



Janvier 2020

Rapport de projet Test-Driven Development

007 RPG

Julien Gorjon

Julien Kirstein

Yannis Argyrakis

Table des matières

1. Introduction	2
2. Projet initial	2
2.1 Description fonctionnelle	2
2.2 Description technique	2
3. Design patterns	2
3.1 Builder	3
3.1.1 Analyse.....	3
3.1.2 Améliorations.....	3
3.2 Singleton	3
3.2.1 Analyse.....	3
3.2.2 Améliorations.....	4
4. Critères de qualité	6
4.1 Fiabilité	6
4.1.1 Analyse.....	6
4.1.2 Améliorations.....	6
4.2 Performance.....	7
4.2.1 Analyse.....	7
4.2.2 Améliorations.....	8
4.3 Extensibilité	9
4.3.1 Analyse.....	9
4.3.2 Améliorations.....	11
4.4 Facilité d'utilisation	12
4.4.1 Analyse.....	12
4.4.2 Améliorations.....	14
4.5 Capacité fonctionnelle	14
4.5.1 Analyse.....	15
4.5.2 Améliorations.....	17
5. Etat final	18
6. Conclusion	19
7. Annexe	21
Diagramme de classe.....	21

1. Introduction

Dans le cadre du cours d'architecture logiciel, nous avons été amenés à récupérer un projet existant ainsi qu'à critiquer sa qualité et mettre en place une série d'éléments afin d'améliorer la qualité de celui-ci. Pour ce faire, nous avons utilisés différents outils tels que Jenkins et Github dont voici les liens ci-dessous.

Github: <https://github.com/JulienKirstein/007RPG>

Jenkins: https://jenkins.ecam.be/job/gr8_007RPG/

2. Projet initial

2.1 Description fonctionnelle

Le projet que nous avons repris est une version rpg¹ du célèbre jeu 007. Dans ce jeu, le joueur incarne un héros de son choix qui possède certaines statistiques de base et qui, tout au long de la partie, doit combattre des Mobs² ayant des armures et des armes aléatoires qui augmentent leurs statistiques. Le joueur a trois coups possibles, soit il attaque, soit il recharge, soit il se défend. Pour terminer la partie, le joueur doit passer 5 paliers comportant chacun 3 combats. A chaque fois que le joueur passe un palier, les Mobs deviennent plus puissants et chaque fois que le joueur tue un Mob, il a 1 chance sur 3 de trouver une arme ou une armure qu'il pourra équiper ou non.

2.2 Description technique

Ce jeu créé en java est donc constitué d'une interface utilisateur assez bien pensée même si lors de notre reprise du projet, nous trouvions qu'elle manquait de clarté restant toutefois fonctionnelle. Pour mettre en œuvre ce jeu, l'équipe de développement initiale a utilisé le design pattern builder où l'on trouve une classe externe qui contrôle l'algorithme de construction et le design pattern singleton dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. De plus, l'équipe de développement a mis en avant différents critères de qualités, à savoir la fiabilité, la performance, l'extensibilité, la facilité d'utilisation ainsi que la capacité fonctionnelle.

3. Design patterns

Le paradigme de programmation utilisé pour réaliser le projet du jeu est celui de l'orienté objet, ce paradigme représente des objets qui interagissent entre eux. On obtient les fonctionnalités attendues dans notre programme grâce aux différentes relations qui lient nos classes. Dans le

¹ Rpg = role playing game

² Mob = monstre

paradigme orientée objet, l'étape de modélisation du programme est la plus importante, en annexe vous trouverez les diagrammes de classe utilisés pour la réalisation du programme

Le paradigme de programmation utilisé pour réaliser le projet du jeu est celui de l'orienté objet, ce paradigme représente des objets qui interagissent entre eux. On obtient les fonctionnalités attendues dans notre programme grâce aux différentes relations qui lient nos classes. Dans le paradigme orientée objet, l'étape de modélisation du programme est la plus importante. Pour modéliser les classes, la précédente équipe s'est basé sur divers design patterns.

Vous retrouverez en annexe les diagrammes de classe utilisés pour la réalisation du programme.

3.1 Builder

Ce design pattern est utile pour la création d'objets complexes dont les différentes parties doivent être créées suivant un certain ordre ou algorithme spécifique. Une classe externe est alors chargée de « construire » ces objets.

Dans ce projet, les mobs créés devant comprendre des statistiques dépendantes de l'avancement du jour dans le jeu, l'utilisation du Builder s'est avérée totalement adéquate.

3.1.1 Analyse

Dans le programme reçu, la classe MobsBuilder.java ne sert choisit aléatoirement, mais en fonction du level dans le jeu tout de même, les différents types de mobs à créer. Il les crée donc et les ajoute au tableau des mobs, précédemment créé grâce à la classe MobsBoards. C'est alors cette dernière qui choisira toutes les caractéristiques à attribuer à ces trois mobs, encore une fois selon le même type algorithme aléatoire.

3.1.2 Améliorations

Aucune amélioration n'a été faite.

3.2 Singleton

L'objectif du design pattern Singleton est de restreindre l'instanciation d'une classe à un seul objet. Il est utilisé lorsqu'on a besoin d'exactly un objet pour coordonner des opérations dans un système. Dans ce projet, ce design pattern s'applique à l'objet game ainsi qu'à l'objet player puisqu'en toute logique, ceux-ci se doivent d'être des instances uniques.

3.2.1 Analyse

La méthode utilisée par le groupe précédent pour réaliser le Singleton, est de créer une instance de la classe en mode static. Les lignes de code concernées sont illustrées ci-dessous, pour la classe Game et Player.

```
static {  
    try {
```

```

        instance = new Game();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

```
private static Player instance = new Player();
```

Aucun constructeur n'étant défini, il est bien impossible de créer plusieurs instances. Pour récupérer ces instances uniques afin de les utiliser, de simples fonctions getInstance() ont été écrites pour renvoyer celles-ci.

3.2.2 Améliorations

Malheureusement, cette méthode ne permet pas de lancer la création de ces instances depuis un test JUnit. En effet, le principe étant justement qu'aucune méthode ne peut être appelée pour instancier ces deux classes. Les instances sont d'ailleurs déjà créées au début de l'exécution des test JUnit.

Et ce dernier est en fait un véritable problème si on veut effectuer des tests sur l'ensemble du programme. Cela s'explique par le fait que seule la création de l'instance game provoque l'affichage de texte sur le terminal dans le but que l'utilisateur entre son nom afin de créer l'instance player.

```
private Game() throws InterruptedException {
    System.out.println("WELCOME IN 007 RPG");
    TimeUnit.SECONDS.sleep(1);
    System.out.println("???WHAT IS YOUR NAME HOMIE???");
    Scanner scanner = new Scanner(System.in);
    String name = scanner.nextLine();
    this.player = Player.getInstance();
    player.setName(name);
}

```

Puisque ces lignes de code sont exécutées suite à la création statique de game, l'entrée du nom du joueur ne peut être effectuée par aucun test. Le programme reste alors bloqué et aucune méthode ne peut être testée.

Afin de résoudre ce problème, nous avons décidé d'utiliser une autre méthode pour le Singleton de l'objet game. Avec cette-ci, l'instanciation est dynamique ; c'est seulement une fois la méthode getInstance() appelée que l'objet game est créé, à condition de ne pas déjà exister.

```
public static Game getInstance(Scanner in) throws InterruptedException
```

```

{
    if (game == null)
    {
        game = new Game(in);
    }
    return game;
}

```

Cette méthode n'étant appelée qu'au lancement du programme, nous avons pu lui mettre en paramètre un Scanner. Celui-ci est ensuite donné en paramètre à Game() qui l'utilisera pour configurer le nom du joueur déjà créé en static :

```

private Game(Scanner sc) throws InterruptedException
{
    System.out.println("WELCOME IN 007 RPG");
    System.out.println("???WHAT IS YOUR NAME HOMIE???");
    setPlayer(sc);
    loading();
}

```

Cette astuce permet finalement à un test de provoquer lui-même la création de l'instance game et la configuration du nom de joueur qui s'en suit :

```

@Test
public void createGameTest() throws InterruptedException
{
    Scanner sc = new Scanner("Yannis");
    Game game = Game.getInstance(sc);
    assertEquals(game==null,false);
}

```

4. Critères de qualité

4.1 Fiabilité

Pour l'équipe précédente, le critère de fiabilité est évalué par le fait que le logiciel doit résister aux erreurs de l'utilisateur ou de services externes.

4.1.1 Analyse

L'équipe précédente tout comme nous, a mis un point d'honneur sur la fiabilité de ce projet. Celle-ci n'avait pas mis en place de tests automatisés mais garantissait que le logiciel résistait à toutes les entrées possible d'un utilisateur. Cependant, lors de nos phases d'analyse, nous avons relevé de nombreux bugs dans l'interface tels que ceux cités ci-dessous:

-Lorsque l'utilisateur n'entrait pas un int, le programme crashait.

```
Curious Poutine : 20/20HP      bullets: 0   attaque: 10
      1 ATTACK, 2 DEFEND, 3 RELOAD
az
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at game.Game.fight(Game.java:126)
    at game.Game.loop(Game.java:228)
    at game.Main.main(Main.java:8)
```

-Quand l'utilisateur perdait, l'utilisateur voyait en boucle et non une seule fois qu'il était mort.

-Bugs d'affichages.

-Quand le joueur changeait d'objet, les dégâts supplémentaires ne se rajoutait pas.

-...

4.1.2 Améliorations

A la vue des problèmes rencontrés sur le projet récupéré et dans le but de par la suite garantir la fiabilité du projet, nous avons utilisé l'outil Jenkins qui nous a permis d'obtenir de nombreuses informations caractérisant la qualité et plus particulièrement la fiabilité du code en permettant de faire des tests automatisés et en relevant automatiquement des informations sur le code. Nous avons donc mis en place une batterie de tests Junit exécutés par Jenkins et résolu tous les problèmes cités plus haut un par un comme le fait que l'utilisateur peut maintenant entrer ce qu'il veut dans la console sans que le programme ne crash grâce à la fonction Checkinput() qui vérifie simplement l'entrée et agit ensuite en fonction. Grâce à Jenkins, nous avons pu quantifier notamment le code coverage qui a bien augmenté comparativement au projet initialement où le code coverage était nul.

4.2 Performance

Pour l'ancienne équipe, l'évaluation du critère de qualité permettant d'affirmer la performance consistait en ce que le jeu se lance rapidement.

4.2.1 Analyse

Le groupe étant à l'origine de ce projet affirme avoir vérifié la performance de leur application grâce à un module time.

Cependant, après avoir analysé le projet nous n'avons nullement trouvé de trace d'une quelconque mesure de temps de lancement du jeu. La caractéristique du temps de lancement de jeu dépend fortement du matériel utilisé par le joueur et non pas de la quantité.

Nous pouvons toutefois analyser les potentiels appels de fonctions qui pourraient influencer sur la rapidité de lancement d'une partie. Une fonction va nous intéresser particulièrement :

```
private void loading(){
    System.out.println("                .");
    wait(300);
    System.out.println("                .");
    wait(300);
    System.out.println("                .");
    wait(300);
}
```

La fonction "loading" dans le fichier "Game.java" est appelé à maintes reprises lors de l'exécution du programme et notamment lors de la première fonction appelée qui consiste en la création de l'instance "game". Elle a pour unique objectif d'ajouter un délai de 900ms en imprimant un point toutes les 300ms, ce qui donne une impression de chargement.

Cette fonction a donc une influence directe sur la performance de l'application car le temps d'attente en millisecondes ralentit le jeu pour l'utilisateur, son appel est donc plus que discutable.

De plus, l'utilisation de la variable de type scanner qui a pour objectif d'appeler les fonctions n'est pas efficace. Dans le projet initial, les membres du groupe ont créé une nouvelle variable de type "Scanner" à chaque fois qu'il avait besoin de récupérer l'entrée de l'utilisateur. Cela ne peut que rajouter de la latence au programme car le scanner occupe de la mémoire ram car celui-ci n'est jamais fermé. Par exemple à la ligne 125 du fichier "Game.java" :


```
private void fight(Mob mob) {

    while(player.getHp() > 0 && mob.getHp() > 0) //while the player and the
mob are alive (the whole loop represents one turn)
    {
        boolean actionIsValid = false;
        while (!actionIsValid) {
            System.out.println("          1 ATTACK, 2 DEFEND, 3 RELOAD");
            Scanner scanner = new Scanner(System.in);
            int action = scanner.nextInt();
        }
    }
}
```

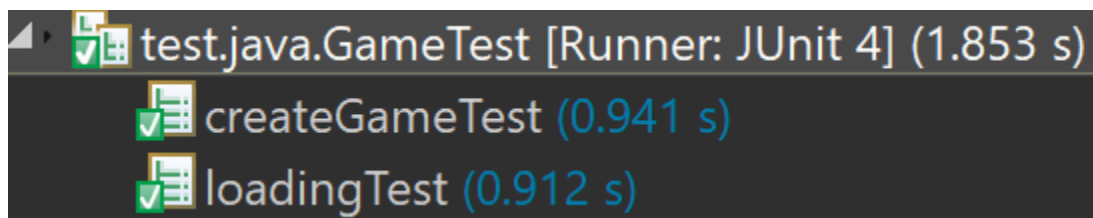
On constate que la fonction fight() qui est appelé lors de chaque début de combat va créer un scanner à chaque fois qu'un combat à lieu dans le jeu. Il est donc clair qu'au plus on aura appelé cette fonction, au plus la mémoire ram va être impacté par la création de nouveau scanner.

4.2.2 Améliorations

Après la reprise et l'analyse du projet, nous avons redéfinis le critère de qualité en précisant que l'application doit également être responsive. Cette redéfinition nous permet de travailler sur la performance globale de toute une partie du jeu et non seulement de son démarrage.

Métrique

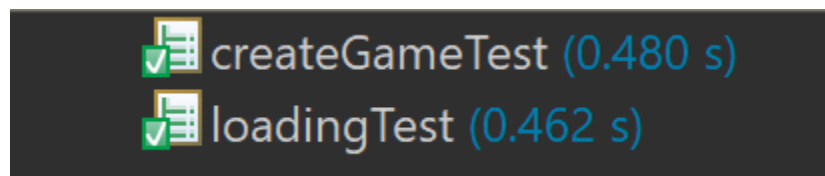
Afin de mesurer la performance du démarrage d'une partie, nous avons créé des tests unitaire avec JUnit qui créent une partie et inscrit un nom de joueur :



test.java.GameTest [Runner: JUnit 4] (1.853 s)

- createGameTest (0.941 s)
- loadingTest (0.912 s)

On peut voir ainsi que la création d'une partie prenait initialement 941ms dont 912ms sont dédié à la fonction loading().



createGameTest (0.480 s)

- loadingTest (0.462 s)

Nous avons donc décidé de réduire volontairement le temps de la fonction loading en réduisant l'attente entre l'impression d'un point de manière à laisser un temps de chargement mais augmenter de moitié le temps de création d'une partie. Voici nos résultats :

On constate donc qu'on a réduit considérablement la durée de lancement d'une partie. Celle-ci se fait donc maintenant en 480ms.

Suppression des scanner inutiles

La création de plusieurs scanner a été évité en ne laissant place qu'à un seul scanner qui est créé dans "Main.java" et passé en paramètre du constructeur de la classe "Game" . comme vous pouvez le voir ci-dessous

```
1) public static Game getInstance(Scanner in) throws InterruptedException
2) public static void main(String[] args) throws InterruptedException
{
    Game game = Game.getInstance(new Scanner(System.in));
    game.loop();
}
```

De cette manière on n'utilisera qu'un seul et même scanner tout au long d'une partie, ce qui évitera à des fonctions comme fight() d'en créer inutilement à chaque appel.

4.3 Extensibilité

Le programme du jeu doit être extensible ; l'ajout de nouvelles features doit être facilement réalisable. L'équipe qui a commencé ce projet a donc choisi une architecture et un design pattern Builder dans l'optique de permettre d'ajouter de nouveaux objets et surtout, de nouvelles mobs facilement.

4.3.1 Analyse

MobsBuilder

Afin de facilement ajouter de nouveaux mobs, une relation d'héritage est utilisée afin de créer rapidement une grande série de nouvelles classes mob. Héritant de la classe Mob qui contient toutes les méthodes communes à chaque mob, les classes de mobs ne doivent en effet contenir que les quelques caractéristiques qui leur sont propres. D'ailleurs les méthodes utilisées pour définir ces caractéristiques sont définies dans la classe Entity, elle-même héritée par la classe Mob.

Cependant, la séparation de tous ces mobs dans des classes différentes a engendré un problème qui dégrade sensiblement le caractère extensible du programme ; pour créer un tableau de 3 mobs, le MobsBuilder utilise un Switch afin de choisir aléatoirement parmi chaque type de mob.

```
public MobsBoard buildMobs(int level){
```

```

MobsBoard mobs = new MobsBoard();
int MAX_MOB = 7;
int MIN_MOB = 1;

for(int i= 0; i< 3; i++){

    switch(MIN_MOB + (int)(Math.random() * ((MAX_MOB - MIN_MOB)+1))) {

        case 1:
            mobs.addMob(new Zombie(level));
            break;

        case 2:
            mobs.addMob(new AngryDuck(level));
            break;

        ...
    }
}

```

D'une part, cela implique qu'à chaque création d'une nouvelle classe de mob, un nouveau « cas » doit être ajouté dans le switch pour ce nouveau mob, et le nombre total de mobs MAX_MOB doit être incrémenté. D'autre part, si la classe d'un mob est supprimée du programme, son « cas » doit être supprimé du switch. Et si on ne souhaite pas avoir de trou dans les indices des différents cas, tous les indices des cas suivants doivent être décrémenté.

Cette façon de faire ne respecte donc pas le critère d'extensibilité puisque, plus le programme prendra de l'ampleur, plus la modification demandera de travail.

MobsBoard

La fonction addMob() de la classe MobsBoard utilisée dans le switch du MobsBuilder pour créer un tableau de 3 mobs ne rend pas non plus le code très modulable. Le rôle de cette fonction est de choisir aléatoirement parmi plusieurs séries de caractéristiques d'attribuer ceux-ci au mob avant de renvoyer celui-ci. Ces listes de caractéristiques étant elle-même constituées de sous-liste correspondant à un certain level atteint dans le jeu, des listes à deux dimensions sont utilisées pour les stocker :

```

private static String[][] qualityBoard = {
    {"Baby", "Happy", "Cute", "Not sad", "Curious"},
    {"Geeky", "DJ", "Seductive", "Kinky", "Meticulous"},
    {"Go-muscu", "Pro-hacker", "Serious", "Sneaky", "Totally incognito"},
    {"Sasuke", "Mega", "Enraged", "Rudeboy", "Fuckboy"},
    {"Super-angry", "The wisest", "Chuck-Norris", "God-like", "Once again that fucking"}
};

```

Le problème dans la fonction addMob() est l'encodage manuel du nombre du nombre d'éléments encodés dans ces listes à deux dimension, ou matrices. Des ajouts ou suppressions d'éléments implique donc forcément une modification de l'indexe maximum pour la caractéristique concernée dans addMob(). De plus, les indexes utilisés ne concernent qu'une dimension de la matrice. Autrement dit, chaque ligne correspondant à un certain level de jeu, doit toujours contenir autant

d'élément ; si une quality est inventée pour le level deux, des quality doivent être inventées pour chacun des autres level avant de pouvoir être ajouté dans le code :

```
void addMob(Mob mob) {  
  
    int MAX_WEAPON = 1;  
    int MIN_WEAPON = 0;  
    int MAX_ARMOR = 1;  
    int MIN_ARMOR = 0;  
    int MAX_QUALITY = 4;  
    int MIN_QUALITY = 0;  
  
    mob.setWeapon(MobsBoard.weaponBoard[mob.getLevel()][MIN_WEAPON +  
(int)(Math.random() * ((MAX_WEAPON - MIN_WEAPON)+1) )]);  
    mob.setArmor(MobsBoard.armorBoard[mob.getLevel()][MIN_ARMOR + (int)(Math.random() *  
((MAX_ARMOR - MIN_ARMOR)+1) )]);  
    mob.setQuality(MobsBoard.qualityBoard[mob.getLevel()][MIN_QUALITY + (int)(Math.random()  
* ((MAX_QUALITY - MIN_QUALITY)+1) )]);  
    mob.setName(mob.getClass().getSimpleName());  
    mobs.add(mob);  
}
```

4.3.2 Améliorations

MobsBuilder

Pour améliorer l'extensibilité, il aurait fallu parvenir à récupérer le nombre de classes existantes de mobs, et récupérer leurs noms afin de les utiliser. Ainsi il n'y aurait plus besoin d'écrire manuellement leurs noms dans le switch. La première ligne du code ci-dessus permet de récupérer le nombre de classes existantes dans le répertoire correspondant, tandis que la deuxième renvoie le nom de la troisième classe parmi toutes celles qui ont pu être listées.

```
System.out.println(new File(".\\src\\main\\java\\game\\entity\\mob").listFiles().  
length);  
System.out.println(new File(".\\src\\main\\java\\game\\entity\\mob").listFiles()[  
2]);
```

Malheureusement, après de nombreuses recherches et tentatives, nous ne sommes pas parvenus à appeler les classes pour les utiliser à partir du nom sous forme de string. Le code suivant illustre le code qui aurait pu être utilisé. Notons que, comme il est présenté ici, il ne peut pas fonctionner vu la conversion nécessaire du nom de classe de type File en String. Mais les tests, encore présent ici en commentaire, utilisant directement des String n'ayant jamais fonctionné, le code permettant cette conversion n'a pas été profondément recherché.

```
try {  
    Class myclass= Class.forName(new File(".\\src\\main\\java\\game\\entity\\mob"  
).listFiles()[2]);
```

```

    //Class clazz = Class.forName(".\\src\\main\\java\\game\\entity\\mob\\Zombie.
java");
} catch (ClassNotFoundException e) {
    System.out.print("Class not found");
}

```

MobsBoard

L'amélioration de la fonction addMob fut relativement simple à mettre en place. Elle consiste à utiliser la véritable taille de la ligne correspondant au niveau de jeu actuel, dans chaque tableau, pour l'attribuer aux variables contenant l'indice maximal de ces listes. Ainsi, on peut maintenant modifier autant qu'on veut ces différents tableaux sans avoir à modifier le code de cette fonction. Il est évident aussi qu'il était inutile d'utiliser des variables contenant les indices minimum, ceux-ci valant toujours zéro.

```

void addMob(Mob mob) {

    int level =mob.getLevel();
    int MAX_WEAPON = MobsBoard.weaponBoard[level].length - 1;
    int MAX_ARMOR = MobsBoard.armorBoard[level].length - 1;
    int MAX_QUALITY=MobsBoard.qualityBoard[level].length - 1;

    mob.setWeapon(MobsBoard.weaponBoard[level][(int)(Math.random() * (MAX_WEAPON +1))]);
    mob.setArmor(MobsBoard.armorBoard[level][(int)(Math.random() * (MAX_ARMOR +1))]);
    mob.setQuality(MobsBoard.qualityBoard[level][(int)(Math.random() * (MAX_QUALITY +1))]);
    mob.setName(mob.getClass().getSimpleName());
    mobs.add(mob);
}

```

4.4 Facilité d'utilisation

Ce critère de qualité implique que l'application soit "User-friendly" afin que le jeu soit clair et accessible à tous.

4.4.1 Analyse

La précédente équipe ayant réalisé l'application a indiqué dans son rapport initial qu'elle avait fait essayer le jeu à des amis afin d'évaluer leur satisfaction quant à son utilisation. Suite aux commentaires reçus l'équipe initiale a décidé d'ajouter les règles du jeu. Ci-dessous vous pouvez voir un extrait des règles se trouvant dans la fonction "loop()" du fichier "Game.java":

```

while (player.getHp() > 0) {

    System.out.println("-----The Rules-----");
}

```

```

        System.out.println("the rules are simple, you have 3 stats:");
        System.out.println("Health points: when it goes to zero you die");

        System.out.println("but don't worry you can drop weapons and armor on
mobs!" );
        loading();
        System.out.println("Press enter when you are ready" );
        Scanner starter = new Scanner(System.in);
        String ready = starter.nextLine();
        System.out.println("-----GOOD LUCK-----
----" );

```

Notre équipe a constaté qu'à chaque fois qu'un palier (ou niveau) a été franchi et que 3 combats sont un succès, le programme vérifie si la vie du joueur est suffisante pour continuer la partie. Dans le cas où sa vie est supérieure à 0, les règles vont à nouveau être affichées. Dès que le joueur est prêt à commencer le nouveau palier, il appuie sur une touche afin de lancer le premier combat.

En testant manuellement l'application et en effectuant plusieurs parties, notre équipe a relevé un caractère d'affichage qui n'était pas en accord avec le caractère user friendly de l'application car celui-ci rendait un combat peu clair. Ci-dessous voici un extrait d'une action effectuée par un joueur :

```

Cute RedHead : 15/15HP      bullets: 0   attaque: 5
      1 ATTACK, 2 DEFEND, 3 RELOAD
1
|
      .
      .
      .
YOU ATTACK                      ENEMY RELOADS
-15.0
      RedHead Ã   perdu 15 hp
Playerheals himself-1 Hp
Yannis :90/100HP                RedHead :0/15HP

```

Après avoir sélectionné l'action (en entrant 1, 2 ou 3), on peut voir un résumé du choix de notre action ainsi que du choix de l'action de l'ennemi. S'en suit un affichage des dégâts respectivement infligés par un joueur à son ennemi ainsi que l'impact qu'a eu l'utilisation de son arme sur sa propre vie.

Comme vous pouvez le constater, l'affichage de l'historique des actions du combat n'est pas clair aux yeux de l'utilisateur et il est très difficile de distinguer les différentes actions du combat. Ce problème d'affichage impacte donc la facilité d'utilisation du jeu qui a pour objectif principal d'être "accessible à tous".

4.4.2 Améliorations

Suite à l'analyse du projet initial, nous avons fait le choix de n'afficher les règles du jeu qu'une fois au tout début de la partie de cette façon suivante :

```
void loop() {
    while (player.getHp() > 0) {
        if(level==0){
            displayRules();
        }
    }
}
```

En créant une fonction “displayRules()” dont l’objectif est uniquement de s’occuper de l’affichage des règles. Cet affichage se fait que lors du premier palier quand “level” est égal à 0. Pour rappel, il y a 5 palier à franchir avec chaque fois 3 montres à vaincre.

Nous considérons donc qu'il est inutile de surcharger la console en affichant donc 5 fois les règles dans le cas où l'utilisateur arrive au dernier palier étant donné qu'il peut toujours remonter l'affichage de la console pour les relire.

De plus, nous avons corrigé l’affichage de l’historique des actions lors d’un combat afin de le rendre l’interface plus claire et donc plus accessible. Voici le nouvel affichage d’une action:

```

      1 ATTACK, 2 DEFEND, 3 RELOAD
1
|
    .
    .
    .
YOU ATTACK                                ENEMY RELOADS
RandomDrunkedGuy lost 15 hp
Yannis heals himself -1 Hp
-----
Yannis :99/100HP                        RandomDrunkedGuy :15/30HP
```

Le nom du joueur est repris lorsque celui-ci effectue l'action et nous n'affichons donc plus "Player heals himself..". Le récapitulatif des actions est séparé pour le joueur et le monstre qu'il affronte.

4.5 Capacité fonctionnelle

Le logiciel doit respecter les spécifications et surtout doit résoudre le problème principal de l'utilisateur, c'est-à-dire lui permettre d'effectuer une partie complète de 007 en affrontant des monstres originaux et en utilisant diverses armes.

4.5.1 Analyse

Suite à de nombreux tests sur le projet initial, nous avons constaté qu'il était possible de jouer une partie entièrement et cela en pouvant utiliser diverses armes contre divers monstres. Le projet était donc déjà fonctionnel

Toutefois, nous avons remarqué une erreur de conception du programme qui peut influencer le bon déroulement d'une partie. Il se trouve dans la fonction loop() de la partie comme vous pouvez le voir ci-dessous :

```
while (fightNumber < 3)
```

Tant que le joueur n'a pas effectué les 3 combats du palier,

```
mob = mobs.getMob(fightNumber);  
loading();  
        mob.getInfoMob();  
        fight(mob);
```

On récupère le monstre affronté en fonction de la variable "fightNumber". Ensuite on appelle la fonction "fight()" qui lance un combat. Cette fonction est une void et engendre un combat uniquement dans ces conditions :

```
while(player.getHp() > 0 && mob.getHp() > 0)
```

Si la vie du joueur et du monstre est supérieure à 0. Dans le cas où le joueur n'a donc plus de vie, cette fonction "fight()" ne fait rien.

De retour dans le "while(fightNumber < 3)", une comparaison est effectuée et le numéro de combat est incrémenté de 1 uniquement si le monstre affronté n'a plus de vie. Dans le cas où c'est le joueur qui n'a plus de vie, cela signifie que la partie est terminée, voici donc ce que le programme affiche :

```
if (player.getHp() <= 0) {  
    System.out.println("-----YOU DIED-----");  
    System.out.println(score);  
}
```


On constate donc que l'information de mort du joueur est envoyée ainsi que son score. Un problème survient donc car la variable "fightNumber" n'est jamais incrémenté, ce qui crée une boucle infinie affichant les informations du dernier monstre affronté en plus du score :

```
NEW ENEMY : Curious RedHead dressed with Tshirt and armed with Fist
.
.
.
Curious RedHead : 2/15HP      bullets: 2   attaque: 5
-----YOU DIED-----
0
NEW ENEMY : Curious RedHead dressed with Tshirt and armed with Fist
.
.
.
Curious RedHead : 2/15HP      bullets: 2   attaque: 5
-----YOU DIED-----
0
NEW ENEMY : Curious RedHead dressed with Tshirt and armed with Fist
.
```

La partie ne se finit donc jamais réellement.

Un autre problème concerne l'ajout d'une armure pour le joueur. En effet, à chaque fin de combat le joueur est censé avoir une chance de ramasser l'armure de l'adversaire et donc d'améliorer ces statistiques grâce à celle-ci. Voici ce que le programme affiche lorsqu'on accepte une armure :

```
Yannis :83/100HP                      Poutine :-7/20HP
-----
Poutine DEAD
You heal yourself7 Hp
.
.
.
???Do you want Tshirt???
Actual armor : Naked---> Damage raduction :0 added maxHP :0 regeneration :5
New weapon : Tshirt---> Damage raduction :15 added maxHP :10 regeneration :1
                        y/n
y
you are equipped with : Tshirt
.
.
.
Yannis : 90/100HP      bullets: 1   attack: 10
Armed with : Fist
Equipped with : Tshirt
```

On constate un "New weapon :." qui contient les caractéristiques de la nouvelle armure, cela est probablement dû à une erreur d'inattention lors de la réalisation du projet. On constate également que le nombre maximum de point de vie du joueur n'a pas augmenté en accord avec ce que lui apporte l'armure ("added maxHP : 10") . Dans l'exemple ci-dessus, il devrait passer à 110HP

4.5.2 Améliorations

Nous avons résolu le problème de la boucle infinie en fin de partie par le changement de la void loop() en fonction de type int. De cette manière, nous pouvons utiliser un return dans la fonction afin de sortir de la boucle dans le cas où le joueur meurt où dans le cas où celui-ci n'a plus de vie. Ci-dessous la preuve que la void "loop()" est devenue une fonction renvoyant un int :

```
public int loop()
```

Si le joueur n'a plus de vie on retourne une valeur ce qui met fin au programme :

```
if (player.getHp() <= 0) {  
    System.out.println("-----YOU DIED-----");  
    System.out.println(score);  
    return 0;  
}
```

Le problème d'utilisation d'une nouvelle armure a également été résolu en mettant à jour le nombre de point de vie maximum dans le cas où le joueur accepte sa nouvelle armure.

```
1 :70/100HP                               RandomDrunkedGuy :-4/30HP  
-----  
RandomDrunkedGuy DEAD  
You heal yourself 7 Hp  
.  
.  
.  
???Do you want Tshirt???  
Actual armor : Naked---> Damage raduction :0| added maxHP :0| regeneration :5  
New armor : Tshirt---> Damage raduction :15| added maxHP :10| regeneration :1  
                               y/n  
y  
you are equipped with : Tshirt  
.  
.  
.  
1 : 77/110HP      bullets: 1   attack: 10  
Armed with : Fist  
Equipped with : Tshirt
```

Vous pouvez voir ci-dessous que le joueur du nom "1" voit désormais ces points de vie maximum augmenté de 10. Nous avons également corrigé le texte pour afficher que cela est bien une "New armor" et ajouté un caractère de séparation entre les caractéristiques proposées par les armures/armes.

5. Etat final

Suite aux modifications effectués sur le projet initial, voici un bilan de l'état final du projet et des actions qui ont été effectuées :

- Redéfinition du singleton de la class "Game"
 - Définition dynamique du singleton
 - Test de la création d'une instance « game » dans un test unitaire
- Ouverture du projet à l'extensibilité
 - Redéfinition du builder "Mobsbuilder" générant la liste des mobs
- Amélioration de la fiabilité de l'application
 - Mise en place de l'outil d'intégration continue Jenkins "gr8_007RPG"
 - Restructuration des fichiers pour l'utilisation de Maven
 - Automatisation de build du projet avec l'outil Maven
 - Mise en place du SLOCCount trend pour compter le nombre de ligne de code
 - Analyse des PMD (réduction à 0) et des checkstyles
 - Analyse du code coverage
 - Mise en place de tests unitaires sous JUnit5 dans le fichier "GameTest.Java"
 - Reporting des résultats des tests dans Jenkins
 - Corrections de bugs sur les inputs de l'utilisateur
 - Vérification par la fonction checkIntInput()
- Evaluation et amélioration du critère de performance
 - Mesure du temps d'exécution de la fonction loadingTest() et createGameTest()
 - Optimisation de performance en réduisant la durée de loading()
 - Suppression des scanners inutiles
- Amélioration de la facilité d'utilisation
 - Création de la fonction displayRules()
 - Affichage des règles qu'une seule fois
 - Amélioration de l'affichage de l'historique des actions
 - Amélioration générale de la clarté de l'interface
 - Correction de la majeure partie des fautes/mauvais caractères
- Amélioration de la capacité fonctionnel
 - Réalisation de tests manuel de plusieurs parties
 - Suppression de la boucle infinie en fin de partie
 - Affichage du score une seule fois
 - Redéfinition de la void loop() en public int loop()

6. Conclusion

Certains objectifs que nous nous sommes fixés lors de la reprise du projet non pas pu être atteints comme :

- La mesure du couplage comme métrique afin d'évaluer le critères de fiabilité des différentes classes tel que la classe Player ou Mob. Cette métrique là nous aurait permis de vérifier le caractère stable ou instable du code. Nous n'avons pas trouvé comment mettre en oeuvre une telle mesure au sein du projet.
- Le refactoring complet de la void loop() dans la classe "Game" . En effet, à la reprise du projet celui-ci était déjà dans un état fonctionnel et la void loop() permettant de gérer une partie était très longue. Cette méthode a été analysé en détail et cela nous a prit énormément de temps de comprendre son fonctionnement. Nous aurions voulu par exemple éviter les doubles appels de fonction, ce qui n'a pas toujours été possible.
- Retirer tous les bads smells du programme. Comme nous n'avons pas refactoriser entièrement les fonctions, nous avons laissé certains bad smells initialement présent qui n'auraient normalement pas lieu d'être dans le projet.
- Le code coverage n'a pas pu atteindre 100%. Ceci dit, nous avons tout de même bien amélioré le code coverage grâce aux test Junits et à l'outils Jenkins sachant que lorsque nous avons récupéré le projet, aucun test automatisé n'était en place.

Les objectifs qui ont été atteints sont les suivants :

- Une amélioration globale de l'ensemble du projet concernant les facteurs de qualités initialement proposés par la précédente équipe.
- La mise en place d'outils pour quantifier les critères de qualité et de tests automatisés.
- Le code est maintenant plus propre, les importations de packages inutiles ont été supprimé notamment. Les variables inutiles ont été supprimées.
- La fiabilité du programme a été augmenté et celui-ci est désormais très robuste contre les erreurs d'entrées de l'utilisateur.
- Le respect du critère d'extensibilité de l'application pour un potentiel ajout de nouveaux Mobs.

- L'application est désormais User-Friendly et l'interface réalisé en console permet de comprendre facilement le déroulement d'une partie.
- Le critère de performance a été retravaillé afin que l'expérience du jeu soit agréable et puisse minimiser l'impact sur l'utilisation de la mémoire RAM. Le jeu se lance plus rapidement qu'avant.
- L'application du concept du "Test Driven Development" qui nous a notamment amené la modification du singleton.

7. Annexe

Diagramme de classe

