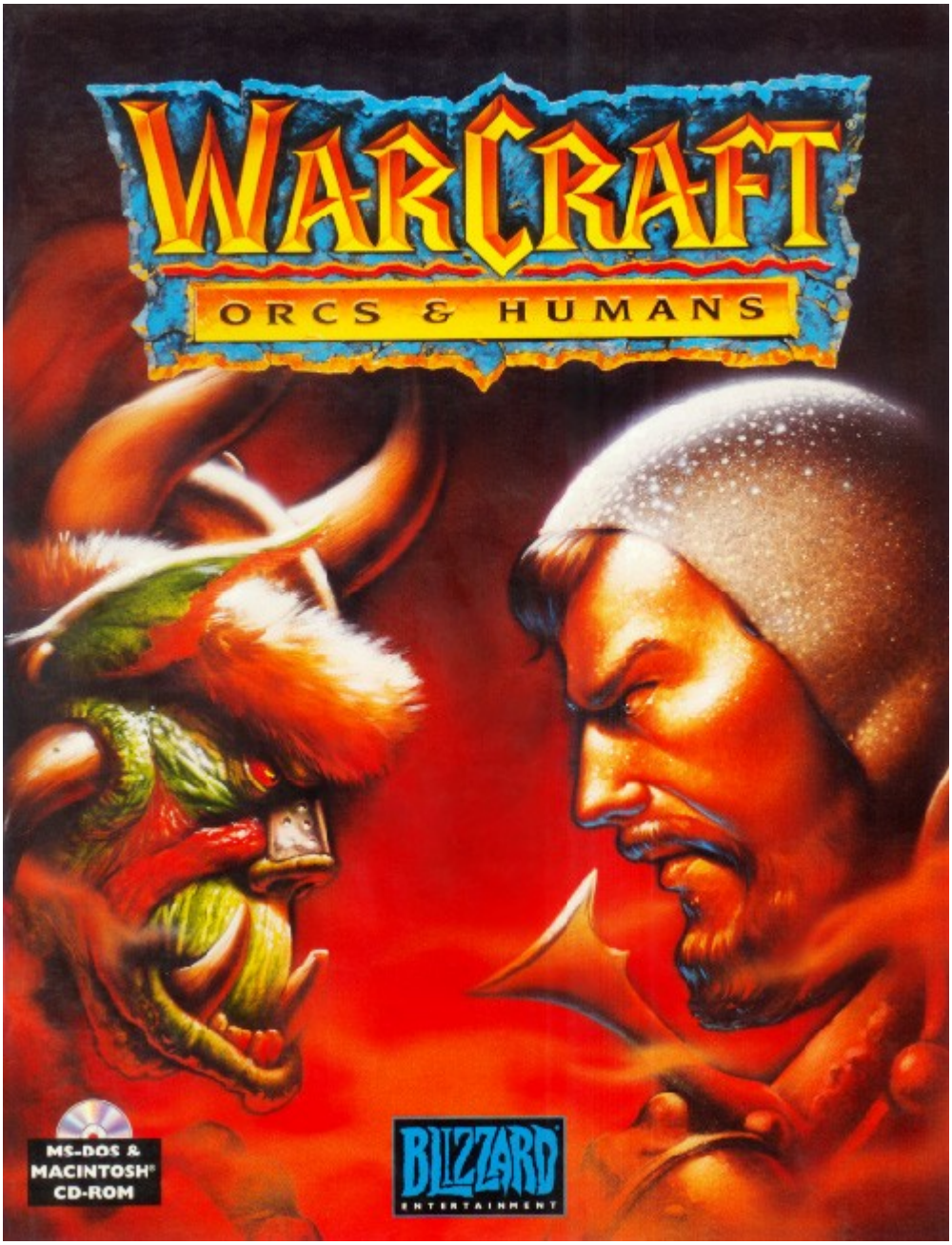
LARBI Adel, M1 STL
LARBI YOUCEF Mohamed Reda, M1 STL

Rapport du projet Warcraft
Composants



UPMC
SORBONNE UNIVERSITÉS

Sommaire

I- Introduction

Warcraft Orcs and Humans, est un jeu de stratégie sorti en 1994, développé par la fameuse société *Blizzard*. Dans ce jeu, on a deux camps qui s'affrontent, principalement en se battant et en récoltant des ressources tels que l'or.

Le but de ce projet est donc d'implémenter une version simplifiée de ce jeu vidéo. On implémentera le moteur de jeu, ainsi que des composants tels que les villageois, les murailles, les routes, ou encore les mines. L'objectif étant d'implémenter un moteur robuste, sûr et fiable.

Pour atteindre cet objectif, nous devons passer par une spécification claire et correcte, une implémentation respectant la spécification à la lettre, l'utilisation de la programmation par contrat pour garantir le respect de la spécification, et enfin un jeu de tests assez large pour tester le moteur avec plusieurs paramètres.

Concernant les tests, nous aurons besoin de tests corrects. Pour cela, nous ajouterons une implémentation buggée du moteur de jeu afin de vérifier cela.

En plus de cet objectif, nous souhaitons que notre moteur soit facilement configurable. Nous ajouterons donc une fonctionnalité permettant de stocker des paramètres prédefinis pour les différents composants dans des fichiers de configuration.

Le langage dans lequel on developpera notre moteur est le langage Java. De plus, le langage qu'on utilisera pour les fichiers de configuration est JSON.

Important : Ce projet est réalisé avec Java 8 et nécessite ant version 1.9.3.

II- Spécification

La première chose à faire est de se mettre d'accord sur une spécification claire, complète, et cohérente. En effet, nous dépendons tout les deux de cette partie, nous devons donc nous assurer que cette partie reste plus ou moins fixe durant le déroulement du projet, afin de garantir un temps de développement efficace.

Pour spécifier notre programme, nous avons utilisé le langage de spécification vu en cours (merci les snippets gedit pour les symboles spéciaux). Nous avons donc pour chaque service, un fichier qui contient sa spécification dans le dossier ***spec***.

Nous avons essayé d'eviter au maximum les redondances pour ce projet, en utilisant entre autre les raffinements. Nous avons donc décomposé notre projet en 8 services :

- Le service Villageois

- Le service Terrain (Pas vraiment un service à part entière, il permet surtout d'eviter les redondances)

- Le service Route qui raffine Terrain

- Le service Muraille qui raffine Terrain

- Le service Structure qui raffine Terrain (De même que Terrain)

- Le service Mine qui raffine Structure

- Le service HotelVille qui raffine Structure

- Le service MoteurJeu

Nous nous sommes pas mal inspiré des spécifications données en correction du partiel. Nous avons notamment ajouté à ces spécifications des informations tels que le camp pour chaque service. Nous avons, par exemple, besoin de savoir à quel camp un villageois appartient, ce qui permettra aux joueurs de ne pas utiliser des villageois qui ne leurs appartiennent pas.

Nous avons en plus ajouté les informations sur les routes et les murailles au moteur de jeu, ainsi que les informations sur leurs positions.

Enfin nous avons ajouté une commande supplémentaire au jeu, qui est la commande FRAPPERMURAILLE, pour permettre à un villageois de frapper une muraille.

III- Organisation du projet

Une fois la spécification faite, nous pouvons nous attaquer au developpement du projet. Mais pour ne pas se perdre, nous devons adopter une bonne organisation du projet. Ce qui passe par une bonne façon de communiquer, et une bonne architecture pour le projet.

Pour faire cela, nous avons décidé de répartir les tâches de façon plus ou moins équitable en terme de temps de développement. Nous avons donc estimer à l'avance le temps de développement de chaque service, et nous avons réparti les tâches en fonction de leurs coût de développement. Nous avons donc décidé qu'Adel ferait les services **Villageois, Mine, et Hotel de Ville**, et Mohamed ferait les services **Muraille, Route et Moteur de Jeu**. Les services Terrain et Structure n'étant pas des services à part entière

Par ailleurs l'architecture du projet est la suivante :

-Un dossier src/ contenant le code source du programme

-Un dossier config/ contenant des fichiers de configuration pour créer les différents composants avec des paramètre prédefinies.

-Un dossier tests/ contenant les fichiers décrivant les tests à implémenter.

-Un dossier spec/ contenant tout les spécifications. On retrouvera pour chaque service, une spécification associée.

-Un dossier jars/ contenant les jars utilisée pour le projet, notamment une archive jar pour utiliser JUnit4, et une autre pour pouvoir utiliser un parser JSON.

De plus, pour pouvoir développer en parallèle de façon efficace, nous avons décidé mettre en place un dépôt **github**.

IV- Implémentations

Tout cela étant fait, nous pouvons donc commencer l'implémentation. Nous avons donc, pour chaque service, deux implémentations associées, une version normale, et une version buggée. Afin de faciliter cela, nous sommes passés par les interfaces Java pour décrire les services, qui se trouvent donc dans le package ***services***. Concernant les implémentations, elles se trouvent dans les packages ***implementations*** et ***implementationsBug***, pour respectivement la version normale et la version buggée. En plus de ces dossiers, on retrouvera un package ***tools*** contenant divers fonctionnalitées utilitaires, comme des classes permettant de représenter plus simplement la position d'une entité (VillageoisPosition pour Villageois par exemple). Ce genre de classe est notamment utile pour le moteur de jeu, afin d'avoir une représentation «propre» des positions des entités. On retrouve aussi dans

ce dossier les différentes énumérations, ou encore des fonctionnalités concernant la géométrie, notamment pour la collision entre deux entités.

V- Programmation par Contrat

Une fois l'implémentation faite, nous devons garantir une bonne fiabilité et une bonne robustesse pour notre programme, afin de répondre aux objectifs que l'on s'étaient fixés au départ du projet. Pour cela, on utilise la très utile programmation par contrat. On implémente cette méthode pour chaque service, excepté les service Terrain et Structure (qui ont surtout pour but d'éviter les redondances).

Pour cela, on décompose cette implémentation en 3 packages. Le package ***error***, qui contient les différents types d'exceptions provoquées par la programmation contrat (erreur de pre-condition, de post-condition, ou d'invariant). On retrouve aussi le package ***decorators***, qui contient, pour chaque service, une implémentation du design pattern Decorator, vu en cours. Enfin, on retrouve les classes qui implémentent les contrats pour chaque service. Dans cette classe, qui hérite de décorator, nous implémentons chaque operateur et constructeur, en utilisant le décorateur.

Pour un operateur/constructeur quelconque nommé «Operation», nous avons :

```
Operation()
{
    //Verification des invariant
    //Verification des pre conditions
    // appel de la fonction Operation avec le decorateur
    //Verification des post conditions
    //Verification des invariants
}
```

VI- Configuration

Afin de créer les instances des différents composants avec des paramètre prédéfinies, nous avons décidé de stocker ces paramètres dans des fichiers de configuration. Le format des fichiers de configuration choisi est JSON. Nous avons donc besoin d'un parser JSON afin de charger ces fichiers de configurations.

Pour implémenter cela, nous avons implémenter une factory afin de créer les composants de façon simple et centralisée, ainsi que des classes pour parser les configuration JSON. Cette factory se trouve dans le package ***factory***. Quant au parsing des configurations JSON, on les trouve dans le package ***presets***, dans lequel chaque service possède son fichier de parsing.

VII- Tests

Maintenant que nous avons implémenter toutes les fonctionnalités, ainsi que la programmation par contrat, nous pouvons nous attaquer aux tests.

Pour commencer, nous devons définir un jeu de tests assez large pour couvrir plus ou moins touts les cas d'utilisations. Pour cela, nous commençons par définir les tests positifs pour chaque service. On retrouve, pour chaque fonctionnalité d'un service, des tests avec plusieurs paramètres, en veillant notamment à tester les cas limites. Une fois les tests positifs définis pour une fonction, nous devons définir les tests négatifs, qui ne respectent pas les préconditions. La programmation par contrat nous permettra de vérifier cela.

L'organisation de chaque test est la suivante :

-Conditions initiales

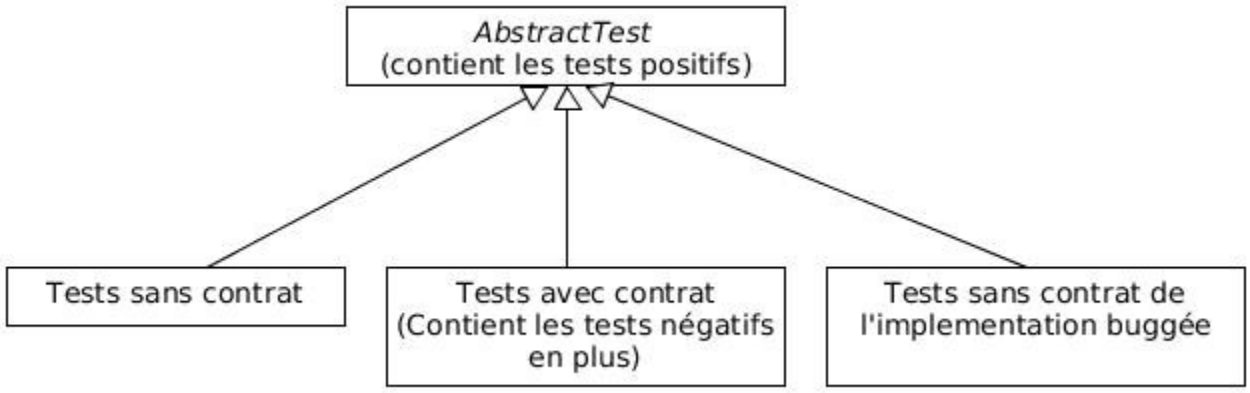
-Opérations

-Oracle

Ces tests se trouvent dans des fichiers se localisés dans le dossier ***tests***, à la racine du projet.

Une fois les tests défini, nous devons bien évidemment les implémenter. Pour cela, nous utilisons JUnit4. Nous avons deux versions de test pour chaque composant : Les tests avec contrat, et les test sans contrat. Pour éviter les redondances, nous avons, pour chaque service, une classe de tests abstraite, qui contient tout les tests positifs, une classe de test sans contrat qui hérite de la classe abstraite, et qui implémente simplement la méthode «before» en instanciant l'implémentation sans contrat, et nous avons une classe de test avec contrat, qui implémente la méthode before, et qui contient en plus les tests négatifs.

Voici un petit schéma illustrant cela :



Nous retrouverons donc les classes de tests avec contrat dans le package ***contractTests***, etles tests sans contrat dans le package ***nonContractTests***.

Une fois les tests implémentés, nous nous intéressons bien évidemment aux résultats. Au départ, nous obtenons des erreurs. Mais à l'aide de ce modèle de programmation claire, ça a été plutôt simple de corriger les erreurs.

Nous obtenons à la fin de ce projet, 48 tests réussis sur 48, pour les tests sans contrats :

[junit] Tests run: 48, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.313 sec

[junit] Tests run: 48, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.313 sec

Et 106 tests réussis sur 106 pour les tests avec contrats :

[junit] Tests run: 106, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.379 sec

[junit] Tests run: 106, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.379 sec

Et enfin, concernant l'implémentation buggée, nous obtenons, 20 tests réussis sur 48, avec un bon nombre de messages d'erreurs :

[junit] Tests run: 48, Failures: 28, Errors: 10, Skipped: 0, Time elapsed: 0.163 sec

[junit] Tests run: 48, Failures: 28, Errors: 10, Skipped: 0, Time elapsed: 0.163 sec

[junit]

[junit] Testcase: mine_Init_1 took 0.026 sec

[junit] FAILED

[junit] Test HotelVille Init :

[junit] junit.framework.AssertionFailedError: Test HotelVille Init :

[junit] at contractTests.AbstractHotelVilleTest.testMineInitWith(AbstractHotelVilleTest.java:41)

[junit] at contractTests.AbstractHotelVilleTest.mine_Init_1(AbstractHotelVilleTest.java:28)

[junit]

[junit] Testcase: hotelVille_Depot_1 took 0 sec

[junit] Testcase: hotelVille_Retrait_1 took 0 sec

[junit] Testcase: hotelVille_Retrait_2 took 0 sec

[junit] Testcase: hotelVille_Retrait_3 took 0 sec

[junit] Testcase: mine_Retrait4 took 0 sec

[junit] Testcase: mine_Init_2 took 0.001 sec

[junit] FAILED

Cette liste de messages d'erreur est évidemment incomplète. Nous ne les avons pas tous mis dû à leur nombre.

VIII- Comment l'utiliser

Pour utiliser notre projet, il faut se mettre à la racine du projet et lancer ant, avec en argument :

- run Afin de lancer le jeu en lui-même. Cependant nous n'avons pas implémenté d'interface, on ne peut dont pas y jouer.

- test Afin de lancer les tests sans contrats pour tout les composants

- ctest Afin de lancer les tests avec contrats pour tout les composants

- btest Afin de lancer les tests sans contrats pour la version buggée.

Attention : Pour que le programme fonctionne, il faut avoir Java 8 et ant version 1.9.3 d'installés.

IX- Conclusion

Pour conclure ce projet, nous pouvons dire que les objectifs initiaux ont été atteints, puisque nous avons défini un jeu de tests assez large, et que ces tests réussissent. Cela dit, nous n'avons pas implémenté d'interface, le jeu n'est donc pas jouable.

Ce projet a tout de même nécessité une quantité colossale de travail (avec quelques nuits blanches). Nous avons néanmoins apprécié ce projet, notamment lors de la partie test où nous avons constaté l'efficacité de ces techniques de programmations.

On pourrait maintenant envisager d'étendre ce moteur, afin de voir l'extensibilité de notre découpe, et de voir si de nouveaux composants s'intègrent bien à notre modèle. Enfin, on pourrait aussi envisager une interface graphique, pour tout simplement jouer au jeu.