**Xabi Avellan TD2 |TP3**

**Dylan Mendiboure TD2 |TP3**

**Arthur Murillo TD2 |TP3**

**Projet Citadelles - M4104C PEL S4 2019/2020**

**Projet n°3**

***Github :*** <https://github.com/dmendiboure/Citadels>

***Trello :*** <https://trello.com/b/thvyX2Wv>

Sommaire

[**Itération 1**](#_heading=h.2d2wjhgq6zko) **3**

[Couvrir par des tests le calcul du score d’une cité](#_heading=h.n1ga5y1rwvez) 3

[Identifier 5 code smells](#_heading=h.aagci8sv4wk7) 3

[Obtenir le diagramme de classes](#_heading=h.3si0p6uxrdlz) 3

[**Itération 2**](#_heading=h.9jvs8zi6o6x5) **6**

[Nettoyer le code du calcul du score d’une cité](#_heading=h.q535br9mx3v3) 6

[Calculer la complexité cyclomatique du main](#_heading=h.dt6zkfy10ln2) 7

[Réduire la complexité cyclomatique du main](#_heading=h.hcpgn244twtu) 7

[**Itération 3**](#_heading=h.fvrciiccm6vb) **8**

[Implémenter le pouvoir “détruire un quartier”](#_heading=h.ysw5yltcfow) 8

[Revoir le code d’un autre groupe](#_heading=h.hs2zldvy69rk) 9

[Automatiser l’exécution des tests avec Github Actions](#_heading=h.b5hm10espm02) 9

[**Itération 4**](#_heading=h.bp7viles0mde) **10**

[Implémenter le pouvoir “détruire un quartier”](#_heading=h.ys4f3wjoox6s) 10

[Réduire la complexité cyclomatique](#_heading=h.w6xyv3gdh3vg) 10

[Revoir le code d’un autre groupe](#_heading=h.m6mxlbbal95l) 11

[Automatiser l’exécution des tests avec Github Actions](#_heading=h.zcyouz5205sj) 12

[**Itération 5**](#_heading=h.r2a5ovdp3rjy) **13**

[Pouvoir jouer avec l’Alchimiste](#_heading=h.pvp4qr11j5of) 13

[Résoudre le bug de la duplication d’actions](#_heading=h.fr3e2caej6j) 14

[Pouvoir jouer avec l’Abbé](#_heading=h.2omr5m9nnrxd) 15

[Pouvoir choisir avec quelle édition du jeu on peut jouer](#_heading=h.ao4619ct9los) 17

[**Conclusion**](#_heading=h.9jvbxqv21u4f) **18**

# Itération 1

**Travail à effectuer durant cette itération :**

* *Couvrir par des tests le calcul du score d’une cité*
* *Identifier 5 code smells*
* *Obtenir le diagramme de classes*

## Couvrir par des tests le calcul du score d’une cité

Nous avons réalisé, dans une classe appropriée située dans un package dédié aux tests, créé une classe CityTest. Dans cette classe, nous avons testé les 5 facteurs pris en compte dans le calcul du score en plus de la possession en or du joueur :

* Le coût de construction des quartiers dans la cité
* La présence d’un quartier de chaque type dans la cité
* Le fait que le joueur ait complété sa cité en premier
* Le fait que d’autres joueur aient complété leur cité
* Des bonus liés aux merveilles (map room, treasury, Dragon Gate, University)

Les tests nous ont permis d’identifier un bug. En effet, un test n’a pas été validé : la présence d’une quartier de chaque type dans la cité : le bonus était fixé à +5 et non pas à +3. Nous devrons donc corriger le bug. Cependant, tous les autres tests ont été validés.

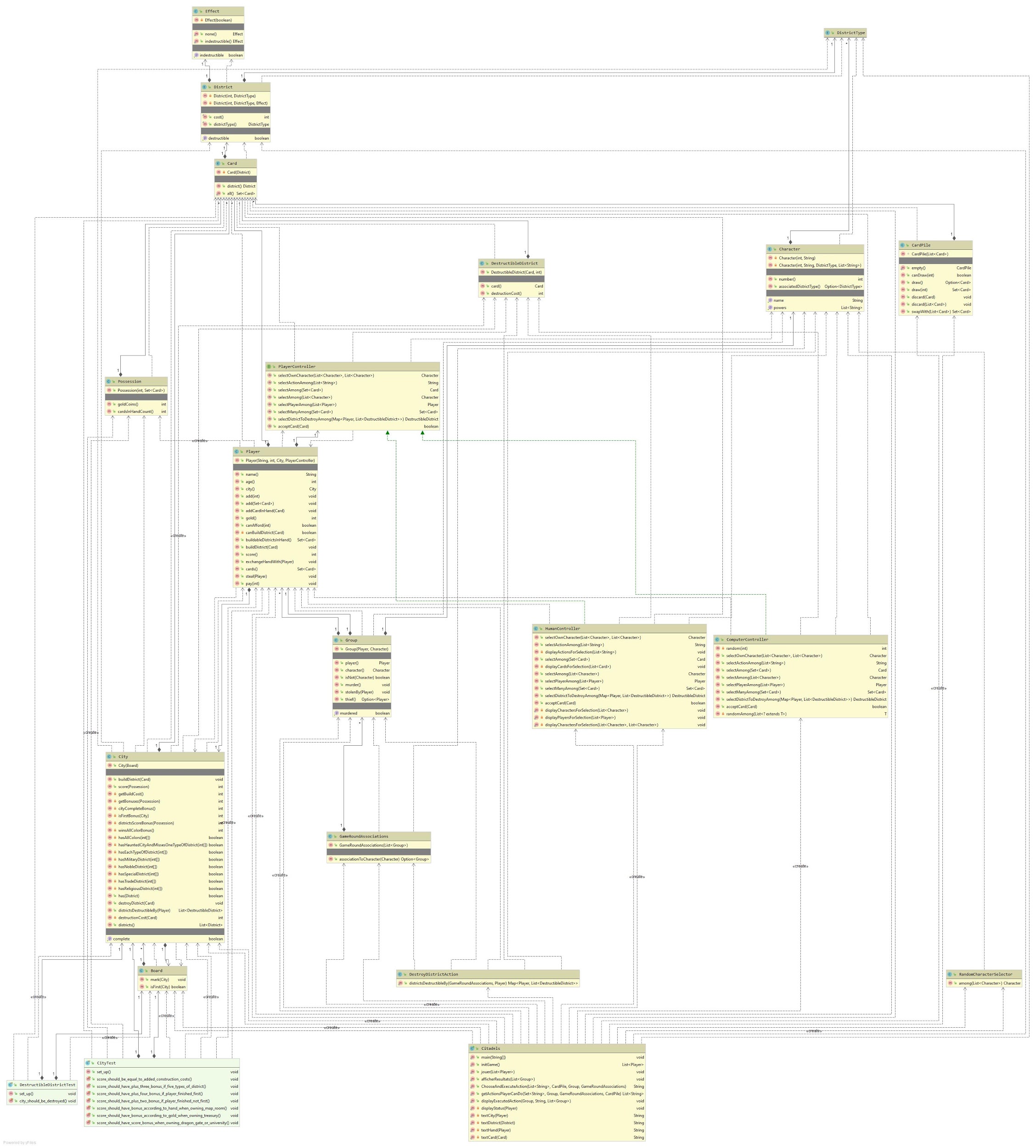
## Identifier 5 code smells

Nous avons, après étude approfondie du code, identifiés les code smells suivants :

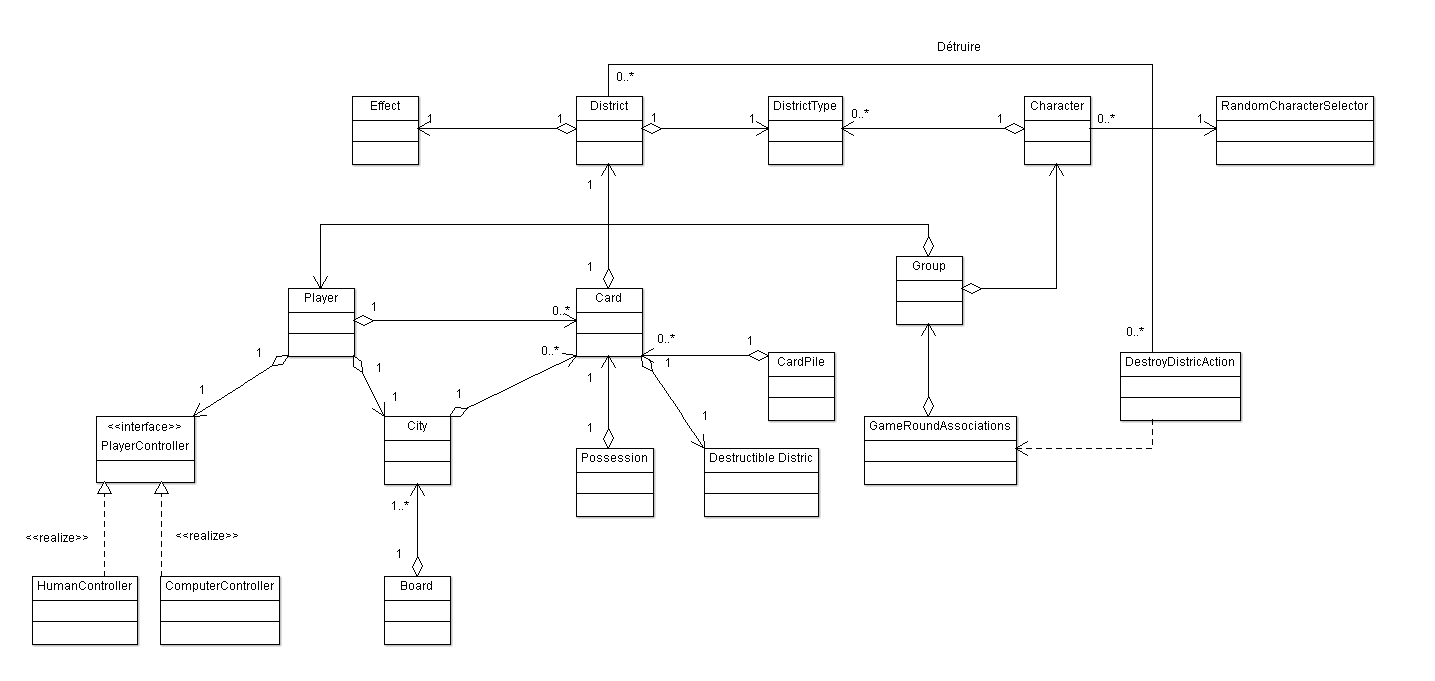
* 1 : Dans Citadels.java instructions switch. Recommandation : encapsuler le comportement dans des classes.
* 2 : La classe main (Citadels.java) est une classe longue, de 350 lignes. Recommandation : éclater en plusieurs classe.
* 3 : Les classes RandomCharacterSelector et DestroyDistrictAction sont des classes fainéantes du fait qu’elles ne contiennent qu'une méthode d'une ligne. Recommandation : Supprimer ces classes et implémenter ces fonctionnalité dans les classes Character et District
* 4 : La classe Board est une classe morte, elle n'est importe dans aucune autre classe. Recommandation : Supprimer cette classe.
* 5 : Obsession des primitifs en représentant le pouvoir d’un personnage par une string

## Obtenir le diagramme de classes

A l’aide du logiciel IntelliJ nous avons pu générer le diagramme de classes correspondant au code qui nous était fourni. En effet, nous n’avons pas réussi à trouver un moyen de représenter le diagramme de classe de manière exhaustive et correcte à la main sans aide. Nous avons pu remarquer que ce diagramme était très fournit, par conséquent nous avons réalisé une autre version simplifiée, donc moins exhaustive, mais plus lisible. Dans ce diagramme de classe figurent également des énumérations (Card et District) qui représentent respectivement toutes les cartes présentes dans le jeu, ainsi que les types de cartes pouvant être ajoutées dans le jeu.



A partir de la version générée par IntelliJ, nous avons donc réalisé la version “allégée” suivante sur ArgoUML :



# Itération 2

**Travail à effectuer durant cette itération :**

* *Nettoyer le code du calcul du score d'une cité*
* *Calculer la complexité cyclomatique du main*
* *Réduire la complexité cyclomatique du main*

## Nettoyer le code du calcul du score d’une cité

Nous avons dans un premier temps analysé la méthode de calcul de score que nous avons récupérée, ainsi que les autres méthode que celle-ci appelle, et que l’on peut voir ci dessous.



Pour nettoyer le calcul de ce score, nous avons donc renommé des variables (par exemple dans la méthode score, ligne 2, renommer l’itérateur de a à carte, pour expliciter le fait que cet itérateur correspond à l’indice d’une carte. De plus, nous avons créé de nombreuses méthodes, correspondant au découpage

## Calculer la complexité cyclomatique du main

A l’aide du plugin Metrics nous avons trouvé une complexité cyclomatique de 148 pour le main Citadels.java. Le plugin identifie également de nombreuses raison pour lesquelles la complexité était élevée, dont la présence de gros blocs de if/else et le regroupement de toutes les actions en une très grande fonction (le main).

## Réduire la complexité cyclomatique du main

Nous n’avons pas réussi durant cette itération à réduire la complexité cyclomatique, nous le reverrons donc en itération 4 ou 5, après plus mûre réflexion.

# 

# Itération 3

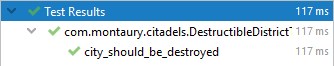
**Travail à effectuer durant cette itération :**

* *Implémenter le pouvoir “détruire un quartier”*
* *Revoir le code d’un autre groupe*
* *Automatiser l'exécution des tests avec Github Actions*

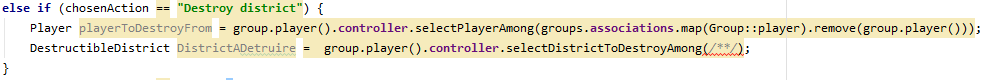
## Implémenter le pouvoir “détruire un quartier”

Pour ce travail, nous avons créé, comme pour le test du calcul du score, une classe de test sur la classe DestructibleDistrict. Nous avons ainsi pu remarquer que l’action de suppression d’un district fonctionnait bien.





Nous avons également commencé l’implémentation du pouvoir dans le jeu. Nous avons donc, à la place de l’ancien commentaire précisant la flemme du créateur du jeu de développer la fonctionnalité, commencé à écrire le code permettant de choisir un joueur pour détruire un quartier, mais nous nous sommes heurté à un problème car nous n'arrivons pas à fournir un paramètre satisfaisant la fonction permettant de choisir un district dans la main d’un adversaire.

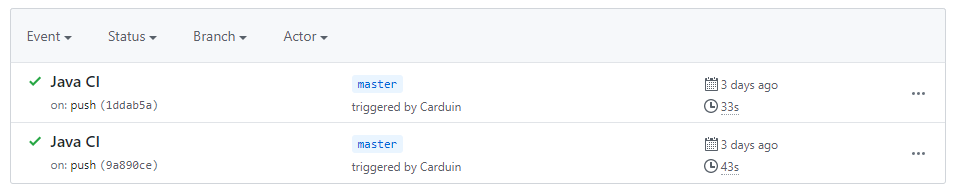


## Revoir le code d’un autre groupe

Nous n’avons pas eu le temps de réaliser cette tâche, nous le reverrons donc en itération 4 ou 5.

## Automatiser l’exécution des tests avec Github Actions

Pour cette tâche, nous avons sur Github créé pour le projet un workflow Gradle, mais n’ayant pas compris comment fonctionnait cette fonctionnalité de Github, nous la reprendrons en itération 4 ou 5 avec l’aide d’un groupe éventuellement plus avancé ou après recherche de notre côté.



# 

# Itération 4

**Travail à effectuer durant cette itération :**

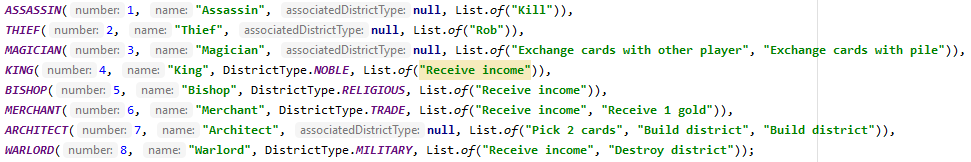
* *Implémenter le pouvoir “détruire un quartier”*
* *Réduire la complexité cyclomatique*
* *Revoir le code d’un autre groupe*
* *Automatiser l'exécution des tests avec Github Actions*

## Implémenter le pouvoir “détruire un quartier”

Grâce à M. Urruty, nous avons compris que pour tester cette fonctionnalité nous pouvions la déplacer dans une classe (par exemple la fonctionnalité DestroyDistrictAction) en lui passant toutes les instances de joueurs, tableaux, cartes dont nous pourrions avoir besoin. Nous avions déjà précédemment créé une classe de test, malheureusement nous avons encore eu des difficulté à comprendre comment nous pourrions tester la fonctionnalité avant de l’implémenter, plus précisément quels tests réaliser. Après son explication à tout le TD sur la solution à mettre en place pour réaliser cette tâche, nous avons compris (avec la solution c’est toujours plus simple) ce que nous aurions dû mettre en place, et comment. Nous abandonnons avec tristesse mais néanmoins soulagement cette tâche au profit du choix libre de ce que nous pourrons faire dans l’itération 5.

## Réduire la complexité cyclomatique

Pour réduire la complexité cyclomatique nous avons identifié des blocs de code l’augmentant grandement, composés d’instructions if/else imbriquées. Ainsi, nous avons par exemple identifié un bloc d’instructions définissant les pouvoirs utilisables par le joueur en fonction du personnage qu’il a choisi (de la form if player.character == Character.BISHOP, Pouvoir = xxx). Pour enlever ce bloc, nous avons encapsulé le pouvoir dans la classe Character en ajoutant comme attribut une liste de pouvoirs (sous forme de string ce qui constitue une primitive obsession, mais nous n’avons pas réussi à trouver un moyen de créer une classe pouvoir qui permettrait une implémentation plus compréhensible) :



Ainsi que la méthode associée pour pouvoir récupérer le pouvoir :



De cette manière, nous avons pu réduire le bloc de if/else en une simple fonction :



De même, nous avons utilisé une méthode permettant de récupérer un éventuel district associé à un personnage, pour réduire un bloc if/else en une ligne :



Ces actions nous ont permis de réduire la complexité cyclomatique de 148 à 125. C’est peu, mais nous avons également identifié d’autres blocs réductibles par la création d’une fonction, ou des blocs if/else réductible par exemple pour l’analyse de l’action choisie par le joueur lors de son tour par la création d’une classe Action, puis l'exécution de l’action.

Ainsi, en créant deux nouvelles fonctions à partir du main (getActionsPlayerCanDo et ChooseAndExecuteAction) nous avons pu séparer le main de ces blocs et réduire la complexité à 54. C’est bien mieux, mais nous n’avons fait que transformer un grand problème en deux petits problèmes. En effet, ces nouvelles méthodes ont respectivement une complexité de 24 et 40. Nous pourrions définitivement réduire cette complexité en intégrant ces actions dans une classe (Action par exemple), mais nous n’avons pas eu le temps de réaliser cette tâche et nous trouvons déjà ce travail satisfaisant par rapport à celui demandé et celui qui nous reste à réaliser.

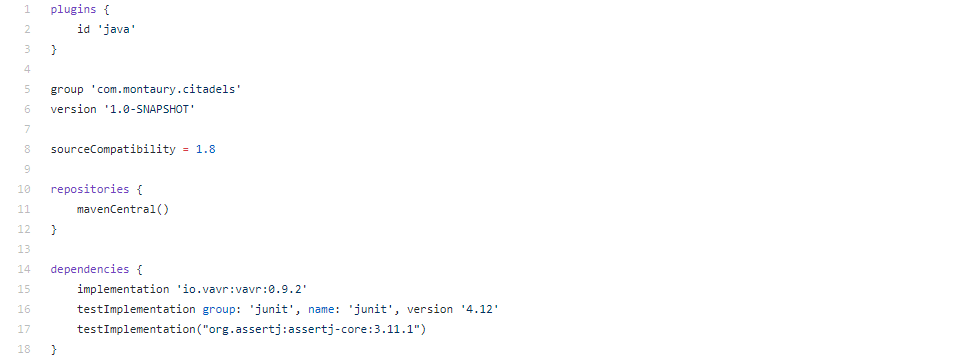
## Revoir le code d’un autre groupe

Pour cette tâche nous avons revu le code du groupe 8 composé de Paul Lafourcade, Malo Le Mestre et Louison Vincent. Nous avons discuté durant une vingtaine de minutes sur la manière dont ils avaient réalisé la réduction cyclomatique du main et nous les avons conseillé sur la manière de l’améliorer par rapport à ce que nous avons pu déjà réaliser, mais également sur leur manière de nommer certaines variables. Nous avons également revu leurs tests qui nous ont paru corrects.

## 

## Automatiser l’exécution des tests avec Github Actions

Pour cette tâche nous avons, à l’aide de M. Urruty paramétré le fichier build.gradle pour exécuter les tests. Les tests s'exécutent maintenant à chaque commit que nous effectuons sur le dépôt.



# Itération 5

**Travail à effectuer durant cette itération :**

* *Pouvoir jouer avec l’Alchimiste*
* *Résoudre le bug de la duplication d’actions*
* *Pouvoir jouer avec l’Abbé*
* *Pouvoir choisir avec quelle édition du jeu on peut jouer*

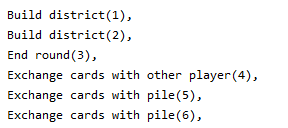
Pour choisir le travail pour cette itération, nous avons évalué les tâches présentes dans la colonne “A faire” du Trello, car ces tâches nous paraissaient plus riches en tests à réaliser. L’interface nous paraissait impossible à réaliser dans le temps imparti, nous avons donc hésité entre l’amélioration des ordinateurs ou l’implémentation de personnages. Nous avons finalement tranchés pour les personnages, et nous en avons pris deux qui possédaient un pouvoir qui nous paraissait intéressant. Nous avions prévu d’en prendre plus mais M. Urruty nous a conseillé de se concentrer sur ceux-ci. De plus, nous allons devoir trouver un moyen de laisser le joueur choisir s’il veut jouer avec l’édition” classique”, ou l’édition “modifiée” avec l’abbé remplaçant l'évêque et l’alchimiste remplaçant le marchand. Enfin, nous avons durant cette phase identifié un bug causé par notre travail, que nous évoquerons plus loin et que nous nous sommes attelés à résoudre.

## Pouvoir jouer avec l’Alchimiste

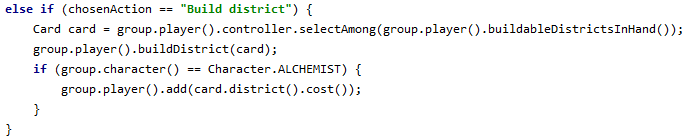
Nous avons commencé par implémenter l’alchimiste. Pour cela nous avons travaillé en pair programming, pendant que le 3e membre rédigeait le compte rendu d’activité. Nous avons dans un premier temps commencé par ajouter le personnage dans le type énuméré prévu à cet effet en précisant son numéro (le même que le marchant car l’alchimiste est censé le remplacer) :



Nous avons ensuite réfléchi pour implémenter un moyen de choisir les personnages que l’on voulait remplacer. Durant des tests nous nous sommes rendus comptes que (probablement lors du nettoyage) un bug était apparu, faisant en sorte que lors de la sélection des actions d’un personnage, certaines actions apparaissent deux fois :



Nous nous sommes donc appliqués à résoudre ce bug en parallèle. Pour l’implémentation du pouvoir, qui est relativement simple, nous avons ajouté dans l’action build district une condition permettant de rembourser le joueur lors de la construction :

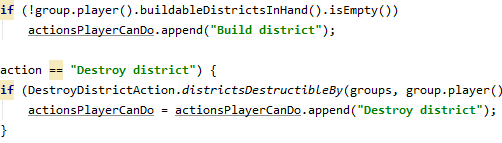


## Résoudre le bug de la duplication d’actions

Pour résoudre ce bug nous avons parcouru les différentes méthodes de choix d’action que nous utilisons pour déterminer les actions que le joueur peut réaliser, et le choix /effectué par le joueur. Nous avons finalement déterminé que le bug provient de la méthode getActionsPlayerCanDo qui renvoie, en fonction de conditions, les actions actuellement réalisables par le joueur. Cependant, la solution que nous avons trouvée pour le résoudre nous paraît assez étrange. En effet, dans cette méthode, si une condition est vérifiée, la liste (qui sera renvoyée par la fonction) des actions réalisables par le joueur est mise à jour avec l’action concernée. Mais sa mise à jour nous paraît étrange :



On voit que la mise à jour s’effectue via la fonction append (qui permet d’ajouter un élément à la liste). Mais l’affectation réalisée nous paraît inutile ici. Nous l’avons donc enlevée, et nous avons constaté que le bug n’était plus présent. Cependant, lorsque nous appliquons cette solution à d’autres actions, cela fait planter le programme dans certains cas. Nous nous disons alors que le problème ne vient peut être pas de ça, et nous avons mis en place cette solution seulement pour les actions pour lesquelles nous constations une duplication. Etant donné ce qu’il nous reste à faire nous avons préféré passer à autre chose mais nous sommes presque certains que ceci est une solution “bancale” faute d’avoir trouvé la racine du problème.



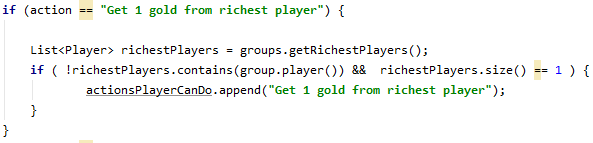
## 

## Pouvoir jouer avec l’Abbé

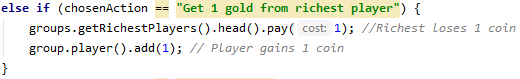
Pour implémenter le pouvoir de l’abbé, qui est de pouvoir recevoir un or de la part du joueur le plus riche (s’il ne s’agit pas du joueur qui possède l’Abbé ou s’il n’y a pas plusieurs les plus riches) , nous avons commencé par ajouter son pouvoir (que nous avons nommé “Get 1 gold from richest player”) dans le type énuméré Characters :



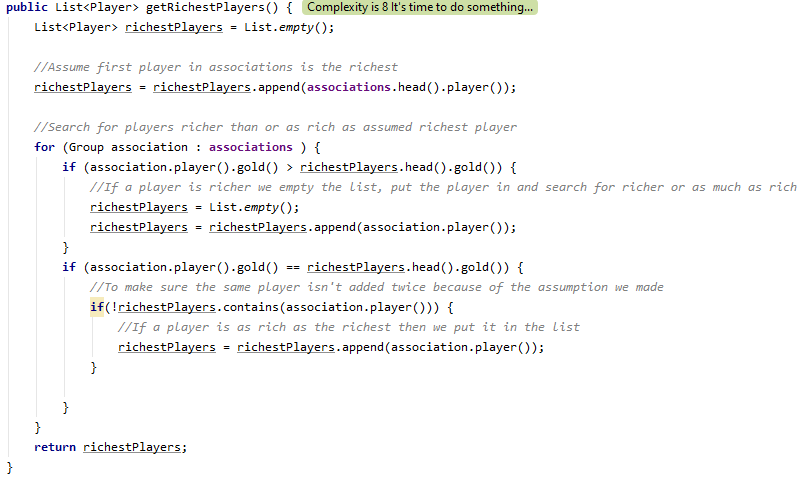
Ensuite, nous l’avons ajouté dans la méthode getActionsPlayerCanDo pour que le joueur puisse choisir d’activer le pouvoir du personnage, avec la condition appropriée :



On remarque que nous avons utilisé la méthode getRichestPlayers que nous avons implémenté et dont nous allons parler. Ensuite, nous ajoutons l’action dans la liste des actions à exécuter en fonction de celle choisie par le joueur (méthode chooseAndExecuteAction) :



Pour exécuter l’action on récupère le premier élément de la liste (car il n’est censé y en avoir qu’un à ce stade du code grâce à la condition précédente), et on le fait payer 1 or. Le joueur se voit alors crédité de 1 or. Pour déterminer le joueur le plus riche, comme nous l’avons dit, nous avons implémenté la méthode getRichestPlayers qui à partir de toutes les associations joueur-personnage du jeu en cours détermine celui avec le plus d’or :

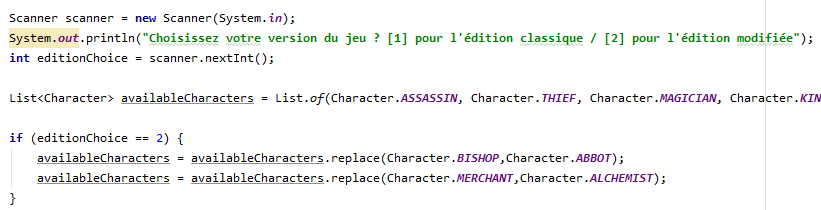


Nous avons implémenté la fonction avec l’algorithme, simple, suivant : on prend la liste de toutes les associations joueurs-personnages. On suppose que le joueur correspondant à la première association de cette liste est le plus riche, on l’ajoute donc à la liste des joueurs les plus riches. Ensuite on parcourt toutes les associations. Si un joueur d’une association possède plus d’or que le premier élément de la liste des joueurs les plus riches jusqu'à maintenant (on prend le premier car on sait qu’il y en a toujours un, et car si il y en a plusieurs ils ont forcément le même nombre d’or), on vide la liste (car les joueurs qui ne sont pas les plus riches ne nous intéressent pas), et on ajoute le joueur plus riche à la liste des joueurs les plus riches. Si un joueur d’une association possède autant d’or que le premier élément de la liste des joueurs les plus riches, on vérifie qu’il n’est pas déjà présent dans la liste (car on a supposé le premier joueur comme le plus riche mais on le traitera à nouveau après cette supposition) puis on l’ajoute à la liste car ca peut vouloir dire qu’il y a plusieurs joueurs “les plus riches”. Enfin, on retourne la liste des joueurs les plus riches maintenant remplie.

Nous avons ajouté cette méthode dans la classe GameRoundsAssociation. Cela faisait sens pour nous car on cherche le joueur le plus riche parmis toutes les associations, et le niveau de la classe Player était trop “bas” : nous n’aurions pas pu connaître tous les joueurs du jeu. Contrairement à l’Alchimiste nous avons choisi une approche TDD pour implémenter l’Abbé, nous avons donc réalisé une classe de test sur la classe GameRoundsAssociation que nous avons rempli au fur et à mesure du développement et que vous pourrez retrouver dans le code.

## Pouvoir choisir avec quelle édition du jeu on peut jouer

Pour implémenter le choix de l’édition du jeu que le joueur veut utiliser, dans la méthode jouer, au moment de créer la variable availableCharacters. Pour cela nous avons utilisé le scanner pour récupérer l’entrée de l’utilisateur. Par défaut on supposera qu’il va jouer avec l’édition classique, et s’il veut jouer avec l’édition modifiée, on remplacera le marchand par l’alchimiste, et le prêtre par l’abbé dans les personnages jouables.



# 

# Conclusion

En conclusion, au long du projet, nous n’avons pas rempli tous les objectifs qui nous étaient fixés de par un manque de temps, mais aussi de compréhension de la globalité du sujet (et de toutes les techniques enseignées) de notre part. Cependant nous avons pu accomplir la majorité des tâches, notamment du nettoyage de l’application, ce qui à défaut de rendre son fonctionnement optimisé, nous a permis de rendre sa compréhension plus simple. Nous n’avons pas effectué de modifications au niveau des classes du programme, le diagramme de classes est donc encore fondamentalement le même qu’au départ du projet, excepté des attributs que nous avons pu rajouter (au niveau des personnages par exemple).