



PROJET DE PROGRAMMATION S6
2019/2020

Acting Shooting Star

Étudiants :

Nathan DECOU
Aymeric FERRON
Aurélien MOINEL
Saad ZOUBAIRI

Encadrant :
M. SCHLICK

14 février 2021

1 Introduction

Acting Shooting Star est un projet de programmation fonctionnelle proposé aux élèves de la filière informatique durant le second semestre de leur première année. L’objectif de ce projet est l’élaboration d’un jeu et d’une interface dans un terminal. Au cours de cet exercice, les élèves sont amenés à mettre en pratique leurs connaissances en programmation et en algorithmique afin de résoudre les problèmes qui leur sont posés.

1.1 Présentation du projet

Le jeu choisi pour l’année 2019/2020 est un *shoot’em up*. Il s’agit d’un jeu solitaire dans lequel le joueur incarne un vaisseau spatial assailli par une armée ennemie. Il doit alors tirer pour éliminer ses adversaires et survivre le plus longtemps possible.

1.2 Cadre de travail

La réalisation de ce projet s’est déroulée à distance en raison de la pandémie du COVID-19. Notre équipe, composée de quatre élèves, s’est réunie au moins une fois par semaine pendant quatre heures grâce à l’outil *Discord* pour avancer dans l’élaboration du jeu. La programmation en binôme (*pair-programming*) ayant été rendue difficile, nous avons veillé à communiquer régulièrement par échanges audios et textuels. Afin d’éviter les conflits sur le dépôt GIT, nous nous sommes répartis des tâches différentes et nous avons travaillé sur des fichiers distincts tout en nous entraînant. Nous avons adopté la méthode Agile Kanban afin de gagner en performance en nous appuyant sur le gestionnaire de projet en ligne *Taïga* (Annexe 3).

Les quatre membres de l’équipe ont utilisé un environnement de travail de type Unix. Nous avons tout d’abord adopté *Drracket* pour coder en *Racket*. Cependant, ce logiciel était instable pour certains d’entre nous, entraînant des problèmes rendant nos machines inopérantes (notamment une surcharge de travail pour le CPU). Nous avons alors migré sur *Emacs* couplé de son `racket-mode`.

Chaque vendredi après-midi, nous avons été encadrés par M. Schlick et M. Renault sur les outils *Slack* et *Discord*. Ainsi, nous avons pu avoir un retour régulier sur notre code ainsi que des réponses à nos questions.

1.3 Plan

Dans un premier temps, nous présenterons l’architecture logicielle de notre projet. Dans un second temps, nous discuterons nos choix d’implémentation et nous

aborderons leur complexité. Enfin, nous présenterons nos tests, notre documentation et des pistes d'améliorations avant de conclure.

2 Projet

Dans cette partie nous présentons tout d'abord notre architecture logicielle puis nos choix d'implémentations avant de décrire nos tests, notre documentation et de proposer des idées d'amélioration.

2.1 Architecture logicielle

Notre projet est composé de plusieurs fichiers que nous avons classés dans différents dossiers. Ainsi, le dossier **doc** contient notre documentation *scribble*, le dossier **tst** contient nos tests et le dossier **src** contient nos fichiers sources.

Le fichier principal de notre projet est **main.rkt**. C'est lui que l'on lance depuis la racine pour faire fonctionner le jeu avec la commande **racket src/main.rkt -f 60** où **-f** est une option qui gère le nombre de FPS (*frame per second*). **main.rkt** inclut des fonctions provenant de plusieurs fichiers sources :

- le fichier **actork.rkt** contient les fonctions relatives aux acteurs et aux messages.
- le fichier **collision.rkt** contient les fonctions relatives à la gestion des collisions entre acteurs.
- le fichier **world.rkt** contient les fonctions relatives au fonctionnement du monde.
- le fichier **world-display.rkt** contient les fonctions relatives à l'affichage et à l'animation du projet.
- le fichier **constants.rkt** contient les variables globales relatives au dimensionnement du terminal.
- enfin, le fichier **main.rkt** contient notre boucle principale qui gère le jeu.

Nous avons inclus les fonctions grâce à des **provide** couplés à des **contrats** afin de rester vigilant quant à l'utilisation correcte des types. En effet, le langage *racket* n'oblige pas à gérer les types mais il est plus prudent de se contraindre à le faire grâce à ces contrats. Ils permettent en outre de vérifier si l'argument passé à une fonction est du bon type et si son exécution renvoie le bon type d'objet.

2.1.1 Approche du modèle MVC

Notre jeu faisant appel à une interface graphique, il était naturel de se pencher sur une conception s'approchant le plus possible du motif MVC (Modèle-Vue-Contrôleur). Le fichier `world-display.rkt` implémente l'ensemble des fonctions destinées à l'affichage, ce qui en fait la **vue** de notre programme. Ensuite, le fichier `world.rkt` joue le rôle de **modèle**. Enfin le **contrôleur** n'est en réalité pas séparé de notre modèle dû aux contraintes de la bibliothèque `lux` d'où l'approche de ce motif.

2.2 Implémentation

Dans cette partie nous allons présenter la conception du projet. Dans un premier temps, nous allons aborder la façon dont nous avons implémenté les acteurs et leurs interactions. Dans un second temps, nous détaillerons l'implémentation des mondes dans lesquels évoluent le joueur. Dans un dernier temps, nous verrons comment nous avons animé le tout.

2.2.1 Gestion des acteurs

La base de ce projet repose sur l'interaction d'**acteurs** entre eux. Notre structure `actor` est composée de 4 champs :

- une chaîne de caractères **type** qui gère le type d'acteur. Ainsi, un acteur peut être un joueur ("**player**"), un mur ("**wall**") ou encore un missile ("**bullet**").
- un entier naturel **life** rendant compte du nombre de vie de l'acteur.
- une paire pointée **position** dont le **car** est l'ordonnée de la position de l'acteur dans le terminal et le **cdr** l'abscisse.
- une liste **msgbox** contenant les messages qui doivent s'appliquer sur l'acteur.

Nos acteurs sont donc des structures qui vont évoluer dans les jeux grâce à des **messages**. Détaillons notre structure `message`. Elle est composée de 4 champs :

- un entier **type** qui détermine le type de message. Il en existe trois :
 - le **type 0** concerne les messages qui se contentent de faire une simple mise à jour de l'acteur. Il s'agit typiquement d'un message visant à bouger l'acteur.
 - le **type 1** concerne les messages qui créent de nouveaux acteurs. Ainsi, il peut s'agir d'un message qui va créer un missile.
 - le **type 2** concerne les messages qui doivent être envoyés à un acteur en particulier. Ainsi, un joueur peut envoyer un malus à tous ses ennemis.

- une chaîne de caractère **target** permettant d'identifier le type d'acteur sur lequel le message va s'appliquer. Si le message n'a pas de cible, **target** = **"nil"**.
- une procédure **action** parmi les différents messages que nous avons créés.
- une liste **arg** des arguments dont a besoin **action** pour fonctionner.

Nous avons créé plusieurs messages différents afin de rendre notre jeu fonctionnel. Ainsi, il est possible d'utiliser le message :

- **move** prenant comme argument un entier x et un entier y . Il permet de faire avancer un acteur de x cases en abscisse et de y cases en ordonnée. C'est un message de type 0.
- **fire** qui permet de tirer un missile en créant un nouvel acteur **"bullet"**. C'est un message de type 1.
- **destroy** qui permet de faire descendre la barre de vie d'un acteur d'une unité. C'est un message de type 0 ou 2.
- **double** qui permet de doubler un acteur. C'est un message de type 1.

Afin de remplir la liste de messages d'un acteur en jeu, nous avons développé une fonction prenant l'acteur en paramètre, ainsi qu'un message, et retournant une copie de l'acteur passé en paramètre avec sa **msgbox** mise à jour.

Puis, afin d'appliquer tous les messages de la **msgbox** de l'acteur, nous avons créé une fonction **actor-update**. Le comportement de cette fonction diffère selon le type de message contenu dans la file de message de l'acteur. Pour gérer tous ces cas, nous avons décidé de passer en paramètre de cette fonction une liste dont le **car** est une liste d'acteurs (qui ne contient à l'état initial que l'acteur que l'on veut mettre à jour) et dont le **cdr** est une liste de messages (qui est vide à l'état initial). Ainsi :

- si la **vie de l'acteur est à 0**, alors il est retiré du jeu. La fonction retourne une liste de deux listes vides.
- si le **message est de type 0**, on retourne le nouvel acteur produit par le message appliqué à l'acteur initial. Il est écrasé, et donc mis à jour. Les tailles des deux listes en paramètre ne bougent pas.
- si le **message est de type 1**, on retourne un nouvel acteur que l'on place à la suite du premier dans la liste des acteurs et on continue à mettre à jour le premier.
- si le **message est de type 2**, on sauvegarde le message dans la seconde liste et on continue à mettre à jour le premier acteur.

Cette fonction retourne une liste dont le **car** est la liste contenant l'acteur mis à jour et les acteurs potentiellement créés. Le **cdr** de la liste contient les messages

potentiellement créés au cours de la mise à jour de l'acteur. Ainsi, il est possible de récupérer des acteurs induits par d'autres acteurs et les messages envoyés d'un acteur à un autre.

Afin d'envoyer ces nouveaux messages, nous avons implémenté une fonction prenant en paramètres la liste des acteurs en jeu et la liste des messages à envoyer et nous actualisons les `msgbox` des acteurs concernés.

La **collision** est un point important dans la programmation de notre jeu. Lorsque deux acteurs entrent en collisions, ils perdent chacun une vie. Nous avons géré cette fonctionnalité grâce au prédicat `collision?` qui prend en paramètre deux acteurs. Il va sauvegarder dans une liste toutes les positions occupées par les deux acteurs au cours d'une unité de temps du jeu (que l'on nomme `tick`) et s'il s'avère que les deux listes possèdent un élément identique, alors les routes des acteurs se sont croisées et ils sont nécessairement entrés en collision.

Nous décrivons un exemple de collision sur la figure 1. Le joueur ">" va se déplacer selon le chemin rouge tandis que la balle "-" va se déplacer selon le chemin bleu pendant un même intervalle de temps. On remarque que leur chemin se croise, cela signifie qu'ils sont entrés en collision.

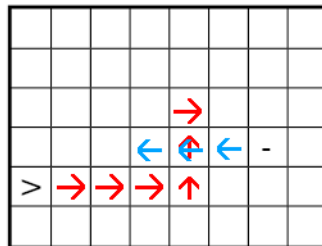


FIGURE 1 – Exemple d'une collision

Enfin, un autre point important concerne la création des acteurs. La première idée consistait en la création d'une fonction par type d'acteur différent. Cette méthode était fonctionnelle, mais ne permettait pas une genericité efficace du code. De plus, l'ajout d'un attribut aux acteurs impliquait une modification de toutes les implémentations. Afin d'améliorer notre code en suivant le **principe ouvert/fermé**, nous avons conçu un patron de conception que nous avons nommé "**Fabrique**" (Figure 2). Par conséquent, pour créer un acteur spécifique, il suffit de faire appel à la fabrique en donnant le type de l'acteur. Celle-ci se charge alors de le créer avec les bons attributs.

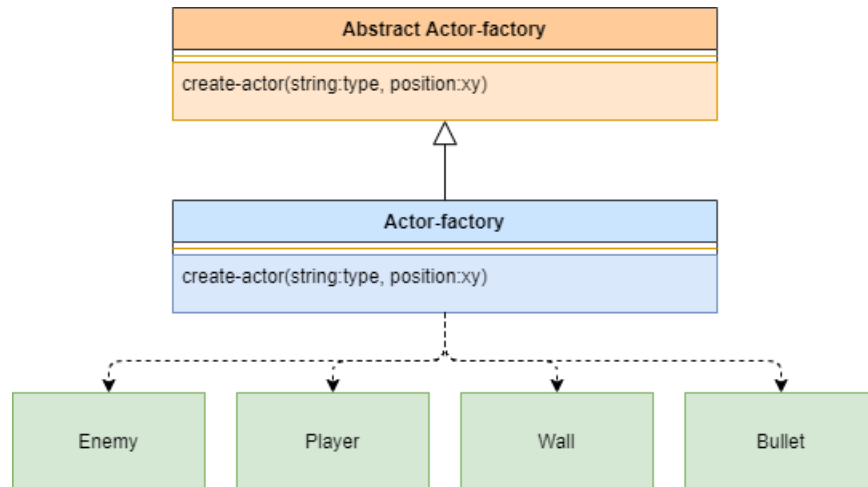


FIGURE 2 – Patron de conception "Fabrique".

2.2.2 Gestion des mondes

La gestion des acteurs et des messages ayant été explicitée, il convient d'aborder la manière dont ceux-ci sont inclus dans les mondes.

Un monde est une structure particulière qui contient toutes les données du jeu à un instant t . Ainsi, le jeu démarre sur un monde initial qui est remplacé par un nouveau monde à une certaine fréquence, mettant à jour les différents acteurs et leurs messages jusqu'à une fin de partie.

Un monde est une structure contenant plusieurs champs :

- un entier **tick** qui caractérise le nombre de mondes s'étant succédés. Chaque monde est défini par son propre **tick** qui s'incrémente à chaque fois qu'un nouveau monde naît.
- une liste **actors** qui contient la liste de tous les acteurs actifs dans le monde.
- une liste **list-old-actors** qui sauvegarde les dix dernières listes d'acteurs des dix derniers mondes afin de remonter dans le temps au besoin.

La structure **world** contient également plusieurs méthodes permettant de gérer les aspects graphiques du jeu que nous aborderons dans la section 2.2.3.

Ce sont les **worlds** qui gèrent la **destruction des acteurs**. Il s'agit de la fonctionnalité à la complexité temporelle la plus élevée de notre projet. En effet, un acteur doit être supprimé de la liste des acteurs si celui-ci ne répond plus à certaines conditions sous peine de rester en mémoire durant toute la durée du jeu.

La première condition qui doit être respectée par les acteurs est de **rester dans**

l'espace du monde. Si un acteur dépasse ces limites, il est détruit par la fonction. Cette situation se produit régulièrement en raison du défilement des acteurs lors de l'animation (cf section 2.2.3).

La deuxième condition concerne **les collisions** entre acteurs. Nous avons vu dans la section 2.2.1 qu'une collision entraînait la perte d'un point de vie. C'est le **world** qui va utiliser le prédicat **collision?** et qui va ôter un point de vie aux acteurs concernés. Cependant, vérifier pour tous les acteurs en jeu s'ils ont subi une collision deux à deux n'est pas du tout optimal. Pour remédier à cela, nous avons pris la décision suivante : les murs sont des acteurs indestructibles. Ils ne peuvent pas subir des dégâts, il est donc inutile de vérifier s'ils ont subi une collision. Les murs étant les acteurs majoritaires, la complexité de la fonction s'en trouve grandement améliorée.

Afin de réaliser cette amélioration, deux filtres ont été mis en place. Le premier permet de récupérer l'ensemble des acteurs de type **wall** et le second récupère tous les autres types d'acteurs. Dans les deux cas, une **curryfication** a été réalisée entre la fonction *filter* et le prédicat associé afin d'obtenir une fonction pure d'un point de vue fonctionnel.

A la fin de chaque **tick** un nouveau monde est généré. La fonction gérant cette génération est en majeure partie **world-actors-update**. Cette fonction prends comme paramètre la liste de tous les acteurs en jeu pendant le **tick** et lui applique une série de changements. Pour cela, plusieurs fonctions sont appelées. Tout d'abord les acteurs qui ne respectent plus les conditions pour être en vie reçoivent un message de type **destroy**. Ensuite, un message de type **move** est envoyé à tous les acteurs (excepté le player) afin de réaliser le défilement du jeu, puis la liste des acteurs est mise à jour en leur appliquant les messages de leur **msgbox** puis on met à jour leur **msgbox** en envoyant les messages de type 2. Enfin, on sauvegarde la liste des acteurs de ce monde, on incrémente le **tick** et on génère un nouveau monde.

2.2.3 Gestion de l'animation

Maintenant que nous avons introduit les mondes et leur fonctionnement, nous allons expliquer comment nous gérons l'affichage du jeu et les interactions avec le joueur.

Pour générer l'affichage du jeu, nous utilisons la bibliothèque *Raart*. Ainsi, un monde est une *frame* sur laquelle on dessine chacun des acteurs en jeu durant un **tick** donné. Lorsque le **tick** s'incrémente, un nouveau monde remplace l'ancien, i.e. l'ensemble des acteurs est mis à jour (position, vie, destruction, etc...).

Afin de gérer l'animation, nous utilisons différentes méthodes que nous avons placées dans la structure `world` :

- **word-fps** permet de gérer le nombre de *frame per second* du jeu.
- **word-label** permet de gérer le nom de la fenêtre dans le terminal.
- **word-event** permet de lier des événements clavier avec des fonctions.
- **word-output** permet de relier la sortie graphique vers le terminal en faisant appel aux fonctions qui dessinent les acteurs.
- **word-tick** permet d'incrémenter le `tick` pour passer du monde n au monde $n + 1$.

La méthode `word-event` est appelée à chaque fois qu'un événement clavier a lieu. Nous en avons implémenté trois :

- **accomplir un déplacement** via une touche « flèche » du clavier. Ceci déclenche l'envoi d'un message `move` au joueur (via la fonction `world-send-player`) en nous appuyant sur le fait que l'acteur "`player`" est toujours le premier de la liste des acteurs en jeu (`actors`).
- **tirer un missile** via la touche T du clavier. Ceci déclenche l'envoi d'un message `fire` au joueur selon le même processus que le déplacement.
- **revenir dans le temps** via la touche Y. Pour cela, nous parcourons la liste `list-old-actors` pour récupérer la liste des acteurs qui existaient dans le passé. Cette liste est alors envoyée dans la fonction de mise à jour afin de réinjecter les acteurs en jeu.

Afin de dessiner plus facilement les acteurs, nous avons développé une fonction particulière `draw-actors` qui prend en argument la liste de tous les acteurs en jeu (`actors`) afin de les dessiner. Cette fonction appelle des sous-fonctions qui permettent de déterminer pour chaque type d'acteur sous quelle forme et sous quelle couleur ils doivent apparaître dans le terminal (un "<" bleu pour les "`enemy`" par exemple).

A chaque fois qu'un nouveau monde est créé, on le dessine de la même manière, créant ainsi l'illusion d'un mouvement.

2.3 Complexités

Cette partie spécifie l'ensemble des fonctions possédant les complexités en temps les plus élevées dans le tableau 1 ci-dessous où m est le nombre maximum de messages (2-3 en général), n le nombre d'acteurs et h la hauteur de la *frame*.

Fonction	Complexité
fichier actor.rkt	
update	$\mathcal{O}(n \times m)$
fichier collision.rkt	
collision ?	$\mathcal{O}(m)$
fichier world.rkt	
world-actors-send-move	$\mathcal{O}(n \times m)$
create-line-wall	$\theta(h)$
will-collide	$\mathcal{O}(n \times m)$
filter-is-wall	$\theta(n)$
filter-no-wall	$\theta(n)$
delete-if	$\mathcal{O}(n^2 \times m)$
world-actors-update	$\mathcal{O}(n^2 \times m)$
draw-actors	$\theta(n)$

TABLE 1 – Complexité en temps des fonctions principales

Ainsi la complexité globale de notre programme pour chaque tick est de $\mathcal{O}(n^2 \times m)$. m pouvant être considérée comme une constante, la complexité est donc quadratique. Enfin, le nombre moyen d'acteurs par `tick`, se traduit par $\mathcal{O}(2 \times L)$ ou $\mathcal{O}(2 \times L + h)$ tous les 100 ticks, avec L la largeur de la **frame**.

2.3.1 Profilage

Afin de déterminer les fonctions les plus chronophages lors d'une exécution de notre jeu, nous avons utilisé la librairie **profile** de **raco**. Celle-ci affiche l'ensemble des fonctions utilisées ainsi que le temps/pourcentage consacré par le CPU pour chacune d'entre-elles. On remarque que les fonctions liées aux librairies **raart** et **lux** sont celles utilisant le plus de processeur ($\geq 95\%$). Il a été compliqué de réellement déterminer l'impact de nos fonctions. Cependant, nous avons été en mesure de remarquer que les fonctions faisant appel à une vérification de collision étaient celles impliquant le plus d'utilisation CPU.

2.4 Tests

Pour chacun de nos fichiers sources, nous avons réalisé un fichier test permettant de vérifier la bonne marche de nos fonctions. Ces tests utilisent la bibliothèque *rackunit* permettant de générer dans le terminal un simple message comptabilisant le

nombre de tests valides. Tous ces tests unitaires permettent une maintenabilité du code accrue, ainsi qu’une documentation supplémentaire. Ils sont le moyen le plus efficace de s’assurer du bon fonctionnement du programme dans son ensemble, et permettent entre autres de maintenir le principe **ouvert/fermé**. En effet, l’ajout d’une fonctionnalité ne doit pas impacter le code déjà existant.

D’autre part, la création de contrats, rendue possible par le typage statique que propose **Racket**, s’avère être une fonctionnalité très utile. En effet, les erreurs sont bien plus faciles à comprendre et à trouver lorsqu’il s’agit d’un problème de paramètre ou de retour de fonction. Par conséquent nous avons appliqué ces contrats à toutes nos fonctions.

2.4.1 Couverture de tests

Dans le but de réaliser une couverture de tests, nous avons utilisé la librairie **cover** de **raco**, et obtenu les résultats du tableau 2.

Fichier	Couverture (%)	Nombres de tests unitaires
actor.rkt	93	14
world.rkt	70	9
collision.rkt	87	
world-display.rkt	78	3

TABLE 2 – Couverture de tests indiquée par l’outil **raco cover**

Nous pouvons remarquer que la couverture de tests semble satisfaisante pour chacun de nos fichiers. Cependant, il est nécessaire de rester critique quant à la manière dont **raco** détermine cette couverture. En effet, celui-ci vérifie le pourcentage de lignes appelées dans un fichier. Ainsi, si une fonction fait appel à l’ensemble des fonctions dans son fichier et qu’un test est réalisé sur cette unique fonction, la couverture est considérée comme avoisinant les 100%. Voilà pourquoi nous considérerons que la couverture de test pour le fichier **world** et **collision** avoisinerait plutôt les 50%. Il y aurait donc un certain nombre de tests à ajouter pour assurer une couverture complète.

2.5 Documentation

Le documentation du projet se décline en trois aspects.

Le premier concerne les **commentaires** dans le code : toutes nos fonctions sont commentées directement au dessus de leur implémentation. Nous avons détaillé pour

chacune d'entre elles leur utilité ainsi que le type de leurs paramètres d'entrée et de sortie.

Le deuxième concerne la documentation *Scribble*. Elle peut être générée via le *Makefile* avec la commande `make docs`. La documentation ainsi créée (une page `html`) peut alors être visualisée via un navigateur. Elle liste toutes les fonctions du projet réparties par fichier et permet d'accéder rapidement à leur utilité ainsi qu'à leurs paramètres d'entrée et de sortie.

Le dernier aspect de la documentation de ce projet est ce **rapport**. Il adopte une approche globale du projet et décrit en profondeur nos choix d'implémentation.

2.6 Amélioration

Actuellement, certaines améliorations peuvent être apportées au code. D'une part, il existe toujours une contrainte liée aux effets de bords concernant la récupération des différentes tailles de notre terminal. En effet, ces constantes sont utilisées lors de la génération aléatoire de nos acteurs. Néanmoins, il n'existe pas de terminal sur la *Forge* par conséquent, il n'était pas possible de faire appel aux fichiers ayant besoins de récupérer ces dimensions. La solution a donc été de mettre en place un comportement de type *try/catch*, selon la réussite ou non lors du chargement de la librairie **charterm**. Dans le cas d'un échec, il y a donc utilisation de constantes. Cependant, la meilleure méthode aurait été de passer en paramètre de la fonction principale l'ensemble des dimensions récupérées. Ainsi, les tests des fonctions ayant besoin des dimensions du terminal auraient pu se réaliser par le passage de constantes en paramètre. Les effets de bords sont ainsi limités.

Avec plus de temps, nous aurions aimé implémenter un compteur de score ainsi qu'une fin de partie. En effet, lorsque le joueur meurt, le jeu continue de fonctionner et il faut alors le quitter avec la touche "Q" alors qu'il serait préférable que le jeu s'arrête de lui-même.

Afin d'améliorer la généricité du code, il pourrait être intéressant de développer notre patron de conception **Factory** afin que celui-ci puisse produire d'autre **Factory** spécifiques à un genre d'acteur (ennemi, allié, widgets, etc...).

3 Conclusion

Pour conclure, nous pouvons affirmer que ce projet a été un moment privilégié pour se rendre compte de l'utilité de la programmation fonctionnelle. En effet, nous avons été en mesure d'appliquer un certain nombre de nouvelles notions, comme l'affranchissement des affectations de variable pour éviter les effets de bord, une refonte

de la manière de penser un programme, ou encore l'application d'une récursivité terminale quasi systématique lorsque celle-ci s'avérerait nécessaire. De plus, l'utilisation du modèle d'acteurs s'est avéré instructif et a changé notre façon de concevoir un code.

Enfin, concernant notre réalisation directe, nous avons conscience des limites et des atouts de notre implémentation. En effet, certains ajouts pourraient contribuer à l'amélioration du projet, notamment d'un point de vue technique et ludique.

4 Annexes



USER STORY	>< NEW	>< IN PROGRESS	>< READY FOR TEST	>< CLOSED
<div><div>✕</div><div>#5 Environnement de Travail</div><div>New</div><div>Not estimated</div><div>+</div><div>☰</div></div>				<div><div>Aurélien MOINEL</div><div>#6 .gitignore</div></div> <div><div>Aurélien MOINEL</div><div>#7 Makefile</div></div> <div><div>Aurélien MOINEL</div><div>#8 Créer le diagramme de classes</div></div>
<div><div>✕</div><div>#4 Tests</div><div>New</div><div>Not estimated</div><div>+</div><div>☰</div></div>				<div><div>#1 Actor</div></div>
<div><div>✕</div><div>#3 Documentation</div><div>New</div><div>Not estimated</div><div>+</div><div>☰</div></div>				<div><div>#2 Créer fichier Scribble</div></div>
<div><div>✕</div><div>Storyless tasks</div><div></div><div></div><div>+</div><div>☰</div></div>				

FIGURE 3 – Interface de gestion de projet Taïga

