{
        var e_i = racine.childNodes[i];
        if (e_i.nodeName == 'me')
        {
                idMe = e_i.attributes.getNamedItem("id").nodeValue;

                for (var j in e_i.childNodes)
                {
                        var e_j = e_i.childNodes[j];

                        if (e_j.nodeName == 'inplay')
                        {
                                inPlay.resetCards(e_j.childNodes);
                        }

                        if (e_j.nodeName == 'main')
                                main.resetCards(e_j.childNodes);

                        if (e_j.nodeName == 'kingdom')
                                setKingdoms(e_j.childNodes);

                        if (e_j.nodeName == 'other')
                                setBases(e_j.childNodes);

                        if (e_j.nodeName == 'defausse')
                                setDefausse(
                                        e_j.attributes.getNamedItem("n").nodeValue,
                                        e_j.attributes.getNamedItem("last").nodeValue
                                );
```

FIGURE 8 - PARCOURS XML

4.4.3. DECOUVERTE DU JQUERY

La première version de l'interface a été réalisée sans jQuery*, soit en JavaScript pur. Nous nous sommes vite rendu compte que le traitement des objets html était très lourd, sans parler des animations. Il fallait stocker l'ensemble des cartes dans des variables au fur et à mesure de leurs créations pour obtenir des pointeurs vers les images, car il était trop lourd de rechercher à chaque fois dans le DOM* la carte que nous cherchions…

L'apprentissage du jQuery* pour la seconde version de l'interface a permis de gagner énormément de temps. Même s'il est plus technique à assimiler, son utilisation apporte beaucoup de confort. On peut ajouter des attributs qui nous aident pour retrouver les éléments nécessaires facilement. Par exemple pour attraper une carte dans la main du joueur qui possède un id précis, on fait maintenant : `$('#main > img[idcard='+idCard+']:first')`

De même pour les animations, le jQuery* nous a permis de réaliser des animations temporisées avec des systèmes de queue, étape par étape.

4.5. CONCEPTION DES CARTES DE JEU

En parallèle du développement du client JavaScript et de son interface, il nous fallait implémenter les fonctionnalités des cartes de jeu, et notamment des cartes actions, présentes en grand nombre dans le jeu, chacune d'elles possédant ses propres spécificités. Le jeu possédant plusieurs extensions avec un nombre très important de cartes, le projet ne

nous demandait pas de toutes les intégrer dans le jeu, il fallait cependant en faire le plus possible et surtout faciliter l'ajout de cartes en prévoyant un jeu comprenant un maximum de fonctionnalités et extensible facilement.

L'ajout de cartes dans le jeu s'est fait par étapes, les trois principales étant celles développées dans le rapport technique, à savoir les cartes modifiant les propriétés du joueur, celles demandant une réponse de leur part ainsi que les cartes qui engageaient une interaction entre joueurs. Un grand nombre d'entre elles imposaient donc un ajout de fonctionnalité du serveur Java ou encore du client JavaScript.

C'est donc lors de cette étape que sont apparues les fonctionnalités concernant les demandes au joueur par le biais des contrôleurs ou encore les gestions d'attaques et de réactions.

CONCLUSION

Au terme de ces mois de projet, nous avons réalisé une version du jeu de cartes Dominion en ligne et jouable par navigateur. Celle-ci comprend l'intégralité des cartes du jeu de base.

Concernant le moteur de jeu Java, celui implémente correctement l'ensemble des règles du jeu tel que le respect du déroulement des phases d'un joueur.

Le client JavaScript, quant à lui, correspond aux attentes fixées dès le départ, à savoir une interface s'approchant au maximum du jeu de plateau réel, avec des animations afin de ne pas perdre le joueur dans la rapidité des actions informatiques effectuées par lui ou bien ses adversaires. La gestion de parties est fonctionnelle, plusieurs d'entre-elles peuvent être jouées simultanément. Le joueur peut en créer, inviter d'autres joueurs ou en rejoindre.

Le client est compatible sur l'ensemble des navigateurs récents les plus utilisés, il est sécurisé suite à la vérification des requêtes qui lui sont envoyées et le jeu est fluide.

Le contenu du jeu complet contient le jeu de bases. Par la suite de nouvelles extensions et d'autres cartes peuvent facilement être ajoutées du fait d'une conception modulable et flexible.

Pour nous, l'ensemble du projet a été très bénéfique. Le sujet proposé nous a semblé très complet et proposant de multiples aspects auquel le développement peut faire appel, comme la gestion d'un serveur, la création d'une interface web, la communication client-serveur et l'implémentation d'un jeu cartes réel. Notre tuteur a également su nous enseigner de nombreux aspects, notamment la communication client-serveur qui a été un des points les plus enrichissants.

Cette expérience a également été l'occasion de découvrir le développement d'un projet dans son intégralité en groupe avec des aspects que l'on ne retrouve pas lorsque l'on est seul comme la répartition de tâches, adapter ce que l'on écrit aux développements des autres membres du groupe ou alors parvenir se mettre d'accord lorsque les avis sont divergents.

Nous avons également appris à conduire un projet sur plusieurs mois, avec des délais à respecter et sous la tutelle d'une personne qui nous suivait régulièrement.

BIBLIOGRAPHIE / SITOGRAPHIE

Base serveur http Java

http://www.prasannatech.net/2008/10/simple-http-server-java.html

Protocole http

http://www.commentcamarche.net/contents/internet/http.php3

Apprentissage jQuery

http://babylon-design.com/apprendre-et-comprendre-jquery-1-3/

Version de Dominion en ligne par Isotropic

http://dominion.isotropic.org/

Liste des cartes

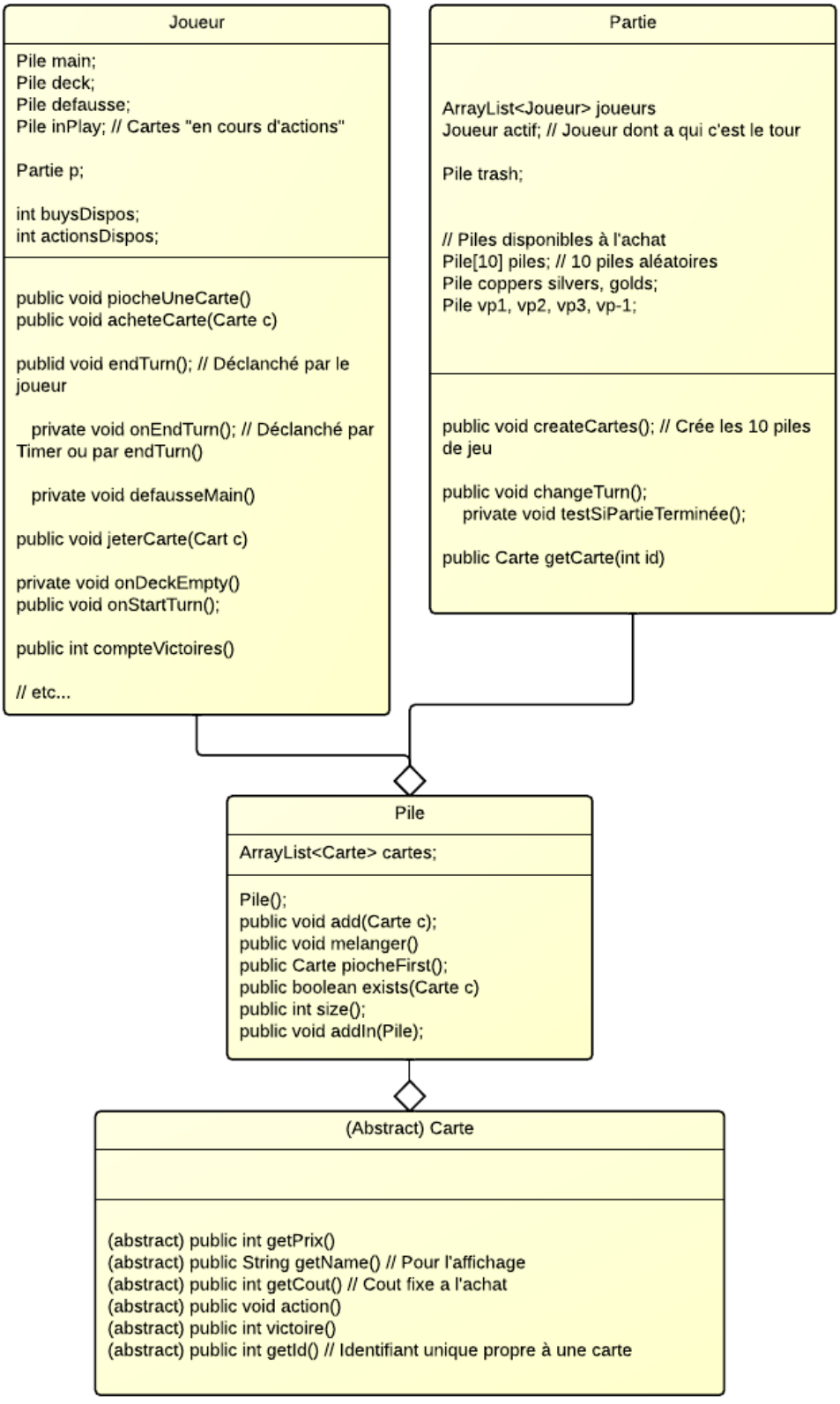
http://dominion.diehrstraits.com/?set=Base

Traduction française des cartes

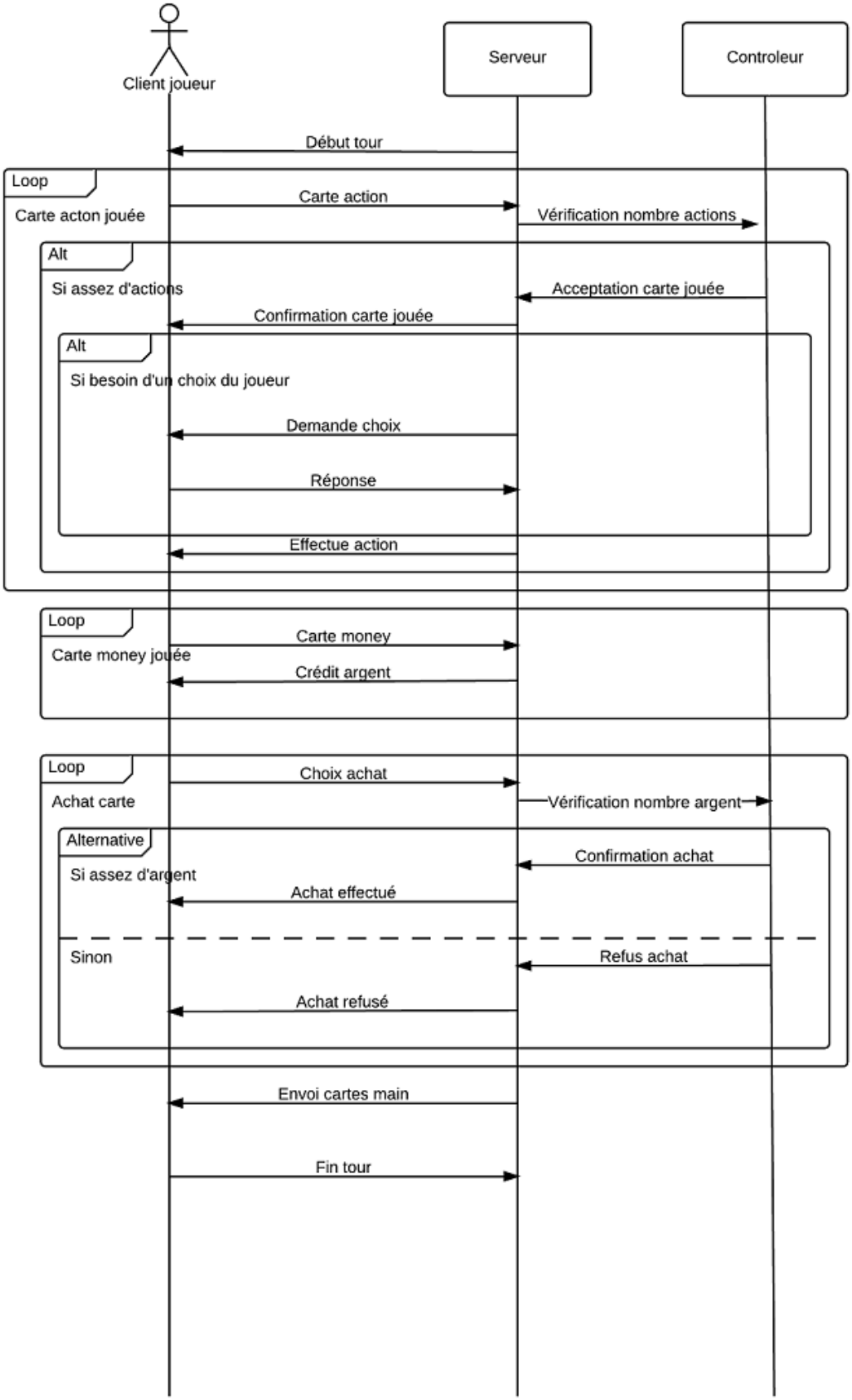
http://taom.eu/temp/domizip/Dominion.string

ANNEXES

ANNEXE N°1 : DIAGRAMME DE CLASSE



ANNEXE N°2 : DIAGRAMME DE SEQUENCE DU TOUR D'UN JOUEUR



ANNEXE N°3 : CLASSE ABSTRAITE CARTE

```
abstract public class Card {

	public final static int TYPE_ACTION = 1;
	public final static int TYPE_TREASURE = 2;
	public final static int TYPE_VICTORY = 3;
	public final static int TYPE_CURSE = 4;
	public final static int TYPE_ATTACK = 5;
	public final static int TYPE_REACTION = 6;
	public int varPrice;

	// Identifiant unique propre à la carte
	abstract public int getId();

	// Nom complet de la carte (affichage)
	abstract public String  getName();

	// Prix inscrit sur la carte.
	abstract public int getFixedPrice();

	// Prix actuel de la carte.
	public int getCurrentPrice() {
	 return getFixedPrice() + varPrice;
	}

	// Les types de la carte (Utiliser les constantes)
	abstract public ArrayList<Integer> getTypes();

	// Vrai si l'attaque de la carte affecte aussi le lanceur
	public boolean attackIncludeLanceur() {
	   return false;
	}

	// Effectue une attaque.
	public void attaque(Player lanceur, Player cible)  {
	   return;
	}

	// Effectue une réaction
	public void reaction() {
	   return;
	}

	// Effectue les actions.
	public void action(Player p {
	   return;
	}

	// Valeur argent
	public int tresor() {
	   return 0;
	}

	// Nombre de points victoires d'une carte.
	public int victory {
	   return 0;
	}
}
```

ANNEXE N°4 : METHODE LANCEMENT D'ATTAQUE

```
public void lauchAttack(Card c)
{
	// Pour chaque joueur.
	for(Player p : getGame().getPlayers())
	{
		// Si le joueur est différent du lanceur ou si l'attaque l'inclut.
		if(c.attackIncludeLanceur() || !p.equals(this))
		{
			// On demande au joueur s'il veut jouer une carte réaction.
			if(!p.getControleur().needChoice(
				"Voulez-vous jouer une carte réaction ?"))
			    c.attaque(this, p);// Si non, on lance l'attaque
			else
			{
				// On cherche les cartes réactions.
				ArrayList<Card> reactions = new ArrayList<Card>();
				for(Card card : p.getMain().cards())
				{
					// Si la carte contient le type réaction, on
				l'ajoute
					if(card.getTypes().contains(Card.TYPE_REACTION))
						reactions.add(card);
				}

				int idCard = 0;
				if(!reactions.isEmpty())
				{
					// On demande au joueur de choisir une raction
					// à jouer.
					reactions =
					p.getControleur().needChooseCards(reactions, 1,
					 "Choisissez une carte reaction à jouer.");

					 if(!reactions.isEmpty())
					 {
						// On exécute la méthode reaction.
						reactions.get(0).reaction();
						idCard = reactions.get(0).getId();
					 }
				}

				if(idCard != 32)  // Si ce n'est pas la carte douves.
					c.attaque(this, p);
			}
		}
	}
 }
```

ANNEXE N°5 : FICHIER XML DE L'ETAT DE LA PARTIE

```
<game id="3211">
	<me id="1">
		<main>
			<card value="1" />
			<card value="4" />
			<card value="1" />
		</main>
		<inplay>
			<card value="1" />
		</inplay>
		<nachat value="1" />
		<naction value="1" />
		<nmoney value="4" />
		<defausse n="5" last="4" />
		<deck n="0" />
	</me>
	<kingdom>
		<card value="15" n="10" />
		<card value="18" n="10" />
		<card value="14" n="10" />
		<card value="19" n="10" />
		<card value="9" n="10" />
	</kingdom>
	<other>
		<card value="1" n="10" />
		<card value="2" n="10" />
		<card value="3" n="10" />
		<card value="4" n="10" />
		<card value="5" n="10" />
		<card value="6" n="10" />
		<card value="7" n="10" />
	</other>
	<joueurs>
		<joueur id="0" name="doelia">
			<main n="5" />
			<deck n="0" />
			<defausse n="5" last="4" />
		</joueur>
		<joueur id="2" name="Player 3">
			<main n="5" />
			<deck n="5" />
			<defausse n="0" last="0" />
		</joueur>
	</joueurs>
	<nactif value="2" />
</game>
```

Résumé

Le projet Dominia a pour but de concevoir une version jouable par navigateur web du jeu de cartes Dominion. Cette version informatisée propose la création de plusieurs parties simultanément, chacune pouvant contenir jusqu'à quatre joueurs.

Le développement a concerné deux parties distinctes : d'une part le serveur en Java qui assure la gestion des parties et le déroulement de chacune d'entre elles et d'autre part une interface web côté client en JavaScript. Nous avons conçu notre propre solution pour leur communication en utilisant le protocole HTTP.

Toutes les cartes du jeu n'ont pas été créées mais l'ajout a été simplifié du fait d'une conception structurée comprenant un maximum de fonctionnalités du jeu.

Abstract

The dominia project was to design a browser game based on the card game Dominion. This electronic version provides multiple games at the same time, with up to four players in each game.

The developpement was divided in two major parts : the Java server, which handles and runs the games and a web interface for the client in Javascript. We built our own solution for the communication between the client and the server, using HTTP.

We didn't add all the cards, but the adding has been simplified thanks to a structured design including as much features as possible.

Mots clés

Dominion, jeu de cartes, serveur HTTP, Java, interface web, JavaScript