Lies Benassou, 14 191 992

Raphael Bernier, 16 124 953

Jonathan Clavet-Grenier, 16 070 207

Philippe Dagenais, 15 168 630

Raphael Laplante, 16 059 239

Patrick Limoges, 16 084 112

Samuel Proulx, 16 038 798

Vincent Quernel, 14 037 357

Projet en programmation agile

Jeu d’échec

Rapport Équipe

Dans le cadre du cours IFT232

Université de Sherbrooke

21 décembre 2016

Table des matières

[Introduction 3](#_Toc470028460)

[Architecture 4](#_Toc470028461)

[Architecture finale vs architecture anticipée 7](#_Toc470028462)

[Patron de conception 7](#_Toc470028463)

[1. Factory 7](#_Toc470028464)

[2. Singleton 8](#_Toc470028465)

[3. MVC 9](#_Toc470028466)

[4. UTILITY 9](#_Toc470028467)

[5. Null Object 10](#_Toc470028468)

[Installation et utilisation 10](#_Toc470028469)

[Conclusion 10](#_Toc470028470)

# Introduction

Pour le cadre du projet final de programmation orientée objet, notre équipe a décidé de programmer et présenter un jeu d’échec. Ce jeu d’échec permettra à deux joueurs de jouer sur un même ordinateur, suivant les règles standards des échecs, tout en y intégrant quelques ajouts, comme la sauvegarde et le chargement de partie.

Pour ce rapport, nous commencerons par présenter l’architecture du jeu d’échec, celle-ci étant présentée grâce à un diagramme, construit selon les standards UML.

En deuxième lieu, nous vous présenterons l’analyse initiale du projet, puis nous la comparerons avec l’analyse de la version finale du projet, pour présenter un compte rendu du travail effectué.

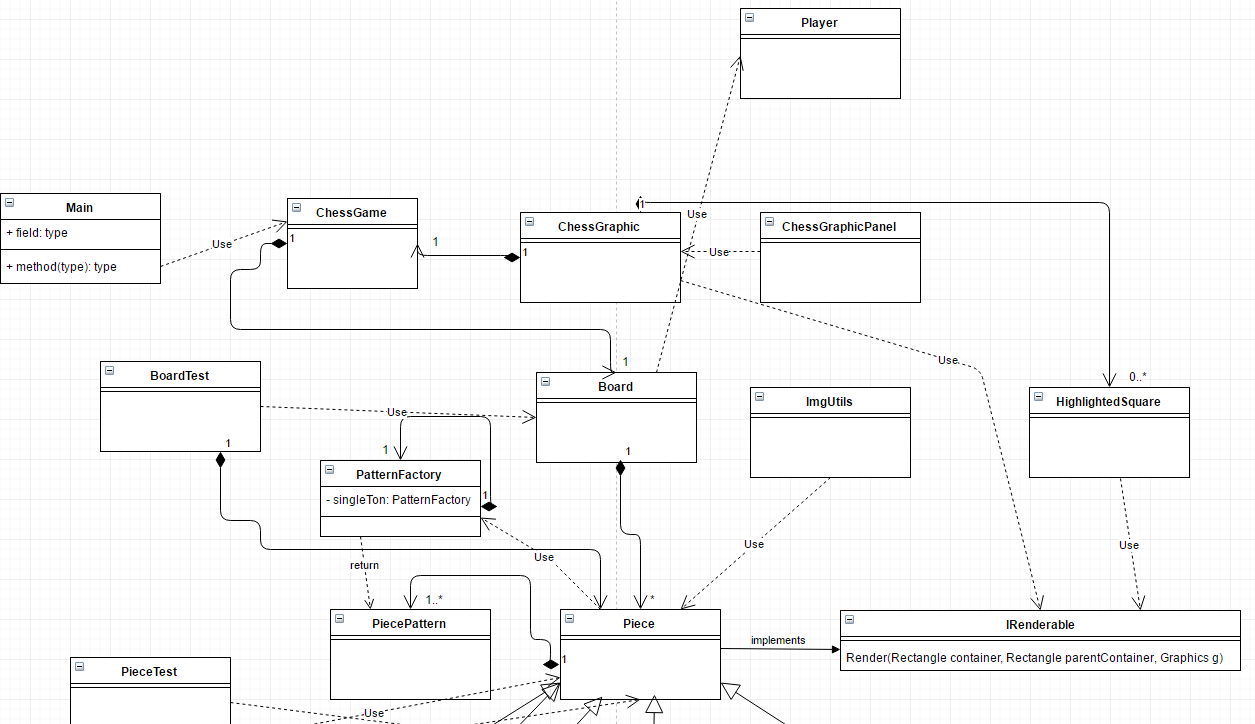
Suivant cela, nous présenterons les différents patrons de conception utilisés dans le projet, incluant une analyse de leurs utilités dans la globalité du projet.

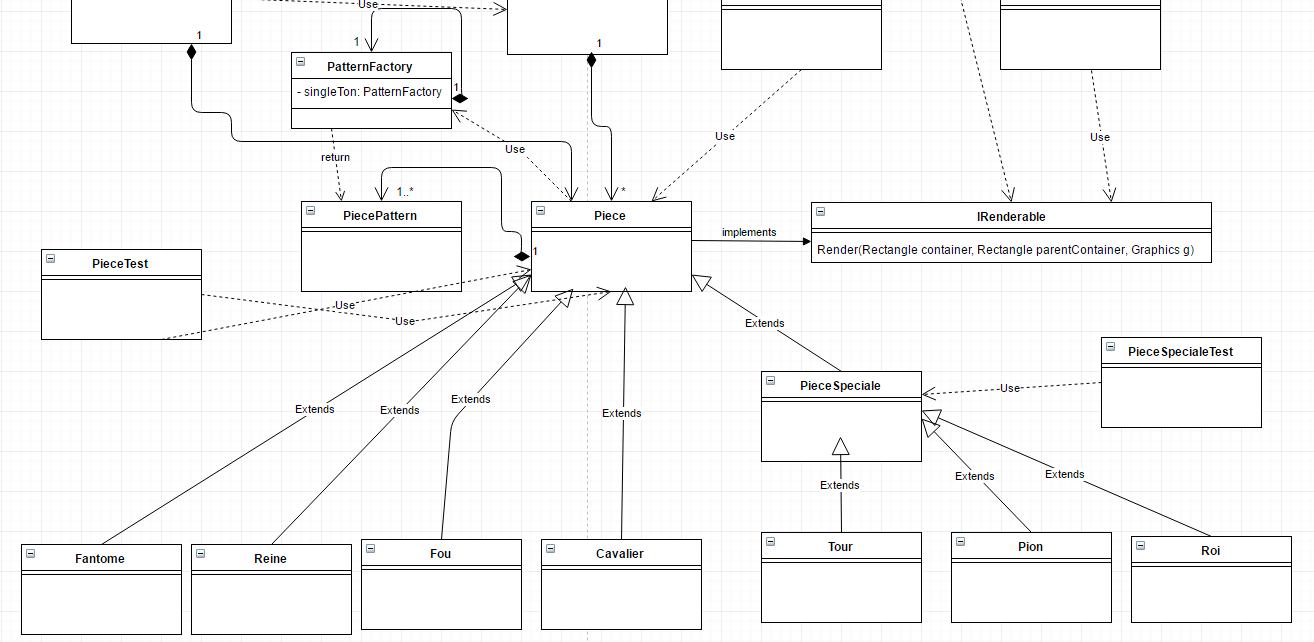
Nous nous attarderons quelque peu sur les différents logiciels utilisés, pour présenter notre méthode de travail.

Puis, pour vous permettre d’apprécier au maximum notre jeu, nous inclurons un petit paragraphe afin de vous permettre d’effectuer une partie d’échec sans désagrément.

Nous finirons alors, par une conclusion qui fera la synthèse du projet, ainsi que de ci document.

# Architecture





**Diagramme UML**

Les classes les plus importantes sont ChessGame, Board et ChessGraphics. Ce sont les classes qui contiennent le plus de code et sans ceux-ci, il aurait beaucoup de travail à faire. Nous ne considérons pas les classes des pièces importantes puisqu’ils ne prennent pas longtemps à coder. En ce qui concerne les instances, ce sont les mêmes que les classes, soit « game » dans ChessGraphics, « Board » dans ChessGame et « graphics » dans ChessGame. Game sert pour la vue, soit ChessGraphics, qui va appeler la fonction « play » de ChessGame qui va exécuter un coup selon le X et le Y reçu en paramètre. « Board » est aussi important, car il permet à ChessGame de déterminer ce qu’il devrait faire selon les informations qu’on reçoit de l’instance « Board ». Finalement, sans « graphics », il serait impossible d’afficher l’échiquier actuel à la personne qui joue au jeu.

Finalement, il n’y a aucun élément externe à prendre en considération. Il suffit simplement d’exécuter le client et le tout va marcher sans aucun soutien externe.

# Architecture finale vs architecture anticipée

Il y a définitivement plus de classes, de méthodes et de données membres de ce que nous avons anticipé. Donc, on peut facilement en déduire que nous avons grandement sous-estimé le nombre de classes, de méthodes et de données membres si on se fie à notre diagramme UML final. Il y a une raison bien simple à cela : Le manque d’expérience. Malgré que certains d’entre nous ont une technique, on peut facilement constater que cela n’est réellement pas suffisant pour bien déterminer toutes les classes et les méthodes du projet le plus précis possible. À la base, nous avions mis une donnée membre « Board » pour chaque pièce ce qui est évidemment très mauvais. Nous avons vite remarqué cela lorsque nous avons appliqué des techniques de refactorisation. Nous avons aussi ajouté plusieurs patrons qui nous sont révélés utile. Par exemple, le *Factory* et le *Utility*. Ceux-ci nous ont été utiles à plusieurs places et rendaient le code vraiment plus beau comparé à l’ancien code.

Bref, nous avons fini par toutes changer la structure de base pour l’accommoder à une nouvelle structure efficace et qui permet maintenant au jeu de respecter plusieurs méthodes de conception telle que vue dans le cours.

# Patron de conception

Dans cette section du rapport, nous ferons l’analyse des différents patrons de conception, utilisés pour résoudre des problèmes ou simplifier l’entretien du jeu.

Nous commencerons donc par démontrer l’efficacité de l’utilisation du patron de la fabrique, couplé à celle du singleton, dans la classe « PieceFactory ».

Puis, nous aborderons le patron de conception des commandes utilisées pour simplifier l’exécution des différents boutons de la barre d’outils du jeu.

## Factory

Le premier patron de conception analysé est celui de la fabrique. Nous l’abordons en premier, car elle est très rapidement distinguable face aux autres patrons de conception, du fait que la classe qui l’utilise porte le nom du patron : « PieceFactory ».

Le fait d’utiliser le patron de la fabrique peut sembler être trivial, mais aussi sembler très inutile.

En effet, dans le cadre d’un jeu d’échec, lorsqu’une nouvelle partie est créée, c’est toujours les mêmes pièces, disposées au même endroit, qui sont placées sur l’échiquier. Cependant, la création de pièce ne s’arrête pas là. Comme dit antérieurement, notre jeu d’échec supporte le chargement de partie, ce qui veut dire que l’on peut avoir de nouvelle partie dont le nombre de pièces peut différer. De plus, cela serait oublier une règle du jeu d’échec – la promotion –qui permet à une pièce de changer.

De ce fait, l’utilisation du patron de conceptions de la fabrique est pleinement justifiée, car la création de pièce peut varier et survenir à n’importe quel moment dans la partie.

## Singleton

Fortement couplé avec la fabrique, il va de soit de le présenter un deuxième. De plus, comme cette dernière, l’utilisation principale du singleton est faite dans la classe : « PieceFactory ».

L’utilisation de ce patron de conception est fortement liée à la conception de la classe qu’il’ utilise. En effet, lors de la conception de « PieceFactory », notre équipe a déterminé qui serait bon d’y implanter ce patron de conception dans un but futur.

En effet, tel qu’il est maintenant, le jeu d’échec ne permet que d’avoir un seul damier d’ouvert à la fois, ce qui rend l’application du singleton inutile, car une instance de « PieceFactory » dans la classe « Board » est suffisante.

Cependant, à cause de l’évolution naturelle du projet, nous avons décidé de placer la classe « PieceFactory » en singleton pour ne pas avoir une redondance inutile entre les instances de « Board » qui utilisent toutes exactement la même instance de « PieceFactory ».

Donc, cela justifie pleinement l’utilisation du singleton.

## MVC

Considérant que l’interaction que notre application doit avoir avec l’utilisateur, nous avons trouvé que la situation était appropriée pour l’utilisation du patron MVC pour mieux séparer notre code pour mieux les identifier en trois sections distinctes.

Les classes dans le package « Piece » ainsi que la classe « Board » représente les modèles, car ce sont eux que nous prenons les données pour les afficher à la vue. « ChessGame » est le contrôleur, car c’est la classe qui gère tout selon ce que le « board » nous retourne (qui prend les informations des pièces). Bref, selon ce que le « board » nous retourne, on fait une série d’action et ensuite, on met à jour notre vue qui est « ChessGraphics ». « ChessGraphics », à l’aide des informations obtenus du contrôleur (qui est « ChessGame ») met à jour le « board » au complet et affiche le « board » le plus à jour à l’utilisateur.

Ce patron nous a permis de mieux organiser notre code et a divisé certaines parties du projet en trois parties spécifiques. Cela a donc permis à certaines personnes de seulement travailler sur les données (« Pieces » et « Board »), d’autres seulement sur le contrôleur et/ou la vue.

## UTILITY

Le patron Utility nous permet de définir une méthode « static » qui sera réutilisée quelquefois dans notre code. C’est pour cela que nous l’avons choisi pour nous faciliter le travail lorsque la fenêtre change de taille. En effet, lorsque la dimension du jeu est modifiée, l’échiquier sera réajusté.

Lorsqu’un utilisateur choisit de modifier la taille du jeu, il faut que nous réajustions la taille de nos images pour chacune de nos pièces, notre échiquier ainsi que les carrés surlignés pour montrer les mouvements possibles de la pièce. Alors, nous avons fait une méthode « static » qui sera appelée pour modifier les dimensions des images pour que le ratio que l’on avait précédemment soit conservé.

Nous avons trouvé que ce patron nous a facilité le travail en ce qui concerne le changement de la taille du jeu, puisque nous avions tout d’abord pensé à garder la fenêtre telle qu’elle est et que l’utilisateur ne puisse pas la « resize », mais ce patron nous a permis d’effectuer la modification sur plusieurs composantes du jeu.

## Null Object

L’utilisation du patron « Null Object » est utilisée pour la création de joueur. Lors de la création d’un joueur, on doit passer en paramètre un énumérateur pour le choix de couleur. Par contre, le choix du nombre de joueurs est défini par « ChessGame », car c’est le jeu qui détermine combien de joueurs peuvent jouer. De ce fait, l’utilisation du patron « Null Object » prend tout son sens. Le constructeur de joueur étant privé, on force l’utilisateur à créer un joueur avec en paramètre la couleur (énumération). Puisque présentement le jeu ne se joue qu’à deux joueurs (blanc et noir), nous créons seulement un joueur si la couleur est blanche ou noire. Sinon, on retourne null afin de s’assurer qu’il n’y ait pas plus que deux joueurs. Finalement, il peut aussi servir si l’on veut comparer un joueur actuel à une couleur en particulier, on a une garantie que le joueur n’aura jamais « null » (sauf s’il a mis l’instance elle-même à null), mais sinon, il aura toujours une couleur d’associé, donc faire une vérification différente de « null » n’est absolument pas nécessaire dans un cas où on souhaite vérifier la couleur du joueur et la comparer à un autre. Au lieu de mettre un joueur à « null » lorsqu’il ne contient aucune équipe et de vérifier si le joueur est différent de « null » pour vérifier s’il appartient à une équipe, on peut immédiatement comparer sa couleur à une autre sans faire la vérification de « null ».

# Installation et utilisation

Étant donné que l’application ne fournit pas un installateur, l’utilisateur peut récupérer l’application directement sous la forme d’une archives et la décompresser à l’endroit souhaité sur son disque dur. L’application est donc facilement portable et peut même être installée sur une clé usb. Puisque nous avons choisi de développer en Java, l’hôte devra avoir préalablement installé Java Runtime Environment, qui permet d’exécuter toute application développée en Java sur n’importe quel système d’exploitation supportant JRE. Finalement, il est nécessaire que l’hôte soit équipé d’un écran et d’une souris, sans quoi il ne sera pas possible pour l’utilisateur d’interagir avec notre application.

# Conclusion

En ce qui concerne l’organisation, le plus difficile est de céduler des rencontres avec tous nos collègues pour faire au moins une session de « peer programming » avec chacun. Les horaires de tous étant différents, il est parfois difficile de déterminer une date de disponibilité dans les jours qui suivent. Ceci dit, ce problème ne devrait pas avoir lieu dans un environnement de travail, considérant que la majorité des employés d’une entreprise travaillent durant la même plage horaire. Concernant le travail en équipe de deux, cela est très enrichissant. Il est très rare que deux personnes aillent s’entendre sur la façon exacte de mettre en place une fonctionnalité puisque chacun a une expérience différente, chacun a ses propres conceptions d’une solution et que chacun ne programme pas de la même façon. On a pu constater que les membres de l’équipe ne respectaient pas la même nomenclature puisque la majorité a une expérience plus approfondie dans un autre langage qui privilégie une nomenclature différente. Sur une petite base de code, cela a peu d’impact, mais sur un projet de grande envergure, il va de soi qu’il est plus facile de naviguer à travers les classes si elles sont toutes écrites de la même façon. En ce qui a trait à la programmation orientée objet, on peut en conclure qu’elle facilite la gestion du code entre les membres de l’équipe lorsque les patrons sont adéquatement appliqués. En effet, lorsqu’une équipe a comme objectif de mettre en place une nouvelle fonctionnalité, cette dernière va généralement utiliser de nouvelles classes qui seront dans des fichiers séparés (une classe par fichier en Java). Cette séparation en classe découle bien souvent de l’application de certains patrons de conception. Puisqu’il s’agit de fichiers différents, il n’y a presque aucune collision entre le code d’une équipe et celui d’une autre équipe, considérant qu’elles ne travaillaient pas sur la même fonctionnalité, ce qui facilite grandement les « commit » avec git. De plus, la division en plusieurs fichiers et l’utilisation de l’héritage avec les patrons de conception permet de réutiliser facilement une interface crée par une équipe dans une nouvelle fonctionnalité gérée par une autre équipe. Tout cela permet de faciliter grandement la maintenance de l’application en permettant d’ajouter de nouvelles implémentations d’une fonctionnalité sans même toucher à l’originale. Suite en cela, il en va de soi que la flexibilité est la principale mission des patrons de conception. Bien que le jeu d’échec ne soit pas complet en soi, on peut tout de même dire que le projet est un succès puisqu’il nous a permis d’expérimenter le « peer programming », plusieurs patrons de conception et le réusinage comme on n’en avait jamais fait par le passé. Finalement, bien que le projet ne soit pas complet, il est tout de même dans une version fonctionnelle, c’est-à-dire qu’il faudrait simplement effectuer quelques itérations supplémentaires pour atteindre une version livrable. Considérant l’expérience obtenue ainsi que le résultat actuel, on peut être satisfait de notre travail dans le cadre de ce cours.