Université Paris Diderot

L2 Informatique année 2019/2020

Rapport de projet PI4
-
CodeLab

Hugo THOMAS

Antoine LIU

Emile ROLLEY

Benjamin NICOLAS

Résumé

Ce document regroupe toutes les informations nécessaires pour faciliter la compréhension de notre projet réalisé dans le cadre du cours `Projet de programmation L2 2019-2020`.

A savoir, les instructions permettant la compilation, l'exécution et l'utilisation de CodeLab4 - L'aventure, une description de son architecture et un résumé de notre organisation.

Table des matières

1 Présentation

CodeLab4 - L'aventure (CL4) est un programme ayant pour objectif d'initier à la programmation de façon ludique et ergonomique. En effet, les problèmes à résoudre sont camouflés par un scénario, permettant l'immersion du joueur.

Le joueur se retrouve dans la peau d'une naufragée qui devra collecter des ressources et fabriquer des outils, afin de construire un radeau qui lui permettra de retrouver la civilisation.

Mais attention ! Elle ne devra ni mourir de faim, ni se faire surprendre par une tempête.

CL4 fournit un environnement de programmation (Figure 1), construit autour d'un langage simple (Section 4), permettant de se familiariser avec les conditions ainsi que les boucles. Le joueur doit écrire un ensemble d'instructions permettant, au fur et à mesure de l'aventure, de remplir des quêtes.

Les quêtes sont regroupées dans plusieurs chapitres nécessitant l'assimilation de nouvelles notions pour pouvoir être complétées.

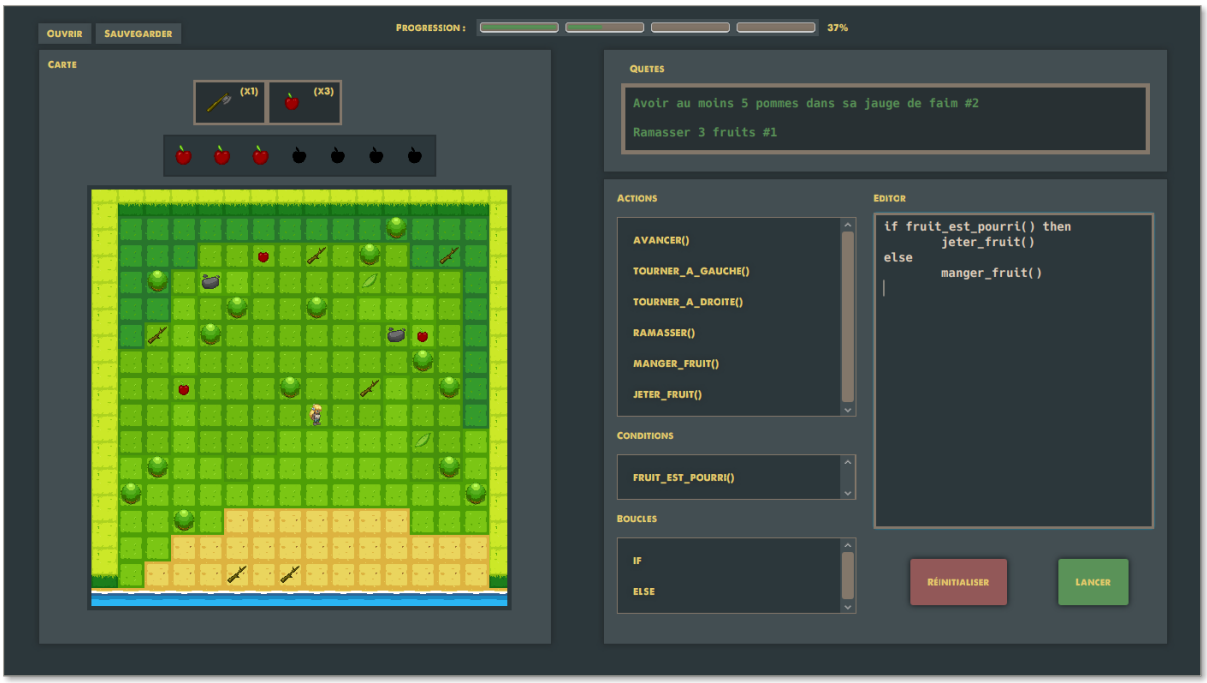

Figure 1: Capture d'écran de la fenêtre principale en *haute résolution*.

2 Utilisation

2.1 Installation

Le dépôt `GitLab` est disponible à l'adresse suivante : https://gaufre.informatique.univ-paris-diderot.fr/sangnier/codelab4

La compilation et l'exécution du projet nécessitent la librairie `JavaFX`. Toutes les informations concernant son installation ainsi que la liste des commandes disponibles se trouvent dans le fichier BUILD.md (situé dans le répertoire `documentation`).

2.2 Création d'une nouvelle session

Une fois compilé, CL4 peut être lancé grâce à la commande : `make run`.
La fenêtre `launcher` apparaît. Depuis cette fenêtre, le joueur a la possibilité de :

- Créer une nouvelle session, en renseignant le nom de la nouvelle session puis en cliquant sur le bouton de validation. (Figure 2.a)
- Supprimer une session existante, en sélectionnant la session correspondante puis en cliquant sur le bouton *trash*. (Figure 2.b)
- Lancer une session existante, en sélectionnant la session correspondante puis en cliquant sur le bouton *play*. (Figure 2.c)

(a) (b) (c)

Figure 2: Captures d'écran du `launcher`

L'état de l'action effectuée depuis le `launcher` est affiché dans le terminal précédé du préfixe `<INFO>`. Par exemple, lors de la création d'une nouvelle session *session2* le message suivant apparaît :

```
$ <INFO> - Session 'session2' created.
```

En revanche, si le joueur essaie de créer une nouvelle fois une session *session2* (ce qui n'est pas possible car chaque nom de session doit être unique), le message suivant apparaît :

```
$ <INFO> - An error occured while creating 'session2' session.
```

2.3 Résoudre une quête

Une fois la session lancée, une fenêtre de quête apparaît (Figure 3). Cette fenêtre contient le numéro et la description du chapitre courant, ainsi qu'un court texte informant le joueur de l'avancée de l'intrigue et de l'objectif de la prochaine quête. La fenêtre peut être fermée en cliquant sur le bouton *Compris !*.

La fenêtre principale se lance alors (Figure 1). A présent, le joueur a accès aux informations suivantes :

- L'état actuel de la carte. (Figure 4)
- L'état de l'inventaire. (Figure 5)
- L'état de la jauge de faim (Figure 6). Lorsqu'elle est débloquée (à partir du deuxième chapitre), toute les 5 actions effectuées, une pomme disparaît. Et lorsque la jauge est vide, l'interprétation échoue, forçant le joueur à recommencer.
- Sa progression actuelle. Chaque région correspond à un chapitre, ainsi la Figure 7 correspond a une session de quatre chapitres, dont le premier est terminé.
- La liste des quêtes accomplies. Le premier objectif de la liste correspond à la quête courante. (Figure 8)

Une fois que le joueur a un objectif de quête, il va devoir écrire un ensemble d'instructions dans l'éditeur (Figure 10). Il peut écrire directement dans l'éditeur, ou bien cliquer sur l'une des instructions listées à sa gauche (Figure 11), qui le remplira automatiquement.

Après avoir écrit les instructions, l'interprétation se lance grâce au bouton *Lancer* et peut être interrompue à tout moment avec le bouton *Réinitialiser*.

Si à la fin de l'interprétation, l'objectif de la quête est atteint, une fenêtre décrivant la prochaine quête apparaît. Sinon, un message apparaît au-dessus de l'éditeur indiquant au joueur de modifier son programme et relancer son interprétation jusqu'à accomplir la quête courante.

A chaque fois qu'une quête est accomplie, la session courante est mise à jour et sauvegardée dans le fichier `JSON` : `./sessions/<nom_de_la_session>.json`.

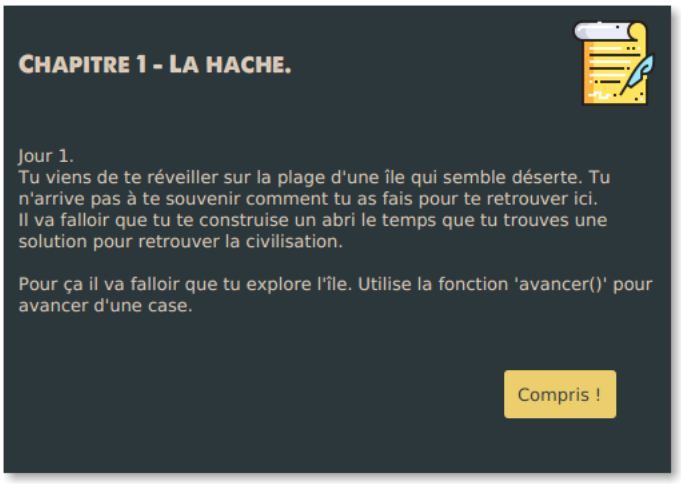

Figure 3: Capture d'écran d'une fenêtre de quête.

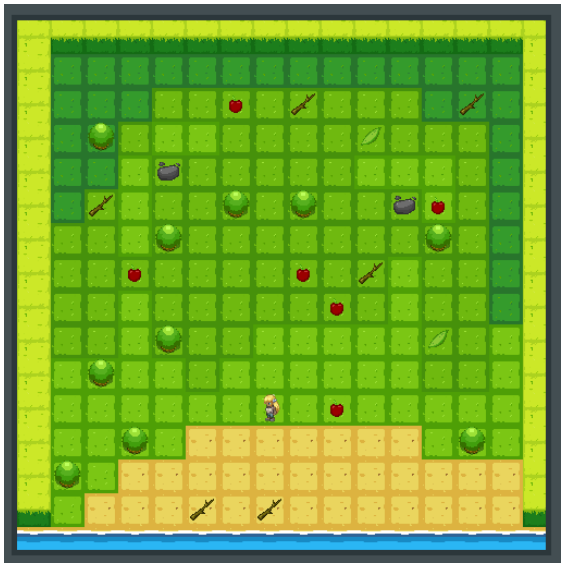

Figure 4: De la carte

Figure 5: De l'inventaire

Figure 6: De la jauge de faim

Figure 7: De la barre de progression

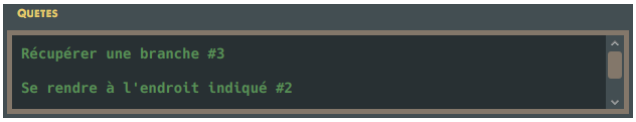

Figure 8: De la liste des objectifs de quête

Figure 9: Captures d'écran

2.4 Gestion des fichiers d'instructions

Grâce aux boutons *Ouvrir* et *Sauvegarder* situés en haut à gauche de la fenêtre principale, le joueur a la possibilité de sauvegarder dans un fichier texte l'ensemble du programme présent dans l'éditeur. Une fois sauvegardé, le fichier peut être ouvert à tout moment et son contenu sera placé dans l'éditeur.

2.5 Solutions

Les solutions complètes sont disponibles dans le fichier ./documentation/solutions.pdf.

Le joueur étant libre de ramasser les objets à la position et dans l'ordre qu'il souhaite, les solutions fournies ne sont pas absolues. Si le joueur ne les suit pas depuis le début, elles peuvent ne pas marcher dans l'état de sa carte !

3 Organisation

Dans la continuité du cours Conduite de projet PréPro suivi au semestre dernier, nous avons choisi d'utiliser la méthode de développement *scrum*.

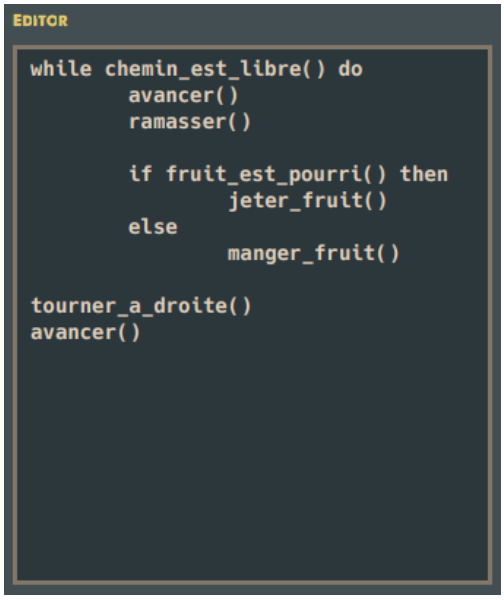

Figure 10: Capture d'écran de l'éditeur

Figure 11: Capture d'écran de la liste des instructions disponibles

3.1 Utilisation de `GitLab`

Chaque *sprint* est associé à un *milestone* regroupant les *issues* et les *merge request* (MR) associées.

Nous avons créé plusieurs labels correspondant à plusieurs parties du projet. Par exemple les *issues* portant le label `GUI` correspondent à l'implémentation de fonctionnalités modifiant l'interface graphique.

Nous avons voulu explorer et utiliser le maximum des outils fournis par `GitLab` tel que `GitLab-CI` qui, une fois configuré, nous a permis de lancer automatiquement une *pipeline* pour chaque *commit push* sur le dépôt, vérifiant que le projet compile et que les tests passent (Section 5.2). Ainsi, pour pouvoir accepter une MR il faut obligatoirement que la *pipeline* passe, ce qui assure d'avoir dans les branches principales (`master` et `develop`) un programme qui compile et qui passe les tests.

3.2 Déroulement d'un sprint

Mis à part les deux premiers *sprints* de conception qui ont duré deux semaines, nos *sprints* duraient une semaine.

Le *sprint-review* se déroulait le jeudi lors du cours de TP d'une durée de deux heures. Nous prenions 15 à 30 minutes pour revenir sur le *sprint* précédent. Le reste du temps était consacré à l'élaboration et l'assignation du *sprint-backlog* pour le prochain *sprint*. Lors du confinement, nous avons maintenu les *sprint-reviews* sur notre serveur `Discord`.

Durant un *sprint*, pour chaque *issue*, une branche était ouverte depuis `develop`. Une fois l'*issue* finie une MR était alors ouverte pour pouvoir être relue par les autres membres de l'équipe. Une fois tous les membres d'accord, la MR était acceptée et l'*issue* considérée finie.

A chaque fois que nous arrivions à une version stable comprenant de nouvelles fonctionnalités dans `develop`, nous mettions à jour `master` et une *release* était créée.

3.3 Répartition globale des tâches

Ayant suivi la méthodologie *scrum*, chacun a travaillé sur toutes les parties du projet. Cependant, chaque membre a approfondi certaines parties.

Hugo s'est occupé en particulier de :

- La conception et l'implémentation du `Parser` avec les différentes `Exceptions`.
- L'implémentation du `MapGenerator`.
- L'implémentation de l'éditeur de texte.

Antoine s'est occupé en particulier de :

— L'implémentation du `datamodel`.

Emile s'est occupé en particulier de :
— La conception et l'implémentation de l'`Interpreter`.
— La mise en place des tests avec `TestsRunner`.
— La mise en place de `GitLab-CI`.
— La création des images présentes dans `./src/resources/png/`.
— La création des *releases*.
— La rédaction du rapport.

Benjamin s'est **seulement** occupé de :
— Écrire la première version des fichiers `JSON` des quêtes.

4 Description du langage

Pour le langage, nous avons voulu le rendre ressemblant aux langages déjà existants pour faciliter la continuité de l'apprentissage, en effet, tous les mots clés sont écrit en anglais. Cependant, nous le voulions assez intuitif pour un enfant parlant couramment français, c'est pour cela que le nom des instructions est en français.

4.1 Sa grammaire

Pour faciliter la lisibilité et le *parsing* du langage, nous avons choisi de limiter le nombre d'instructions à une seule par ligne, et la délimitation des blocs de code se fait par une indentation (similaire au `Python`).

Nous avons classé les mots du langage en trois catégories :
— Les *actions* sont représentées sous la forme d'un appel de fonction `nom_action()` et permettent l'interaction avec le personnage ou les éléments du décor.
— Les *conditions* sont également représentées sous la forme d'un appel de fonction et permettent de tester l'état du décor à un moment donné.
— Les *boucles* sont représentées sous la forme de mots clés écrits en anglais encadrant une condition `if condition() then` et permettent d'effectuer des actions en fonction de la validité d'une condition.

Les actions disponibles sont :
— `avancer()` déplace le personnage sur la case située en face de lui (si celle ci est libre).
— `tourner_a_droite/gauche()` oriente le personnage vers sa droite/gauche.
— `ramasser()` place l'item situé devant le personnage dans son inventaire (s'il y a un item et qu'il est ramassable).
— `couper()` transforme l'arbre situé en face du personnage en bûche et la place dans son inventaire (action réalisable uniquement si une hache est présente dans l'inventaire).
— `manger_fruit()` mange un fruit de son inventaire, si ce dernier est *sain* alors la jauge de faim augmente de 2 pommes, sinon elle diminue d'une.
— `jeter_fruit()` supprime un fruit de l'inventaire.

Les conditions disponibles sont :
— `chemin_est_libre()` renvoie *vrai* si la case située devant le personnage est vide.
— `fruit_est_pourri()` renvoie *vrai* si le dernier fruit ramassé est pourri.

Les boucles disponibles sont :
— `if [condition] then [instructions] else [instructions]` exécute l'ensemble des instructions indentées si et seulement si la condition renvoie *vrai*, sinon, exécute l'ensemble des instructions indentées après le `else`.
— `while [condition] do` exécute les instructions indentées tant que la condition renvoie *vrai*.

Remarque : les boucles peuvent être imbriquées sans restriction.

4.2 Quelques exemples

Afin de faciliter la compréhension de la grammaire, voici quelques exemples de programmes valides. *Fais avancer de deux cases le personnage* :

```
avancer()
avancer()
```

Jette un fruit s'il est pourri sinon le mange :

```
if fruit_est_pourri() then
    jeter_fruit()
else
    manger_fruit()
```

Tant que la case devant le personnage est vide le fait avancer, ramasser l'objet situé à sa droite et manger un fruit s'il n'est pas pourri sinon jette un fruit :

```
while chemin_est_libre() do
    avancer()
    tourner_a_droite()
    ramasser()
    if fruit_est_pourri() then
        jeter_fruit()
    else
        manger_fruit()
    tourner_a_gauche()
```

5 Architecture

CL4 a été conçu en ayant l'objectif d'être modulaire, en effet, grâce à l'utilisation de la librairie `JavaFX`, la séparation entre les vues (Section 5.5) et les contrôleurs se fait naturellement.

5.1 Les classes `Java`

Nous avons fractionné le code source du projet de la façon suivante :

- Le *package* `codelab4` contient les classes :
  - `App` qui lance le `launcher`.
  - `LauncherController` qui contrôle la fenêtre du `launcher` et qui lance la fenêtre principale.
  - `MainWindowController` qui contrôle une fenêtre principale et son instance associée.
  - `Instance` qui manipule la session courante.
- Le *package* `codelab4.alert` contient les classes :
  - `UserAlert` (classe abstraite).
  - `UserAlertQuestDescription` qui hérite de `UserAlert` et qui permet de lancer les fenêtres quêtes.
  - `UserAlertChapterDoAtEnd` qui hérite de `UserAlert` et qui permet de lancer les fenêtres de fin de chapitres.
  - `UserAlertEndOfGame` qui hérite de `UserAlert` et qui permet de lancer la fenêtre de fin.
- Le *package* `codelab4.commons` contient la classe :
  - `Tools` qui regroupe toutes les fonctions utilisées dans plusieurs classes.
- Le *package* `codelab4.datamodel` contient les classes :
  - `Quest` qui représente une quête.
  - `Chapter` qui représente un chapitre et qui manipule des quêtes
  - `Chapters` qui manipule les chapitres.
  - `MapState` qui représente l'état d'une carte.
  - `Session` qui représente une session.
  - `Sessions` qui manipule les sessions.
- Le *package* `codelab4.filemanager` contient les classes :
  - `FileManager` qui gère la sauvegarde et l'ouverture des fichiers.
  - `JSONConverter` qui permet de charger un fichier `JSON` depuis une archive `JAR`.
- Le *package* `codelab4.process` contient les classes :
  - `Token` qui est une énumération représentant les *tokens*.
  - `Parser` qui transforme une chaîne de caractères en une liste de `Token`.

  - `Interpreter` qui transforme la *MapState* courante en utilisant la liste de *tokens*
- Le *package* `codelab4.process.exceptions` contient les classes :
  - `InfiniteLoopException` qui hérite de la classe `Exception` et qui est lancée depuis la classe `Interpreter` lorsqu'une boucle a effectué plus de 50 itérations.
  - `InstructionCountException` qui hérite de la classe `Exception` et qui est lancée depuis la classe `Parser` lorsque le nombre d'instructions dépasse la limite donnée par la quête courante.
  - `SyntaxException` qui hérite de la classe `Exception` et qui est lancée depuis la classe `Parser` lorsque une fraction de la chaîne de caractères à *parser* ne correspond à aucun `Token`.
- Le *package* `codelab4.vizualisation` contient les classes :
  - `MapGenerator` qui représente le `MapState` courant en agrégeant les différentes images stockées dans `./src/resources/png`.

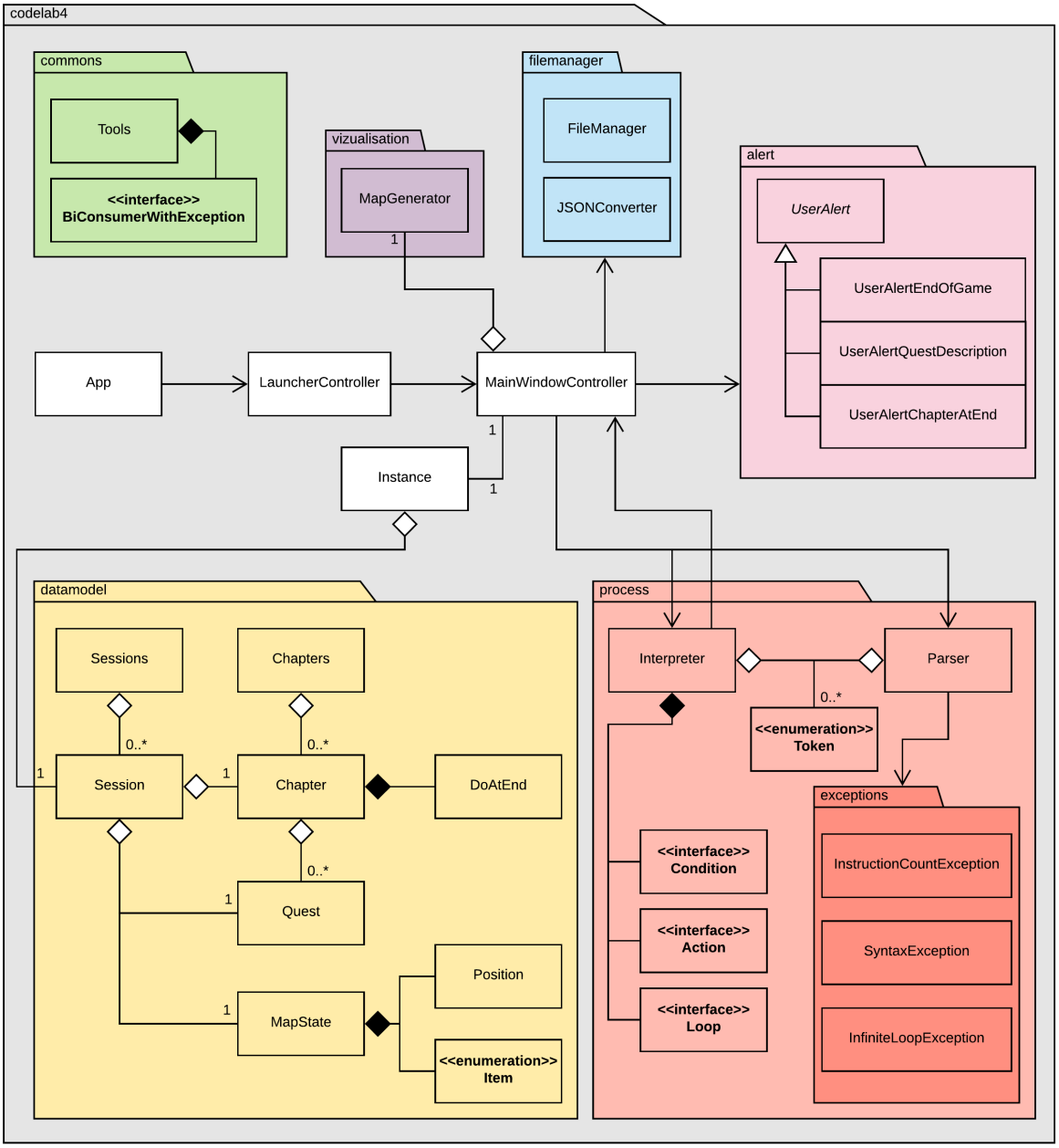

Figure 12: Diagramme UML simplifié du *package* `codelab4` (sans les classes de tests).

5.2 Les classes de tests

Nous avons voulu utiliser la méthode de développement dirigé par les tests (DDT). Pour cela nous avons utilisé la librairie `JUnit 4`. Ne pouvant utiliser `Maven` qui automatise l'exécution des tests, nous avons dû écrire une classe qui s'en occupe : `codelab4.TestsRunner`.

Chaque classe présente dans les *subpackages* de `codelab4` possède une classe de tests dans le répertoire `./test/codelab4/<subpackage_name>/`.

Cependant, ce sont uniquement des tests unitaires non exhaustifs et nous n'avons pas mis en place de tests pour les contrôleurs car cela aurait pris trop de temps et aurait ajouté l'utilisation d'une librairie externe supplémentaire.

5.3 Les librairies externes

CL4 a nécessité l'utilisation de plusieurs librairies externes stockées dans `./lib/` :

- `javafx` contient les différents composants nécessaires à la compilation de CL4 dans le `GitLab runner`. (Section 3.1)
- `commons-io-2.0.jar` nécessaire pour la gestion des fichiers.
- `hamcrest-core-1.3.jar` et `junit-4.13.jar` nécessaire pour la compilation et l'exécution des tests.
- `org.json.jar` nécessaire pour la gestion des fichiers et objets `JSON`.

5.4 La documentation

Le répertoire `./documentation/` regroupe tous les fichiers produits lors de la conception.

Attention, mis à part les fichiers BUILD.md et solutions.pdf, les autres fichiers ne sont pas à jour et ne sont qu'une trace de notre conception.

5.5 Les ressources

Le répertoire `./src/resources/` est décomposé en plusieurs sous-répertoires en fonction des types de fichiers :

- `css/` contient les feuilles de style `CSS`.
- `font/` contient les polices.
- `fxml/` contient les vues au format `FXML`.
- `icon/` contient les logos.
- `json/` contient les fichiers `JSON` des chapitres et de la carte de départ.
- `png/` contient les images nécessaires à la visualisation.

6 Produit final

Tous les objectifs fixés dans le cahier des charges ont été réalisés. A savoir :

- Un `launcher` permettant de créer, supprimer et lancer des sessions.
- Un module permettant d'ouvrir et sauvegarder des fichiers d'instructions.
- Un éditeur permettant de rédiger un ensemble d'instructions.
- Un module permettant de visualiser l'exécution d'un ensemble d'instructions.
- Un module permettant de sauvegarder la progression des sessions.
- Une interface regroupant tous ces modules.

En plus de ces objectifs initiaux, de nouvelles fonctionnalités ont été ajoutées :

- La visualisation des erreurs de syntaxe. (Figure 13)
- L'affichage de la progression dans la fenêtre principale. (Figure 7)
- L'affichage des instructions cliquable en fonction des quêtes. (Figure 11)

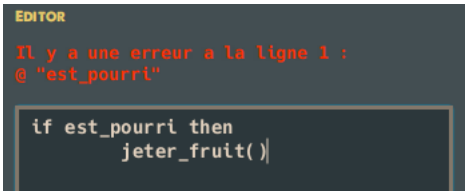

Figure 13: Capture d'écran d'une erreur de syntaxe (oublis de parenthèses).

7 Conclusion

Au terme de 11 *sprints*, nous avons pu réaliser un programme qui répond à l'énoncé (CodeLab). CL4 propose un environnement de programmation graphique stable, accompagné d'un langage textuel simple.

Le fait d'offrir une expérience type *"open world"* accompagnée d'une histoire dans le but de rendre l'expérience plus prenante, a posé quelques problèmes de *game design*. Par exemple, le chapitre 2 qui apporte une nouvelle mécanique avec la barre de faim ainsi que l'utilisation des boucles `if` et `else` (à partir de la deuxième quête) n'est pas très intuitif et l'utilisation des boucles peut être facilement contournée.

Malgré cela, le langage reste totalement fonctionnel tout comme la `mécanique` de faim. De plus, pour ajouter un chapitre, il suffit de rédiger un fichier `JSON` dans `./src/resources/json/chapters/`, l'utilisateur n'a donc pas besoin de modifier le code source. Chacun peut donc créer sa propre histoire.