**Drasus: un jeu de stratégie au tour par tour.**

***Projet Programmation JAVA***

**DAVID Aurélien, PEQUIGNOT Charlotte, RAMOS Adrien**



SOMMAIRE

[Drasus en quelques chiffres : 2](#_Toc345782299)

[Introduction 2](#_Toc345782300)

[Description de l'application. 3](#_Toc345782301)

[Diagramme UML des cas d’utilisation 5](#_Toc345782302)

[Diagramme UML de classes 6](#_Toc345782303)

[Réalisation de l’application 8](#_Toc345782304)

[Emploi de Design Pattern 8](#_Toc345782305)

[Liste des fonctions développées 8](#_Toc345782306)

[Liste des tests automatiques effectués 8](#_Toc345782307)

[Difficultés rencontrés 9](#_Toc345782308)

[Conclusion 10](#_Toc345782309)

# Drasus en quelques chiffres :

* 1 Jeu Open-source et un Wiki d'explication des règles.
* 3 différents design patterns utilisés parmi les plus connus.
* Environs 60 classes dont de nombreuses classes abstraites, interfaces ainsi que des exceptions personnalisées.
* Plus de 100 commits sur notre gestionnaire de version git.
* Environs 150 heures de code et de conception pour 3000 lignes de code rédigé intégralement par les 3 développeurs et concepteurs du jeu.

# Introduction

En cette troisième année de l’EFREI, le cours de Programmation en Java est venu en remplacement de l’enseignement en programmation C++ dispensé l’année précédente. Ce changement a impliqué de nouvelles attentes: changer sa façon de concevoir un programme informatique, celui-ci passe d'abord par une phase d'analyse puis de conception.

Le sujet étant libre, nous avons rapidement choisi de coder un jeu stratégique en tour par tour que nous affectionnons tout particulièrement. Au-delà d’un projet scolaire de Java, nous souhaitons avant tout réalisé un jeu qui pourrait évoluer post-rendu de projet.

Il nous a donc fallu étudier longuement la façon dont nous allions aborder le jeu, le concevoir et enfin l'optimiser. Cela a été rendu plus simple du fait que nous travaillons pour la première fois en trinôme.

Le dernier projet de l’année précédente, le générateur d’emploi du temps en C++ a été pour nous l’occasion de travailler à une conception objet avancée, cette fois-ci nous choisi d’y consacrer encore plus de temps en utilisant notamment plusieurs design pattern.

# Description de l'application.

Nous avons décidé de faire de Drasus un jeu de stratégie au tour par tour avec la particularité qu’il serait en réseau, c’est-à-dire que deux joueurs humains s’affronteraient. Le principe est simple, chaque joueur doit gérer ses unités en plus de son général (dragon ou pegasus). Les joueurs peuvent choisir leurs généraux avant de lancer la partie. Chaque unité est unique, et nous avons inclus quelques principes du RPG: la monté en puissance lorsqu’une unité en tue une autre, elle gagne un pouvoir particulier. Les joueurs peuvent ainsi élaborer leurs stratégies sur une spécificité de certaines unités et leurs placements. Ces placements seront importants car il faudra jouer avec les éléments de la carte. Vous pourrez bloquer vos ennemis dans une concave ou alors utiliser les forts présents pour vous régénérer. Une fois le général ennemi éliminé, la partie est remportée.

Ce projet a été réalisé dans un nouveau langage pour nous, le Java en utilisant la librairie graphique dédiée aux jeux Slick2D ainsi que Kryonet pour la gestion de la partie réseau et Junit pour les tests unitaires. Ces choix nous ont permis de réaliser une interface utilisateur complète et agréable tout en utilisant le modèle MVC qui facilite grandement l'organisation du code.

Kryonet est dérivé de Kryo, qui est un serialiseur qui nous a été bien utile, car avant d’envoyer des objets par le réseau via Kryonet, il faut les sérialiser tous. Notre architecture réseau est très simple, l’architecture client-serveur. Les deux clients vont se connecter à un serveur dont le rôle sera uniquement de faire transiter les informations entre les deux clients. Aucun calculs ni aucune opération n’est faite sur le serveur.

Ainsi nos sources se composent de quatre packages, eux même divisés en plusieurs subpackages :

* model, qui contient la partie données du jeux, c'est à dire les joueurs et les unités (model.units) et les attaques (model.attack) ainsi que les exceptions personnalisées (model.exception).
* view, qui comprend notre interface graphique les différents états du jeu et les actions des utilisateurs.
* Le package controller, qui gère et vérifie les actions permettant de relier des packages sus-jacents à l'aide de plusieurs classes de contrôle.
* Et enfin notre package test qui contient les tests unitaires de notre programme.

Afin de concevoir ce projet facilement nous avons utilisé comme à notre habitude le gestionnaire de version Git. Le système de commit nous permettant de savoir sur quelle partie du code chacun des membres du trinôme avait travaillé et de comprendre les modifications. De plus l'utilisation de différentes branches nous a permis de travailler chacun de notre côté sans gêner les autres ni attendre la fin de leurs modifications... De plus le développement du réseau impliquait des contraintes lors des tests: lancement du serveur et des 2 clients, c'est pour quoi nous avons dédié le réseau à une branche parallèle à master tout au long du développement et accéléré ainsi le développement du programme, tout en s'assurant que la partie réseau était fonctionnelle au fur et à mesure de notre progression.

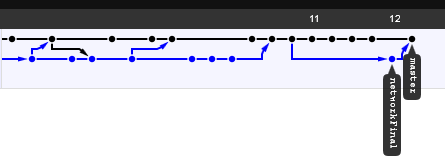


Figure 1 - Illustration du développement

Nous veillons à rendre notre projet Open-Source une priorité, c'est pourquoi nous attachons une grande importance au code source qui a été rédigé intégralement en anglais, sauf pour les classes concrètes des unités qui reprennent le nom de l'unité en jeu par question de clarté. Les commentaires respectent la convention Javadoc afin de générer facilement une documentation complète.

## Diagramme UML des cas d’utilisation



## Diagramme UML de classes du model



# Réalisation de l’application

## Emploi de Design Pattern

En codant ce jeu nous sommes partis du principe que notre jeu allait être amené à évoluer et prendre de l'ampleur au fil de la conception via de nouvelles idées mais également au delà du rendu de projet ! Nous concevons en effet ce projet avec la prévision de continuer son développement...

Chaque Boss permet d’avoir des unités personnalisés c’est pourquoi nous avons utilisé le Design Pattern de double fabrique abstraite afin d’implémenter facilement la personnalisation des unités en fonction du choix du boss mais également rendre simple l’ajout d’un autre boss, d’une nouvelle unité propre à un seul boss ou bien les deux.

Chaque unité possède son propre mode d’attaque qui peut varier au cour de la partie en fonction des améliorations, c’est pourquoi nous avons employé le pattern stratégie afin de privilégier l’encapsulation des attaques plutôt que de se baser uniquement sur l’héritage qui aurait compliqué le changement de mode d’attaque.

MainController est notre classe principale, elle contient le model, instancie la vue et fait la liaison entre les deux. Il est donc important de s’assurer qu’elle n’est instancié qu’une seule fois tout en restant utilisable au sein de plusieurs classes, nous avons donc utilisé le pattern singleton.

## Liste des fonctions développées

Chaque joueur choisi un Boss et se connecte au serveur du jeu qui va le mettre en relation avec un autre joueur. Chaque joueur place ces unités de son coté de la carte puis la partie peut commencer.

A chaque tour un joueur peut déplacer, faire attaquer ces unités, activer le pouvoir d’une unité…

Lorsqu'un joueur tue le boss adverse il obtient le menu de victoire et l'autre joueur le menu de défaite.

## Liste des tests automatiques effectués

De part notre architecture nous contrôlons chacune des actions du joueur avant de modifier le model, ce qui implique de nombreux tests, comme ne pas déplacer une unité sur une case du jeu qui est bloquée ou sur laquelle est déjà présente une unité. On ne peut évidemment pas attaquer ses propres unités mais uniquement celles de son adversaire. Pour rechercher une case accessible lors d'un déplacement de joueur nous effectuons une recherche de tous les chemins possible.

En plus de ces tests nous avons accordé une importance aux tests unitaires via la bibliothèque dédiée aux tests Junit. Ces derniers sont destinés à vérifier que le jeu fonctionne malgré un ajout ou une suppression de code. Par exemple si l'on ajoute une unité à la vue il s'assurera qu'une fabrique permet de la créer. Les tests s'assurent également que toutes les unités ont la possibilité d'être supprimée ou bien d'activer un pouvoir, etc...

## Difficultés rencontrés

Un jeu comme Drasus est bien plus complexe à programmer que nous le pensions lorsque nous avons commencé à rédiger les fonctionnalités du programme. Avant de commencer à implémenter il est essentiel de s’assurer d’avoir une structure solide qui va être amené à évoluer et répondre à tous les besoins du jeu en sachant que ceux-ci seront amenés à évoluer.

La partie réseau, bien que courte, a nécessité beaucoup de logique lors de son implémentation. C’était la première fois pour nous que nous devions faire gérer des clients par un serveur. Nous avons donc créé une nouvelle classe Msg possédant 3 attributs : une chaine de caractère (qui sert à échanger des messages de combat durant la partie), et deux booléens, qui sont utilisé au début lors de la mise en relation avec le serveur pour s’assurer que c’est la première connexion qu’ils tentent et par la suite lors de l’échange des messages. Le serveur contrôle donc les booléens afin de savoir comment gérer le message qu’il vient de recevoir.

# Conclusion

Pour un premier projet libre de Java nous sommes fiers du résultat obtenu et ceux-malgré un bon nombre d’heure de travail conséquent.

Programmer ce jeu nous a permis avant tout de consolider notre programmation en Java mais aussi d’avoir un certain recul sur la difficulté à produire un jeux vidéo, étant donné le nombre de contraintes importants.

Pour terminer, c'était certainement le projet le plus ambitieux que nous ayons eu la chance de réaliser...

# Annexe

Github

L'intégralité du gestionnaire de version du projet est disponible de façon open source à l'adresse suivante:

<https://github.com/spyl94/drasus>

Wiki

Le wiki contenant l'intégralité des règles du jeu et les caractéristiques des unités:

<https://github.com/spyl94/drasus/wiki>

Sources

* [http://www.slick2d.org/wiki/index.php](http://www.slick2d.org/wiki/index.php%20) Le wiki de slick2d pour comprendre l'utilisation de la librairie.
* <http://www.slick2d.org/javadoc/> La Javadoc de slick2d.