CPS
RAPPORT GÉNÉRAL DU PROJET

8 mai 2019

GERDAY Nathan

Table des matières

Introduction

Ce document est le rapport pour le projet de CPS 2019, présentant notamment les points pertinents du projet. Le rapport de spécification formelle ainsi qui la description formelle des tests MBT ont été placés dans des fichiers séparés afin d'améliorer la lisibilité. L'ensemble de ces fichiers est trouvable à la racine du projet ainsi que dans le mail en pièce jointe.

Le projet a été réalisé en Java 1.8 pour l'implémentation ainsi que les contrats. Les tests ont eux été réalisés en JUnit 4.
Le projet a été testé sur la version 18.10 de Ubuntu, il devrait cependant être compatible avec la majorité des systèmes ayant une version de Java similaire et Ant d'installé.

Manuel

Installation

Les pré-requis pour exécuter le projet sont :

— Une version de Java proche de 1.8

— Ant

Le jar de JUnit étant inclus avec l'ensemble du projet, il ne devrait pas y avoir d'installation supplémentaire à effectuer.

Utilisation

La compilation sera faite automatiquement par *Ant* à l'exécution d'une cible, il n'est donc pas nécessaire de s'en occuper manuellement.

Pour exécuter le projet, on peut utiliser la cible "run" de Ant :

```
$ ant run
```

Le jeu se lance alors dans le terminal, avec une fenêtre qui s'ouvre pour les entrées claviers (Voir la prochaine section "Comment jouer", pour plus de détails sur le gameplay). Pour terminer l'exécution, il faut soit finir la partie, soit envoyer le signal Ctrl+C.

Pour tester le projet grâce aux tests JUnit, on peut utiliser la cible "test" de *Ant* :

```
$ ant test
```

Cette commande a alors deux effets :

1. Elle génère un fichier "report<NomDeClasse>.txt" contenant toutes les informations liées au test de cette classe. Ces fichiers se situent à la racine du projet.
2. Elle affiche directement dans le terminal un résumé rapide des résultats des tests indiquant le nombre d'échecs et d'erreurs (au moment de l'écriture de ce rapport, l'ensemble des tests fonctionnent).

Comment jouer ?

Au lancement du jeu, une petite fenêtre sur fond blanc s'ouvre et prend le focus. C'est dans cette fenêtre qu'il faut entrer les inputs claviers qui auront une influence sur le jeu. Le jeu s'affiche dans le terminal après chaque *Step* du moteur.

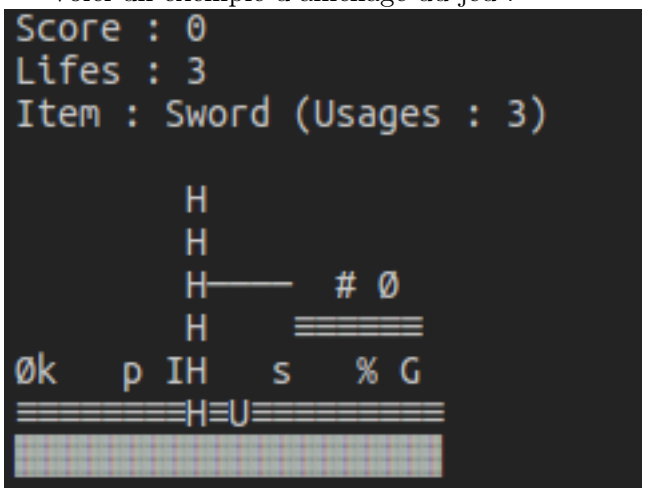
Liste des commandes :

- **Z** : Se déplacer vers le haut
- **S** : Se déplacer vers le bas
- **Q** : Se déplacer vers la gauche
- **D** : Se déplacer vers la droite
- **A** : Creuser à gauche
- **E** : Creuser à droite
- **SPACE** : Utiliser l'item actuellement équipé

Comprendre l'affichage :

Les représentations graphiques de chaque éléments ont été pensées pour essayer d'être rapidement identifiables malgré le fait que l'affichage soit fait via des caractères ASCII dans un terminal.

Voici un exemple d'affichage du jeu :



On peut tout d'abord remarquer les 3 lignes en haut qui indiquent le score, le nombre de vies restantes ainsi que l'item actuellement équipé et son nombre d'utilisations restantes.

Voici maintenant une explication du contenu de chaque case :

— Le symbole **p** correspond au joueur que l'on contrôle. Il s'affiche **p** lorsqu'on regarde à droite et **q** lorsqu'on regarde à gauche.
— Le symbole **G** correspond à un garde. Lorsque le garde ramasse un trésor et se déplace avec, son symbole devient alors **T**.
— Le symbole **Ø** correspond à un trésor permettant au joueur d'incrémenter son score et étant ramassable par des gardes.
— La ligne de carrés blancs tout en bas, correspond aux cases *MTL* qui ne pourront pas changer.
— La ligne de carrés juste au dessus (les carrés composés de 3 lignes), correspondent aux cases *PLT* que le joueur peut creuser. Elle peut également être une case *TRP* dans lequel le joueur tombera en marchant dessus.
— Le symbole **U** correspond à une case *HOL* dans lesquels les gardes resteront coincés quelques tours.
— Le symbole **H** correspond à une case *LAD* sur laquelle les personnages peuvent grimper.
— Le symbole **-** de trait horizontal correspond à une case *HDR* sur lequel le joueur peut se déplacer à gauche et à droite ou se laisser tomber.
— Le symbole # correspond à une case *NGU* dans lesquels les gardes ne peuvent pas passer.
— Le symbole **%** correspond à une case *NPL* dans lesquels le joueur ne peut pas passer.
— Le symbole **I** correspond à une case *DOR* qui sont les portes dans lesquels aucun personnage ne peut passer tant qu'elles n'ont pas été ouvertes par une clé.
— Les symboles **k**, **f**, **s**, **g** correspondent aux objets ramassables et équipables par le joueur. Elles correspondent chacune à la première lettre minuscule du mot anglais correspondant à l'objet : **(k)**ey, **(f)**lash, **(s)**word, **(g)**un.

Déroulement du jeu

Le but du jeu est de ramasser tout les trésors sans se faire attraper par les gardes. Lorsqu'on tout les trésors sont ramassés, on passe au niveau suivant s'il existe, sinon on dit que la partie est gagnée. Lorsque le joueur meurt, (s'il se fait attraper ou qu'il est dans une case trou au moment où elle se remplie), son nombre de vie diminue et le niveau recommence au début. S'il n'a plus de vies restantes, on déclare la partie perdue et on termine le programme.

Création du niveau

Toute la gestion de la création des niveaux se fait dans la classe principale *LodeRunner*. On peut décider de la nature des cases, de leur contenu, de la position des joueurs et des gardes, etc...

Description du projet

Travail réalisé et extensions

L'ensemble du sujet d'examen a été réalisé avec notamment la gestion des écrans, le contact entre un garde et un joueur, un maintien d'un nombre de vies et du score, ainsi que le fait que les gardes puissent ramasser des trésors et que leur IA ait été un peu améliorée pour éviter de se bloquer dans des situations évidentes.
Le choix a été fait de ne pas faire d'éditeur interactif car cela aurait été peu agréable à utiliser dans le terminal et il n'aurait pas eu d'éléments très intéressant à spécifier en lui même, la classe *EditableScreen* étant déjà existante et spécifiée.

Quelques extensions ont également été ajoutées :

1. Le fait que le joueur regarde dans une direction (à droite ou à gauche). On change la direction dans laquelle on regarde lorsque l'on se déplace ou que l'on creuse dans cette direction.
2. Des objets ramassables permettant le combat :
   — Un **Pistolet** : Il permet de tirer en ligne droite en face du joueur et de tuer tout les gardes jusqu'à un obstacle ou au bord de l'écran.
   — Une **Épée** : Elle permet de réaliser une attaque rapprochée qui tue les gardes à la même hauteur que le joueur et qui se trouvent à 2 cases ou moins.
   — Une **Bombe Aveuglante** : Elle permet d'immobiliser tout les gardes du niveau pendant un certain temps.
3. Une **clé** permettant d'ouvrir des portes
4. Le fait que chaque objet ramassable possède un nombre d'utilisation donné.
5. De nouveaux types de cases, avec différentes propriétés :
   — Les **Portes** : Elles bloquent le passage à tous les personnages jusqu'à ce que le joueur l'ouvre à l'aide d'une clé.
   — Les **Pieges** : Ces cases disparaissent lorsque le joueur marche dessus.
   — Les **Membranes Anti-Joueur** : Qui laissent passer tous les personnages sauf les joueurs.
   — Les **Membranes Anti-Gardes** : Qui laissent passer tous les personnages sauf les gardes.

Architecture générale du code

L'objectif de cette partie est de survoler les différents packages et classes du projet afin de pouvoir s'orienter dedans. L'ensemble du code du projet se trouve dans le dossier "*src*" à la racine du projet.

Voici la répartition des classes / packages et leurs raisons d'êtres :

1. **services** : Contient l'ensemble des interfaces de chaque services, avec une spécification intermédiaire en commentaire, permettant de faire le lien entre le spécification et les contrats.
2. **decorators** : Contient les décorateurs permettant de faire les contrats.
3. **contracts** :
   — Contient l'ensemble des contrats pour chaque services (préconditions, postconditions, invariants).
   — Contient les classes permettant la gestion d'erreur en cas de rupture de contrats (*Contractor* et *ContractError*) .
   — Contient une classe avec une unique méthode statique représentant l'opérateur d'implication et permettant d'améliorer la lisibilité des contrats (*Checker*).
4. **impl** : Contient toutes les implémentations réelles et fonctionnelles des services ainsi que les implémentations avec bugs (les bugs sont marqués par un commentaire "//BUG").
5. **test** : Contient toutes les classes permettant de réaliser les tests JUnit sur le projet.
6. **data** : Contient différentes énumérations et classes de représentation de données étant trop simple pour nécessiter une spécification qui n'apporterait pas grand chose. Ces classes ne contiennent, en général, aucune opération ou alors une opération très simple (*exemple : Le fait d'incrementer le temps d'existence sur un trou*).
7. **utils** :
   — *CommandManager* : Permet de gérer les inputs utilisateurs dans une fenêtre spéciale indépendante du terminal. Le moteur pourra utiliser cette classe pour récupérer la dernière commande faites par l'utilisateur.
   — *Factory* : Contient des méthodes statiques permettant d'instancier chaque services. Elle permet de regrouper toutes les instanciations en un seul endroit et simplifie le fait de choisir si on souhaite créer des contrats, des implémentations fonctionnelles ou des implémentations avec bugs.
   — *Util* : Contient différentes méthodes statiques pouvant être utilisés dans tout le projet et permettant d'améliorer la lisibilité. Ces méthodes sont pincipalemement au contenu des cases de l'environnement et permettent de récupérer ou de vérifier facilement l'existence d'éléments précis.
8. **main** : Contient la classe principale *LodeRunner* permettant la création de différents niveaux, la création du moteur et le fait de faire tourner le moteur jusqu'à la fin de la partie.

Choix et difficultés intéressantes

Au niveau de la spécification

Le Step du Joueur Le *Step* du joueur consiste à appliquer un comportement, en fonction de l'utilisateur. Il est donc différent des autres opérations car il ne fait rien lui-même, il se charge simplement d'appeler la "bonne" opération de player. Cela donne donc une implémentaiton très simple, si on tombe, on fait *GoDown* et sinon on suit l'indication du joueur. Cela pose cependant un problème au niveau des contrats. En effet, étant donné que le contrat va appeler la méthode *step()* sur son délégué qui est une implémentation. L'implémentation va ensuite appeler sa propre méthode *GoDown()*. On remarque alors, qu'on ne passe pas par les contrats de *GoDown()*. Afin de régler ce problème ; il faut donc faire une capture du délégué avant de faire l'opération *step()* dans le contrat et créer un nouveau PlayerContract sur cette copie du délégué. On appelle ensuite *step()* sur le délégué et *GoDown()* sur la copie (dans le cas où on veut tester d'aller en bas) et on compare ensuite que les 2 instances sont bien arrivées au même endroit. Cela nous permet au final de bien avoir vérifié les pré et post conditions de GoDown() et de vérifier également que c'est bien *GoDown()* qui a été faite et pas une autre méthode.

Ramasser un objet Lorsque le joueur est sur une case contenant un objet ramassable, le moteur va appeler la méthode *pickupItem(ItemType i)* du joueur. Si c'est le même item, que l'item actuellement équipé on augmente simplement le nombre d'utilisations. Si c'est un item différent, on perd l'item actuellement équipé et on équipe le nouveau. Le nombre d'utilisation pouvant varier selon les items il faut donc pour pouvoir être complet, définir pour chaque type d'objet, une post-conditions si l'objet est déjà équipé et une post-condition s'il ne l'est pas encore.

Utiliser un item Afin d'éviter le problème précédent dans la gestion du *Step*, il a été choisi de faire en sorte d'avoir une méthode unique *UseItem()* qui appliquera le bon résultat en fonction de l'item équipé. Cela implique donc de spécifier de nombreuses post-conditions au sein de cette méthode. Ce n'est cependant pas un réel problème car ces post-conditions sont assez indépendantes les unes des autres étant donné qu'il suffit de déterminer l'item équipé.
La majorité des spécifications des objets ne posent pas de problème majeur. Pour la clé, on regarde si la case auquel le joueur fait face est une porte et si oui, on dit qu'elle se transforme en case vide. Pour la bombe aveuglante, on vérifie que tout les gardes sont bien paralysés pour un temps donné. Pour l'épée, la portée est fixe donc il suffit de vérifier que les gardes à portée sont bien morts (et donc qu'ils sont bien retournés à leur position initiale).
Le pistolet a cependant une propriété qui pose un problème supplémentaire de spécification. En effet, on ne connaît pas à l'avance sa portée étant donnée qu'il touche tout les gardes dans la direction qu'il regarde jusqu'au bord de l'écran ou jusqu'à un obstacle. La solution choisie est de définir dans la spécification une méthode indiquant si un obstacle est présent entre 2 colonnes sur une hauteur particulière. Pour cela, on fait l'union de toutes les cases entres ces 2 colonnes et on regarde si cette union contient l'une des cases que l'on considère comme un obstacle. On sait alors que s'il n'y a pas d'obstacles entre un garde et un joueur à la même hauteur, le garde doit être mort et donc doit retourner à sa position initiale.

Le Step du Garde À l'opposé du *step* du joueur, celui du garde est assez complexe car en plus d'appeler des méthodes de déplacements, il doit gérer tout les impacts que le niveau et le joueur peuvent avoir sur lui (être dans un trou ou paralysé, ramasser ou laisser tomber un trésor...). Afin de limiter la taille des contrats, il faut donc réussir à minimiser les effets de l'opération sur chaque observateur. La gestion de la paralysie n'est pas dépendante du reste puisque la paralysie empêche la majorité des actions. Le fait de ramasser un trésor ou non est déjà une opération plus compliqué. En effet, afin de garder la cohérence du niveau, il ne faut pas qu'un garde puisse ramasser 2 trésors, ou bien qu'il lâche un trésor dans une case en contenant déjà un. Il faut donc bien spécifier tout les cas en fonction de la présence d'un trésor sur la garde, la présence d'un trésor dans la case du garde et la présence d'un trou sous le garde. Le temps passé dans un endroit, lui, ne pourra augmenter que si le garde n'est pas paralysé. Enfin, la gestion des mouvements dépend de beaucoup d'autres éléments : le temps passé paralysé, le temps passé dans un trou, la nature de la case sur laquelle il se trouve. Et le même problème que pour Player se pose, il faut donc également cloner le délégué et en faire un contrat.

Sortir d'un trou lorsque le joueur est juste au dessus Dans le comportement de base, lorsque le joueur marche sur un garde se trouvant dans un trou, le garde ne sort pas tant que le joueurs n'a pas bougé. Il a donc déjà fallu changer se comportement afin que le garde essaie d'exécuter *climbRight* ou *climbLeft* lorsque le joueur se trouve au dessus. Cependant, un autre problème se pose alors : par défaut, le fait de grimper à droite ou à gauche placera directement le joueur en diagonale de sa position dans le trou, il ne passe donc pas par la case contenant le joueur et ne lui fait pas perdre une vie. Or, il faudrait punir le joueur lorsqu'il reste sur un garde dans un trou. Il faut donc trouver une solution permettant de déclencher une mort mais sans faire en sorte que le garde puisse être dans 2 cases en même temps, ce qui casserait la cohérence. Les méthodes *climbLeft* et *climbRight* ont donc été changées pour permettre de grimper directement au dessus du trou, lorsque le joueur s'y trouve.

Contact entre joueur et garde A l'origine, 2 personnages ne pouvaient jamais entrer dans la même case. Or, le moteur utilise cette condition pour déclencher la perte d'une vie, le jeu n'était donc pas très intéressant. Le choix a donc été fait d'autoriser le déplacement d'un personnage dans une case à la condition qu'elle ne contienne pas un garde. Cela permet donc aux garde d'entrer en contact avec le joueur mais interdit un autre garde ou un joueur d'aller sur une case contenant un garde (on interdit donc le suicide au joueur). On peut remarquer que cela permet aussi au joueur de marcher sur la tête d'un garde sans pénalité, ce qui peut donner des concepts de niveaux intéressants.

Retour d'un garde à ses coordonnées initiales Lorsque le garde meurt, il a été décidé qu'il retourne à ses coordonnées initiales. Cela crée cependant un problème se pose si un autre garde se trouve à ces coordonnées. La solution choisie est que si jamais un autre garde se trouve sur ces coordonnées au moment où le garde qui meurt y retourne, l'autre garde retourne également à ses coordonnées initiales, et ce, jusqu'à ce qu'il n'y ait plus d'incohérence dans le niveau (On est garantie de ne pas entrer dans une boucle, car tous les gardes ont des coordonnées initiales différentes). Au niveau des contrats / spécification, on regarde donc si un garde existe à l'emplacement des coordonnées initiales avant de s'y téléporter et si c'est le cas, on vérifie que ce garde existe bien dans la case à ses coordonnées initiales.

Différence entre Joueur, Garde et Personnage Avec l'ajout de l'extension qui permettait de créer des cases dans lesquelles certains types de personnages ne peuvent pas passer, il faut donc interdire l'interchangeabilité entre ces classes. Il faut donc faire en sorte que Player et Guard incluent Character et ne le raffine pas.

Le Step du Moteur Le step du Moteur étant déjà un opération compliquée à cause du fait qu'il gère toutes les autres entités, il a été décidé qu'il n'était pas raisonnable de cloner totalement le moteur et de vérifier qu'il appelait bien les opérations de *Step* des gardes et du joueur. On vérifie cependant tout de même l'état de la majorité des observateurs qui peuvent être amenés à beaucoup changé, notammeent à cause du fait que le step puisse charger un nouveau niveau.

Les principales post-conditions consisent à vérifier que dans les cas où le joueur meurt, le niveau est bien recharger avec le score initial, et dans le cas où le joueur gagne, le niveau suivant est bien chargé s'il existe. Pour cela, on défini dans la spécification une méthode *loadlevel(int i)* qui permet de beaucoup synthétiser le code. Il reste ensuite pleins d'autres petits élements à verfier au sein d'un même niveau, tels que le fait que le joueur ramasse bien un trésor ou un item lorsqu'il passe dessus, que les cases piégées sont bien révélées, que l'ensemble des trous présents vieillises à chaque step, etc...

Au niveau des tests MBT

Les dépendances de Player et Guard avec Engine La principale difficulté au niveau des tests se trouve dans le fait qu'un joueur ou un garde doivent nécessairement se trouver dans un moteur de jeu. De plus, c'est le moteur de jeu lui même qui se charge de la création des gardes et des joueurs. Afin de pouvoir créer ces classes il est donc nécessaire de créer un moteur de jeu avec tout ce qui va avec. Mais étant donné que l'on peut vouloir un environnement différent en fonction du test, on ne peut pas initialiser le moteur avant chaque test. Dans les conditions initiales, il faut donc, à chaque fois, faire les modifications nécessaires sur l'EditableScreen qu'on donne au moteur, ajouter l'EditableScreen dans un gestionnaire d'écran, initialiser le moteur avec ce gestionnaire d'écran puis récupérer l'instance sur laquelle on veut faire des tests dans le moteur et en faire un contrat pour tester les conditions et invariants.