IUT Nancy Charlemagne

Université de Lorraine

2 ter Boulevard Charlemagne

54052 Nancy Cedex

Dépt. Informatique

Conception et réalisation d'un éditeur d'exercices de programmation

Rapport de stage DUT Informatique

Laboratoire lorrain de recherche en informatique et ses applications

Benjamin THIRION

Promotion

IUT Nancy Charlemagne
Université de Lorraine
2 ter Boulevard Charlemagne
54052 Nancy Cedex
Dépt. Informatique

Conception et réalisation d'un éditeur d'exercices de programmation

Rapport de stage DUT Informatique
LORIA
Campus scientifique
BP 239
54506 Vandoeuvre-lès-Nancy Cedex

Benjamin THIRON

Gérald OSTER et Martin QUINSON
Isabelle DEBLED-RENNESSON

Remerciements

Dans un premier temps, je remercie tous les enseignants de l'IUT Nancy Charlemagne qui m'ont accompagnés tout au long de mes études.

Je remercie Madame Debled-Rennesson de m'avoir conseillé ce stage et qui en tant que marraine de stage m'a suivi et conseillé durant cette période.

Je remercie Gérald Oster et Martin Quinson de m'avoir accepté en tant que stagiaire au LORIA et de m'avoir guidé dans mon projet. Ils m'ont ainsi offert une expérience enrichissante.

Je remercie Matthieu Nicolas pour son soutient et toutes les précieuses informations qu'il m'a fournies pour réussir ce projet.

Plus largement, je remercie toutes les personnes que j'ai pu croiser et qui ont contribué à faire en sorte que ces 10 semaines se passent dans une ambiance agréable et tout particulièrement Luc André, Baptiste Mounier et Théodore Lambolez.

Table des matières

Introduction

Mon stage de fin d'étude à l'IUT s'est déroulé au LORIA (laboratoire lorrain de recherche en informatique et ses applications) où mon travail a consisté en la conception et la réalisation d'un éditeur d'exercices pour le projet Programmer's Learning Machine. Durant mes 10 semaines de stage, j'ai ainsi du réalisé une interface sur l'application web du projet permettant de créer des exercices ayant divers composants comme une carte du monde, une mission décrivant les objectifs de l'exercice, ainsi que plusieurs solutions dans différents langages informatiques. La programmation s'est effectuée dans les langages Java, Scala, JavaScript, HTML5 et CSS3.

Dans un premier temps, je présenterai le LORIA. Puis je parlerai du projet Programmer's Learning Machine pour ensuite décrire ma mission sur ce projet. Dans un second temps, je parlerai du travail que j'ai réalisé. Il s'agit tout d'abord de présenter l'éditeur de mondes, puis l'éditeur de missions, l'éditeur de solutions, la sauvegarde et le chargement des exercices. Enfin, j'ouvrirai sur les améliorations qui seraient bienvenues sur l'état actuel de l'éditeur.

Présentation du LORIA et du projet PLM

Le LORIA

Présentation

Le LORIA est le laboratoire lorrain de recherche en informatique et ses applications. Il a été créé en 1997 lors de l'association du CNRS, de l'INRIA (Institut national de recherche en informatique et en automatique), et des universités de Nancy. Il est l'un des plus grands laboratoires lorrains. Sa mission est la recherche fondamentale et appliquée dans le domaine de l'informatique. Il se compose de 30 équipes structurées en 5 départements, représentant un total de plus de 450 personnes.

Environnement de travail

Lors de mon stage, j'ai intégré l'équipe COAST du département Réseaux, systèmes et services. J'ai travaillé sous la responsabilité de Gérald Oster et Martin Quinson, chercheurs au LORIA. Je partageais un bureau avec trois autres personnes : Matthieu Nicolas, un ingénieur récemment diplômé de TELECOM Nancy, Luc André, qui prépare une thèse, et Théodore Lambolez, un autre stagiaire de l'IUT. J'avais à ma disposition un espace de travail constitué d'un bureau et d'un ordinateur. L'environnement était agréable, calme, bénéficiant d'une bonne clarté, et donc propice à la concentration pour mon travail. Il y avait possibilité de restauration sur place. L'ambiance de travail était sérieuse et décontractée. Mes horaires de travail était de 9 heures 30 jusqu'à environ 18 heures, avec une pause déjeuner à 12 heures et une pause café vers 15 heures.

La Programmer’s Learning Machine

Présentation

La Programmer’s Learning Machine est un environnement permettant l’apprentissage de la programmation et de l’algorithmique à travers différents exercices. Le logiciel est développé depuis 2008 par Martin Quinson et Gérald Oster, deux chercheurs au LORIA. L’origine du projet est le besoin d’enseigner la programmation à l’école TELECOM Nancy, où Martin Quinson et Gérald Oster enseignent, à des étudiants n’ayant pas encore de notion dans ce domaine. Il a ainsi été lancé comme projet de réaliser un logiciel destiné à ces étudiants, qui permettrait un apprentissage ludique et attrayant de la programmation, et qui puisse se faire avec une certaine autonomie.

Le logiciel propose plus de 190 exercices organisés dans différentes leçons. Les exercices se constituent d’un monde, d’un éditeur de code, et d’un texte contenant la mission pour réussir l’exercice. L’image 1 montre l’affichage d’un exercice au sein de PLM. On peut y voir la mission à gauche, et le monde à droite. L’onglet *SourceCode* permet d'écrire le code solution.

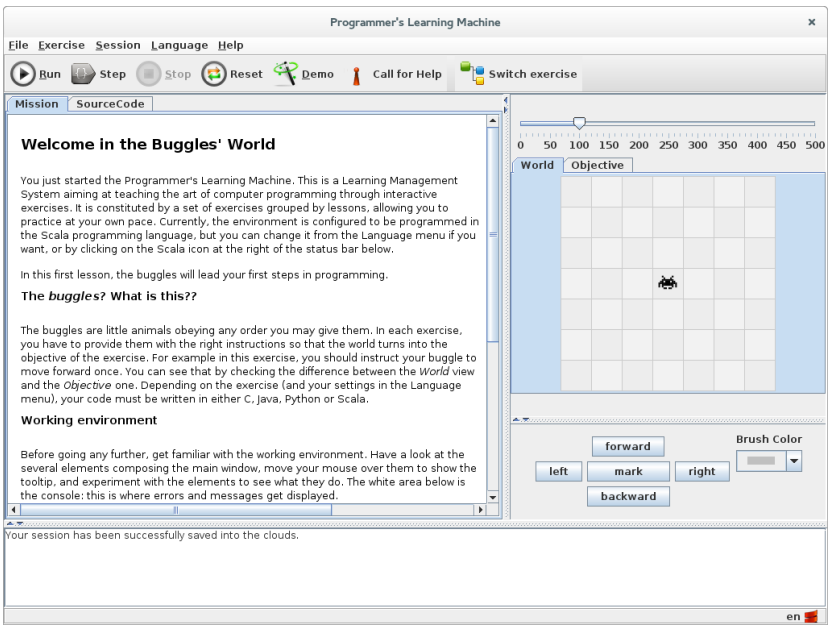

Image 1 : Un exercice sur la Programmer’s Learning Machine

Les mondes sont pour la plupart des mondes graphiques, que l'élève va devoir modifier. Chacun de ces mondes possède une ou plusieurs entités. L'élève doit interagir avec ces entités en écrivant un code informatique utilisant les différentes méthodes disponibles. Le but est pour l'élève d'écrire un code modifiant le monde à travers les entités afin d'arriver à une solution qui correspond à une certaine configuration du monde, le monde objectif. L'élève peut à tout moment comparer le monde résultant de son code à celui du monde objectif.

Les mondes semblables, appartenant à la même famille, sont regroupés dans différents univers. On trouve par exemple l'univers des Buggles. Dans cet univers, les entités sont les Buggles. Ils se déplacent dans une grille et peuvent interagir de différentes manières avec leur environnement, par exemple, en ramassant des objets. L'image 2 montre un monde de type Buggle. Le Buggle se trouve sur le case centrale de la ligne du bas. Il peut se déplacer dans la grille, en ramassant les objets et en évitant les murs.

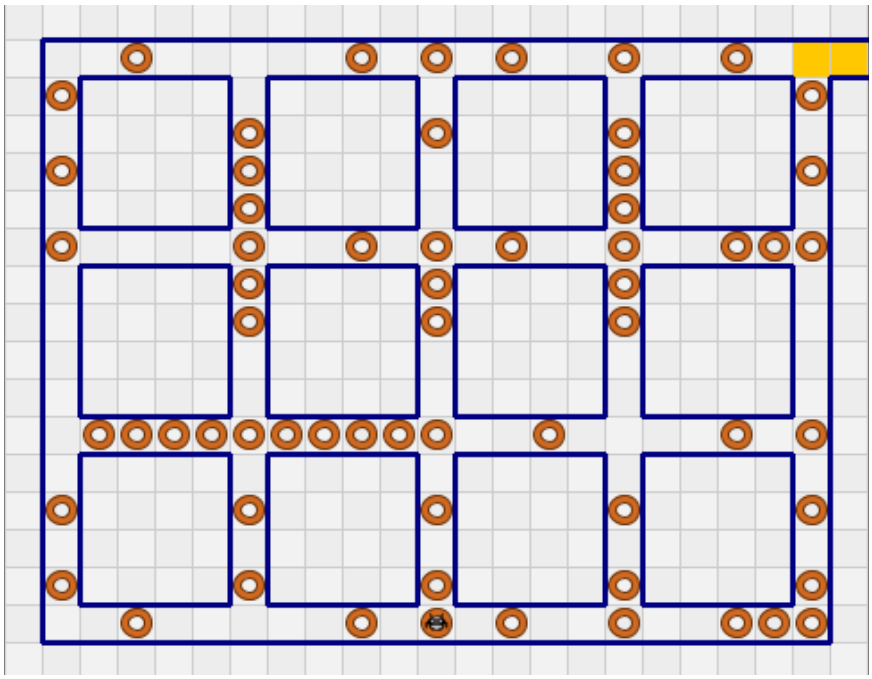

Image 2 : Un monde de type Buggle

Le code source visant à atteindre le monde solution peut être écrit dans différents langages de programmation. Le Java, le Scala, le C est le Python sont ceux disponibles actuellement, d'autres devraient également faire leur apparition. La Programmer's Learning Machine est capable de cacher certaines parties du code indispensables au bon fonctionnement du

programme, mais qui ne sont pas forcément liées à l'exercice en lui-même ou confrontent l'élève à des notions trop avancées pour le niveau auquel il est.
Enfin, chaque exercice s'accompagne d'une mission qui décrit le but de l'exercice et donne des indications à l'élève pour y parvenir. Les missions ainsi que l'interface de la Programmer's Learning Machine sont disponibles en plusieurs langues.

La nouvelle version de la Programmer's Learning Machine

La Programmer's Learning Machine est développée en Java. Les utilisateurs doivent télécharger un client lourd donnant l'accès aux exercices. Il a été décidé de passer de ce client lourd à une architecture 3 tiers. Les trois couches sont les suivantes : la partie client qui devient une application web, qui sera en interaction avec la partie serveur elle même en interaction avec la dernière partie, le coeur de la Programmer's Learning Machine.
L'application web est développée dans différents langages. Le HTML5 est utilisé pour la déclaration des éléments de l'interface. Le CSS3 et le framework Foundation pour l'apparence graphique. Enfin, le JavaScript pour l'interaction avec l'utilisateur, avec l'aide du framework AngularJS.
AngularJS permet notamment de lier la vue et le modèle de manière à ce que chaque changement dans le modèle soit appliqué à la vue et inversement. La vue et le modèle sont ainsi parfaitement synchronisés. Par exemple, on peut lier la valeur d'un champ de texte <input> à une variable dans le modèle de manière extrêmement simple :

```
<input type="text" name="nom" ng-model="name">
```

Cela aura pour effet de lier ce champ de texte à la variable "name" du modèle. ng-model est un attribut n'existant pas dans le HTML. AngularJS permet en fait d'étendre le langage HTML afin d'en modifier le comportement. Les nouveaux attributs et balises sont nommés directives. AngularJS permet de définir ses propres directives et d'utiliser celles déjà définies dans le framework. Avec ce système, on incite à manipuler le DOM dans les directives et non dans le code correspondant à la logique de notre application, ce qui permet un code plus propre et plus

facile à maintenir. AngularJS facilite donc le développement d'application web dynamique et monopage.
Le serveur est développé à l'aide du Play Framework. Il s'agit d'un framework permettant le développement rapide d'une application dans le langage Java ou Scala. Dans le cas de la Programmer's Learning Machine, c'est le Scala qui a été choisi. Le client et le serveur communiquent à l'aide du protocole WebSocket, permettant une communication bidirectionnelle, ainsi que l'envoie de données au client sans que celui ci n'ait à les demander au serveur. Le client peut par exemple envoyer le code l'élève au serveur et attendre le résultat.
Enfin, l'ancienne version de la Programmer's Learning Machine a été modifiée afin de pouvoir être utilisée par le serveur. L'interface graphique a d'abord été supprimée. Le serveur communique avec cette version pour accéder et modifier les données, par exemple en lui faisant exécuter le code de l'élève. A chaque fois que le monde est modifié, une opération décrivant la modification est générée et communiquée au serveur. Le serveur envoie les opérations au fur et à mesure au client, qui modifie le monde en conséquence.

Ma mission

Besoin de créer des exercices plus facilement

Pour ajouter de nouveaux exercices, il faut créer plusieurs fichiers à la main, puis les rassembler dans un package Java. Les différents fichiers correspondent à la classe principale de l'exercice initialisant le monde (en le chargeant à partir d'un fichier .map ou en le créant à partir d'une nouvelle instance de la classe BuggleWorld), à la mission au format HTML, et au différents fichiers sources correspondant à la solution exprimée dans les langages correspondant.
Il devrait être possible de créer de nouveaux exercices pour PLM de manière plus simple et intuitive. C'est pourquoi il a été décidé d'ajouter un éditeur de mondes et un éditeur de missions à la Programmer's Learning Machine. Ces deux éditeurs existent sur l'ancienne version de PLM, mais ne sont pas implémentés dans la version web. De plus, il n'y a aucun

éditeur de solutions existant, et les éditeurs sont dans des interfaces séparées. On pourrait imaginer les rassembler pour avoir un véritable éditeur d'exercices.

Réalisation d'un éditeur d'exercices

Mon objectif était de concevoir et réaliser un éditeur d'exercices pour la nouvelle version de PLM. Un tel éditeur doit permettre de créer les nouveaux exercices avec leur monde, leur mission et leurs codes solutions. Cet objectif a été divisé en plusieurs étapes.

Tout d'abord, la création du monde. Durant mon stage, je me suis concentré sur la réalisation d'un éditeur de mondes de type Buggle. L'objectif était de d'avoir une zone pour visualiser le monde et différentes commandes pour ajouter ou supprimer des éléments à ce monde. Il devait également y avoir un tableau des propriétés du monde (comme la largeur ou la hauteur de la grille), éditable. Il existait déjà un éditeur de monde dans la version Java. J'ai pu m'inspirer de cette version pour définir les fonctionnalités et l'interface de la version web.

L'image 3 montre l'éditeur de monde de l'ancienne version de PLM. On y voit la visualisation du monde à gauche, la barre des commandes disponibles en haut, et le tableau des propriétés du monde à droite.

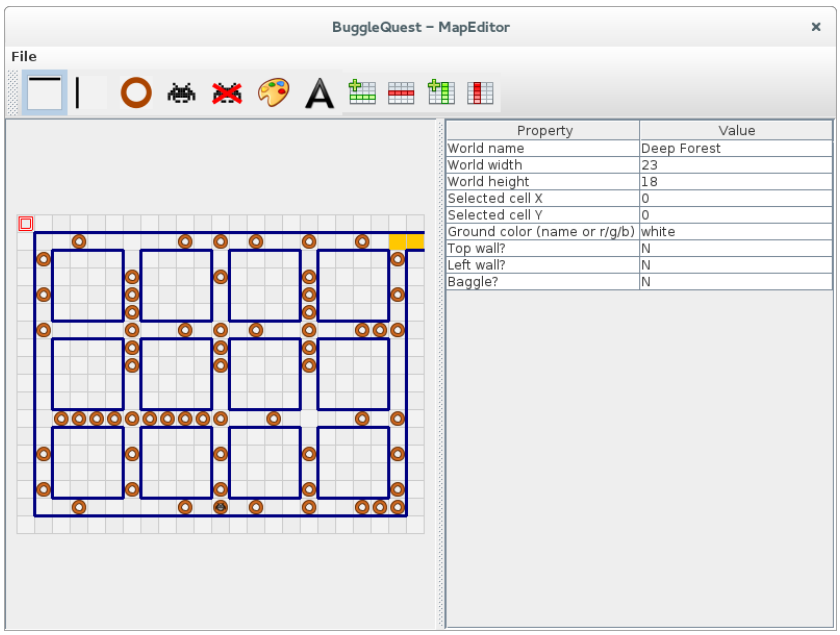

Image 3 : Éditeur de mondes de l'ancienne version de PLM

Le seconde objectif était la création de la missions. Le but est de pouvoir écrire les consignes de l'exercice en HTML. Des balises personnalisées permettent d'écrire et d'afficher la mission selon les différents langages de programmation disponibles dans PLM. Une zone doit permettre de voir en temps réel le rendu de la mission en cours de création, selon les langages sélectionnés. Un éditeur de missions existait également sur l'ancienne version de PLM. L'image 4 montre cet éditeur. On peut y voir la mission en cours de rédaction à gauche, et le rendu à droite. Les cases à cocher en haut à droite permettent de choisir les langages pour lesquels la mission doit être affichée.

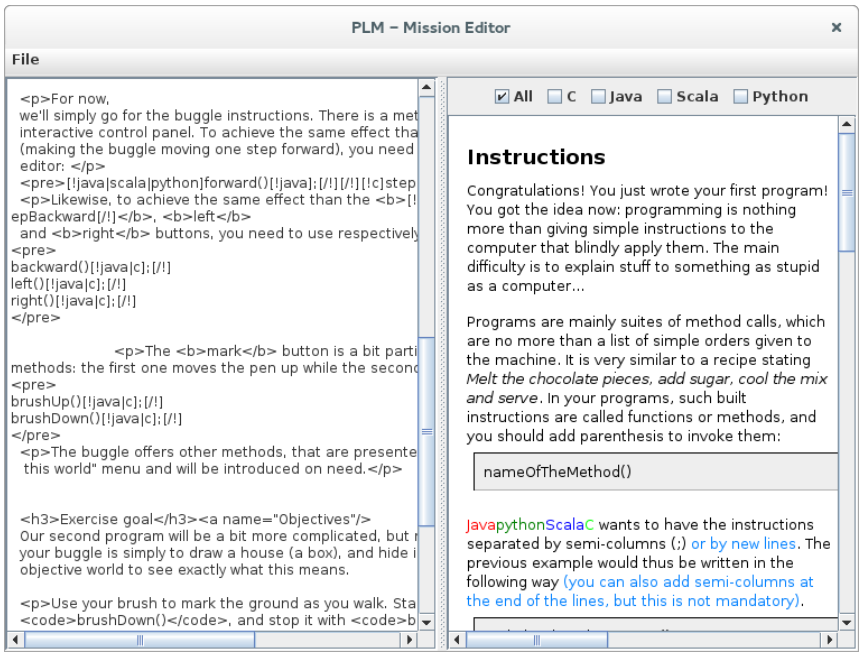

Image 4 : Éditeur de missions de l'ancienne version de PLM

Mon troisième objectif était la création de la partie solution de l'éditeur, où l'on doit pouvoir écrire le code solution, avec une partie définie qui sera affichée à l'élève et une autre qu'il devra écrire de lui-même. On doit pouvoir écrire la solution dans les différents langages de programmation de PLM et pouvoir observer le résultat sur le monde que l'on a associé à l'exercice.

Le dernier objectif concernait le chargement et la sauvegarde d'exercices. Premièrement, on doit pouvoir démarrer la création d'un nouvel exercice à partir des données d'un exercice existant. On doit ainsi récupérer les mondes, la mission, et les codes solution constituant l'exercice afin d'initialiser l'éditeur avec ces données. Ensuite, on doit pouvoir sauvegarder

l'exercice sur le serveur. Mon objectif était d'exporter les mondes dans un format déjà implémenté dans PLM, la mission en HTML, et les solutions dans des fichiers sources correspondant aux langages de programmation.

Présentation du travail réalisé

Installation des outils, méthode de travail et apprentissage sur AngularJS

La première partie de mon travail, sur les trois premiers jours, a consisté en l'installation des outils de développement et à l'apprentissage du framework AngularJS. Le premier jour a été consacré à AngularJS. Ce framework a une orientation et des concepts différents de la bibliothèque JavaScript JQuery, que l'on utilisait à l'IUT. Cela a donc nécessité quelques heures de prise en main avant de pouvoir coder efficacement avec. Pour ce faire, un site proposant un cours ainsi que des exercices sur AngularJS nous a été indiqué. Une fois le cours suivi et les exercices réussis, j'ai pu continuer sur l'installation des outils de travail.

Afin de pouvoir travailler sur le projet PLM, il m'a fallu tout d'abord installer le Play Framework sur ma machine, afin de faire tourner le serveur. Ensuite, j'ai du réaliser un fork du dépôt GitHub webPLM, contenant le code du serveur et du nouveau client web. J'ai également créé un fork du dépôt PLM contenant le code de l'ancienne version de PLM, dont la branche library correspond à sa version utilisée par le serveur.

La dernière étape a consisté à la mise en place d'un reporting quotidien sur les travaux effectués. Il se présente sous la forme d'un fichier texte utilisant Org mode pour l'éditeur de texte GNU Emacs. Org mode permet de prendre des notes et réaliser des listes "TODO", avec un système de cases à cocher. Chaque jour, j'ai créé une nouvelle entrée où j'ai indiqué le travail réalisé et terminé sous forme de liste. Je mettais aussi les questions que je me pose, par exemple sur des problèmes rencontrés. Enfin, j'indiquais le travail que je prévoyais de faire sous forme de listes "TODO". Une fois le reporting du jour écrit, je le mettais en ligne sur un dépôt GitHub afin de le rendre visible. Ce reporting permet de mieux s'organiser en ayant une bonne visibilité du travail réalisé et planifié.

Une fois tout ceci mis en place, j'ai pris un peu de temps pour étudier le fonctionnement général de PLM. J'ai également testé les deux éditeurs existants sur le client lourd. Finalement, j'ai pu commencer mon travail sur l'éditeur de monde.

L'éditeur de mondes

La première partie de l'éditeur d'exercices que j'ai développé est l'éditeur de mondes. L'éditeur de monde doit permettre la création et la modification des mondes associés à l'exercice.

Dans l'application web de PLM, les zones principales de l'application sont regroupées dans des contrôleurs AngularJS. Il existe par exemple un contrôleur pour la page d'accueil et un autre contrôleur pour l'affichage d'un exercice à résoudre. Ils sont chacun liés à un template, c'est-à-dire la vue au format HTML. Le contrôleur a pour but principal d'exposer les données du modèle à cette vue, et d'initier ainsi le mécanisme d'AngularJS permettant la synchronisation modèle / vue. La première chose que j'ai faite est donc de créer un contrôleur nommé editor, dont le rôle était de gérer l'éditeur.

Une fois le contrôleur créé, la première étape consistait à créer une zone de visualisation du monde. Cette zone devait permettre l'affichage de la grille et de tous les éléments qu'elle contient. Une telle zone d'affichage existait déjà dans la version web, dans la partie de résolution des exercices. Elle est définie dans la vue par l'élément HTML5 canvas. Cet élément représente sur la page une zone dans laquelle il est possible de dessiner en JavaScript. Afin d'y dessiner le monde, un service AngularJS canvas était à ma disposition. Les services AngularJS sont des objets JavaScript donnant accès à différentes méthodes. Dans le cas du service canvas, on l'initialise en indiquant l'élément canvas qui va afficher le monde, sa largeur, sa hauteur, et une fonction de dessin. La fonction de dessin est définie dans BuggleWorldView, un autre service donnant accès à ce qui concerne l'affichage d'un BuggleWorld. Ensuite, pour afficher un monde on donne au service canvas l'objet BuggleWorld, représentant notre monde actuel. La mise à jour automatique de la vue et du modèle par AngularJS n'était pas possible sur l'élément canvas, mais une méthode update du service canvas permet de mettre à jour la vue dès que cela devient nécessaire.

J’ai donc utilisé ce code existant dans l’éditeur. J’ai créé une méthode initEditor dans le contrôleur qui s'exécute au lancement de l’éditeur. Elle créé tout d’abord un nouveau monde, vide. J’ai du modifier le constructeur de BuggleWorld afin qu’il créé un monde vide si il est appelé sans paramètre. En effet, il était jusqu’à présent prévu de l’utiliser en créant un nouveau BuggleWorld à partir d’un monde au format JSON communiqué par le serveur. La méthode initEditor s’occupe ensuite de l’initialisation du canvas. Ainsi, un monde vide s’affiche au lancement de l’éditeur.

La prochaine étape consistait à pouvoir sélectionner une cellule dans la grille. Pour ce faire, j’ai d’abord créé une méthode selectCell dans le contrôleur editor. Lorsque on clique sur le canvas, cette méthode est appelée à l’aide de la directive ngClick d’AngularJS. Elle est l’équivalent de onclick en JavaScript, qui permet l’appel d’une fonction spécifique lors du clic sur un élément. Elle reçoit en paramètres les données sur l’événement du clic, et notamment la position du curseur sur la page. Afin de déterminer la case sélectionnée, un petit calcul est nécessaire. Il consiste à soustraire le décalage en pixels de l’élément canvas par rapport aux coordonnées d’origine du document à la coordonnée du clic, puis de diviser par la hauteur en pixels ou la largeur en pixels d’une cellule afin d’obtenir la coordonnée X ou la coordonnée Y voulue. Je mets à jour un attribut isSelected dans la cellule ainsi sélectionnée. Enfin, j’ai mis à jour la méthode draw de dessin afin de faire apparaître la cellule actuellement sélectionnée.

Une fois qu’il fut possible de sélectionner les cellules, il me fallait débuter la création des boutons correspondant aux différentes commandes disponibles. J’ai créé en premier les boutons pour ajouter les murs qui me semblaient être les plus simples. Un clic sur un de ces boutons change la variable qui correspond à la commande actuelle dans l’éditeur. Ensuite, dans la méthode selectCell, j’appelle une autre méthode en fonction de la commande actuelle. Dans le cas des murs, il suffit de créer une méthode qui change l’attribut hasTopWall ou hasLeftWall dans l’objet BuggleWorldCell correspondant à notre cellule de true à false ou de false à true.

Après les murs, je me suis occupé de la commande visant à ajouter un Buggle. Cette méthode est un peu plus compliquée. En effet, la liste des Buggles du monde est un objet JavaScript, attribut de l’objet BuggleWorld, dont les clés sont les noms des Buggles et les valeurs un objet représentant le Buggle. Il faut ainsi parcourir cet objet pour chercher si un Buggle existe déjà sur la case demandée, en regardant les coordonnées X et Y de chaque Buggles. Si c’est

le cas, le Buggle est sélectionné. Dans le cas contraire, le Buggle est ajouté. La commande pour supprimer un Buggle marche de manière similaire. On recherche si un Buggle existe sur la case, et si oui, on le supprime.

Une fois ces premières commandes réalisées, j'ai débuté la création du tableau des propriétés du monde. Dans un premier temps ce tableau servait juste à afficher les propriétés générales du monde comme la largeur et la hauteur. AngularJS permettait d'avoir ces valeurs automatiquement synchronisées avec celles du monde. J'ai ensuite continué l'ajout de commandes et de propriétés dans le tableau.

La commande Color permet l'ajout d'une couleur de fond sur les cases du monde. Lorsqu'on clique sur le bouton Color, une liste déroulante présente les couleurs proposées par défaut. Cette liste se trouve dans un service Color que j'ai créé. Il se charge de donner la liste des couleurs disponibles. Lorsque l'utilisateur choisi une couleur par défaut, le service se charge de convertir cette couleur en tableau RGBA (Red Green Blue Alpha), le format utilisé pour représenter la couleur d'une case. De plus, dans la liste déroulante, une option permet de choisir d'utiliser une couleur personnalisée. Si cette option est choisie, une fenêtre s'affiche à l'écran demandant à l'utilisateur d'entrer un code RGB sous la forme d'une chaîne de caractères sous la forme "rouge/vert/bleu". À l'aide d'une expression régulière, j'ai créé une méthode dans le service Color permettant d'extraire les trois valeurs de couleur et les convertir dans en un tableau RGBA. Lorsque une cellule est sélectionnée, on voit sa couleur dans le tableau des propriétés. On peut également changer la couleur directement depuis le tableau. L'utilisateur a le choix d'entrer un nom de couleur ou une couleur sous la forme "rouge/vert/bleu". Si jamais l'utilisateur rentre une valeur rouge/vert/bleu qui correspond à celle d'une couleur par défaut, j'affiche le nom de la couleur à la place. Enfin, Martin Quison m'a suggéré d'ajouter la possibilité de choisir une couleur à l'aide d'une fonctionnalité pipette, qui permettrait de cliquer sur une cellule pour en récupérer la couleur. Il m'a pour cela suffit de récupérer la valeur de l'attribut couleur de la cellule présente aux coordonnées du clic. Enfin, il devait être possible de visualiser la couleur actuellement choisie pour la commande couleur. Nous ne savions au départ pas vraiment où afficher cette indication, car l'interface était déjà assez chargée. J'ai alors choisi d'utiliser un petit rond coloré qui apparaît par dessus le bouton Color si la commande actuellement sélectionnée est la commande couleur. AngularJS propose les directives ngHide et ngShow afin d'afficher ou nom un bloc

HTML en fonction de la valeur d'une variable, valeur de la variable commande en l'occurrence.

La prochaine commande réalisée est la commande texte. Lorsqu'on clique sur une cellule avec cette commande de sélectionnée, une fenêtre apparaît et demande le texte que l'on souhaite inscrire dans la cellule. J'ai également fait en sorte que l'on puisse changer le texte de la cellule actuelle directement dans le tableau des propriétés, à la manière de la propriété couleur.

Les dernières commandes concernent l'ajout et la suppression de lignes et de colonnes. Pour réaliser ces commandes, j'ai ajouté une méthode addColumn et une méthode addLine aux objets BuggleWorld. Ces méthodes permettent d'ajouter (paramètre nb positif) ou supprimer (paramètre nb négatif) un certains nombre de cellules à partir de la cellule indiquée. Cela revient en premier lieu à supprimer ou ajouter des colonnes ou des lignes dans un tableau JavaScript, le monde étant un tableau à deux dimensions. La méthode splice en JavaScript permet de s'en sortir, car elle a justement pour but de supprimer ou ajouter des entrées dans un tableau. Une fois les cellules supprimées du tableau, il m'a fallu écrire le code pour mettre à jour les cellules dont les coordonnées changent lors de la modification. Mais il ne fallait pas oublier de mettre à jour l'objet contenant la liste des Buggles. J'ai donc du parcourir ce tableau afin de détecter les Buggles à décaler en cas d'ajouts de cases. Il fallait également détecter les Buggles à supprimer en cas de suppressions de cases. Une première boucle recherche dans l'objet les Buggles dont les coordonnées ne sont plus à jours ou ceux qui doivent être supprimés, et les garde en mémoire dans une variable. Une seconde boucle permet ensuite de les mettre à jour. Il est également possible de modifier la dimension du monde directement depuis le tableau des propriétés. Pour réaliser cette fonctionnalité, j'ai pu réutiliser mes méthodes addColumn et addLine en spécifiant comme ligne ou colonne de départ celle à l'extrémité du monde et comme nombre d'ajout ou de suppression la nouvelle dimension du monde dont on soustrait l'ancienne.

Une fois les commandes et le tableau des propriétés terminés, il restait à donner la possibilité de gérer plusieurs mondes. J'ai donc créé un tableau de mondes dans le contrôleur. À partir de ce tableau, une liste déroulante se met automatiquement à jour en fonction des mondes disponibles. Cette liste permet également de passer d'un monde à l'autre. La directive ngOptions d'AngularJS permet de réaliser une telle liste déroulante. Appliquée à un élément HTML select, elle va créer un élément option dans ce select pour chaque entrée dans le

tableau. À côté de la liste, un bouton permet d'ajouter un monde en créant un nouvel objet BuggleWorld et en l'insérant dans le tableau des mondes. Le bouton supprimer demande une confirmation, puis supprime un monde en l'enlevant du tableau. Lorsque l'utilisateur change de monde, on indique également au canvas le nouveau monde à dessiner.

L'image 5 montre l'apparence finale de l'éditeur de mondes. On peut y voir en haut les boutons donnant accès aux différentes commandes. Les commandes proches comme ajouter un mur en haut ou ajouter un mur à gauche sont regroupées dans des boutons qui affichent un menu déroulant au clic. Juste en dessous se trouve le sélecteur de monde, ainsi que les boutons de création et de suppression. En dessous, à gauche, on voit l'affichage de l'état actuel du monde en cours de création. Enfin, sur la droite, se trouve le tableau des propriétés. Le tableau de gauche permet d'éditer le monde ainsi que les propriétés de la cellule sélectionnée. Celui de droite permet d'éditer le Buggle actuellement sélectionné.

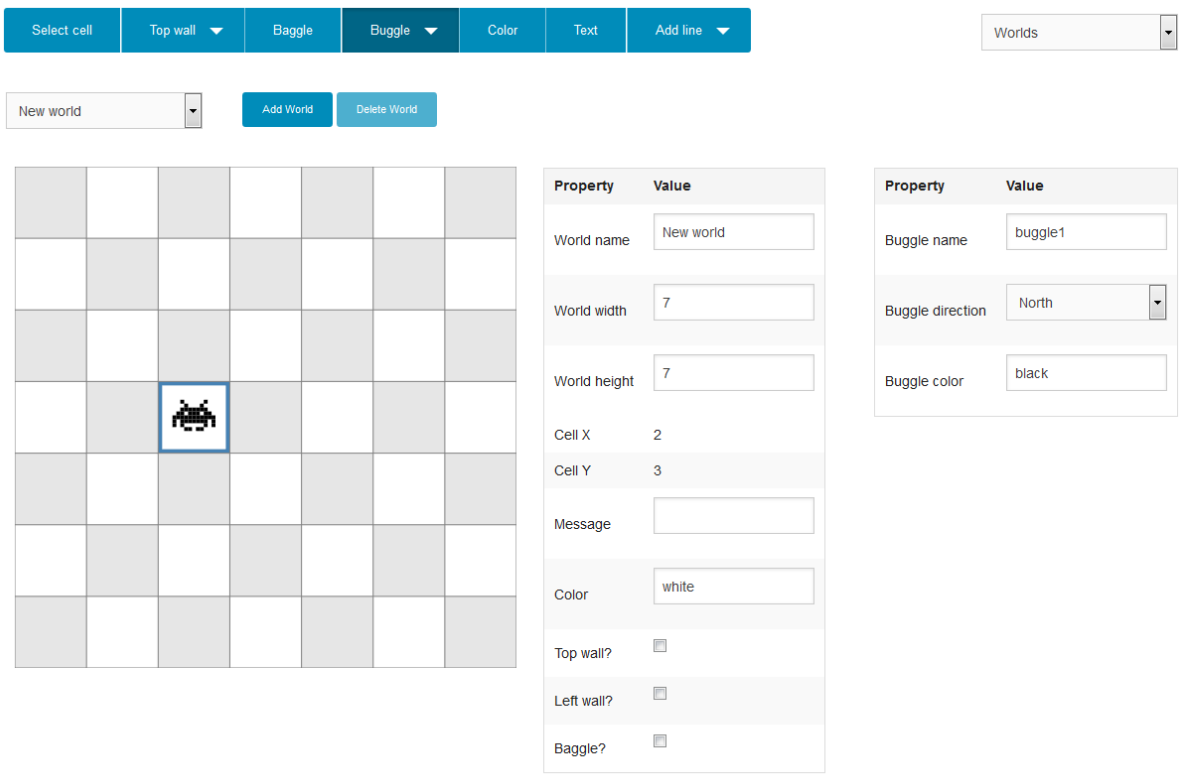

Image 5 : Interface finale de l'éditeur de mondes

L’éditeur de missions

Après l’éditeur de monde, la seconde fonctionnalité de l’éditeur d’exercices est l’éditeur de missions. Il s’agit de rédiger la mission que verra l’élève lors de la résolution d’un exercice. Elle décrit l’exercice, en donne l’objectif et des indications sur la manière d’y parvenir.

La première étape a été de créer une large zone de saisie dans un élément HTML textarea afin de saisir le texte de la mission. La mission est saisie dans le langage HTML. Des balises personnalisées permettent en plus l’affichage du langage courant (balise `[!thelang]`), ou un affichage différent selon le langage dans lequel l’élève a choisi de résoudre l’exercice. Par exemple, en écrivant `[!java|scala|]code[/!]`, le texte “code” ne sera affiché que si le langage choisi est le Java ou le Scala. En effet, les différences entre les langages font que la mission doit être adaptée à chacun.

La seconde étape a été d’afficher la mission avec toutes les balises filtrées dans une zone de visualisation à droite de la zone de saisie. La visualisation devait se faire en temps réel. Au niveau du code existant me permettant d’arriver à mon but, il y a dans PLM une méthode filterHTML permettant de filtrer la mission.

J’ai d’abord créé des cases à cocher correspondant au différents langages dans lesquels on peut afficher la mission. Une case “All” permet d’afficher la mission dans tous les langages à la fois. Lorsque on coche une case, un attribut qui représente la volonté d’afficher la mission dans le langage correspondant passe à la valeur true. Ensuite, à l’aide de la directive AngularJS ngChange, j’envoie la mission au serveur à travers la WebSocket à chaque fois qu’elle est modifiée. Cela pouvant représenter un grand nombre de requêtes, j’utilise une option de la directive ngChange permettant de n’envoyer la mission que si l’utilisateur fait une pause de 300 ms dans sa saisie. Cela permet d’alléger le nombre de requêtes tout en gardant la sensation d’une mise à jour en temps réel. Une fois la modification récupérée sur le serveur, j’utilise la méthode filterHTML qui va filtrer la mission. Une fois filtrée, la mission est renvoyée au serveur dans la WebSocket. Lorsque le serveur reçoit la mission, j’affiche le HTML récupéré à l’utilisateur, en utilisant le module AngularJS ngSanitize afin que les scripts que pourrait avoir écrit un utilisateur soient désactivés.

L’image 6 montre l’interface finale de l’éditeur de missions. À droite se trouve la zone de saisie. En haut à gauche les cases à cocher pour choisir le langage dans lequel affiché la mission. Enfin, à gauche en dessous des cases, le rendu final de la mission filtrée.

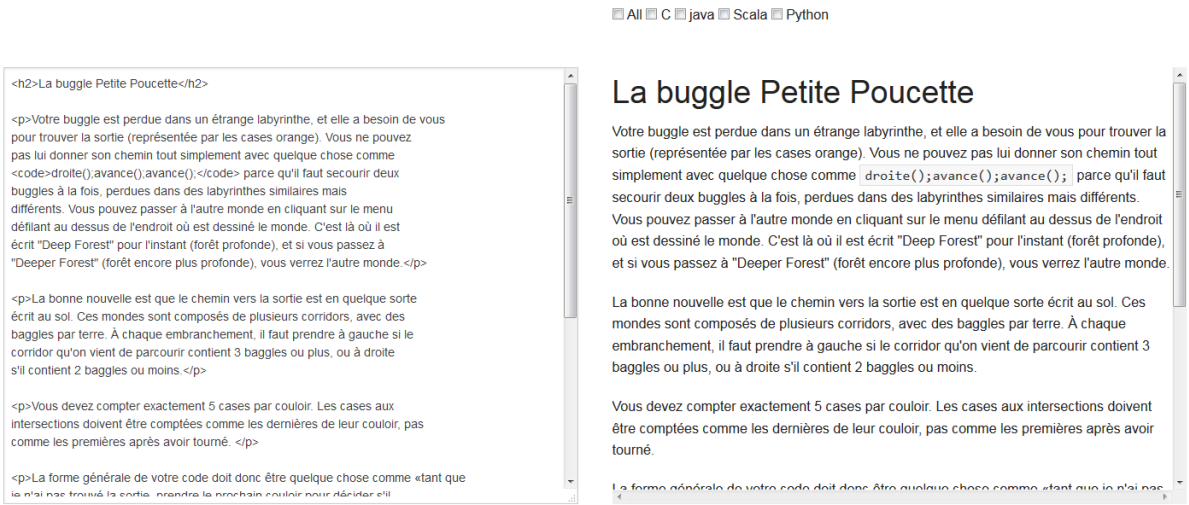

Image 6 : Interface finale de l’éditeur de missions

L’éditeur de solutions

La troisième partie de l’éditeur est l’éditeur de solutions. Le but est de pouvoir écrire la solution dans chacun des langages et de l’exécuter sur le monde créé afin de voir le résultat. La fonctionnalité étant assez proche de celle consistant à l’élève d’entrer sa solution dans la partie résolutions des exercices de l’application, j’ai choisi de me baser sur le code existant.
Mon but était alors d’envoyer la solution au serveur comme si c’était l’élève qui l’envoyait. Cela nécessitait donc un exercice ayant pour mondes ceux que l’on venait de créer. Pour ce faire, j’ai créé un “faux exercice” dans le coeur de PLM. Cet exercice ne sera pas affiché aux élèves. Il ne possède au départ un monde et des solutions par défaut.
Une fois ce faux exercice créé, j’ai commencé la réalisation de l’interface sur l’application web. J’ai dans un premier temps inséré dans ma vue HTML les directives personnalisées qui avaient été créées pour la partie résolution des exercices. Ces directives permettent d’afficher le sélecteur de langage, l’éditeur de code, et les contrôles d'exécution. Les contrôles d'exécution permettent de voir étape par étape l’évolution du monde après qu’on code lui ait été soumis.

Ensuite, j'ai copié les parties du code du contrôleur exercice qui permettent d'utiliser les directives. Cela comprend notamment la gestion du code écrit dans l'éditeur de code, l'envoie de la solution au serveur, et l'affichage étape par étape de la solution sur les différents mondes. Malheureusement, cela faisait beaucoup trop de codes commun entre les deux contrôleurs, plus de 300 lignes. Il a donc fallu réfléchir à une solution pour mettre se code en commun. Après quelques temps de réflexions, nous avons décidé d'utiliser les services AngularJS. En effet les services permettent notamment la mise en commun de code, car se sont des objets partagés entre les contrôleurs. Le code commun concernant l'éditeur de code a été réécrit dans un service nommé ide, et celui concernant la gestion des mondes et l'affichage de la solution étape par étape, dans un service worlds. Réaliser cette séparation m'a pris un peu de temps, car il a fallu adapter le code des deux contrôleurs afin qu'ils aillent chercher les fonctionnalités dans les services.

Je me suis ensuite consacré au changement à effectuer au serveur afin qu'il puisse renvoyer le résultat à l'application. Le serveur reçoit à la fois les mondes que l'on a créé, ainsi que les codes solutions écris. La première étape est d'associer les mondes au faux exercice. Les mondes sont reçus au format JSON. J'ai donc du créer des méthodes permettant de convertir un monde JSON vers un objet Java BuggleWorld. Cela comprend la création de méthodes pour transformer les cellules en BuggleWorldCell et les Buggles en SimpleBuggles. Play Framework propose un outil pour lire un fichier JSON. Le JSON récupéré est stocké dans une variable de type JsObject. On peut ensuite demander la conversation des types fréquemment rencontrés comme les nombres ou les chaînes, dans des variables du type équivalent en Scala. Une fois toutes les valeurs de bases récupérées, on les utilise pour créer les objets Java voulus. Enfin, la méthode setWorlds permet de définir les mondes de l'exercice, en lui donnant en paramètre un tableau contenant les mondes voulus.

Il ne reste plus qu'à exécuter le code de la solution avec la méthode runExercise. Les opérations correspondant à chaque étape de la modification du monde sont ainsi calculées par PLM, et envoyé au fur et à mesure à l'application web. Le code mis en commun dans le service worlds récupère ces opérations et met à jour le monde en conséquence.

L'image 7 présente la version finale de l'interface de l'éditeur de solutions. On peut voir à gauche l'éditeur de code, avec l'onglet "Solution code" pour écrire la solution, et l'onglet "API" qui indique les méthodes disponibles. À côté des onglets se trouve le bouton pour sélectionner la langage pour lequel on souhaite écrire le code solution. On voit à gauche la

liste déroulante pour sélectionner le monde, la visualisation du monde, le bouton executer, et les contrôles pour afficher la solution étape par étape.

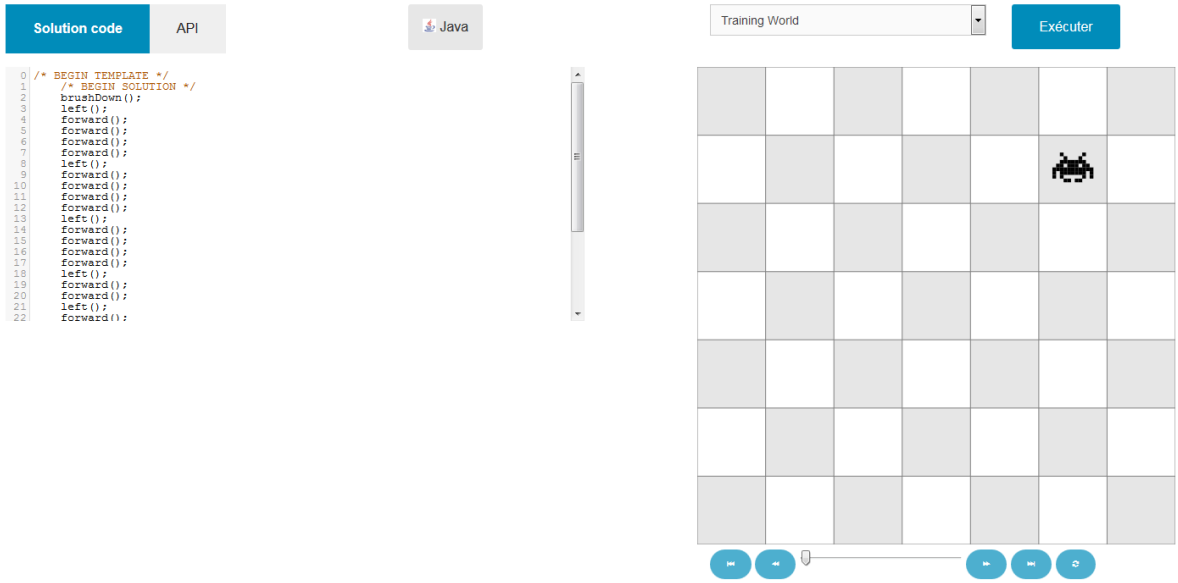

Image 7 : Interface finale de l'éditeur de solutions

Chargement et sauvegarde d'exercices

Une fois toutes les fonctionnalités de création réalisées, il restait à donner la possibilité de charger l'éditeur avec les données d'un exercice existant, ainsi que la possibilité de sauvegarder l'exercice créé.

En ce qui concerne le chargement, le serveur possède déjà la possibilité d'envoyer toutes les données d'un exercice. Cette fonctionnalité est utilisée pour charger un exercice dans la partie résolution des exercices. Pour charger un exercice dans l'éditeur, j'ai donné la possibilité d'indiquer l'identifiant de l'exercice dans l'URL. Si l'éditeur est chargé sans paramètre dans l'URL, je l'initialise avec un monde vide. Dans le cas contraire, je demande au serveur de m'envoyer les données de l'exercice. Une seule modification a du être apportée à la version de PLM utilisée par le serveur. Il s'agit de l'ajout d'une méthode getRawMission permettant d'obtenir le fichier HTML brut de la mission. Une fois que le serveur a envoyé les données et qu'elles sont réceptionnées par l'application web, j'utilise les données pour initialiser le monde, la mission, et les codes solutions.

Pour la sauvegarde, j'ai choisi de sauvegarder les exercices dans des dossiers portant leurs noms. En effet, les exercices implémentés dans PLM étant sous forme de package Java directement intégré au code du logiciel, qui reste toujours en cours d'exécution sur le serveur, il aurait été très difficile de les intégrer directement dedans. À la place, je sauvegarde dans le dossier les données de l'exercice dans les même fichiers que l'on retrouve dans un exercice packagé. Cela nécessite un fichier .map par monde, un fichier code source par solution, un fichier HTML pour la mission, et un fichier correspondant à la classe principale de l'exercice. La classe principale de l'exercice se charge de charger les mondes et de les initialiser.

Dans un premier temps, j'ai créé un champ de texte dans lequel l'utilisateur entre le nom voulu pour l'exercice. Puis j'ai ajouté un bouton qui permet de déclencher la sauvegarde. Au clic, l'application envoie au serveur toutes les données concernant les mondes, la mission, et les codes solution.

Dans un second temps, j'ai ajouté au serveur le code pour sauvegarder l'exercice. La première étape est de convertir les mondes en objet BuggleWorld, en utilisant les méthodes de conversion que j'avais précédemment créées. Ensuite, à l'aide de la méthode walkFileTree de la classe Files de Java, je supprime les données contenues dans le dossier portant le nom de l'exercice, si ce dossier existe, car elles ne sont plus à jour. La prochaine étape consiste à créer la classe principale. Pour cela, j'ai créé un fichier sur le serveur nommé classTemplate, qui contient le code commun à la classe principale de tous les exercices. Dans le code du serveur, je charge le contenu de ce fichier et je remplace le nom de la classe par le nom de l'exercice. J'ajoute également les méthodes qui chargent les mondes, en indiquant le nom des mondes créés. Une fois ce fichier créé, je m'occupe d'écrire les fichiers mondes. Une méthode writeToFile de la classe BuggleWorld correspondait exactement à cette fonctionnalité, je l'ai donc utilisée. J'écris ensuite le contenu de la mission dans un fichier HTML. Il ne me reste plus qu'à écrire le code des solutions. La méthode getCompilableContent permet d'obtenir les fichiers sources de l'exercice courant, en l'occurence le faux exercice pour lequel j'ai renseigné les codes solutions. Il restait à modifier dans ces fichiers le nom de la classe par défaut par celui indiqué par l'utilisateur. Or la méthode getCompilableContent me permet de donner des données textuelles à remplacer avant de retourner le fichier, j'ai donc utilisé cette méthode. Une fois tous ces fichiers écris, l'exercice est sauvegardé sur le serveur.

Les améliorations possibles

En l'état actuel, l'éditeur permet de créer ses mondes, sa mission, et ses codes solutions, puis de sauvegarder le tout. Mais il reste néanmoins diverses améliorations et ajouts nécessaires ou potentiellement intéressants.

Au niveau de l'éditeur de solution, il manque la compatibilité avec les exercices dont le code solution se trouve en dehors de la méthode run de l'entité, méthode qui est appellée pour lancer la solution. En effet, sur certains exercices la solution est entièrement comprise dans la méthode run, mais sur d'autres, il peut y avoir des méthodes externes. Dans le faux exercice créé pour l'éditeur, le placement des marqueurs délimitant où le code solution de l'élève doit être inséré se trouve dans la méthode run. Il devrait être possible de les déplacer en dehors de la méthode run afin de rendre l'éditeur compatible. Cela nécessitera alors pour la personne créant la solution de mieux connaître la structure de la classe d'une solution.

Au niveau de la sauvegarde, il est possible de sauvegarder un monde mais il n'est pas possible de charger un monde sauvegardé. Il devrait être possible de demander au serveur de lire les données d'un monde sauvegardé puis de les envoyer à l'application web afin qu'elle initialise l'éditeur avec ces données.

De plus, si l'on donne un nom d'un exercice déjà utilisé pour sauvegarder un autre exercice, l'ancien sera supprimé car on considère que c'est une mise à jour de l'exercice réalisée par la même personne. Il devrait y avoir un système pour voir la liste des exercices sauvegardé et prévenir si l'on risque d'en écraser un.

Enfin, il avait été suggéré de pouvoir créer une galerie d'exercices réalisés par les élèves. Les exercices sauvegardés seraient disponibles sous forme d'une galerie un peu à part des autres exercices. Il pourrait par exemple être possible de voter pour les exercices qui nous semblent intéressant afin de rendre la galerie vraiment interactive.

Conclusion

Durant ce stage, j'ai pu contribuer à un projet qui m'a semblé très intéressant pour son but pédagogique. Je suis heureux d'avoir pu y ajouter un éditeur permettant une création plus simple d'exercices. Cet éditeur permet finalement la création de toutes les composantes d'un exercice ainsi que de sa sauvegarde.

Sur le plan personnel ce stage m'a permis d'améliorer mes connaissances en Scala et en JavaScript. J'ai pu découvrir et utiliser le framework AngularJS, qui est très intéressant et plaisant pour réaliser des applications web dynamiques de qualité. J'ai pu découvrir d'autres moyens pour réaliser un serveur d'application web à travers le Play Framework qui m'était inconnu. J'ai également appris à mettre en place un reporting sur mon travail ce qui aide beaucoup dans l'avancement de son projet. Travailler sur un logiciel comme PLM et voir son évolution vers une application web fut très enrichissant.

Ce stage m'a également permis de me faire une meilleur idée du monde professionnel et notamment celui de la recherche. J'ai pu assister à des présentations de jeunes ingénieurs qui travaillent sur des projets du LORIA, ainsi qu'à une soutenance de thèse, ce qui fut très intéressant.

Bibliographie

Sites

- AngularJS, https://angularjs.org, 13 juin 2015
- LORIA, http://www.loria.fr, 13 juin 2015
- Programmer's Learning Machine, http://www.loria.fr/~quinson/Teaching/JLM/, 13 juin 2015

Documents

- A Teaching System To Learn Programming: the Programmer's Learning Machine, http://webloria.loria.fr/~quinson/Research/Publications/2015-itiCSE-plm.pdf, 6 pages
- Programmer's Learning Machine Campagne ADT 2015, 13 pages