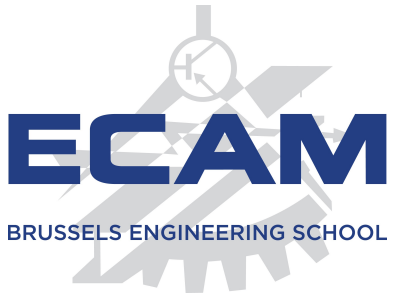


Test Driven Development Project

Décembre 2019

DENIS Maxime 15089
SMITS Victor 16107
TOURNEUR Bernard 16148

Ecole centrale des arts et métiers

MA1 - Groupe 1

Année académique 2019-2020

Contents

1 Introduction

Dans le cadre du cours *Architecture logicielle*, il nous a été demandé, dans un premier temps d'établir un projet en *Java*. L'important lors de la réalisation de ce projet était de mettre en place différents critères de qualité et de contrôler ceux-ci. Après 4 séances, les différents projets ont été redistribués à travers les groupes dans l'optique d'améliorer la qualité du programme et non sa fonctionnalité.

Une fois le nouveau projet récupéré, il convient d'en faire une analyse approfondie ainsi qu'une analyse des différents critères de qualité mis en place. C'est sur base de ces critères que nous effectuerons notre travail de refactoring du code. Dans ce dossier se trouve les différentes analyses, problématiques, solutions et améliorations relatives à ce projet. Les objectifs atteints ou non y sont détaillés ainsi que les modifications apportées.

L'ensemble du projet peut être trouver sur *Github* en cliquant ici ou à l'adresse suivante: https://github.com/victorsmits/Ludotheque.

L'intégration continue du projet a été réaliser grâce au système Jenkins et est retrouvable au en cliquant ici ou en vous rendant sur la platform jenkins de l'ecam dans le projet *Ludotech*.

2 Analyse de l'existant

Le projet du groupe 3[1] consiste en la gestion d'une ludothèque, un centre de prêt de jeux en tout genre; jeux vidéos, jeux de société ou autres jouets.

Selon les spécificités décidées par le groupe 3, un jeu doit avoir un nom, un statut de disponibilité et un fabricant, quel que soit son type.
Certains types de jeux spécifiques ont des caractéristiques propres, ce qui entraîne l'utilisation de l'héritage, décrite au point 2.1.

L'application devrait permettre au gérant de s'identifier, et de gérer la ludothèque; ajouter des jeux, faire des recherches, obtenir une liste de prêts, etc.
Elle devrait également permettre aux adhérents de faire des recherches de jeux, voyant lesquels sont disponibles ou non et de déclarer un emprunt.

L'application, codée en java, utilise les concepts de programmation orientée objet. La structure du programme est donc composée de classes représentant les différents objets ou acteurs intervenant.

2.1 Structure

Classes et fonctionnalités

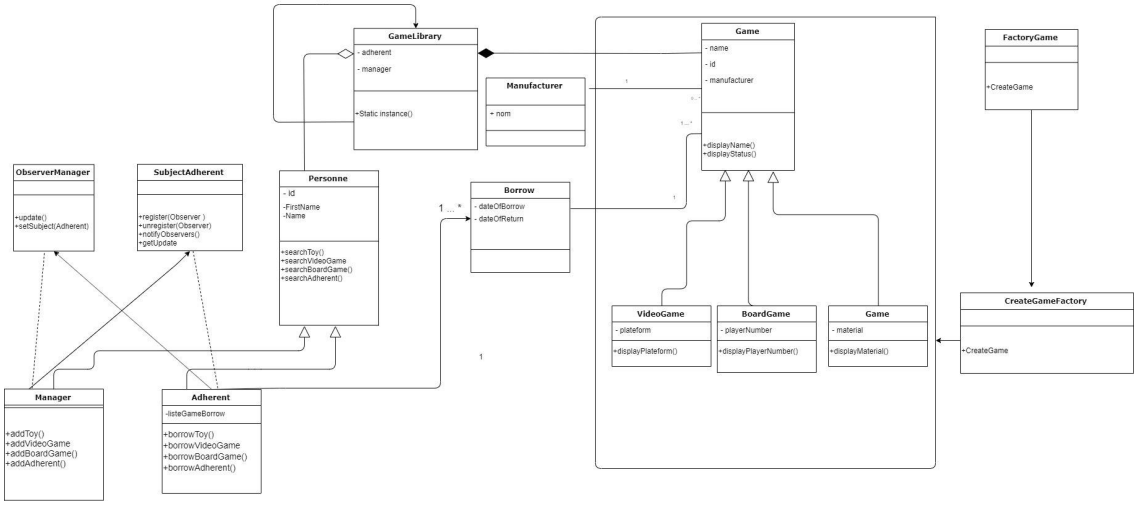


Figure 2.1.1: Diagramme de classe

[1]L'équipe composée de Louis Randriamora Andriantsiory, Kolawole Abdoulaye et Rudy Itunime Masala

Le diagramme de classe en figure 2.1.1 représente la structure du projet que nous avons auditionné, et est repris en annexe en figure 9.1.15 pour plus de lisibilité.

On observe sur ce diagramme que la première équipe de dévelopement avait créé une classe Game, de laquelle héritaient 3 sous-classes Game, BoardGame et VideoGame. Le diagramme ne semble pas à jour[2], puisque la classe-fille Game a été remplacée par une classe Toy. La super-classe Game est une classe abstraite et ne peut pas être implémentée.

4 différentes interfaces étaient présentes dans le code et ne sont pas reprises dans le diagramme de classe[3]. Ces interfaces sont des parties de design patterns qui seront décrits en sous-section 2.1 : Design.

En outre, les relations entre les classes ne sont pas toutes claires, comme celle entre la classe Borrow et la classe Game, et celle entre la classe Manufacturer et la classe Game. Les aggrégations et compositions sont quant à elles plus claires; une GameLibrary est composée de Game, et a des Personne[4], soit Manager soit Adherent.

Design patterns

Le groupe 3 a choisi d'utiliser plusieurs designs patterns : l'observer, le facade, et le factory.
Le design pattern observateur est utilisé pour envoyer des signaux à des modules, les observateurs. En cas de notification, les observateurs peuvent effectuer l'action adéquate en fonction des informations reçues. En pratique, le groupe 3 a implémenté l'observateur mais ne l'utilise pas.
En effet, les interfaces ObserverManager et SubjectAdherent existent et présentent des méthodes typiques du design pattern (notifyObserver chez le subjectAdherent, setSubject qui est l'action de l'observateur), mais ne sont jamais utilisées et leurs méthodes jamais appelées.
Le design pattern observer suit typiquement la structure représentée en figure 2.1.2.

[2]Par rapport à l'état dans lequel nous avons récupéré le projet
[3]AdherentFacade, ObserverManager, SubjectAdherent et UserFacade
[4]Noté en français dans le diagramme de classe

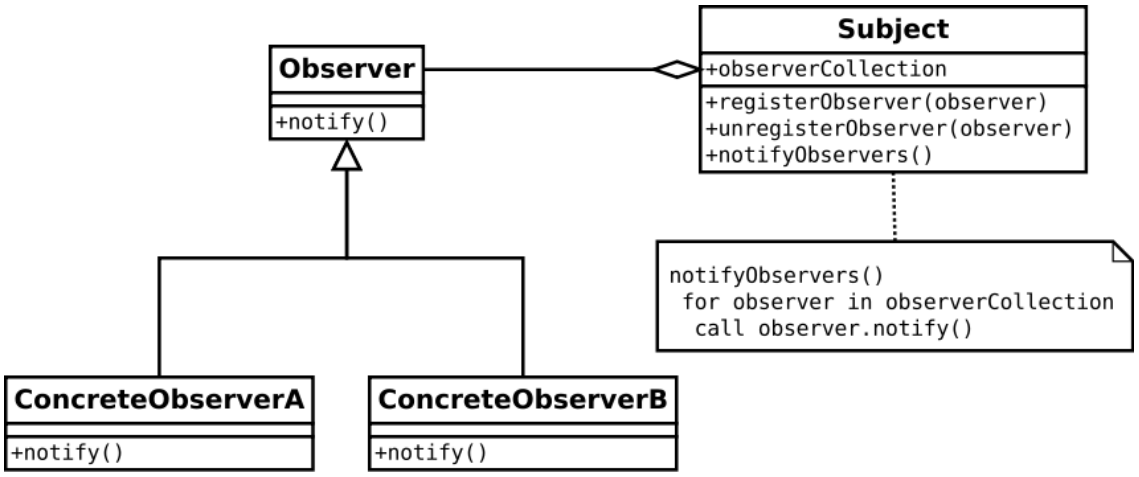


Figure 2.1.2: Design pattern observer

Le design pattern factory permet d'instancier des objets héritant d'une classe abstraite, et donc d'instancier des objets dont la classe exacte est inconnue.
Le factory est ici utilisé pour instancier des jeux, quel que soit leur types. Cependant, l'implémentation de ce design pattern n'était pas terminée et nous avons dû y apporter de nombreuses modifications.

Le design pattern facade doit cacher une interface et conception complexe, en proposant au développeur ou utilisateur une interface plus simple et générique.
Le UserFacadeImpl, qui implémentait une interface UserFacade[5]. A nouveau, de nombreuses modifications ont été apportées à la classe UserFacadeImpl.

Paradigmes

Le projet de ludothèque répondant à un problème concret, représentant une situation du monde réel, et comprenant des représentations d'objets physiques, la programmation orientée objet semble tout indiquée.
Il existe une multitude de langages de programmation orientée objet; l'utilisation de Java est pertinente puisque c'est un langage de POO pour lequel de nombreux outils de qualités et d'intégration existent[6].

Justifions la structure du projet et son design en fonction du problème auquel il répond.

[5] L'interface oblige le développeur à implémenter une méthode managerMenu, et une méthode adherentMenu, méthodes dont le rôle est de gérer un menu utilisateur

[6] C'est bien sûr le cas d'autres langages, mais la programmation Java faisait partie des exigences du client, le professeur

Le design pattern observateur semble servir à notifier le manager lorsqu'un adhérent s'enregistre, requièrant une action de ce dernier, probablement mettre à jour la base de données avec les informations de l'adhérent. Il est malheureusement difficile d'en être certain puisque le design pattern n'était pas utilisé dans le projet.

La classe GameFactory permet de créer une instance de jeu, quel que soit son sous-type. La méthode principale (et unique) de cette classe est createGame, qui reçoit en paramètre les informations nécessaires à l'instanciation du jeu. Cette méthode ne respecte pas l'extensibilité, puisque si un développeur souhaite ajouter un autre type de jeu, il devra obligatoirement modifier la méthode createGame également.

Le rôle de la classe UserFacadeImpl est de gérer l'interface utilisateur du programme. Puisque ce programme est exclusivement exécuté en terminal, sans interface graphique, cette classe appelle principalement des fonctions et classes terminales telles que println ou Scanner, qui alourdissent la lecture du code; il est donc intéressant de les placer dans une classe à part et non directement dans le main, ou dans une autre classe.

2.2 Critères de qualité

La première équipe de développement a déclaré vouloir respecter les critères de qualité suivants : le projet doit être maintenable, respecter le concept d'intégrité, être "simple" et être extensible.
Le critère de simplicité est décrit par le groupe[7] comme la facilité à comprendre la structure du projet et la logique des algorithmes et du code.

La maintenabilité d'un logiciel décrit l'effort nécessaire pour comprendre, corriger ou modifier ce logiciel pour un développeur, extérieur au projet ou non.
Le rapport préliminaire du groupe 3 indique qu'ils respectent ce critère par l'utilisation de la programmation orientée objet; il s'agit d'une erreur, puisque l'utilisation de la POO n'est pas garante de maintenabilité en soit.

Le critère de simplicité découle en réalité de celui de maintenabilité, et n'est donc pas un critère de qualité à part entière.

L'intégrité, que le groupe déclare respecter à l'aide de tests unitaires, est la nécessité de conserver à tout moment une cohérence, pertinence et un fonctionnement correct du programme et des données exploitées.
Les tests unitaires, s'ils sont eux mêmes intègres et fonctionnels, peuvent valider que le programme a le fonctionnement attendu et que les données générées ou traitées seront intègres. Ils ne sont cependant pas une garantie absolue d'intégrité, un aspect imprévu et non testé du logiciel pourrait passer au travers de la vérification par test.

[7] Voir page 3 du rapport préliminaire du groupe

L'extensibilité est un sous-critère de la maintenabilité, et est la facilité avec laquelle un développeur peut ajouter des classes ou méthodes ou autres fonctionnalités au logiciel, en apportant un minimum d'autres modifications au code existant. Pour respecter ce critère, l'équipe avait choisi d'appliquer l'encapsulation, c'est-à-dire de masquer le fonctionnement interne des classes en proposant des interfaces; d'utiliser les relations d'héritage; et d'appliquer les designs patterns mentionnés précédemment.

Métriques

Lorsque notre équipe a récupéré le projet, aucun système de mesure de métrique n'était mis en place.

Nous avons donc configuré un projet Jenkins permettant de relever certaines valeurs de qualité, notamment l'intégrité avec la réalisation automatique des tests unitaires; la qualité du style de codage (bien qu'elle ne se trouvait pas dans les critères du groupe 3) avec des tests automatiques détectant la duplication de code et appliquant le checkstyle Java de Google; nous avons utilisé le plugin CodeMR d'intellij, localement, pour réaliser une analyse des métriques de qualité du projet [8].

Puisque de nombreux métriques sont utilisés par le plugin pour déterminer les différents niveaux de qualité (complexité, longueur, cohésion, couplage, etc), nous n'en expliquerons ici qu'une partie, concernant les critères choisis par le premier groupe.

L'indice de McCabe est une valeur entière caractérisant la maintenabilité du code en déterminant , pour chaque méthode utilisée, la complexité de cette dernière. Ce calcul est fait en comptant le nombre de structures conditionnelles, de boucles et structures itératives, et de conditions bouléennes (&&, ||, etc). Plus l'index est grand, plus il sera difficile de coder un test unitaire couvrant correctement la méthode et plus cette méthode sera difficile à comprendre et/ou modifier. Une valeur acceptable serait de 4 ou 5[9]. Cet indice est utilisé dans le *Weighted Method Count* du plugin, lui même utilisé pour déterminer la complexité des méthodes, donc des classes et donc du programme. Plus le programme est complexe, moins il sera maintenable.

Le couplage entre deux classes est l'intensité de la liaison entre ces classes, la dépendance qui existe entre ces classes. Ainsi, si une modification est appliquée à une des classes, il y aura des répercussions sur l'autre, ce qui peut entraîner une augmentation du temps de développement nécessaire.

[8] Une liste des métriques mesurés par ce plugin peut être trouvée ici : `https://plugins.jetbrains.com/plugin/10811-codemr`

[9] Source : `https://www.theserverside.com/feature/How-to-calculate-McCabe-cyclomatic-complexity-in-Java`

Le couplage est donc un indicateur de l'extensibilité du programme. Dans le plugin, il est défini par plusieurs métriques, notamment le *Number of Children* qui indique le nombre de sous-classes directes d'une classe; le *Coupling Between Object Classes* qui compte le nombre de classes à laquelle une certaine classe est couplée, en comptant le nombre de classes qui utilisent ses attributs ou méthodes; et le *Access to Foreign Data* qui indique le nombre de classes dont les attributs sont atteignables par la classe en question.

Puisque lorsque nous avons récupéré le projet, ce dernier n'était pas terminé et ne compilait pas, nous n'avons pas pu examiner l'intégrité du projet à l'aide de Jenkins; certains tests unitaires échouaient, et le projet Jenkins ne compilait pas initialement. Nous avons cependant rapidement découvert de nombreuses erreurs de styles qui seront décrites en détail plus loin. Nous avons réalisé des analyses lors du développement et des modifications que nous avons apporté, ainsi qu'une analyse finale de la qualité selon les métriques sus-mentionnés qui sera décrite dans la section 6, et comparée avec les résultats des mêmes métriques sur l'ancienne version du projet, en l'état auquel nous l'avons récupéré.

3 Objectifs non atteints

3.1 Encapsulation

Comme expliqué ci-dessus, le programme utilise la programmation orientée objet. Cependant la règle "*Liskov substitution principle*" qui permet de regrouper différents objets sous un même type et ainsi pouvoir éviter la duplication de code et assurer l'extensibilité du programme n'est pas respectée.

Les interfaces sont existantes mais ne sont pas utilisées correctement. Par exemple, les 3 types de jeux sont tous regroupés dans un type global, *l'interface "Game"*. Mais un client a pour attributs 3 listes de jeux contenant chacune un type de jeux, au lieu de les regrouper dans une liste de *Game*.

Cela a donc créé de nombreuses duplications de codes et erreurs d'exécution. L'encapsulation est donc présente mais pas utilisée correctement.

3.2 Intégrité

L'intégrité du programme n'est pas atteinte. En effet lors de l'ajout d'un jeux dans le système, le programme lui octroie un id qui se veut unique. Cependant, l'id est généré sur base de la fonction "*Math.random*" de la manière suivante: *this.id = (long) ((Math.random() * (9999 - 1000) + 1) + 1000);*. Ce qui peut techniquement ne pas donner des id uniques. En effet, il y a une probabilité non nulle que plusieurs id générés par le code soient identiques.

Ceci peut entraîner des erreurs lors d'une recherche dans le système en fonction de l'id et ainsi donc affecter l'intégrité du système.

3.3 Simplicité

Certaine partie du projet ne sont pas très compréhensible. Ces dernières nous ont rendus la tâche plus compliquée. Par exemple, l'utilisation du design pattern observer dans le projet. Son utilité n'est jamais expliquée et les classes observer sont intégrée au projet et implémentées mais jamais utilisées.

3.4 Utilisation des façades

Le projet utilise le design pattern façade pour créer l'interface dans le terminal. Cependant, dans certaines classes, par exemple dans la classe *Manager*, nous pouvons retrouver la présence d'interface utilisateur. Ceci va à l'encontre du design pattern.

3.5 Maintenabilité

Le projet contient de nombreuses duplications de code, ce qui va à l'encontre de la maintenabilité du programme. Certaines de ces duplications n'étaient pas correctes et ont du être corriger pour pouvoir être refactorisées par la suite.

3.6 L'extensibilité

L'utilisation des designs patterns Factory et Façades ainsi que de la programmation orienté objet permette de favoriser l'extensibilité du projet tout en ne devant pas intervenir dans l'interface utilisateur via l'utilisation de façade ou dans la création de jeux via GameFactory.

La classe GameFactory permet de créer une instance de jeu, quel que soit son sous-type. La méthode principale (et unique) de cette classe est createGame, qui reçoiten paramètre les informations nécessaires à l'instanciation du jeu. Cette méthodene respecte pas l'extensibilité, puisque si un développeur souhaite ajouter un autretype de jeu, il devra obligatoirement modifier la méthode createGame également.

En conclusion on peut dire que l'extensibilité est atteints et respecter dans certain cas mais que d'autre ne le sont pas du tout.

4 Objectifs atteints

4.1 Simplicité

L'utilisation du design pattern singleton est un choix intelligent en ce qui concerne les performances du programme. En effet l'utilisation d'un singleton améliore l'occupation en mémoire et la vitesse d'exécution en ne générant qu'une seul instance de l'objet *Manager*.

4.2 Test unitaire

L'intégrité de l'entièreté du projet est assurée par les tests unitaires. Ils nous permettent de vérifier que chaque modifications ou améliorations apportées n'amène pas le programme à un dysfonctionnement.

4.3 Rôle des utilisateurs

Chaque utilisateur a des droits précis quant aux actions qu'il peut réaliser dans le programme. La distinction entre un client et un manager est assurée par la présence de 2 classe (*Adherent , Manager*) ayant chacune des actions différentes sur le programme. Toutefois, certaines actions restent communes aux deux types d'utilisateurs. Pour rendre cela possible, l'équipe a appliqué l'héritage sur ces 2 classes en créant une classe commune, la classe *Person*. Ainsi nous pouvons partager certaines méthodes entre le manger et l'adhérent tout en évitant la duplication de code.

5 Modifications apportées

Une fois l'analyse du projet et des objectifs terminée, nous avons commencé le *refactoring* du code.

5.1 Anglais et commentaires

Pour commencer nous avons, tout en parcourant le code, corrigé les erreurs d'anglais telles que "*with success*" à la place de "*with successfull*" ou "*first name*" pour "*firstname*". Dans un premier temps nous nous sommes arrêtés aux commentaires et noms de variables globales et locales afin de ne pas créer d'erreurs par la modification d'une méthode.

Toujours en parcourant le code, nous avons jugé la pertinence des différents commentaires. Dès lors nous avons supprimé ceux qui semblaient inutiles. En effet, *un bon code ne nécessite pas de commentaire*, il est donc inutile de placer un commentaire comme en figure 9.1.10.

De plus, tout au long du code on retrouve des commentaires copiés-collés. Ceux-ci ne sont donc pas pertinents comme on peut le voir sur la figure 9.1.11. Le second commentaire "*// Enter username and press Enter*" vient du fait qu'il y a une duplication de code, ou au moins deux codes très similaires, probablement qu'un copié-collé a été fait sans modifier le commentaire. D'autre part, comme dis dans le paragraphe précédent, un commentaire de ce type n'est pas nécessaire à cet endroit. Une équipe de développement pourrait aisément comprendre ces quelques lignes de code.

5.2 Noms des méthodes, classes et variables

Comme précisé ci-dessus, la première équipe de développement a mis en place certaines conventions par rapport aux noms des classes et des variables. Pour vérifier ces conventions nous avons appliqué la même méthode qu'au point précédent, c'est à dire que tout en parcourant le code nous avons corrigé les différentes erreurs remarquées. Contrairement au point précédent, ces conventions étaient généralement respectées. Il a toutefois fallu corriger certains noms de variables, méthodes et classes, par exemple en figure 9.1.12 à la ligne 2 pour "*boardGame*". Pour finir, il n'y avait rien de préciser quant aux méthodes, nous avons donc appliqués la même règle que pour les classes.

5.3 Duplication de code

A de nombreuses reprises nous avons observé des duplications de code. Ceux-ci sont dûs au fait que les relations d'héritage n'ont pas été utilisées comme elles l'auraient dû. Comme on peut le voir dans les figures 9.1.12 et 9.1.13, ces deux méthodes sont quasiment identiques. Les seules choses qui peuvent différencier ces

deux méthodes sont la variable "*boardGame*" ou "*Toy*" et les chaînes de caractères dans les "*switch*". Dans le code d'origine il existait une troisième méthode, identique aux deux premières mais pour "*VideoGame*".

Pour palier à ce problème, nous avons réécrit une nouvelle méthode qui remplace les trois précédentes. Pour ce faire, nous avons utilisé les relations d'héritage entre "*Game*" et "*boardGame*", "*toy*" et "*VideoGame*". Cette méthode prend en paramètre le type du jeu en *string*, une liste de *Game* qui correspond à la base de données et un id de type *long* caractérisant le jeu emprunté. Cela nous permet de faire une méthode générale qui sera valable pour les trois types de *Game*. Elle renvoie un string qui nous informe sur le résultat de la tentative d'emprunt. Cette méthode peut être observée en figure 9.1.14.

5.4 Erreurs de *checkstyle*

Lorsque nous avons pris en main le projet nous avions un nombre élevé d'erreurs de *checkstyle*. La plupart étaient des erreurs de longueur de ligne ou de d'espacement entre les différentes boucles, méthodes, variables et classes. Pour corriger celles-ci nous avons utilisé *Jenkins* qui permet de repérer ces différentes erreurs sur base d'un template. Une fois les erreurs définies, il a suffit d'appliquer les bonnes règles aux différents endroit renseignés sur *Jenkins*.

6 Qualité du projet

L'intégration avec Jenkins nous a permis de valider l'intégrité du projet au fur et à mesure que nous développions. Les tests unitaires étaient en effet effectués automatiquement.....
Pour réaliser la mesure de métriques de qualité du projet, nous avons utilisé un plugin d'intellij nommé CodeMR, qui génère automatiquement un rapport contenant les résultats d'un grand nombre de métriques de qualité selon plusieurs critères.
Les fichiers html générés sont placés automatiquement dans le sous-dossier `/codemr/Ludotheque/html` [10].

6.1 Taille

(a) Ancienne taille.

(b) Nouvelle taille.

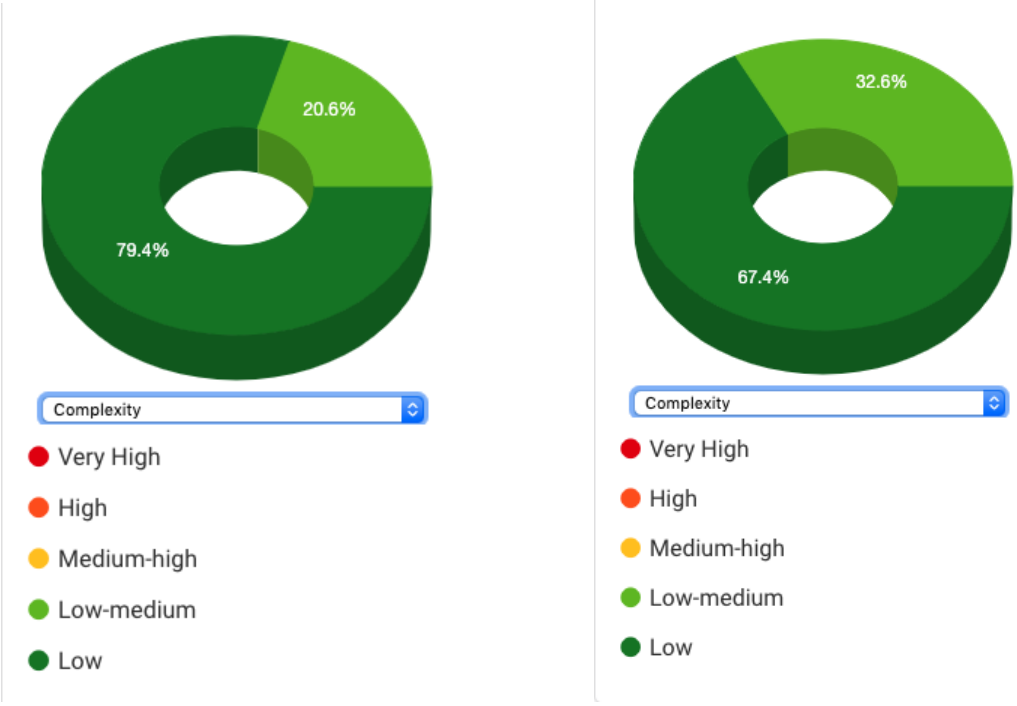
Figure 6.1.3: Taille du projet.

La taille du code a diminué de manière non négligeable; diminution de 16% du nombre de lignes de code, expliquée principalement par la suppression de nombreuses duplications de code ; et diminution de 19% du nombre de classes dans le projet, expliqué par la suppression de plusieurs classes entièrement redondantes et non implémentées.

La diminution de la taille du code pour une même qualité globale entraîne souvent une meilleure lisibilité du code, et donc une augmentation de sa maintenabilité.

[10] Rapport qualité : `https://github.com/victorsmits/Ludotheque/tree/master/codemr/Ludotheque/html/main_report`

6.2 Complexité



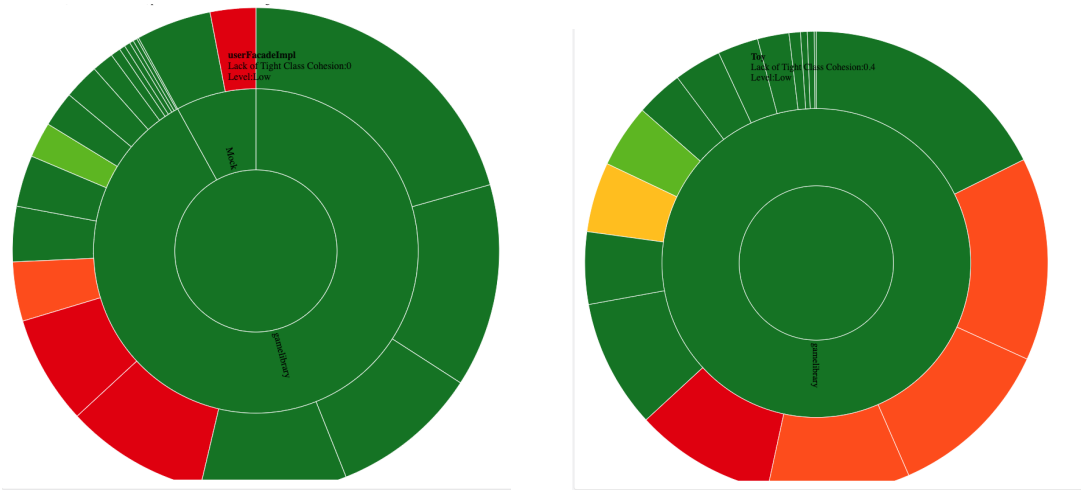
(a) Ancienne Complexité interne.

(b) Nouvelle complexité interne.

Figure 6.2.4: Complexité générale.

On constate que la complexité du projet a augmenté avec nos modifications, donc que la qualité du projet a décru en ce qui concerne la maintenabilité; ceci est probablement expliqué par le fait que nous avons implémenté plusieurs classes et méthodes qui étaient laissées vides par l'équipe précédente, nous avons notamment fait augmenter les indices de McCabe, les interactions entre les classes, etc. Les valeurs de métriques de complexité du projet tel que nous l'avons reçu ne sont donc pas représentatives de la complexité réelle du design initial.

6.3 Cohésion



(a) Ancien manque de cohésion interne.

(b) Nouveau manque de cohésion interne.

Figure 6.3.5: Manque de cohésion. Chaque portion de disque représente une classe du projet; sa couleur, le taux de manque de cohésion.

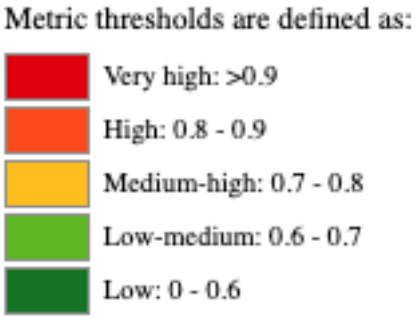
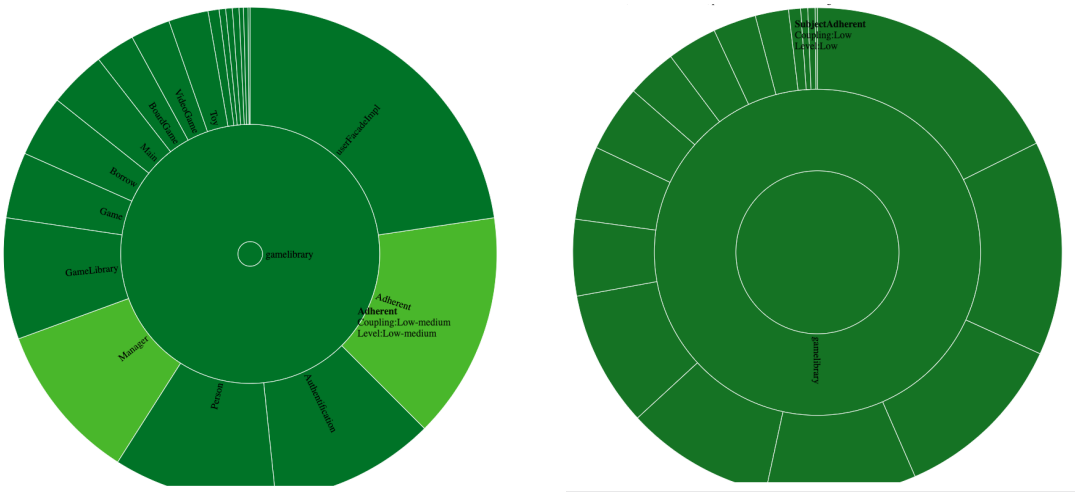


Figure 6.3.6: Légende

Ce paramètre décrit la qualité des relations à l'intérieur d'une classe (utilisation de tous les attributs par les méthodes de la classe donc pas d'attribut inutile, par exemple); une cohésion élevée (donc vert, manque de cohésion bas) est associé avec des qualités telles que la robustesse et la maintenabilité. En modifiant le code, nous avons globalement amélioré la cohésion du programme, même si certaines classes telles que Person ou Authentification se sont dégradées, ce qui s'explique par le fait que nous avons ajouté de nombreuses méthodes à cette classe.

6.4 Couplage



(a) Ancien couplage interne. (b) Nouveau couplage interne.

Figure 6.4.7: Couplage interne.

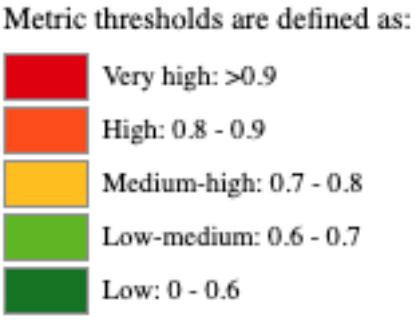
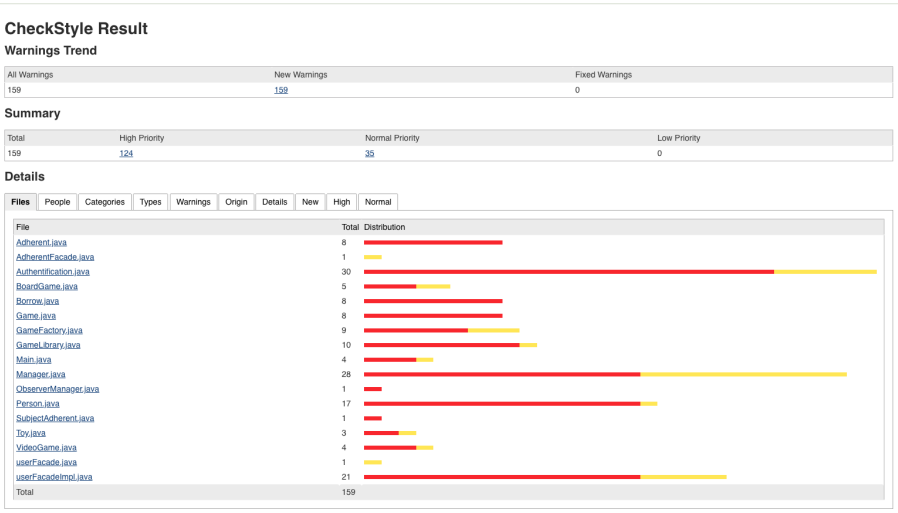


Figure 6.4.8: Légende

Nous pouvons constater que le couplage entre les classes a légèrement diminué [11]. Cependant, puisque nous n'avons presque pas changé la structure du programme, il semble logique que le couplage entre les classes n'ait presque pas changé. Ce couplage relativement bas implique que le projet devrait être facilement extensible.

[11] Le couplage des classes Adherent et Manager a diminué

6.5 Erreur de Style



CheckStyle Result

Warnings Trend

All Warnings	New Warnings	Fixed Warnings
159	159	0

Summary

Total	High Priority	Normal Priority	Low Priority
159	124	35	0

Details

Files | People | Categories | Types | Warnings | Origin | Details | New | High | Normal

File	Total	Distribution
Adherent.java	8	
AdherentFacade.java	1	
Authentification.java	30	
BoardGame.java	5	
Borrow.java	8	
Game.java	8	
GameFactory.java	9	
GameLibrary.java	10	
Main.java	4	
Manager.java	28	
ObserverManager.java	1	
Person.java	17	
SubjectAdherent.java	1	
Toy.java	3	
VideoGame.java	4	
userFacade.java	1	
userFacadeImpl.java	21	
Total	159	

Figure 6.5.9: Ancienne erreur de style.

Comme nous pouvons le voire dans la figure 6.5.9 la première version du code contenait de nombreuse erreur de style de code. Nous avons donc résolut toutes les erreurs de check Style qui ont été repérée par Jenkins via la comparaison aux règles de codage de Google.

7 Conclusion

Pour clôturer ce travail, il convient de rappeler que l'analyse du projet existant nous a permis de déterminer la qualité du code. Ceci grâce aux différents critères de qualité qui ont été mis en place et décidés par la première équipe de développement. Certains de ces critères étaient respectés et appliqués correctement dans le code. Le couplage entre classes est en effet assez faible, ce qui est un indicateur qu'il ne sera pas trop difficile d'apporter des modifications au projet; il est relativement extensible, malgré quelques défauts.
Toutefois une partie des critères n'ont pas été respectés ou pas entièrement. Il convenait donc de modifier le code pour en augmenter la qualité et respecter les critères mis en place. La complexité du projet est également relativement basse ou moyenne; cependant la cohésion est globalement assez mauvaise, ce qui indique que des améliorations peuvent être apportées au projet pour augmenter sa maintenabilité.

Enfin, il convient de dire qu'il n'y pas de "recette" quant à la qualité d'une application. Les critères qui détermine cette qualité doivent être défini au préalable par l'équipe de développement selon les besoins de l'application. Il ne faut pas se fier à un seul critère de qualité ni se jeter sur ceux que l'on connaît. Il est donc important d'effectuer une analyse approfondie des besoins que l'application doit remplir afin d'en déterminer les critères de qualité adéquats.

8 Glossaire

S O L I D : Single responsibility principle, Open/closed principle, Liskov substitution principle, Interface segregation principle, Dependency inversion principle.

9 Annexes

9.1 Annexe 1 - Codes

```
1 System.out.println("Enter your name"); //Enter name
```

Figure 9.1.10: Commentaire inutile

```
1  //put  username
2  Scanner user = new Scanner(System.in);
3  String username;
4  System.out.println("Enter your username"); // Enter username and press Enter
5  username = user.nextLine();
6
7  // put password
8  Scanner pass = new Scanner(System.in);
9  String password;
10 System.out.println("Enter your password");  // Enter username and press Enter
11 password = pass.nextLine();
```

Figure 9.1.11: Commentaires copiés-collés

```java
public String borrowBoardGame(long id){
    BoardGame boardgame;
    Borrow borrow;

    int count = 0;

    if(GameLibrary.getBoardGameList().isEmpty()){ // if database empty
        return "No board game in database";
    }

    for (int i = 0; i < GameLibrary.getBoardGameList().size(); i++) {
        if(GameLibrary.getBoardGameList().get(i).getId() == id ){ // if found

            if(GameLibrary.getBoardGameList().get(i).getStatut() == true){
                boardgame = GameLibrary.getBoardGameList().get(i);
                boardgame.setStatut(false);

                borrow = new Borrow(this, boardgame);

                GameLibrary.getAllBorrowList().add(borrow);
                borrowList.add(borrow);

                count = 1;
            }else{
                count = 2;
            }

        }
    }

    switch (count) {
        case 1:
            System.out.println("Please, go pick your borrow");
            return "Borrow with successfull";
        case 2:
            return "this game is not available";
        default:
            // if no found
            return "No found";
    }

}
```

Figure 9.1.12: Méthode "borrowBoardGame"

```java
public String borrowToy(long id){
    Toy toy;
    Borrow borrow;

    int count = 0;

    if(GameLibrary.getToyList().isEmpty()){ // if database empty
        return "No toy in database";
    }

    for (int i = 0; i < GameLibrary.getToyList().size(); i++) {
        if(GameLibrary.getToyList().get(i).getId() == id ){ // if found

            if(GameLibrary.getToyList().get(i).getStatut() == true){
                toy = GameLibrary.getToyList().get(i);
                toy.setStatut(false);

                borrow = new Borrow(this, toy);

                GameLibrary.getAllBorrowList().add(borrow);
                borrowList.add(borrow);

                count = 1;
            }else{
                count = 2;
            }

        }
    }

    switch (count) {
        case 1:
            System.out.println("Please, go pick your borrow");
            return "Borrow with successfull";
        case 2:
            return "this toy is not available";
        default:
            // if no found
            return "No found";
    }

}
```

Figure 9.1.13: Méthode "borrowToy"

```java
private String borrowGame(String gameType,
                          ArrayList<Game> database,
                          long id) {
    Game game;
    Borrow borrow;
    int count = 0;

    if (GameLibrary.getVideoGameList().isEmpty()) {
        return "No video game in database";
    }

    for (Game value : database) {
        //Check if game exist
        if (value.getId() == id) {
            //Check the game status
            if (value.getStatus()) {
                game = value;
                game.setStatus(false);

                borrow = new Borrow(this, game);

                GameLibrary.getAllBorrowList().add(borrow);
                borrowList.add(borrow);

                count = 1;
                break;
            } else {
                count = 2;
            }

        }
    }
    switch (count) {
        case 1:
            return "You can pick up your " + gameType + ".";
        case 2:
            return "This " + gameType + " is not available.";
        default:
            return "This " + gameType + " was not found.";
    }
}
```

Figure 9.1.14: Méthode "borrowGame"

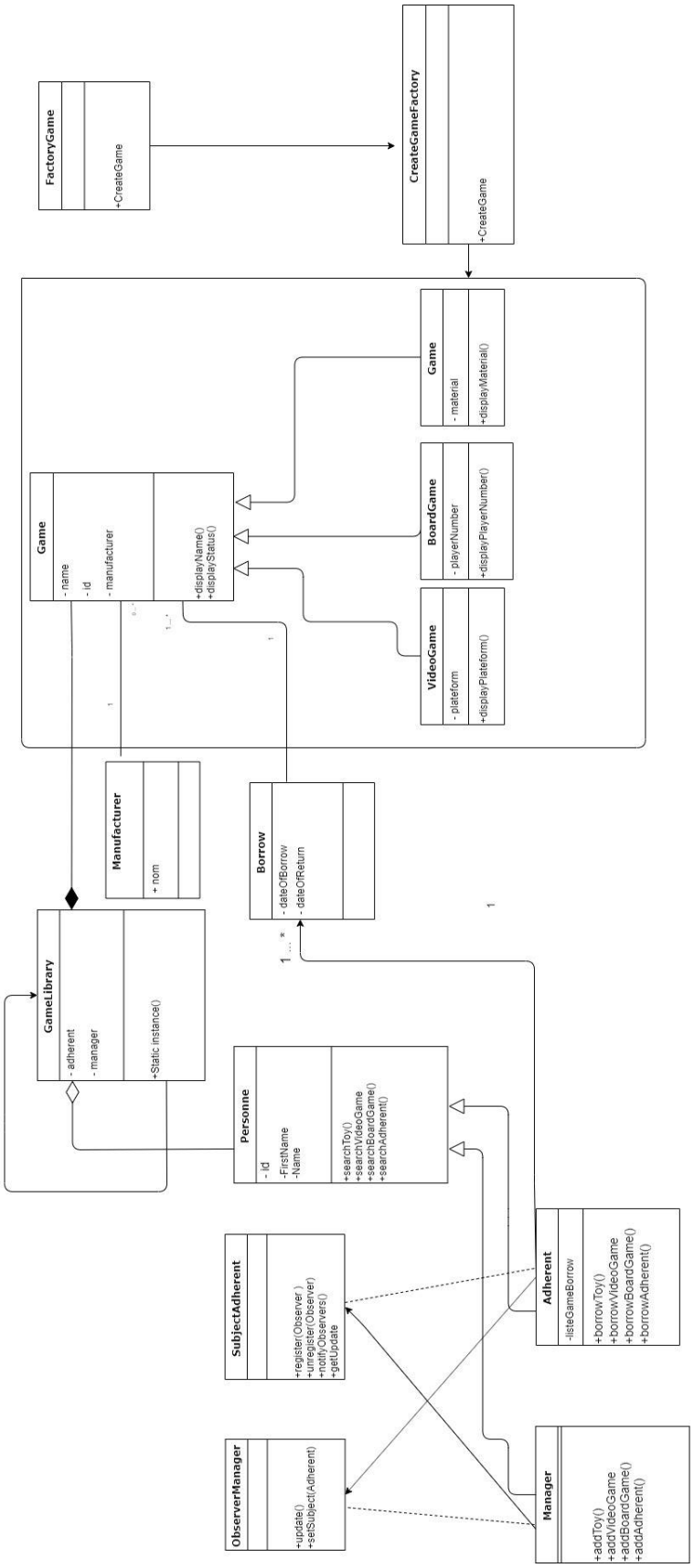


Figure 9.1.15: Diagramme de classe