Composant

Rapport de fin de projet

Lonely Runner

BAH Thierno

3408625

Sommaire

Introduction

Le but de ce projet est de créer le jeu Lode Runner en suivant le modèle de Design by Contract.
Il s'agit de voir une nouvelle manière de concevoir des programmes informatiques robustes.

En partant d'une spécification des services qui vont composer notre application, nous mettons en place des contrats qui vérifient que l'application reste dans un état cohérent.

Choix d'implémentations

Ma version de Lode Runner : *Lonely Runner* est un jeu qui se déroule pas à pas, c'est-à-dire que l'affichage ne se met à jour qu'une fois que le joueur a fait une action, il a donc le temps de réfléchir à ses prochaines actions.

Les utilitaires

Dans le package ***lonelyrunner.service.utils*** il y a un ensemble de classes et d'énumérations qui vont me permettre d'implémenter plus facilement les services.

Il y a par exemple la classe **Hole** qui va permettre de représenter un trou du point de vue de **Engine**. Un objet **Hole** contient les coordonnées d'un trou créé par le joueur et le temps depuis lequel ce trou a été créé (le temps ici est le nombre de pas).

On y retrouve aussi une classe très importante **SetCharItem** qui est une classe qui représente le contenu d'une case dans un **Environment**, c'est-à-dire est-ce qu'une case contient un **Item**, un **Player** ou un **Guard** ?

Enfin on y trouve **Cell** qui représente les différents types de case qu'il y a dans le jeu, **Move** qui représente les différentes actions réalisable par les **Character** (**Player** et **Guard**), **Status** qui représente l'état d'une partie (si elle est en cours, gagné ou perdu) et enfin la classe **Item** qui représente un objet, celui-ci peut avoir différente nature, pour l'instant il n'y a que des **Treasures** dans Lonely Runner.

Les écrans

Les écrans sont les services qui décrivent les objets sur lesquelles vont évoluer nos entités, le service **Screen** est le service de base qui va être ensuite raffiné par les services **Environment** et **EditableScreen**.

EditableScreen est le service qui nous permet de créer un niveau de jeu en modifiant les cases du jeu grâce à son opérateur *setNature.*

Pour créer un niveau je me sers d'un fichier texte où je donne d'abord la taille de l'écran sur la première ligne (hauteur et largeur) puis la position initiale du **Player** sur la deuxième ligne (la position en x puis en y). Sur la troisième ligne il y a la position initiale de chaque **Guard** séparé par des virgules, sur la quatrième ligne la position initiale de chaque **Treasures** et dans le reste du fichier il y a le niveau en lui même.

En lisant ce fichier j'ai toutes les informations nécessaires pour créer un écran, chaque caractère correspond à un type de case il me suffit donc de l'ajouter à mon EditableScreen.

```
50 11
5 2
15 2,6 9
10 2,49 2,1 9
--------------------------------------------------
--------------------------------------------------
===H=====---------------------------------HTTTTTTT
---H--------------------------------------H-------
---H-----TTTTT==H==TTTTT------------------H-------
---H------------H-------------------------H-------
---H-----=======H=======H======-----------H-------
---H--------------------H-----==-------H==--------
---H--------------------H-----===----H==----------
==================================================
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Exemple de fichier décrivant un niveau

```
Lives : 2 Score : 0

 *    G
===H=====                                 H-------
   H                                      H
   H     -----==H==-----                  H
   H            H                         H
   H     =======H=======H======           H
   H                    H     ==       H==
   H P    *    G        H     ===    H==         *
==================================================
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
>>>
```

Exemple de niveau généré avec ce fichier

Environement est le service qui va contenir toutes les entités et les objets du jeu qui apparaissent sur l'écran, il est initialisé grâce à un **EditableScreen** et contient un tableau de **SetCharItem.** Grâce à **Environment** on peut avoir le contenu de chaque case de l'écran et **Screen** nous permet d’avoir la nature de chaque case de l'écran.

Les entités

Character

Ce service est la classe de base qui va représenter toutes les entités qui vont être dans le jeu, c'est à partir de **Character** que les services **Guard** et **Player** vont être faits.

J'ai décidé de changer la spécification de **Character** pour permettre à celui-ci d'être sur la même case qu'un autre, cela va être utile pour permettre à un **Guard** d'être sur la même case qu'un **Player** tout en évitant que deux **Guard** soit sur la même case en ajoutant des post conditions sur les méthodes du service **Guard**.

De plus dans chacune des opérations de **Character** (c'est-à-dire les opérations qui s'occupe du déplacement des **Character**) j'ai ajouté une post condition qui fait en sorte que tout **Character** qui est en état de "*chute libre*" va forcément tomber et cela peut importe la commande qu'il essaie d'exécuter.

Donc quand un **Character** fait un *GoLeft* et qu'il est en train de tomber, il ne reste pas en l'air ou ne va pas à gauche dans les airs mais tombe forcément d'une case.

J'ai pensé que faire les choses de cette façon serait intéressante pour gérer les chutes étant donné que le jeu se déroule au fur et à mesure que le joueur rentre des commandes.

Player

Player est un service qui va inclure **Character** et qui va apporter de nouvelles commandes. Il est initialisé comme **Character** sauf qu'il a **Engine** en plus, ce dernier est le service du moteur du jeu.

Un **Player** est un **Character** qui peut creuser à gauche et à droite grâce à deux nouvelles opérations *digL* et *digR*.

J'ai ajouté des conditions sur ces deux opérations pour que le joueur puisse creuser quand il est au dessus d'une échelle et aussi pour l'empêcher de creuser une case si au dessus elle est déjà occupée par exemple il ne peut pas creuser en dessous d'un trésor, d'un guard d'une case **PLT** ou **MTL**.

J'ai rajouté une opération *doNeutral* qui va me permettre de ne rien faire à part subir la gravité dans les cas où le joueur rentre des commandes qui n'existe pas. Un des pièges du jeu qui fait avancer le jeu sans que le player n'ait bougé.

Un **Player** possède un opérateur *step* qui en fonction de la commande entré par le joueur (récupérable depuis l'observateur sur **Engine)** va lancer la méthode qui correspond à cette commande, un *Left* vas déclencher un *GoLeft*.

Guard

Guard est aussi un service qui va inclure **Character** et qui va aussi apporter de nouveaux opérateurs : *climbLeft*, *cimbRight.* **Guard** est initialisé avec une target qui sera un **Player**.

Guard va avoir des conditions en plus pour pouvoir utiliser ses méthodes de déplacements.

Les nouvelles post conditions vont être là pour empêcher un **Guard** d'être sur la même case qu'un autre **Guard**. De la même façon qu'un *GoLeft* au bord de l'écran ou dans une case non libre, quand un **Guard** va faire un *GoLeft* sur une case qui contient un autre **Guard** il ne se passera rien pour lui.

Les opérations *climbLeft* et *climbRight* ne se déclenche que dans le cas où un **Guard** est dans un trou. J'ai ajouté cette précondition pour éviter de faire des *climbLeft* ou *climbRight* à la place d'un *GoLeft* ou d'un *GoRight* alors qu'il n'est pas dans un trou.

L'opération *step* permet au **Guard** de faire la méthode qu'il faut selon la commande qu'il y a dans leur *getBehaviour.* Après avoir utilisé la méthode adéquate, j'ai décidé de mettre à jour la valeur de ce *getBehaviour* par rapport à la position de leur target.

C'est dans cette opération *step* que réside l'intelligence des **Guard**.

Le moteur du jeu et le jeu

Engine

Engine est le service qui représente le moteur de jeu. Il a plusieurs observateurs qui permettent d'observer l'état courant du jeu. Par rapport à l'énoncé, j'ai rajouté quelques observateurs comme *getNbLives*, *getScore* ou encore *guardInitPos*.

Avec ses nouveaux opérateurs je peux gérer le nombre de vie du joueur, son score et je peux obtenir la position initiale de chaque **Guard.** Ce qui va être utile lorsqu'il va falloir faire respawn un **Guard** qui vient d'être détruit parce qu'il était dans un trou qui s'est refermé.

Ce service a comme opérations *init*, *step*, *setCommand setNbLives* et *setScore*. Les opérateurs *seNbLives* et *setScore* permettent de gérer la vie du joueur quand elle diminue et de gérer son score quand il ramasse un trésor.

L'opération *step* est le coeur du jeu. C'est dans cette opérations que le **Player** va agir selon la commande que le joueur rentre et qui est mis à jour grâce à *setCommand.*

C'est aussi ici que les gardes vont exécuter leur opération *step*, que je vérifie si un joueur a ramassé un trésors, s'il est dans un trou qui se referme, s'il est sur la même case qu'un garde, si la partie est gagné ou perdu, si les gardes ont déplacé des trésors ou pas, si les trous qu'il y a dans l'observateur *getHoles* vont être reboucher ou pas.

J'ai choisi de faire en sorte que tous les **Guard** puissent déplacer un trésor si jamais ils sont sur un trésor et que la prochaine case sur laquelle ils vont être est valide pour contenir un trésor, si la case sur laquelle ils vont être ne peut pas contenir un trésor alors ils le laissent dans la case d'avant et continuent leur chemin.

Le jeu évolue au fil des *step*.

Lonely Runner Main

LonelyRunner et **LonelyRunnerMain** sont les classes qui vont me permettre d'initialiser un Engine et de lancer le jeu et de recueillir les commandes du joueur entre chaque step.

Elles sont chargées de créer l'**EditableScreen** qui va être utilisé pour initialiser *Engine* ainsi que tous les autres paramètres que Engine a besoin pour être initialiser.

Elles obtiennent ces informations en lisant le fichier texte qui décrit un niveau. Elles sont aussi charger de changer de niveau quand le joueur en a fini un et de relancer un niveau quand le joueur a perdu une vie.

Pour jouer au jeu il faut exécuter la classe **LonelyRunnerMain**.

Tests

Les test se trouvent dans le package ***lonelyrunner.test.***

Je vais présenter quelque tests Junit que j'ai trouvé intéressant à implémenter et qui permettent de valider quelques opérations importantes dans les services.

Les objets utilisés dans les tests ont été créés dans une méthode *BeforeTest*.

Test Dig/Fill Positif

```
@Test
public void testDigPositif() {
    editscreen.init(10, 15);
    Cell changeTo = Cell.PLT;
    editscreen.setNature(1, 1, changeTo);
    env.init(editscreen);
    ArrayList<Cell> avants = new ArrayList<>();
    for (int x = 0; x < env.getWidth(); x++) {
        for (int y = 0; y < env.getHeight(); y++) {
            avants.add(env.getCellNature(x, y));
        }
    }
    Cell beforeDig = env.getCellNature(1, 1);
    assertTrue(beforeDig == Cell.PLT);

    this.testInvariant();
    env.dig(1, 1);
    this.testInvariant();
```

```
    Cell afterDig = env.getCellNature(1, 1);
    assertTrue(afterDig == Cell.HOL);
    ArrayList<Cell> after = new ArrayList<>();
    for (int x = 0; x < env.getWidth(); x++) {
        for (int y = 0; y < env.getHeight(); y++) {
            after.add(env.getCellNature(x, y));
        }
    }
    int v = 0;
    for (int x = 0; x < env.getWidth(); x++) {
        for (int y = 0; y < env.getHeight(); y++) {
            if (x == 1 && y == 1) {
                v++;
                continue;
            }
            assertTrue(avants.get(v).compareTo(after.get(v)) == 0);
            v++;
        }
    }
```

Le test ci-dessus concerne le service **Environment** ici je veux tester si l'opération *dig* de **Screen** fonctionne. Je ne pouvais pas la tester dans **Screen** car celle-ci ne peut pas modifier la nature des cases et qu'à l'initialisation d'un **Screen** il n'y avait que des cases **MTL** et **EMP**, des cases qui ne peuvent pas être creusés.

Il a fallu donc avoir **EditableScreen** qui grâce à la méthode *setNature* peut introduire des cases **PLT** creusables dans notre screen.

J'ai choisi de présenter ce test depuis **Environment** car **Environment** est ce qui est utilisé par **Engine** et ce sur quoi les entités vont évoluer et parfois agir (*digL* et *digR* font appel à cette opération *dig* il faut donc voir si elle fonctionne comme il faut).

Dans ce test je crée mon **EditableScreen** avec une case **PLT** en (1,1) ensuite je me sers de cet **EditableScreen** pour initialiser mon **Environment** et après avoir vérifier les invariants (grâce à *testInvariant*) je fais appel a *dig* sur la case (1,1) et je vérifie ensuite les invariants et que cette case est bien devenu un **HOL** et qu'aucune autre case n'a été modifié.

Pour tester le cas positif de l'opération *fill* c'est exactement le même test il faut juste mettre un **HOL** et vérifier qu'il devient un **PLT** après l'opération sur cette case.

Il y a les même tests dans les tests de **EditableScreen**.

Test Dig/Fill Négatif

```
@Test
public void testFillNegatif() {
    Cell mtl, hdr, emp, lad, plt;
    mtl = Cell.MTL;
    hdr = Cell.HDR;
    emp = Cell.EMP;
    lad = Cell.LAD;
    plt = Cell.PLT;
    editscreen.init(10, 15);
    editscreen.setNature(0, 1, mtl);
    editscreen.setNature(1, 1, hdr);
    editscreen.setNature(2, 1, emp);
    editscreen.setNature(3, 1, lad);
    editscreen.setNature(4, 1, plt);
    env.init(editscreen);
    testInvariant();
    boolean caught = false;
    try {
        env.fill(0, 1);
    } catch (ContractError e) {
        caught = true;
    }
    assertTrue(caught);
    caught = false;
    try {
        env.fill(1, 1);
    } catch (ContractError e) {
        caught = true;
    }
    assertTrue(caught);
    caught = false;
    try {
        env.fill(2, 1);
    } catch (ContractError e) {
        caught = true;
    }
    assertTrue(caught);
    caught = false;
    try {
        env.fill(3, 1);
    } catch (ContractError e) {
        caught = true;
    }
    assertTrue(caught);
    caught = false;
    try {
        env.fill(4, 1);
    } catch (ContractError e) {
        caught = true;
    }
    assertTrue(caught);
```

Le but de ce test est de vérifier que faire un *fill* sur une case qui n'est pas **HOL** déclenche bien une erreur de contrat, je me sers du booléen *caught* pour vérifier que toutes les erreurs ont bien été déclenchées.

Je présente le test de *fill* négatif mais celui de *dig* est le symétrique de celui ci, faire un *dig* sur une case autre qu'un **PLT** déclenche les même erreurs.

Test GoRight/GoLeft/GoUp/GoDown sur Guard Player et Character

J'ai choisi de présenter les tests de déplacement du **Guard** dans le rapport car ils sont extrêmements similaire à ceux de **Character** et de **Player**.

La seule différence est qu'il y a un cas en plus à traiter pour le **Guard** en effet il a obstacle en plus de **Character** et **Player.** Un **Guard** ne peut pas aller sur une case déjà occupée par un autre **Guard.** de la même façon qu'un **Character** ou un **Player** ne peut pas aller dans une case **MTL** ou **PLT** par exemple.

```
@Test
public void testGoLeftEdge() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    env.init(editscreen);
    target.init(env, 9, 2);
    guard.init(env, 0, 2,target);

    testInvariant();
    int xbefore = guard.getWdt();
    int ybefore = guard.getHgt();

    testInvariant();
    guard.goLeft();
    testInvariant();
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore == guard.getHgt());
}
```

Voici un test qui se déroule au bord gauche de l'écran, on initialise le **Guard** au bord de l'écran et on fait un *GoLeft*, on s'attend à ce que sa position n'ait pas changé.

De même les tests *GoRightEdge*, *GoUpEdge* initialise un Guard au bord droit ou au sommet de l'écran et applique l'opération qui correspond puis vérifie que la position n'a pas changée.

Il y a tout de même une petite différence car pour pouvoir faire un *GoUp* il faut être sur une échelle, on vas donc initialiser le **Guard** sur une case vide le déplacer manuellement avec des *GoLeft/Right* sur une échelle et faire un *GoUp*.

Je n'ai pas testé *GoDownEdge* car un écran a toujours une rangée de case **MTL** en bas, on ne peut jamais se trouver dans le cas ou on doit faire un GoDown sur le bord en bas de l'écran.

```
@Test
public void testGoLeftObstacle() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    editscreen.setNature(0, 2, Cell.PLT);
    env.init(editscreen);
    target.init(env, 9, 2);
    guard.init(env, 1, 2,target);
    int xbefore = guard.getWdt();
    int ybefore = guard.getHgt();
    testInvariant();
    guard.goLeft();
    testInvariant();
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore == guard.getHgt());
    editscreen.setNature(0, 2, Cell.MTL);
    env.init(editscreen);
    target.init(env, 9, 2);
    guard.init(env, 1, 2,target);
    xbefore = guard.getWdt();
    ybefore = guard.getHgt();
    testInvariant();
    guard.goLeft();
    testInvariant();
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore == guard.getHgt());
```

```
    editscreen.setNature(0, 2, Cell.EMP);
    env.init(editscreen);
    GuardContract g2 = new GuardContract(new GuardImpl());

    target.init(env, 9, 2);
    g2.init(env, 0, 2,target);
    guard.init(env, 1, 2,target);

    xbefore = guard.getWdt();
    ybefore = guard.getHgt();
    testInvariant();
    guard.goLeft();
    testInvariant();
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore == guard.getHgt());
```

Ce test est un peu similaire à celui des edges sauf que ici je teste le déplacement dans un obstacle, pour le **Guard** un obstacle est une case **MTL**, **PLT** ou un autre **Guard**.

Ce test est semblable pour les opérations *GoUp*, *GoDown* et *GoRight* et par rapport à **Character** et **Player** la seule chose qui change c'est que Character et Player considère uniquement les cases **PLT** et **MTL** comme obstacle.

On s'attend à ce qu'une opération de déplacement dans un obstacle ne change pas la position de l'entité qui l'appelle.

```
@Test
public void testGoLeftFalling() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    env.init(editscreen);
    target.init(env, 9, 2);
    guard.init(env, 5, 5,target);

    int xbefore = guard.getWdt();
    int ybefore = guard.getHgt();
    testInvariant();
    guard.goLeft();
    testInvariant();
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore - 1 == guard.getHgt());
}
```

Ce test *GoLeftFalling* est là pour vérifier que faire une opération pendant une chute libre nous fait forcément tomber, j'initialise mon **Guard** sur une case vide sans support en bas et je fait appelle à une opération, l'opération peut être un *GoLeft,* un *GoRight*, un *GoUp* ou un *GoDown* on s'attend à ce que l'entité tombe toujours.

Pour le **Player** faire un *digL* ou un *digR* en chute libre fait que le joueur tombe aussi, ce test est pareil pour **Player** et **Character**.

```
@Test
public void testGoUpNormal() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 1; x < editscreen.getHeight(); x++) {
        editscreen.setNature(1, x, Cell.LAD);
    }
    env.init(editscreen);
    target.init(env, 9, 1);
    guard.init(env, 0, 1,target);
    testInvariant();
    guard.goRight();
    testInvariant();
    int xbefore = guard.getWdt();
    int ybefore = guard.getHgt();
    testInvariant();
    int j = 0;
    for (int x = 2; x < 9; x++) {
        guard.goUp();
        testInvariant();
        j++;
    }
    assertTrue(xbefore == guard.getWdt());
    assertTrue(ybefore + j == guard.getHgt());
}
```

Tous les tests *Normal* sont des tests qui sont fait dans le cas où l'opération se déroule sans problèmes, c'est à dire pas d'obstacle, pas de bord et ne pas être en chute libre.

Ici j'ai mit le test pour *GoUp* de **Guard**, mais ceux de *GoDown*, *GoRight*, et *GoLeft* sont similaire pour **Guard**, **Character** et **Player**.

J'initialise le **Guard** je lui fais prendre une échelle et je fait 5 *GoUp* puis je vérifie que sa hauteur a bien changer de 5.

```java
@Test
public void testCharacterOnCharacter() {
    editscreen.init(10, 15);

    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    editscreen.setNature(1, 2, Cell.PLT);
    editscreen.setNature(3, 2, Cell.PLT);

    env.init(editscreen);
    CharacterContract c2 = new CharacterContract(new CharacterImpl());
    c2.init(env, 2, 2);
    character.init(env, 1, 3);
    testInvariant();
    character.goRight();

    testInvariant();
    assertTrue(c2.getWdt() == character.getWdt());
    assertTrue(c2.getHgt() + 1 == character.getHgt());

    testInvariant();
    character.goRight();
    testInvariant();
    assertTrue(character.getWdt() == 3);
    assertTrue(character.getHgt() == 3);
}
```

Ce test est la pour vérifier que tout **Character** peut marcher sur un autre **Character** sans tomber, un **Character** est une surface valide sur laquelle un autre **Character** peut faire des deplacements.

Pour les tests de **Guard** il y avait aussi les tests de *climbLeft* et *climbRight* positifs et négatifs qui vérifient qu'un **Guard** doit être dans un **HOL** avant de faire appelle a ses methodes et que l'endroit où le **Guard** sera est bien celui qui correspond.

Il y aussi un test *stuckInHole* dans *GuardTest* qui vérifie qu'un **Guard** reste dans un trou et ne tombe pas à travers contrairement au **Player** ou au **Character**.

Test Player Dig/Step

Dans la classe de test *PlayerTest* il y a plein de cas où je teste les opérations *dig* de **Player**, j'ai choisi dans ce rapport de montrer les tests dans le cas où on essaie de *dig* en dessous d'une case qui contient un *Item* ou un **Character** (le test qui vérifie qu'il est impossible de dig en dessous d'un **MTL** ou d'un **PLT** est similaire).

Je vais juste présenter *DigL* car *DigR* est symétrique.

```
@Test
public void testDigLUnderItemOrCharacter() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    env.init(editscreen);
    PlayerContract p2 = new PlayerContract(new PlayerImpl());
    ArrayList<Couple<Integer, Integer>> treasures = new ArrayList<C
    treasures.add(new Couple<Integer,Integer>(13,2));
    engine.init(editscreen, new Couple<Integer, Integer>(14, 2),
            new ArrayList<Couple<Integer, Integer>>(), treasures);
    env = engine.getEnvironment();

    player.init(env, 14, 2, engine);
    Cell beforeDig = env.getCellNature(13, 1);
    int xbefore = player.getWdt();
    int ybefore = player.getHgt();
    testInvariant();
    player.digR();
    testInvariant();
    Cell afterDig = env.getCellNature(13, 1);
    assertTrue(afterDig == beforeDig);
    assertTrue(xbefore == player.getWdt());
    assertTrue(ybefore == player.getHgt());
```

```
    env.init(editscreen);
    engine.init(editscreen, new Couple<Integer, Integer>(14, 2),
            new ArrayList<Couple<Integer, Integer>>(), treasures);
    env = engine.getEnvironment();
    player.init(env, 14, 2, engine);
    p2.init(env, 13, 2);
    env.getCellContent(13, 2).addCar(p2);
    beforeDig = env.getCellNature(13, 1);
    xbefore = player.getWdt();
    ybefore = player.getHgt();
    testInvariant();
    player.digL();
    testInvariant();
    afterDig = env.getCellNature(13, 1);
    assertTrue(afterDig == beforeDig);
    assertTrue(xbefore == player.getWdt());
    assertTrue(ybefore == player.getHgt());
```

Dans un premier temps j'initialise un **Player** et j'ajoute un *Item* a la case juste à sa gauche, ensuite je fais un *digL* et je m'attends à ce que la case en dessous de l'*Item* n'ai pas changé.

Ensuite je crée un deuxième **Player** juste sur la case à gauche de mon premier **Player** et je fait un *digL* ici aussi je m'attends à ce que rien n'ait changé.

```java
@Test
public void testStep() {

    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);
    }
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);
    }
    editscreen.setNature(0, 2, Cell.LAD);
    ArrayList<Couple<Integer, Integer>> treasures = new ArrayList<Couple<Integer, Integer>>();
    treasures.add(new Couple<Integer,Integer>(9,2));
    engine.init(editscreen, new Couple<Integer, Integer>(1, 2),
            new ArrayList<Couple<Integer, Integer>>(), treasures);
    env = engine.getEnvironment();
    player.init(env, 1, 2, engine);
    testInvariant();
    engine.setCommand(Move.LEFT);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(0==player.getWdt());
    assertTrue(2==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.LAD);
    engine.setCommand(Move.UP);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(0==player.getWdt());
    assertTrue(3==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.EMP);
    engine.setCommand(Move.NEUTRAL);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(0==player.getWdt());
    assertTrue(3==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.EMP);
    engine.setCommand(Move.DOWN);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(0==player.getWdt());
    assertTrue(2==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.LAD);
    engine.setCommand(Move.RIGHT);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(1==player.getWdt());
    assertTrue(2==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.EMP);
    engine.setCommand(Move.RIGHT);
    testInvariant();
    player.step();
    testInvariant();
    assertTrue(2==player.getWdt());
    assertTrue(2==player.getHgt());
    assertTrue(player.getEnvi().getCellNature(player.getWdt(), player.getHgt())==Cell.EMP);
```

Dans ce test je vérifie que l'opération qui est appelé dans le step de **Player** correspond bien à la commande présente dans le *getNextCommand* de l'**Engine** avec lequel le **Player** est initialisé, et je test pour chaque commande.

Je fais donc faire un petit chemin au **Player** et je vérifie bien entre chaque *step* qu'il est sur la case sur laquelle je veux qu'il soit.

Lors de ce test pour les commandes **DigL** et **DigR** j'ai pu découvrir un bug dans le contrat car je faisais des clones qui creusait un peu n'importe comment, cela m'a permis de corriger ce contrat au moins sur ce point la.

Test Guard Step

```java
@Test
public void testStepWhileInHOL2() {
    editscreen.init(10, 15);
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 0, Cell.MTL);}
    for (int x = 0; x < editscreen.getWidth(); x++) {
        editscreen.setNature(x, 1, Cell.PLT);}
    editscreen.setNature(4, 1, Cell.HOL);

    env.init(editscreen);
    target.init(env, 2, 2);
    guard.init(env, 5, 2,target);

    assertTrue(guard.getBehaviour()==Move.NEUTRAL) ;
    guard.step();
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.HOL);
    assertTrue(guard.getBehaviour()==Move.LEFT);
    guard.step();
    assertTrue(guard.getWdt()==3 && guard.getHgt()==2);
    assertTrue(guard.getEnvi().getCellNature(guard.getWdt(), guard.getHgt())==Cell.EMP);
    guard.step();
    assertTrue(guard.getBehaviour()==Move.NEUTRAL) ;
    assertTrue(guard.getWdt()==target.getWdt() && guard.getHgt()==target.getHgt());
```

Dans ce test je vérifie que le **Guard** suit bien le scénario qu'il devrait suivre et que à chaque *step* la commande qu'il exécute est celle qui le rapproche de la target, sur le chemin j'ai mit un **HOL** pour voir que le **Guard** tombe bien dedans et reste coincé dedans pour 5 *step* avant de faire un *climbLeft* et de sortir.

Test Engine Initialisation

Dans le test *testInitialisationPositif* de *EngineTest*, je vérifie que les préconditions et les post conditions de l'opération init de **Engine** sont valider. Comme il y a des préconditions on peut faire plusieurs tests négatifs pour vérifier que si on ne respecte pas ses préconditions on déclenche des erreurs de contrats.

Tous les *testInitialisationNegatif* vérifient une de ces préconditions non valides, par exemple : fournir un **EditableScreen** non valide ou encore initialiser un **Player** ou un **Guard** sur une case non vide.

Difficultés

J'ai rencontré beaucoup de difficultés notamment pour écrire dans un contrat la vérifications de certaines post conditions.

Pour les gardes j'aurais aimé que le choix du prochain *behaviour* soit un invariant du service mais je n'ai pas réussi à définir des conditions assez précise pour choisir un tel ou tel *behaviour*, j'ai été contraint d'enlever énormément de choses et j'ai abouti à une IA très simplifiée.

Il y a beaucoup de choix dans les services qui mériterait plus de réflexions, et pour la forme il y a aussi toute la partie où j'utilise des clones pour simuler les déplacements théoriques je n'étais pas sur de la façon dont rédiger ces post conditions.

Trouver des tests intéressants était difficile je n'ai pas encore reussi a trouver de scénario intéressant pour tester l'opération *step* de **Engine**, et niveau implémentation gérer les interactions entre les objets étaient un peu délicates (pour éviter de modifier un environnement qu'on ne devait pas modifier par exemple).

Conclusion

J'ai reussi a faire une version jouable du jeu que je voulais faire. Il reste encore des choses à corriger et des bugs à trouver en ajoutant plus de test et en réfléchissant mieux sur les contrats et la spécification des services.

Il est difficile de bien spécifier des services, le Design by Contract est une approche un peu fastidieuse mais avec de bonne garantie sur la robustesse de l'application.

L'implémentation des contrats ressemble fortement à l'écriture de l'implémentation et beaucoup de bug peuvent être découvert grâce au test MBT.

C'est un moyen de concevoir des programmes très intéressants et je pense utiliser certaines de ses techniques dans de futur projet.

Annexe

- **Les descriptions formelles de tous les services sont dans le dossier Specification.**
- **Les services sont tous dans le package *lonelyrunner.service.* (mais ils ont au propre dans le dossier Specification).**
- **Les contrats sont tous dans le package *lonelyrunner.contract.***
- **Les implémentations sont toutes dans le package *lonelyrunner.impl.***
- **Les 115 tests sont tous dans le package *lonelyrunner.test.***
- **Les main LonelyRunnerMain à exécuter se trouve dans le package *lonelyrunner.main.***