université de BORDEAUX

Collège Sciences et technologies

Data Mining
Dominion game Logs

Auteur :
Willian Ver Valen Paiva
Liliana Lopez Farfan
Elmer Bayol
Khaoula Tagnaouti

Client :
Yvan Le Borgne

6 avril 2016

Résumé

Ce projet porte sur l'analyse de parties d'un jeu de cartes appelé *Dominion* enregistrées sous formes de *Logs*. L'objectif principal est d'en interpréter les données pour les rendre utilisables facilement afin de pouvoir effectuer des vérifications et des interprétations sur la base de celles-ci. Pour cela une première partie du projet porte surle *Parsing* de ces *Logs*, puis dans un second temps le stockage des données extraites. La seconde partie du programme est constitué d'une bibliothèque capable de récupérer ces données, de les analyser et/ou de les reconstituer pour les compléter.

Table des matières

Chapitre 1

Présentation du projet

Le jeu de cartes *Dominion* a été mis à disposition, sur un serveur de jeu, d'octobre 2010 à mars 2013. Les parties effectuées via ce serveur ont été enregistrées dans des *logs* qui mémorisaient toutes les actions des joueurs ; ces *logs* ont été mis à disposition du public. Mais ces enregistrements n'ont pas conservé un certain nombre d'informations ayant trait à des décisions prises par les joueurs.

Un wiki[1] relatif au jeu a été élaboré ; il offre des avis d'experts susceptibles de servir d'aide à la décision pour les joueurs. Il propose des conseils, notamment au niveau stratégique. Le but de ce projet, proposé par Yvan Le Borgne, chercheur au Labri, est de traiter ces *logs* (soit plus de 12 millions de parties) afin de répondre à plusieurs interrogations. Il s'agira principalement de comparer les données recueillies aux préconisations de ce wiki, afin de valider, à travers les contenus des parties, l'éventuelle efficacité des avis et suggestions fournies par les experts. Dans cette optique, on cherchera à élaborer des outils de validation présentant une efficacité suffisante.

A cet effet, le projet a pour but de créer une base de données recensant les parties ; puis un travail d'analyse sera effectué à partir de cette base de données. Pour cela, notre programme devra tout d'abord extraire les données présentes dans les logs, car ceux-ci sont trop volumineux et trop compliqués pour pouvoir les utiliser directement (structure non standardisée). Il s'agira donc, dans un premier temps, de mettre au point des structures de données permettant de les stocker de façon plus efficace. Une manière simple de représenter les données extraites et analysées sera proposée à l'utilisateur.

Par ailleurs, il manque des informations dans les enregistrements (*logs*). On cherchera à trouver par quelle méthode on peut les reconstituer et les mettre dans un format contenant toutes les informations. En outre, on cherchera à optimiser les opérations menées dans l'analyse des donnnées, en recherchant le meilleur équilibre possible entre la mémorisation des recherches, et le re-calcul.

Enfin, on esssaiera de déterminer quelle stratégie a été utilisée dans chaque partie, voire de faire émerger les changements de stratégie en cours de partie. Si on prend par exemple une des stratégies (la *Penultimate Province Rule*, peut-on détecter à quel moment cette règle a été appliquée ou contournée ? ou qui a mis au point cette stratégie ? (découverte qui pourrait permettre d'organiser un classement des joueurs les plus performants). Dans la mesure où on parviendrait à mettre au point un nombre suffisant de ces démarches stratégiques, il s'agirait de déterminer le processus habituel d'apprentissage des joueurs. Le client a demandé de pouvoir représenter la progression des joueurs, de ce fait un système d'ELO [1][2] pourrait être mis en place.

1.1 Règles du jeu

Chaque joueur possède un deck de cartes et a accès à un «marché» où différentes cartes d'action sont disponibles. Il y a également une pile de "cartes de victoire" (en anglais *victory cards*).

Les joueurs commencent avec un deck contenant 3 cartes de monnaie permettant d'acheter les autres cartes mises à la disposition des joueurs ainsi que 2 cartes de victoires a valeur la plus basse. Les

1. le système de prédiction ELO, mis au point par le Pr Arpad Elo est un système mathématique simple qui permet de prédire la probabilité qu'un joueur d'échec, ou une équipe de baseball ou de basket, en gagne un(e) autre

joueurs commencent leur tour avec 5 cartes en main, et le tour d'un joueur se déroule en deux phase : premièrement, la phase d'action, où le joueur peut jouer une carte d'action ; puis une phase d'achat, où le joueur peut acheter des cartes du marché, des cartes de monnaie ou des "cartes victoire". Lorsque le tour se termine, les cartes en main sont mises dans la pile défausse qui resservira de deck une fois celui-ci vidé.

La partie se termine dans deux cas : si la réserve de cartes «Province» (carte de victoire au score le plus élevé) est vide ; ou bien si 3 piles du marché sont vides. Quand la partie est terminée, les points de victoire (découlant des "cartes de victoire") de chaque deck sont comptés et le vainqueur est celui qui a le meilleur score.

1.2 Introduction au datamining

Avec l'augmentation des capacités de stockage des supports informatiques, et le recueil généralisé de données dans le cadre du commerce et de l'entreprise, la question de leur exploitation se pose et est résolue par le *data mining*. Il s'agit de découvrir dans de grandes masses de données d'apparence chaotique des structures et des corrélations non perceptibles à première vue. Le *data mining* permet de faire surgir des tendances non encore discernables, et des phénomènes qui ne sont pas encore perceptibles pour fonder des stratégies décisionnelles. Comportement d'achat, caractéristiques de produits, création de l'histoire des productions, etc... sont les domaines principaux d'application du *data mining*.

Il existe deux types de *data mining* ; une forme de vérification, qui permet de valider une intuition ou une idée générale en exploitant les données disponibles de confirmer une idée préalable ; cette approche est plus limitée que celle de la variante de découverte, qui permet de déccouvrir de l'information cachée, impossible à découvrir par un analyste humain, tant la quantité des données à exploiter est importante.

Une démarche de *data mining* s'organise selon 5 grandes étapes :

— la définition du problème, qui consiste à cibler les besoins auxquels il faut répondre
— la collecte des données, lesquelles devront être "nettoyées" pour les rendre exploitables ; cette phase est essentielle pour obtenir des résultats exploitables ; elle doit être effectuée avec le plus grand soin, et fournir une quantité suffisante de données consolidées
— la construction du modèle d'analyse testé pour vérifier sa pertinence et modifier si besoin les deux premiers points
— l'étude des résultats, en corrigeant là aussi les étapes précédentes si les résultats ne sont pas satisfaisants
— la formalisation et la diffusion, laquelle fait des résultats une connaissance partagée, et justifie de ce fait le soin qu'on a apporté aux étapes précédentes.

On n'oubliera cependant pas que les corrélations ne veulent pas nécessairement dire causalités, et que le *data mining* doit être exploité prudemment pour ne pas faire des déductions inappropriées.

1.3 Positionnement du projet

Le projet proposé concerne les phases de consolidation des données fournies pour les rendre exploitables, et de construction d'un modèle d'analyse qui serait testé pour valider la pertinence de son approche et revenir le cas échéant sur la phase précédente. En effet, le sujet tel qu'il a été proposé signalait notamment qu'un certain nombre de données n'avait pas été enregistrées, et qu'il fallait donc les reconstituer, et créer un format valide pour l'ensemble. Par ailleurs, on devait également vérifier les modalités d'apprentissage des joueurs, et leur classement compétitif relatif. Il se situe dans le cadre d'une analyse de vérification, puisqu'il s'agit de valider les affirmations du wiki qui prétend fournir des stratégies pour les joueurs.

Chapitre 2

Analyse de l'existant

2.1 Analyse de l'existant

2.1.1 Description des données

Les données qui ont été fournies par le client avaient été compressées au format *tar.bz2*. Un fichier contenait chaque journée du Log, et le total des données compressées se montait à 13 Go. Une fois décompressées, les données faisaient un total de 400 Go. Chaque journée du Log est au format html. Chaque *Log* doit contenir :

- une en-tête contenant le numéro du jeu, et le gagnant
- un résumé du match contenant les cartes utilisées pendant le match, et comment le match a fini
- un résumé du joueur, contenant toutes les "cartes bonus" du joueur, le *deck* et les points
- l'enregistrement des différentes étapes du jeu, qui contient tous les détails des mouvements effectués par les joueurs au cours de la partie.

2.1.2 Incohérences dans les données

Les*Logs* présentent des incohérences qui peuvent créer des problèmes pour le développement du parser ; quelques uns des problèmes rencontrés ont été les suivants :

- **Numéro de *Log*** : la numération des logs n'était pas unique, ce qui voulait dire qu'on ne pouvait pas l'utiliser pour identifier un log.
- **Nom d'utilisateur** : les logs font apparaître qu'il n'y avait pas de restriction quant aux noms des joueurs, et un certain nombre de noms de joueurs contiennent des mots-clés et utilisent des caractères spéciaux qui entrent en conflit avec le parser.
- **Données manquantes** : dans quelques logs, il manque une partie des données(comme par exemple l'en-tête, le résumé du joueur, etc...)
- **Format du *Log*** : la syntaxe utilisée pour saisir les *Logs* n'est pas cohérente, et offre des différences entre différents *Logs*
- **Compression des données** : comme les données sont compressées, et que le matériel fourni pour travailler sur le projet ne peut gérer les données une fois décompressées, il faudra travailler sur le projet avec des données compressées. Un premier essai de décompression a montré qu'il fallait un minimum de 4 heures pour tout décompresser. Il allait falloir en outre décompresser par tranches et effacer les contenus au fur et à mesure pour pouvoir fonctionner avec le matériel disponible.

Chapitre 3

Analyse des besoins

Comme on l'a vu dans l'approche théorique du *Data Mining*, comme dans le cas de tout projet d'ailleurs, la définition des besoins est une étape cruciale du démarrage de l'activité. On va donc évoquer tout d'abord les besoins fonctionnels, puis les besoins non-fonctionnels.

3.1 Besoins fonctionnels

Les besoins fonctionnels sont ceux qui permettent de remplir les conditions du cahier des charges ; ils sont incontournables. Il s'agit donc ici de voir comment on va répondre à la demande formulée à l'origine du projet.

3.1.1 Interface utilisateur

L'interface utilisateur se compose d'une bilibliothèque qui permettra à l'utilisateur d'exploiter les données récupérées par le parser. L'utilisateur pourra appeler les fonctions proposées par le programme ou bien écrire ses propres fonctions et les appliquer aux données.

3.1.1.1 Se connecter à la base de données orientée documents

Description et piorité Niveau de priorité = haute
Cette bibliothèque offrira des fonctions simples d'accès en lecture/écriture sur la base de données ; de ce fait ; cette librairie pourra :

— récupérer un *log* ou les informations générales d'un joueur.
— mettre à jour les données relatives à un *log* ou aux informations générales d'un joueur.
— créer de nouvelles données issues des analyses effectuées.

3.1.1.2 Outils d'analyse

Description et piorité Niveau de priorité = haute
Une seconde fonctionnalité de la bibliothèque proposera une solution à l'utilisateur pour effectuer des analyses prédéfinies sur les données récupérées, ainsi qu'un moyen d'appliquer de manière générale une analyse créée par l'utilisateur. Elle pourra donc :

— appliquer une fonction spécifiée par l'utilisateur à une liste de documents récupérés par une requête qu'il aura posée. La fonction spécifiée recevra un *log* et pourra travailler sur le document en question.
— générer l'ELO de chaque joueur pour chaque partie, ainsi qu'un ELO global de chaque joueur.
— détecter les différentes stratégies utilisées par les joueurs lors d'une partie ou de l'ensemble des parties.
— détecter le moment auquel le *greening* survient, c'est a dire le tour ou les joueurs commencent à acheter des «cartes de victoire».

3.1.2 Modélisation des données

La première démarche consistait à modéliser les données disponibles pour créer une cohérence entre tous les formats existants. Il fallait commencer par décompresser les données, même si, comme on l'a vu,

cela ne pouvait être fait que par "tranches" de temps (journées de *Logs*) à cause des ressources mémoire disponibles.

3.1.2.1 Décompression des logs

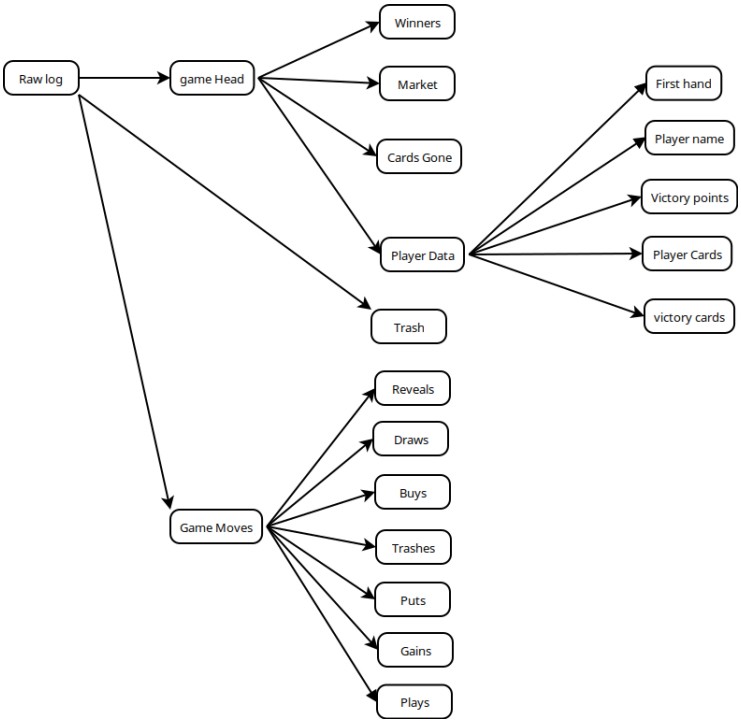
Description et priorité Niveau de priorité = haute

- Les fichiers *tar.bz2* seront stockés dans un dossier.
- Le programme va décompresser un fichier spécifié à partir de ce dossier dans un dossier temporaire.
- Le programme va supprimer les *Logs* décompressés à la demande du parser.

3.1.2.2 Parser

Description et priorité Niveau de priorité = haute

Compte tenu des problèmes mis en évidence en analysant un échantillon de *Logs* (comme par exemple des incohérences de syntaxe), l'utilisation d'un parser (comme *YACC* et *LEX*) n'est pas recommandée. C'est pour cette raison que l'on va créer un parser qui utilisera les mots-clés et l'HTML déjà présents dans les *Logs*. Ce parser sera responsable de la lecture et de la collecte des informations importantes sans perdre la structure du *Log*. Un aperçu des données devant être reconnues par le parser est représenté dans l'illustration suivante :

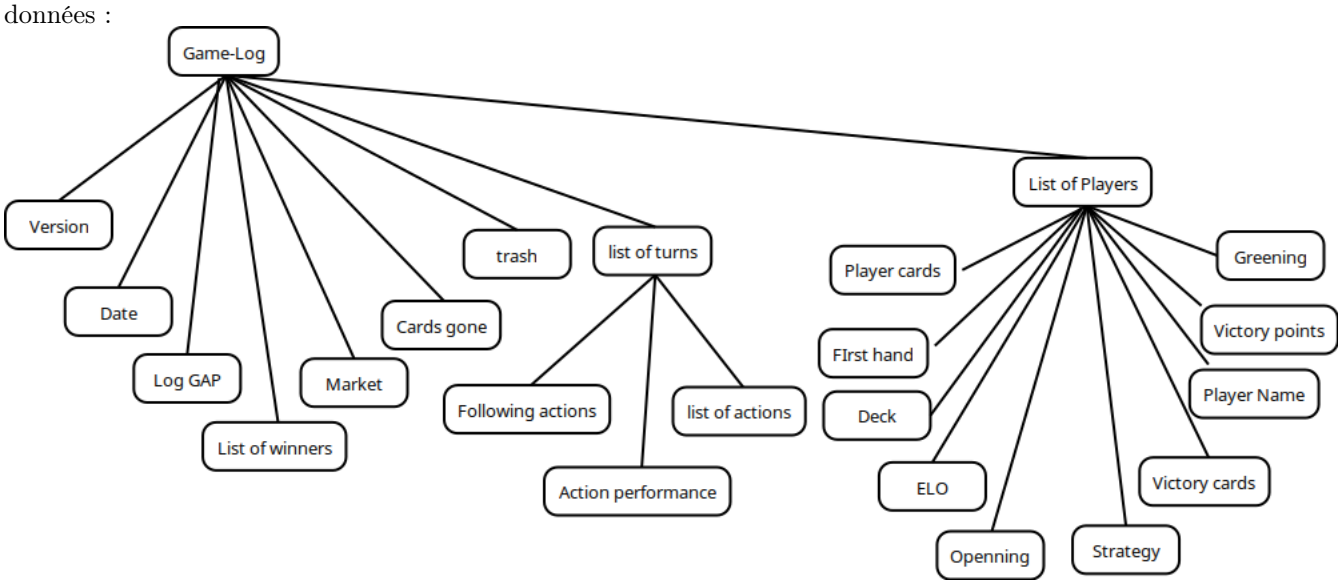


- Winners : liste des gagnants du jeu ; il peut y avoir plusieurs gagnants en cas d'égalité.
- Market : liste des 10 cartes disponibles pour être achetées par les joueurs.
- Cards Gone : liste des cartes qui ont été entièrement acquises à la fin de la partie.
- First hand : liste des cartes obtenues au début de la partie.
- Player Name : nom du joueur.
- Victory points : nombre de points à la fin de la partie.
- Player cards : liste des cartes obtenues à la fin de jeu avec des noms et de quantités differents.
- Victory cards : liste des cartes de victoire que le joueur a acheté.
- Trash : liste des cartes qui sont jetés à la fin de la partie.
- Game Moves : liste des actions effectuées pendant chaque tour d'un joueur.

Lorsque l'utilisateur exécute le parser, celui-ci va identifier les types d'informations concernant chacun des éléments représentés dans le graphique précédent, les trier, et les charger dans une structure de données du type "document", puis alimenter la base de données avec le document.

3.1.2.3 Créer le game-log

Description et priorité Niveau de priorité = haute
Cette fonctionnalité consiste à créer une structure de données dans laquelle les données du *Log* seront chargées. La figure suivante est la représentation graphique d'une vue d'ensemble de la structure des données :



3.1.2.4 Créer une base de données orientée document

Description et priorité Niveau de priorité = haute
Ce module sera responsable de la création de la base de données orientée document, contenant des données au format JSON.

3.1.2.5 Se connecter à la base de données orientée document

Description et priorité Niveau de priorité = haute
Un interface complet de communication avec la base de données orientée document permettra d'envoyer et recevoir des données.

3.1.2.6 Compression

Description et priorité Niveau de priorité = moyen

La base de données orientée document contiendra la totalité des données et un certain niveau de compression devra être appliqué. La base de données que nous allons utiliser est *mongoDB*, et elle offre 2 niveaux de compression *Zlib* et *Snappy*. Au cours du développement, on utilise *Snappy*, car il offre une meilleure performance. Mais le programme final devrait donner à l'utilisateur la possibilité de choisir le niveau de compression lors du démarrage de l'analyse du *Log*.

3.1.2.7 Sauvegarder le game-log

Description et priorité Niveau de priorité = haute

Cette fonctionnalité permet d'envoyer les documents au format JSON et de les stocker dans la base de données.

3.1.2.8 Restorer le *game-log*

Description et priorité Niveau de priorité = moyen

Cette fonction a pour responsabilité de restaurer l'information manquante sur les *Logs* analysés, par déduction, sur la base des informations déjà récoltées dans le *Log*. Si les échanges exécutés pendant la partie sont fournis par le *Log* on va les utiliser pour simuler la partie et reconstituer les éléments

manquants.

En cas d'un en-tête du jeu manquant par exemple, on utilisera la stratégie suivante :
— *Gagnants / "cartes bonus" / points "bonus" manquants* : le programme peut garder la trace des cartes détenues par les joueurs pendant le jeu afin de savoir combien de "cartes bonus" ils ont à la fin de la partie.
— Market manquant : le programme peut garder la trace des cartes achetées au cours du jeu afin de reconstituer partiellement ou totalement les cartes disponibles sur *Market*.
— *Cartes Gone* : comme avec les données de *Market* manquantes, le programme permet de garder une trace des cartes achetées et compter la quantité achetée pour chaque carte ; si le montant maximum de cartes est acheté, la carte a disparu à la fin de la partie.
— *Nom du joueur manquant* : le jeu signale à chaque mouvement le nom du joueur qui agit. Le programme sera donc tout simplement chargé de les restaurer dans l'en-tête.
— *Cartes de joueur* : le programme garde une trace des actions effectuées au cours du jeu en ce qui concerne les cartes de joueurs (actions, tirage,...) et d'en garder la liste.

3.1.2.9 Calculer l'*ELO*

Description et priorité Niveau de priorité = haute

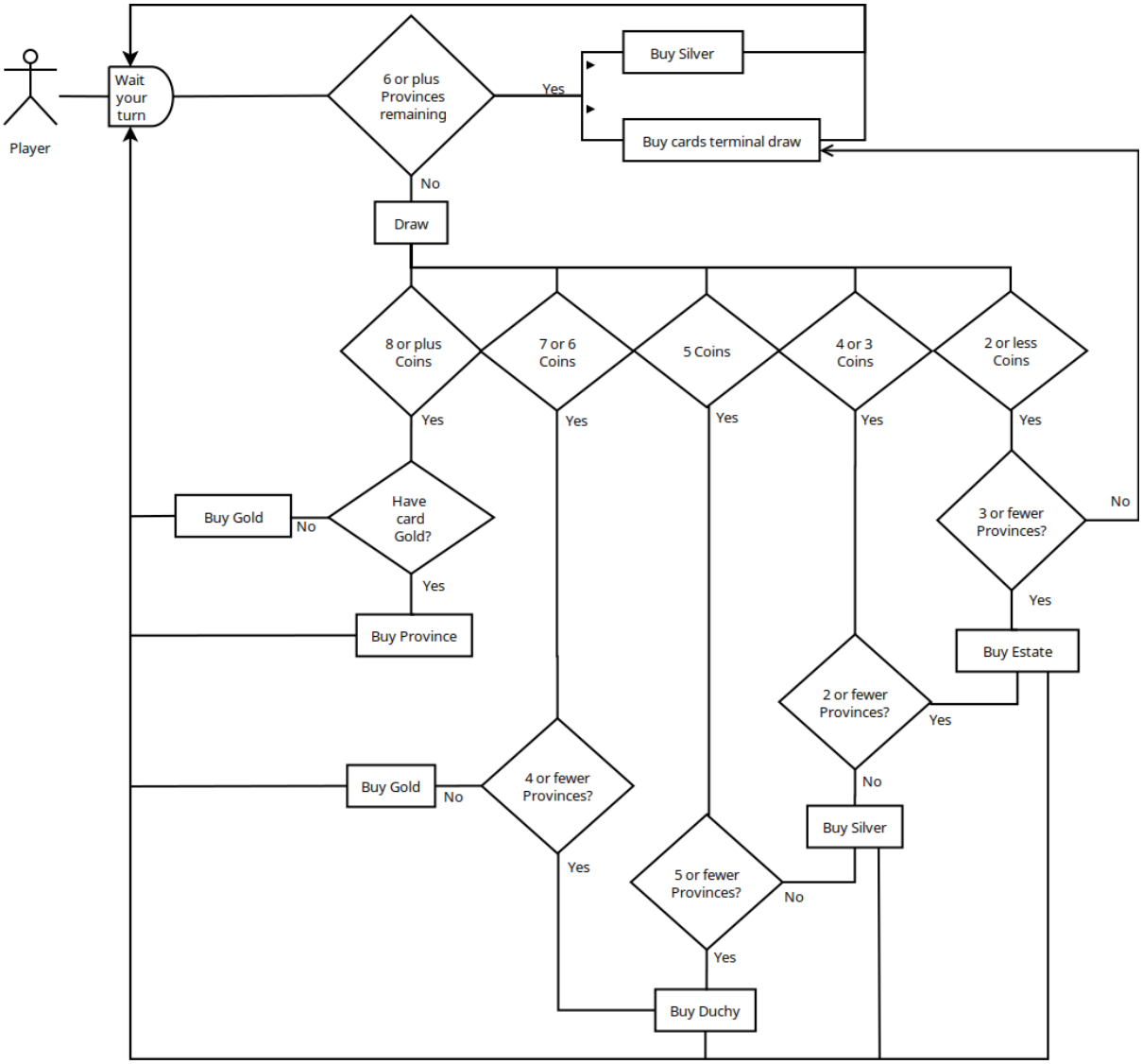
L'analyseur devra calculer l'ELO final de chaque joueur ainsi que son ELO temporaire à l'issue de chaque partie.
Pour chaque partie ; dans l'ordre chronologique, l'analyseur va calculer l'ELO de chaque joueur à l'issue de cette partie et l'ajouter aux données relatives au *Log* en question ; il va également mettre à jour l'ELO général des joueurs concernés afin de pouvoir continuer à calculer leur évolution.

3.1.2.10 Reconnaître les stratégies

Description et priorité Niveau de priorité = moyen

Le wiki de dominion décrit quelques stratégies qui peuvent être utilisés dans le jeu.
Par exemple :
Big Money
Beyond Silver
Penultimate Province Rule
L'intérêt du présent projet, comme on l'a vu, est de vérifier la validité des stratégies conseillées par le wiki. Celles qui seraient déduites par le projet seraient fondeées sur un calcul approfondi dépassant les simples intuitions ou calculs statistiques concernant ces mêmes stratégies. Mais cette méthode, pour efficace qu'elle soit, est bien entendu beaucoup plus coûteuse en temps et en efforts.

Cette image décrit le processus de décision associé à la stratégie Big Money



L'analyseur devra donc être en mesure de reconnaître quelle stratégie a été utilisée sur un match donné (si une stratégie a été utilisée), et d'attribuer à chaque partie le nom de la stratégie utilisée par chaque joueur, puis d'enregistrer le tout sur la base de données.

Afin de reconnaître la stratégie *Beyond Silver* ou *Big Money*, l'analyseur doit reconnaître lorsque certains types de cartes sont achetées (la liste de ces cartes peut être trouvée sur : http://wiki.dominionstrategy.com/index.php/Silver#Beyond_Silver et http://wiki.dominionstrategy.com/index.php/Big_Money)

Afin de reconnaître que *Penultimate Province Rule* a été utilisée, (comme expliqué à http://dominionstrategy.com/2011/03/28/the-penultimate-province-rule/), l'analyseur doit garder une trace de la quantité de "Cartes bonus" de chaque joueur à chaque tour.

3.1.2.11 Reconnaître le *Greening*

Description et priorité Niveau de priorité = haute

L'analyseur doit être capable de reconnaître à quel moment les joueures commencent à investir dans les "Cartes bonus", et donc à pratiquer le ***greening*** dans chaque match. En savoir plus à propos de ***greening*** sur http://wiki.dominionstrategy.com/index.php/Greening.

3.2 Besoins non-fonctionnels

Les besoins non-fonctionnels sont des possibilités qui n'ont pas été expressément réclamées par le client, et donc n'appartiennent pas au cahier des charges. Néanmoins, si on peut les mettre en oeuvre, elles constituent un plus appréciable et augmentent la dimension qualitative du travail.

3.2.1 Besoins de performance

Aucun besoin de performance spécifique n'a été exprimé par le client. Cependant, il tombe sous le sens qu'il convient d'optimiser l'exécution des différentes tâches en termes de rapidité et d'accessibilité pour l'utilisateur, ainsi que pour faciliter le développement.

3.2.2 Fiabilité

Le client demande qu'un maximum de *Logs* soient récupérés, la totalité des *Logs* ne pourra pas être parsée mais une tolérance de 5 à 10% de *Logs* peut être acceptée.
En revanche lorsque un *Log* est parsé, le programme devra récuperer 100% des données présentes le concernant, et si possible restaurer les informations non présentes dans le *Log* de départ.

3.2.3 Attributs de qualité de logiciels

Le programme devra pouvoir lancer la phase initiale de parsing en une seule ligne de commande.
Les fonctions présentes dans la librairie à l'attention de l'utilisateur devront avoir un nom clair et facile à comprendre.
Les données générées par l'analyseur n'ont pas besoin à être lisibles par l'homme.
Les game log générés devront pouvoir être analysés par l'utilisateur afin de permettre la plus grande souplesse possible dans le traitement des logs.

3.3 Besoins organisationnels

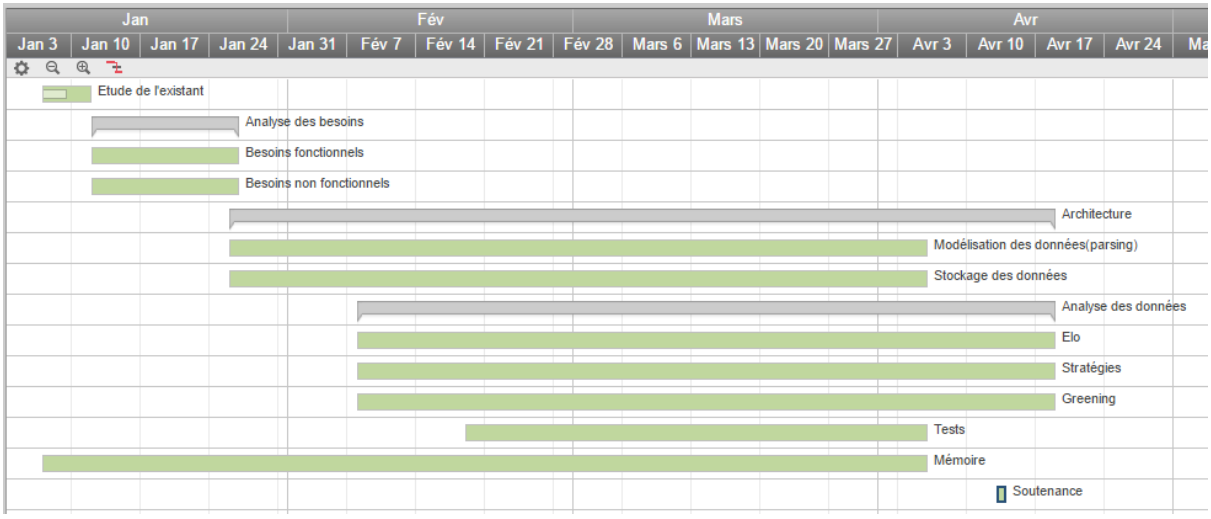
La répartition des tâches du projet et l'estimation de la durée de chacune d'elle sont présentées sur le planning suivant :

3.3.1 Planning prévisionnel

Tâches

Nom	Date de début	Date de fin
Formulation de groupe	30/11/2015	30/11/2015
Réunion1 avec le Client	02/12/2015	02/12/2015
Réunion1 entre les membres du Groupe	04/12/2015	04/12/2015
Création de dépôt svn + GitHub	04/12/2015	04/12/2015
Réunion2 avec le Client	18/12/2015	18/12/2015
Réunion2 entre les membres du Groupe	06/01/2016	06/01/2016
Etude de l'existant	07/01/2016	11/01/2016
Réunion1 avec le chargé de TD	25/01/2016	25/01/2016
Réunion3 entre les membres du Groupe	29/01/2016	29/01/2016
Réunion3 avec le Client	05/02/2016	05/02/2016
Réunion4 entre les membres du Groupe	18/02/2016	18/02/2016
Réunion2 avec le chargé de TD	04/03/2016	04/03/2016
Réunion5 entre les membres du Groupe	07/03/2016	07/03/2016
Réunion3 avec le chargé de TD	18/03/2016	18/03/2016
Réunion6entre les membres du Groupe	19/03/2016	19/03/2016
Réunion7 entre les membres du Groupe	21/03/2016	21/03/2016
Réunion4 avec le chargé de TD	31/03/2016	31/03/2016

3.3.2 Diagramme de Gantt



Chapitre 4

Architecture et description du logiciel

Cette section concerne la description de l'architecture du projet, en abordant tout d'abord le parsing, puis l'analyse des données et le rendu graphique.

4.1 Architecture globale

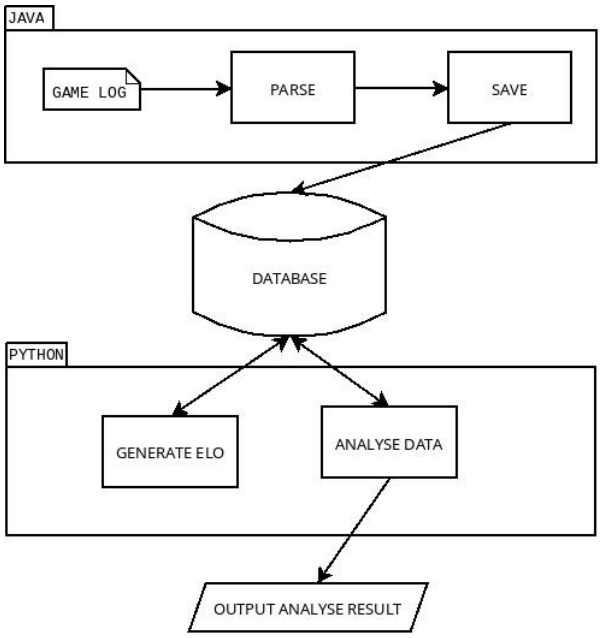


FIGURE 4.1 – représentation de l'architecture globale du projet

L'architecture du projet se décompose en 3 parties distinctes et indépendantes. Un premier élément, développé en Java, est responsable du parsing destiné à analyser et structurer les données, dans un format compatible avec la base de données, et de s'en servir pour l'alimenter.

La base de données stocke ces données et peut en recevoir de nouvelles ou bien répondre aux requetes qui lui sont soumises.

La dernière partie correspond à une bibliothèque en Python qui met à disposition des outils permettant de raffiner et d'analyser les données disponibles dans la base.

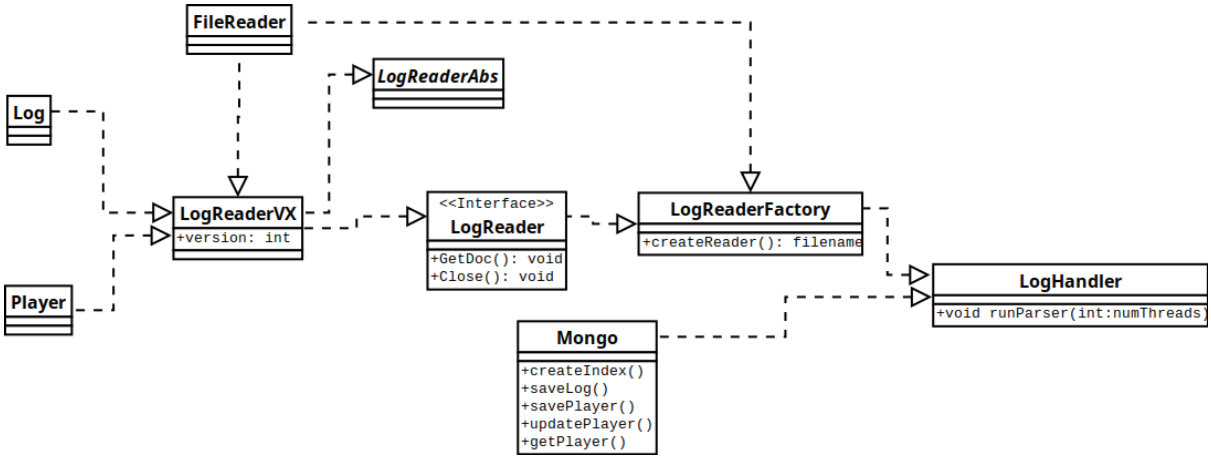


FIGURE 4.2 – Représentation de l'architecture de la partie parser

4.2 Parser

En raison d'évolutions constantes du serveur du jeu, les formats des *Logs* présentent de petites différences. Plusieurs tranches de *Logs* existent, qui traduisent les changements apportés aux formats au fil du temps. Pour résoudre ce problème il a fallu identifier les tranches, leur donner un numéro de version, et créer un *LogReader* adapté à chaque version. A cette fin, il a été utilisé un *design pattern factory*. Pour la création des *LogReaders*, une classe abstraite, la *LogReaderAbs* a été créée, comportant toutes les méthodes utilisées pour le parsing qui sont communes à l'ensemble des *LogReaders*.

La classe *LogHandler* est chargée de demander au *LogReaderFactory* la création d'un *LogReader* correspondant à la version à traiter et de demander au parser de générer les documents puis de les envoyer à la base de données.

4.3 Base de données

MongoDB a été choisi pour gérer les documents ainsi générés car ses caractéristiques correspondent aux besoins du projet : quantité volumineuse de données, grande vitesse de recherche, format des données adapté.

4.4 Bibliothèque

Cette partie est écrite en Python à la demande du client. Plusieurs bibliothèques, contenant des outils nécessaires pour travailler sur les données acquises lors du parsing, sont proposées. Il s'agira de la structure des données, de l'interface de communication avec la base de données, et d'outils utilisables par le client.

Voici un schéma regroupant les différents modules et leurs liens :

4.4.1 Structure des données

Il s'agissait de créer des structures de données facilitant le travail à effectuer sur celles-ci. Un type de données est appelé *Match* ; il correspond aux documents "Log" ; un autre type est dénommé *Simplified-Player*, et correspond aux données concernant les joueurs et leur ELO final.

4.4.2 Interface de communication avec la base de données

Cette interface a été créée pour faciliter la communication du client avec la base de données :elle comprend des outils comme la récupération de documents, la recherche de "logs", la recherche de joueurs. Le nom de ce module est *MongoInterface*.

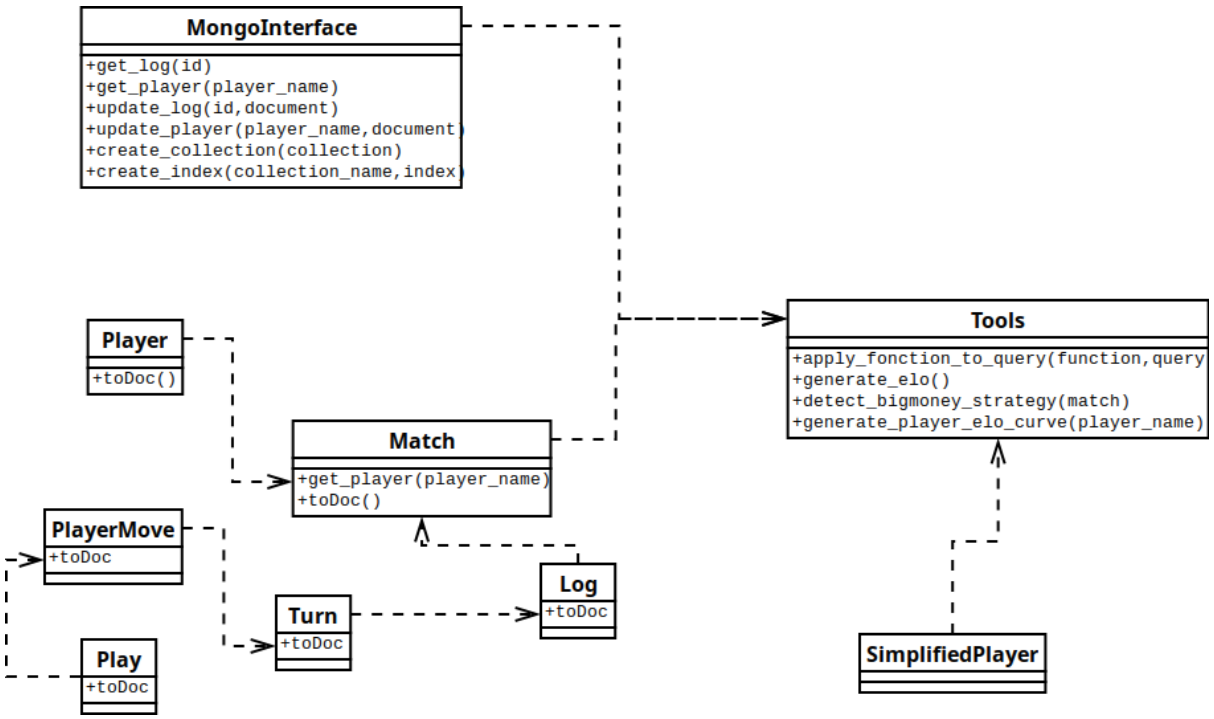


FIGURE 4.3 – Représentation de l'architecture de la partie Analyse des données

4.4.3 Outils

Il s'agit d'un ensemble de fonctions destinées à raffiner et analyser les données de la base. On y trouve les fonctionnalités suivantes :

— génération de *SimplifiedPlayers* dans la base de données
— calcul d'ELO pour chaque *SimplifiedPlayers* et chaque *Log*
— génération de courbes d'ELO pour un joueur donné
— génération de listes contenant les *greenings* pour chague *Log* de la base de données
— tentative de reconnaissance des stratégies *BigMoney* dans les *Logs* (outil non fonctionnel)

4.5 Extensions possibles

Le projet étant arrivé à la phase où on pourrait commencer à exploiter les données, de nombreuses extensions sont certainement possibles. Dans l'état actuel des choses, il n'est malheureusement pas possible de les évoquer avec précision.

Chapitre 5

Fonctionnement et Tests

Dans cette partie nous allons tout d'abord nous pencher sur le fonctionnement et les tests concernant la partie parser du projet, puis nous aborderons les mêmes points pour la partie analyse des données.

5.1 Parser

Cette partie du projet est écrite en Java. Nous avons utilisé *JUnit* afin de mener à bien ces tests unitaires. De plus, le dépôt *GIT* est muni d'un outil d'intégration continu (*Travis*), qui garantit l'intégrité du code préexistant.

De plus, nous avons effectué des tests de couverture en utilisant le plugin *Cobertura Maven* qui nous a aidé à détecter les points plus faibles au niveau des tests et donc à proposer d'autres tests unitaires pour avoir une meilleure couverture.

5.1.1 Politique de tests

Le but de nos test est de verifier l'extraction des donnéees des *Logs*. Pour ce faire, nous avons choisi deux *Logs* réels représentatifs de chaque version et avons appliqué la stratégie suivante :

— En premier lieu ; nous avons réalisé des tests pour vérifier la bonne détection de version en présence de laquelle on se trouve.

— Ensuite, nous avons pris chaque donnée de la structure et avons validé qu'elle ne soit pas vide. Dans le cas où l'élément est une liste, nous vérifions que le nombre d'éléments de la liste correspond au nombre d'éléments dans le *Log*,
— Et finalement, pour les éléments carte/quantité, comme par exemple *Victory Cards* ou *Market* nous vérifions que le couple (nom, valeur) correspond.

5.1.2 Tests unitaires

Les tests proposés pour la validation du module Java sont fait avec des *Logs* réels et sans aucune altération.
Nous avons utilisé six *Logs*, deux représentatifs de chaque type de version de *Logs* (3 versions actuellement).

Dans la classe *LogReaderFactoryTest*, nous voulons tester la lecture du *Log*, la création de l'objet *LogReader* qui instancie la version correspondante du parser et la construction de l'objet *Document* complet avec toutes les donnéées du *Log* à envoyer à la base de données.

Dans les classes *LogReaderV1HeaderTest*, *LogReaderV2HeaderTest* et *LogReaderV3HeaderTest*, on rentre dans le détail de l'objet Document, et on vérifie que les attributs Date, Name, Winners, Cards Gone, Cards Market et Cards Trash sont corrects au niveau des noms, nombre et quantite par rapport aux informations contenues dans les *Logs*.

Dans la classe *LogReaderPlayerTest*, on valide les données propres du joueur. Pour les tests, on utilise trois *Logs*, un pour chaque version de parser et pendant les test, on prend un joueur au hasard pour chaque *Log*. On verifie que le nombre de joueurs correspond ou nombre de joueurs du *Log*. Les noms des joueurs doivent correspondre au nom des joueurs du *Log* ? Le même principe est appliqué pour les points et le nombre de tours.
Pour les données de *Cards Victory*, *Cards Deck*, *Cards FirstHand* et *Cards Opening*, on valide le nombre, le nom et la quantite par rapport aux *Logs*.

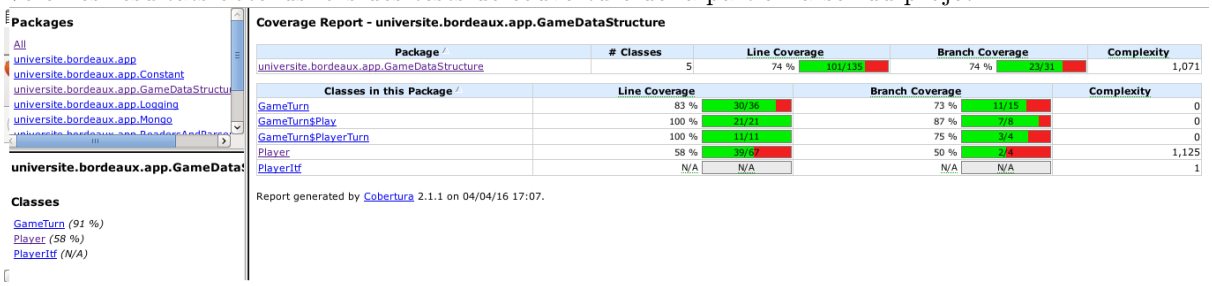
Finalement, on test la classe *PlayerTest* afin qu'il n'y ait pas de duplications de joueurs.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running universite.bordeaux.app.GameDataStructure.PlayerTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.048 sec
Running universite.bordeaux.app.ReadersAndParser.Readers.LogReaderPlayerTest
[chapel, treasuremap]
Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 3.152 sec
Running universite.bordeaux.app.ReadersAndParser.Readers.LogReaderV2HeaderTest
Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.911 sec
Running universite.bordeaux.app.ReadersAndParser.Readers.LogReaderV3HeaderTest
Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.321 sec
Running universite.bordeaux.app.ReadersAndParser.Readers.LogReaderV1HeaderTest
Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.338 sec
Running universite.bordeaux.app.ReadersAndParser.LogReaderFactoryTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.119 sec
Running universite.bordeaux.app.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec

Results :

Tests run: 51, Failures: 0, Errors: 0, Skipped: 0
```

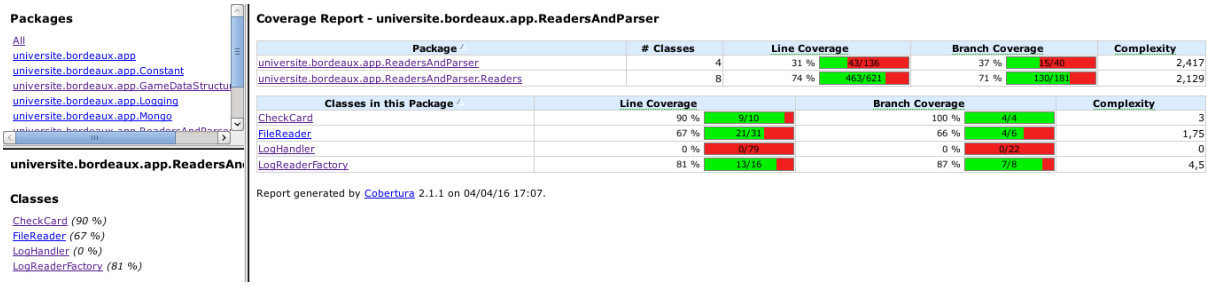
Avec un total de 51 test executés, aucune erreur ou aucun échec n'ont été détectés.

5.1.3 Tests de couverture

Voici les resultats obtenus lors des tests de couverture de la partie Parser du projet :



Dans la partie structure de données, nous n'avons pas testé l'utilisation d'actions successives dans un même tour (combo). L'autre portion de code non testée correspond à la conversion des données d'un joueur contenues dans le serveur vers un objet de type *Player*. Ce cas de figure ne se produit jamais avec l'utilisation envisagée pour la partie Java. Mais si jamais le projet venait à évoluer et utiliser des données de joueurs déja présentes dans le serveur, il faudrait effectuer des tests sur cette partie.



Packages

All
universite.bordeaux.app
universite.bordeaux.app.Constant
universite.bordeaux.app.GameDataStructu
universite.bordeaux.app.Logging
universite.bordeaux.app.Mongo

universite.bordeaux.app.ReadersAn

Classes

CheckCard (90 %)
FileReader (67 %)
LogHandler (0 %)
LogReaderFactory (81 %)

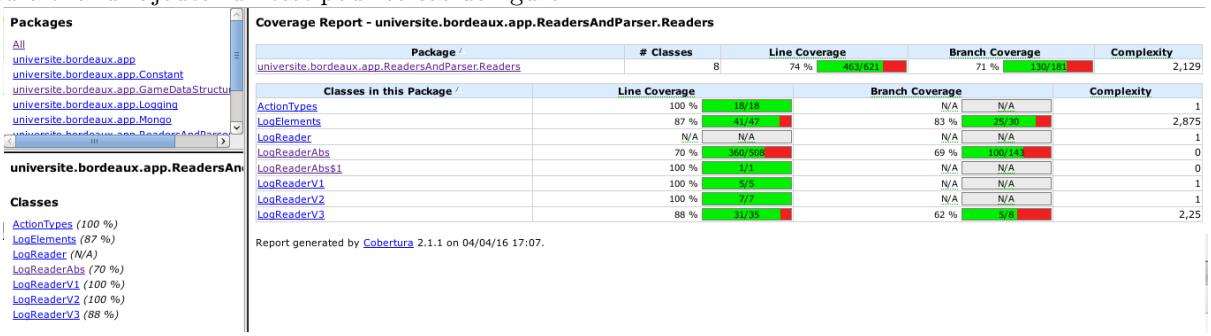
Coverage Report - universite.bordeaux.app.ReadersAndParser

Package	# Classes	Line Coverage	Branch Coverage	Complexity
universite.bordeaux.app.ReadersAndParser	4	31 % 43/136	37 % 15/40	2,417
universite.bordeaux.app.ReadersAndParser.Readers	8	74 % 463/621	71 % 130/181	2,129

Classes in this Package	Line Coverage	Branch Coverage	Complexity
CheckCard	90 % 9/10	100 % 4/4	3
FileReader	67 % 21/31	66 % 4/6	1,75
LogHandler	0 % 0/29	0 % 0/22	0
LogReaderFactory	81 % 13/16	87 % 7/8	4,5

Report generated by Cobertura 2.1.1 on 04/04/16 17:07.

Pour la partie préparant le parsing (c'est-à-dire avant l'utilisation du parser), disons "par manque de temps", nous n'avons pas pu tester le *LogHandler*. Pour le module *FileReader*, nous n'avons pas testé les cas aux limites, ce qui entraine la non-couverture des lancement d'exceptions. Pour le module *LogReaderFactory*, la portion non traitée correspond au cas où une partie du header d'un *Log* est manquante. Il aurait fallu rajouter un test pour ce cas de figure.



Packages

All
universite.bordeaux.app
universite.bordeaux.app.Constant
universite.bordeaux.app.GameDataStructu
universite.bordeaux.app.Logging
universite.bordeaux.app.Mongo

universite.bordeaux.app.ReadersAn

Classes

ActionTypes (100 %)
LogElements (87 %)
LogReader (N/A)
LogReaderAbs (70 %)
LogReaderV1 (100 %)
LogReaderV2 (100 %)
LogReaderV3 (88 %)

Coverage Report - universite.bordeaux.app.ReadersAndParser.Readers

Package	# Classes	Line Coverage	Branch Coverage	Complexity
universite.bordeaux.app.ReadersAndParser.Readers	8	74 % 463/621	71 % 130/181	2,129

Classes in this Package	Line Coverage	Branch Coverage	Complexity
ActionTypes	100 % 18/18	N/A N/A	1
LogElements	87 % 41/47	83 % 25/30	2,875
LogReader	N/A N/A	N/A N/A	1
LogReaderAbs	70 % 360/508	69 % 100/143	0
LogReaderAbs$1	100 % 1/1	N/A N/A	0
LogReaderV1	100 % 5/5	N/A N/A	1
LogReaderV2	100 % 7/7	N/A N/A	1
LogReaderV3	88 % 31/35	62 % 5/8	2,25

Report generated by Cobertura 2.1.1 on 04/04/16 17:07.

Pour le module *LogReaderAbs*, nous n'avons pas testé le cas de figure où plusieurs joueurs gagnent en étant ex-aequo. Le reste du code non couvert correspond à des cas de figures plus rare. Pour obtenir une couverture de ces portions de code, il aurait fallu trouver ou bien construire des *Logs* plus variés. Un certain nombre de facteurs, dont le temps nous a empêché d'effectuer ce travail assez long.

5.2 Bugs rencontrés

Dictionnaire de Cartes Il existe une classe qui sert à verifier l'existence d'une carte et redonner en même temps le nom générique de la carte donnée.
Il y a des problèmes avec les cartes dont le pluriel change radicalement du singulier, par exemple les cartes Colonies (Colony) et Duchies (Duchy).

Pour résoudre le bug il faut changer le dictionaire actuellemement implementé, pour ajouter la conversion de cartes dont le nom au pluriel est plus compliqué que le simple ajout d'un "s".

Pour la classe *LogReaderAbs*, il y a des actions qui ne sont pas testées car, dans les *Logs* selectionnés pour les tests, il n'existe pas toutes les actions possibles pour les Joueurs, Une solution serait d'augmenter la sélection des *Logs* utilisés, pour disposer d'une gamme plus significative d'exemples.

5.3 Module d'analyse des donnée

Cette partie du projet écrite en Python est cruciale, car c'est elle qui sera mise à disposition de l'utilisateur. Nous avons utilisé le module *pytest* pour effectuer nos tests unitaires ainsi que *mongomock* pour simuler l'utilisation d'un serveur *MongoDB*. Les tests de couverture ont été effectués à l'aide de *pytest-cov*.

5.3.1 Politique de tests

Ka stratégie choisie pour les tests a été de les effectuer après le développement. Nouys avons utilisé les outils suivants :

— *Pytest*, un framework destiné à faciliter la création de petits tests, efficaces même pour des fonctions complexes
— *Mongomock*, petite bibliothèque destinée à aider au test de code Python en interaction avec MongoDB

— un fichier Python sur lequel on crée un dictionnaire représentant un document dans la base de données (*Logtest*

5.3.2 Tests unitaires

5.3.2.0.1 module Match Trois tests unitaires ont été effectués :
— un test de comparaison du nom des joueurs.
— un test de comparaison des *Logs* à proprement parler par rapport à un *Log* standard, c'est à dire la vérification que les actions des joueurs au cours de la partie sont bien chargées dans la structure de données.
— un test de comparaison des informations du *header* par le même procédé.
Ces trois tests passent sans poser de problèmes.

5.3.2.0.2 Module Tools Trois tests unitaires ont été effectués :
— un premier test permet de vérifier la bonne création de la pseudo base de données (utilisation de *mongomock*).
— un second test vérifie la bonne création des informations gloabales pour chaque joueur présent dans le log test (2 joueurs).
— le troisième test vérifie que la fonction de calcul d'ELO s'applique correctement, de même que la détection du *greening*.

5.3.3 Tests de couverture

Le résultat des tests de couverture est le suivant :

```
-------------- coverage: platform linux, python 3.5.1-final-0 --------------
Name                                             Stmts   Miss  Cover
--------------------------------------------------------------------
DominionAnalyser/Analysers/Tools.py                 97     13    87%
DominionAnalyser/Analysers/__init__.py               0      0   100%
DominionAnalyser/Match/Log.py                       27      0   100%
DominionAnalyser/Match/Match.py                     21      0   100%
DominionAnalyser/Match/Player.py                    14      0   100%
DominionAnalyser/Match/SimplifiedPlayer.py          12      0   100%
DominionAnalyser/Match/__init__.py                   0      0   100%
DominionAnalyser/Mongo/MongoInterface.py            20      3    85%
DominionAnalyser/Mongo/__init__.py                   0      0   100%
DominionAnalyser/__init__.py                         0      0   100%
--------------------------------------------------------------------
TOTAL                                              191     16    92%
=============== 6 passed, 3 pytest-warnings in 1.20 seconds ================
```

Nous ne couvrons pas totalement le module *Tools* ; cela s'explique par le fait que nous n'avons pas fait de tests sur la reconaissance des stratégies.
Le module *MongoInterface* n'est pas entièrement couvert non plus, car nous ne testons pas la création d'index et de collections.

5.3.3.0.1 Conclusions La partie programmation a demandé beaucoup de travail et a été quasiment le fait d'un seul membre du projet, qui ne pouvait également réaliser la partie test. La répartition du travail ayant abouti à confier les tests aux autres membres de l'équipe, la faible quantité de tests réalisée ne fait que traduire une implication trop limitée et très tardive de ceux-là.

Chapitre 6

Choix effectués et problèmes rencontrés

La nature des données à traiter et les attentes du client ont conduit à étudier plusieurs possibilités et à sélectionner celles qui paraissaient les plus appropriées pour résoudre les difficultés spécifiques du projet.On évoquera dans cette section les choix techniques effectués durant le développement du projet et les répercussions que ces derniers ont eu sur les résultats obtenus.

6.1 Problème de choix de langages

Le choix du langage Java a été réalisé avec un consensus du groupe qui a estimé que ce langage présentait le meilleur rapport programmation / résultat. De fait, ce langage offre de bonnes bibliothèques, et de bonnes ressources de travail adaptées aux besoins du projet. Pour le développement de la bibliothèque, le client a imposé l'utilisation du langage Python, et ce après le début du projet, à un moment où un volume de travail important avait déjà été réalisé. Ajouté à cela le fait que Python soit un langage lent a imposé de rechercher comment en accélérer l'exploitation. C'est à ce moment que le choix de Java s'est révélé inopportun. Car pour que Python fonctionne plus rapidement, il faut utiliser *C-types* et *C-modules*. Il aurait donc mieux valu travailler en C qu'en Java parce qu"ainsi, nous aurions pu lire directement les modules en C. Ces changements ont eu d'importantes répercussions, car à ce moment là, le choix préalable de Java ne s'avérait plus pertinent.

6.2 Problèmes de matériel

La quantité massive de données, et l'espace limité alloué par l'Université pour traiter les données, a obligé à fonctionner sur la base de données avec le maximum de compression, ce qui a rendu le traitement excessivement long.

Chapitre 7

Résultats

L'ensemble des activités menées durant les quatre mois de la durée du projet ont permis d'obtenir un certain nombre de résultats. Il s'agissait d'un projet complexe, offrant à la fois un volume considérable de données à traiter, et des données de structure irrégulière.

7.1 Parser

A l'heure actuelle, les derniers tests réalisés avec le parser lancé dans les serveurs de calcul du CREMI ont montré que le programme a réussi à parser 11742348 *Logs* sur une quantité originelle de un peu plus de 12 millions.

7.2 Bibliothèques

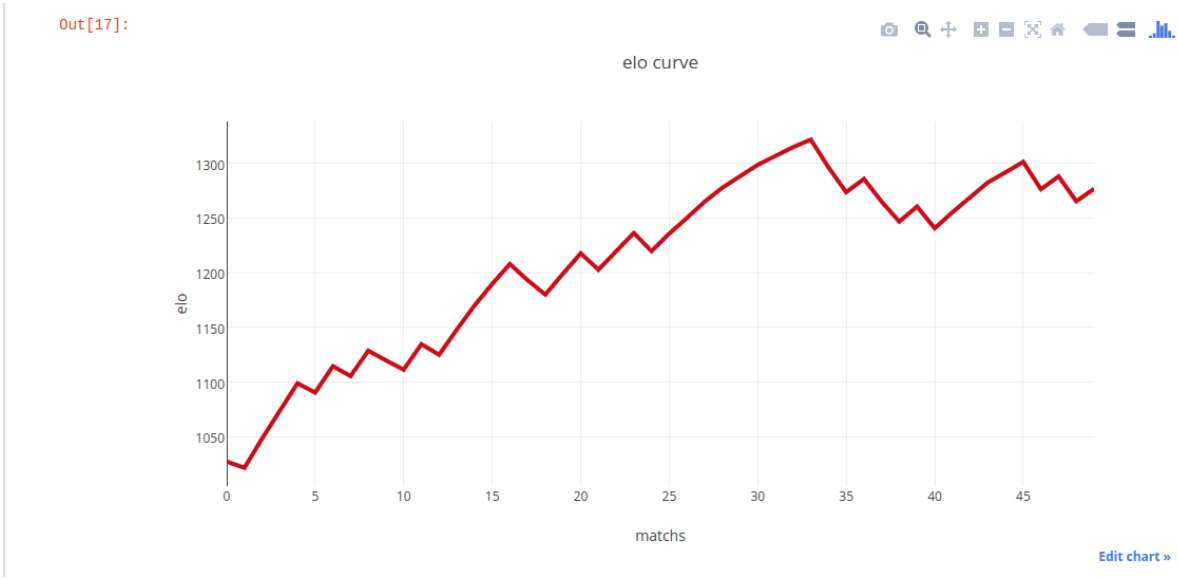
Les figures suivantes montrent des exemples de possibilités d'utilisation fournies par la bibliothèque.

```
In [12]: from DominionAnalyser.Analysers import Tools
from DominionAnalyser.Mongo import MongoInterface
import plotly.plotly as py
import plotly.graph_objs as go

In [2]: Tools.generate_player_table()

0%                                                                          100%
[###########################################################################] | ETA: 00:00:00 | Item ID
Total time elapsed: 00:05:51

In [3]: Tools.generate_elo()

0%                                                                          100%
[###########################################################################] | ETA: 00:00:00 | Item ID
Total time elapsed: 00:15:16

In [5]: print(MongoInterface.logs_col.find_one().get("winners"))

['searsjs']

In [6]: playerCurve = Tools.generate_player_elo_curve("searsjs")

0%                                                  100%
[###################################################] | ETA: 00:00:00 | Item ID: 5703e4ff2336eb5ea0
Total time elapsed: 00:00:00

In [10]: index = list(range(len(playerCurve)))
```

```
In [13]: trace0 = go.Scatter(
x = index,
y = playerCurve,
name = 'searsjs elo curve',
line = dict(
color = ('rgb(205, 12, 24)'),
width = 4)
)

In [14]: data = [trace0]

In [15]: layout = dict(title = 'elo curve',
xaxis = dict(title = 'matchs'),
yaxis = dict(title = 'elo'),
)

In [17]: py.sign_in('******', '*****')
fig = dict(data=data, layout=layout)
py.iplot(fig, filename='styled-line')
```

```
High five! You successfuly sent some data to your account on plotly. View your plot in your browse
```

Out[17]:




```
In [18]: greening = Tools.generate_greening_list()
```

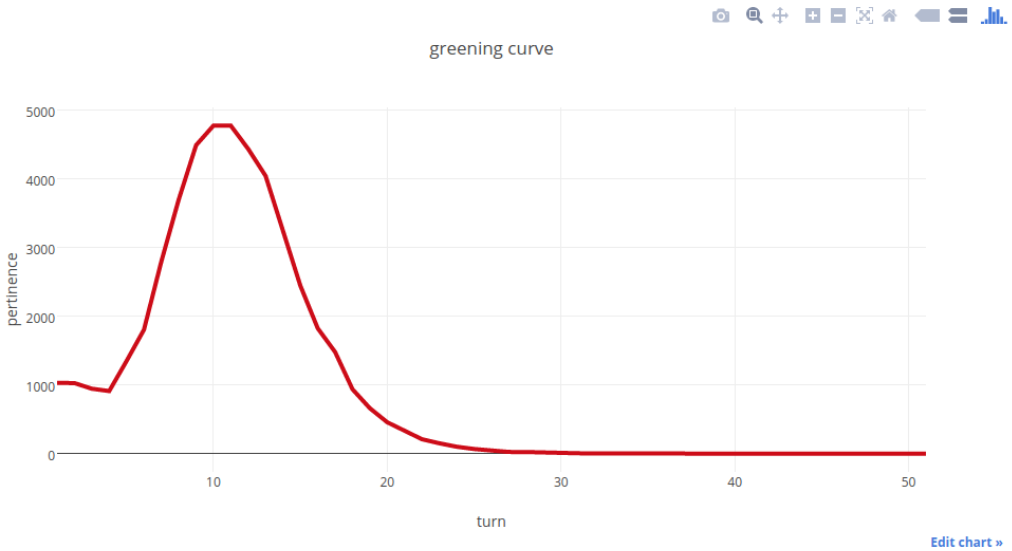
```
0%                                                                                100%
[##############################################################################] | ETA: 00:00:00 | Item ID
Total time elapsed: 00:02:26
```

```
In [25]: index =[]
value =[]
for k in sorted(greening):
index.append(k)
value.append(greening[k])


trace0 = go.Scatter(
x = index,
y = value,
name = 'greening curve',
line = dict(
```

```
color = ('rgb(205, 12, 24)'),
width = 4)
)

data = [trace0]

layout = dict(title = 'greening curve',
xaxis = dict(title = 'turn'),
yaxis = dict(title = 'pertinence'),
)
py.sign_in('*******', '********')
fig = dict(data=data, layout=layout)
py.iplot(fig, filename='styled-line')
```

```
High five! You successfuly sent some data to your account on plotly. View your plot in your browse
```

Out[25]:



In [ ]:

In [ ]:

Bibliographie

[1] Wiki contributors. Wiki for dominion strategy. <http://wiki.dominionstrategy.com/index.php/Main_Page>, 2016. [Online ; accessed 05-April-2016].

[2] Glickman Mark E. Rating the chess rating system. <http://www.glicko.net/research/chance.pdf>, 1999. [Online ; accessed 05-April-2016].

[3] G. Fernandez. Data mining using SAS Applications. Chapman and Hall/CRCChapman and Hall/CRC, 2003.

[4] Jiawei Han. Data mining : concepts and techniques. Elsevier, 2012.

[5] Daniel T. Larose. Discovering Knowledge in Data : An Introduction to Data Mining. Wiley, 2004.

[6] Daniel T. Larose. Exploration de donnees : methodes et modeles du data mining. Vuibert, 2012.

[7] S. Tuffery. Data Mining et statistique decisionnelle : l'intelligence dans les bases de donnees. Technip, 2007.