



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

Langage visuel pour un exerciseur

Rapport de stage DUT informatique
LORIA

Baptiste MOUNIER
47^{ème} promotion - 2015

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

Langage visuel pour un exerciseur

Rapport de stage DUT informatique
LORIA
Campus Scientifique
BP 239
54506 Vandœuvre-lès-Nancy

Auteur : Baptiste MOUNIER

Maîtres de stage : Gérald OSTER Matthieu NICOLAS Martin QUINSON
Marraine : Isabelle DEBLED-RENNESSON

Remerciements

Il me semble judicieux avant de débiter ce rapport de remercier tous ceux et celles qui m'ont permis de réaliser ce stage dans ces conditions.

Je commencerai par Matthieu Nicolas, Gérard Oster et Martin Quinson qui m'ont accepté au sein de leur projet et qui m'ont accompagné tout au long de la réalisation de ma contribution.

Je remercierai également Mme. Debled-Rennesson, ma marraine de stage qui a effectué le suivi et l'encadrement de mon stage.

De même que toute l'équipe COAST qui m'a accueilli dans leurs locaux dans la joie et la bonne humeur, éléments essentiels d'un travail efficace au quotidien, notamment Claudia-Lavinla Ignat et Luc André, et par extension le LORIA dans son ensemble.

Je n'oublierai pas l'IUT Nancy-Charlemagne et ses enseignants qui m'ont apporté, tout au long de ces deux années d'études, les capacités et les connaissances nécessaires pour réaliser ce genre de projet.

Table des matières

Remerciements	1
Introduction	4
Sujet	4
Mise en œuvre	4
LORIA	6
Laboratoire	6
Equipe COAST	6
Contexte	7
Etat du projet	7
Blockly	8
Outils, technologies et méthodes	8
Développement	10
Découverte	10
Insertion	10
Sélection.....	10
Affichage.....	11
Génération	12
Toolbox	12
Bloc	12
Liaison.....	14
Stockage	15
Traduction	17
Conclusion	19
Bibliographie.....	20
Sites web	20
Rapport.....	20

Introduction

Sujet

Le stage consiste à la réalisation de l'ajout d'un langage de programmation visuel au sein de l'application web de PLM (Programmer's Learning Machine). Le service utilise les langages Java, Python, Scala et C pour la résolution des exercices. Ce genre de langage permet d'ajouter une couche visuelle afin de faciliter la première approche vis-à-vis de la programmation. Il en existe plusieurs notamment Scratch, mais nous avons décidé d'implémenter Blockly.

Blockly est elle-même une application réalisée par Google sous licence Apache 2.0. C'est une librairie de développement pour logiciels et applications d'apprentissage de la programmation. Son utilisation est très visuelle et sous forme de « drag and drop » de blocs représentant des instructions, comme une affectation d'un entier à une variable, dans un espace de travail.

PLM est une application, récemment portée sur le web, d'apprentissage de la programmation développée conjointement par des membres des équipes COAST et VERIDIS du LORIA sous licence publique générale GNU AGPL v3.0. L'application est actuellement utilisée en école d'ingénieur, Télécom Nancy, pour une progression rapide des étudiants découvrant la programmation. Le portage est un atout d'accessibilité et de gestion. Dans la suite de ce rapport j'utiliserai « webPLM » comme terme générique pour désigner le côté web de l'application et « PLM » pour désigner le côté logiciel existant déjà avant le portage web PLM.

PLM a été développé comme un outil afin de faciliter l'apprentissage de la programmation pour une utilisation avant tout scolaire même s'il est également disponible pour un usage personnel. Le portage web améliore cette accessibilité en rendant l'application disponible partout sans téléchargement de client lourd.

Mise en œuvre

Mon travail s'est effectué en cinq grands axes que je développerai dans cet ordre dans ce dossier (réf : Annexe 1 – Diagramme de Gantt).

Avant tout, j'ai dû découvrir un projet en cours de réalisation et effectuer une légère remise à niveau sur les technologies employées dans sa conception.

Pour ma contribution, il m'a d'abord fallu insérer Blockly dans l'architecture du projet, tant au niveau de PLM que webPLM, ceci afin de pouvoir dans un premier temps créer l'emplacement dans la structure qui allait l'accueillir et ensuite avoir une base sur laquelle poursuivre l'implémentation.

L'étape suivante a été de mettre en place la barre d'outils proposant les blocs utilisables pour la réalisation de l'exercice. C'est à partir de cette étape que j'ai commencé à avoir une réelle interaction entre l'utilisateur et l'application via Blockly. Cette partie regroupe également le fonctionnement des blocs, au niveau de leur définition visuelle mais également leur traduction dans le langage choisi pour l'exercice.

S'ensuivit la gestion du stockage du code généré par les blocs. Utilisé pour la simple sauvegarde de l'avancement dans les exercices, le stockage était aussi très important et indispensable pour réaliser le suivi des étudiants par les enseignants.

La dernière étape fut la gestion de la traduction, de l'anglais qui est le statut « par défaut » de l'application au français. Le projet a une portée supérieure à la simple application en milieu scolaire local, et même dans ce cas, le portage en anglais et en français de l'application est un véritable plus d'accessibilité.

Dans tout ce rapport je suivrai un exemple, celui de l'exercice « Moria » de l'univers « Welcome ».

LORIA

Laboratoire

Le LORIA est le Laboratoire Lorrain de Recherche en Informatique et ses Applications. Créé en 1997, il est membre de la Fédération Charles Hermite avec les deux autres principaux laboratoires de recherche en mathématiques et de Science et Technologie de



Figure 1 : Logo LORIA



Figure 2 : Logo Inria

l'Information et de la Communication (STIC) de Lorraine. Le LORIA est une Unité Mixte de Recherche (UMR 7503) entre le CNRS (Centre National de la Recherche Scientifique), l'Université de Lorraine et Inria (Institut National de Recherche en Informatique et en Automatique).



Figure 3 : Logo CNRS

Le laboratoire est structuré autour d'une 30^{aine} d'équipes répartis en 5 départements, chacun représentant depuis 2010 une thématique (réf : Annexe 1 - Organigramme 2014 LORIA) pour un total de plus de 450 personnes.

Equipe COAST

Le stage s'est déroulé au sein de l'équipe COAST du LORIA et plus particulièrement en collaboration directe avec Gérald Oster, enseignant chercheur, et Matthieu Nicolas, ingénieur interne. Deux autres stagiaires, travaillant également sur PLM mais sur des problématiques différentes, Benjamin Thirion et Theodore Lambolez m'ont accompagné durant ces dix semaines.

L'équipe, dirigée par François Charoy, fait partie du 3^{ème} département « Réseaux, système et services » et travaille majoritairement sur la conception de service de partage d'objets, de communication, de gestion de tâches, de maintien d'une conscience de groupe, d'aide à la prise de décisions, notamment des applications de co-conception et/ou de co-ingénierie en Génie Logiciel, Architecture, Formation-Apprentissage.

Contexte

Etat du projet

PLM est un programme débuté en 2008 et en amélioration constante avec une mise à jour régulière suite aux retours des utilisateurs. L'application est développée en Java. Elle permet la réalisation en différents langages (Java, Python, C, Scala) de plus de 190 exercices répartis en 14 univers afin d'apprendre la programmation à son rythme. Il est également possible de choisir entre l'anglais, le français, l'italien, et le portugais.

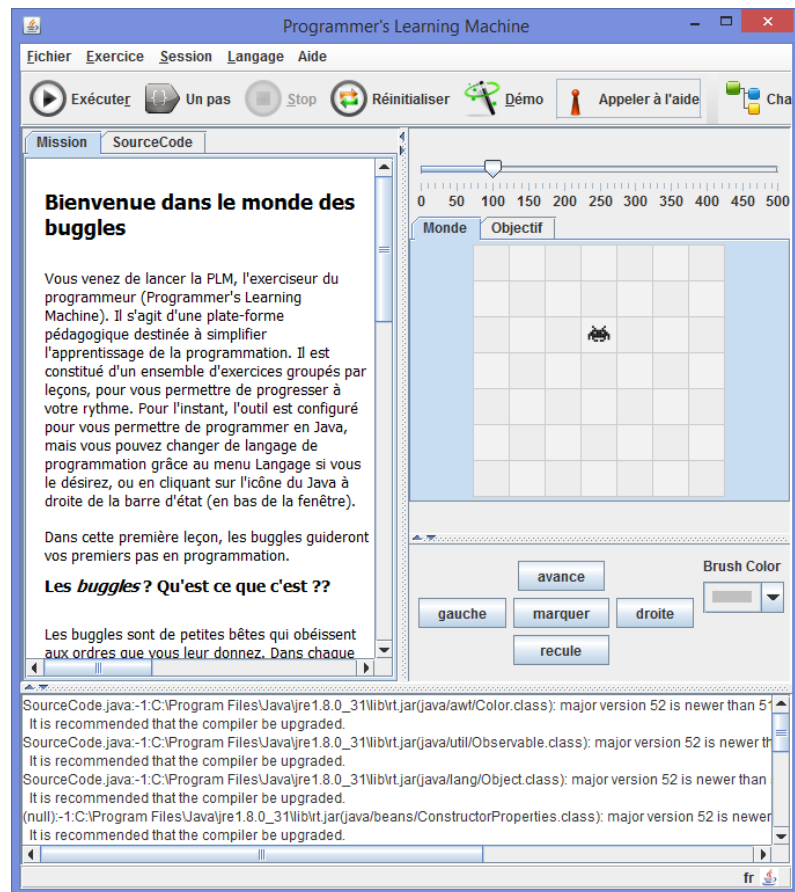


Figure 4 : PLM

Cette année, PLM est petit à petit portée comme service web et se tient prête à être utilisée pour la rentrée 2015. Gain d'accessibilité tant pour les utilisations encadrées que libres. WebPLM souhaite se séparer de la nécessité d'être téléchargée pour son utilisation et ainsi devenir accessible à tout poste disposant d'une connexion internet. Il reprend les mêmes exercices et options que PLM à la différence des langages humains qui sont pour le moment restreints à l'anglais ou au français. L'application web utilise Scala pour le serveur et JavaScript, HTML et CSS.

La structure est relativement simple. Nous avons d'un côté le serveur qui est composé de classe Scala, du Framework PlayFramework et qui se sert d'un fichier JAR généré à partir de l'application PLM de base en Java. De l'autre se trouve le client développé en JavaScript avec le Framework AngularJS. Mon implémentation de Blockly touche à ces trois axes, le JAR qui définit le comportement des langages utilisables pour les exercices, le serveur qui permet la

liaison entre les données du JAR et l'utilisation par le client, le Client qui donne directement l'affichage à l'utilisateur.

Blockly

La librairie Blockly dispose de toute une gamme d'outils, d'options et de documentations permettant son utilisation dans d'autres projets. Une fois en main, elle permet une création simple de ses propres blocs tout en

utilisant les blocs déjà existant. Blockly peut générer un code traduisant ses blocs dans différents langages, JavaScript, Python, Dart et prochainement PHP qui est actuellement en développement. Elle dispose également de divers outils notamment sur les traductions et le stockage sur lesquels je me suis basé pour correspondre au besoin et à l'implémentation actuelle de webPLM. Blockly possède également un grand avantage dans l'apprentissage de la programmation : les erreurs de syntaxe sont impossibles, évitant ainsi de ne pas trouver le point-virgule manquant.



Figure 5 : Blockly

Outils, technologies et méthodes

Dans le cadre de mon stage, je me suis servi du système de développement collaboratif Git via GitHub où étaient déjà présent les projets PLM, webPLM et Blockly, et des éditeurs de code Brackets et Eclipse suivant les langages.

Afin de ne pas interférer entre les différentes réalisations développées en parallèles, nous avons tous effectué un doublon du dépôt GitHub principal.

Cela nous a permis en effet de développer sans risquer de compromettre le travail du reste de l'équipe et inversement. L'envoi de ma contribution pour son ajout dans le dépôt principal pour la mise en commun s'est réalisé pour ma part vers la fin du stage afin d'avoir une fonctionnalité complète et stable.



Figure 6 :
GitHub

Le projet est composé de plusieurs langages de programmation. Dans le cadre de ma contribution j'ai utilisé principalement le Java et le JavaScript, les langages principaux de PLM et webPLM.

Pour une organisation et une continuité plus simple dans mon travail, il m'a été demandé de réaliser un « reporting », permettant également un suivi plus simple. Il consiste à noter en fin de journée tout ce que j'ai réalisé, les problèmes que j'ai rencontré, et l'évolution de mes objectifs, le tout étant envoyé sur un dépôt GitHub.

Développement

Découverte

Dans un premier temps, il a fallu que je découvre l'environnement de travail, PLM, webPLM et Blockly. J'ai donc réalisé quelques exercices sur PLM puis sur webPLM et j'ai terminé sur des utilisations de Blockly via ses démos avant d'explorer le code et sa structure. Le projet utilisant le Framework AngularJS, nous avons également réalisé une remise à niveau à travers un tutoriel sur le site Code School.

Insertion

Sélection

Pour ajouter Blockly dans l'application, il a d'abord fallu l'insérer au même niveau que les autres langages. Comme on peut le voir sur la figure 7 ci-contre, l'éditeur de code sur la partie gauche contient une instruction

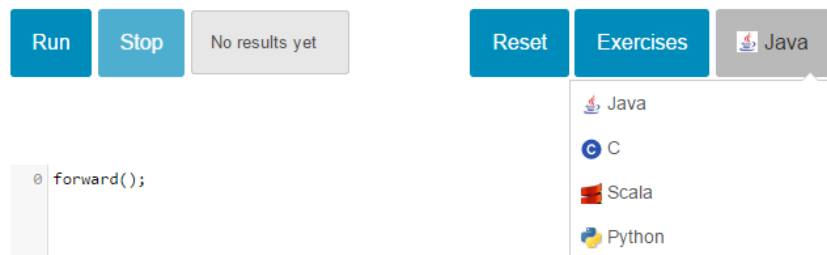


Figure 7 : Affichage de la zone de code et de la sélection du langage avant l'ajout de Blockly

« forward() ; ». On voit également sur la droite, que nous sommes en Java et qu'un clic sur le langage affiche un panneau qui nous offre le choix entre Java, C, Scala et Python.

Etant donné que le Python est un langage de résolution d'exercices déjà maîtrisé côté serveur, j'ai fait le choix de générer le code de Blockly dans celui-ci, je me suis donc basé sur son implémentation actuelle.

Les choix de langage de programmation disponibles sont chargés en fonctions de différents paramètres :

- Existence d'un fichier nommé *LangNomLangage.java* qui définit les caractéristiques du langage en question, sur la façon de l'exécuter notamment.
- Existence d'un fichier *NomExerciceEntity.extensionLangage* dans le package qui définit un exercice dans PLM comme *MoriaEntity.java* ou encore *MoriaEntity.blockly* qui indique que l'exercice Moria est réalisable en Java et avec Blockly.

J'ai également dû modifier certains des fichiers principaux qui s'occupent de la gestion des langages comme le fichier « Game.java ». En effet, il possède une liste de tous les langages possibles dans laquelle j'ai dû ajouter le choix de Blockly ou encore l'ajout d'une icône désignant Blockly pour rester dans le même rendu que pour les autres langages. La figure 8 montre le rendu dans le menu déroulant après l'ajout de Blockly, avec les mêmes caractéristiques que les autres langages, le nom et le logo.



Figure 8 : Sélection du langage après ajout

Affichage

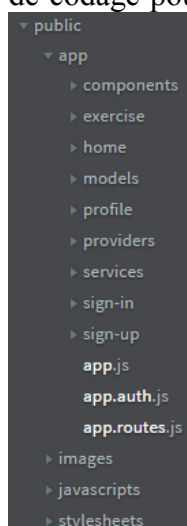
A ce stade, la sélection de Blockly comme langage est devenu possible bien que n'ayant pas de véritable impact sans l'éditeur associé. Il me faut à présent insérer la vue Blockly à la place de la zone de code. Pour se faire, Blockly propose des outils d'insertion, en taille fixe, dynamique ou via une iframe. Ces trois choix, bien que fonctionnels dans une structure lambda, ne correspondent pas à l'implémentation nécessaire à webPLM, structuré avec AngularJS. Je me suis donc intéressé, après conseil, à une implémentation déjà existante de Blockly via AngularJS partagé sur GitHub.

PLM se sert
actuellement de
CodeMirror
comme interface

```
20 <ui-codemirror ng-show="exercice.id == 'codemirror'" ng-
    model="exercice.code" ui-codemirror-opts="{ onLoad: codemirrorLoaded,
    lineWrapping : true, lineNumbers: true, tabSize: 2, firstLineNumber:
    0, autoCloseBrackets: true, mode: 'text/x-java' }"></ui-codemirror>
21 <blockly ng-show="exercice.id == 'blockly'"></blockly>
```

Figure 9 : Partie traitant le choix de l'IDE dans le fichier ide.directive

de codage pour permettre la création d'un code propre pour l'utilisateur c'est-à-dire un code avec une indentation et coloration s'adaptant au langage désiré. J'ai donc dû appliquer mes récentes connaissances en AngularJS afin de modifier la directive affichant CodeMirror et de faire de la place pour créer une directive pour Blockly en m'inspirant du projet AngularJS-Blockly. Pour se faire, je me suis servi, comme vous pouvez le voir sur la figure 9, des options « ng-show » et de la variable « exercice.id » du contrôleur d'exercice décrivant si je me sers de CodeMirror ou de Blockly pour éditer le code. Si la condition est validée alors la directive s'affiche.



**Figure 10 :
Partie
structure
webPLM**

Puis je me suis servi de l'implémentation AngularJS de Blockly, je l'ai décomposé et restructuré sous forme d'une directive (réf : Annexe 1 – Directive de Blockly) et d'un service (réf : Annexe 2 – Service de Blockly).

Afin de distribuer les fonctionnalités à leur place, par exemple, les services ont été placés dans un fichier `blockly.service.js`, rangé avec les autres services.

C'est à ce moment que j'ai eu des problèmes d'affichage de Blockly. Cela a commencé par l'apparition de Blockly seulement après une modification de la taille du navigateur puis par une augmentation de la hauteur de la zone de travail à chaque itération, clic ou ajout d'un bloc, avec cette dernière ou de changement de taille du navigateur. Pour les corriger, j'ai dû ajouter une contrainte de hauteur fixe de 500px, taille raisonnable étant donné que Blockly intègre deux scrollbars.

Génération

Cette troisième phase de l'ajout de Blockly consiste à lier la génération de code avec la gestion actuelle de webPLM. C'est dans le fichier `exercise.controller.js` qu'on définit le fonctionnement principal du client et de sa communication avec le serveur.

Tout d'abord, je dois regarder dans la fonction appelée lors de l'exécution, le bouton « Run », (réf : Annexe 2 – Fonction `runCode`) quel est l'éditeur de code actif entre CodeMirror et Blockly. Dans le cas de CodeMirror rien ne change, je garde l'implémentation actuelle. Mais dans le cas de Blockly je dois lui demander de générer en Python le code correspondant aux blocs présents dans l'espace de travail. Il possède une fonction « `Blockly.Python.workspaceToCode()` » qui renvoie le code généré par l'ensemble des blocs de l'espace de travail. Il ne me reste plus qu'à stocker le retour dans la variable recevant habituellement le code extrait de CodeMirror, le reste étant déjà implémenté, le code est envoyé au serveur qui le traite et renvoie l'évolution de l'exercice ou les éventuelles erreurs.

Toolbox

Bloc

Dans un deuxième temps, je me suis attelé à la création d'une barre d'outils composée des blocs pour la réalisation de l'exercice en cours. Pour ce faire, j'ai conçu de nouveaux blocs correspondant aux fonctions reconnus par le serveur et nécessaires à Blockly comme les blocs de mouvement « forward », « backward », ou d'interaction avec l'environnement « pickupBaggle », « isFacingWall ».

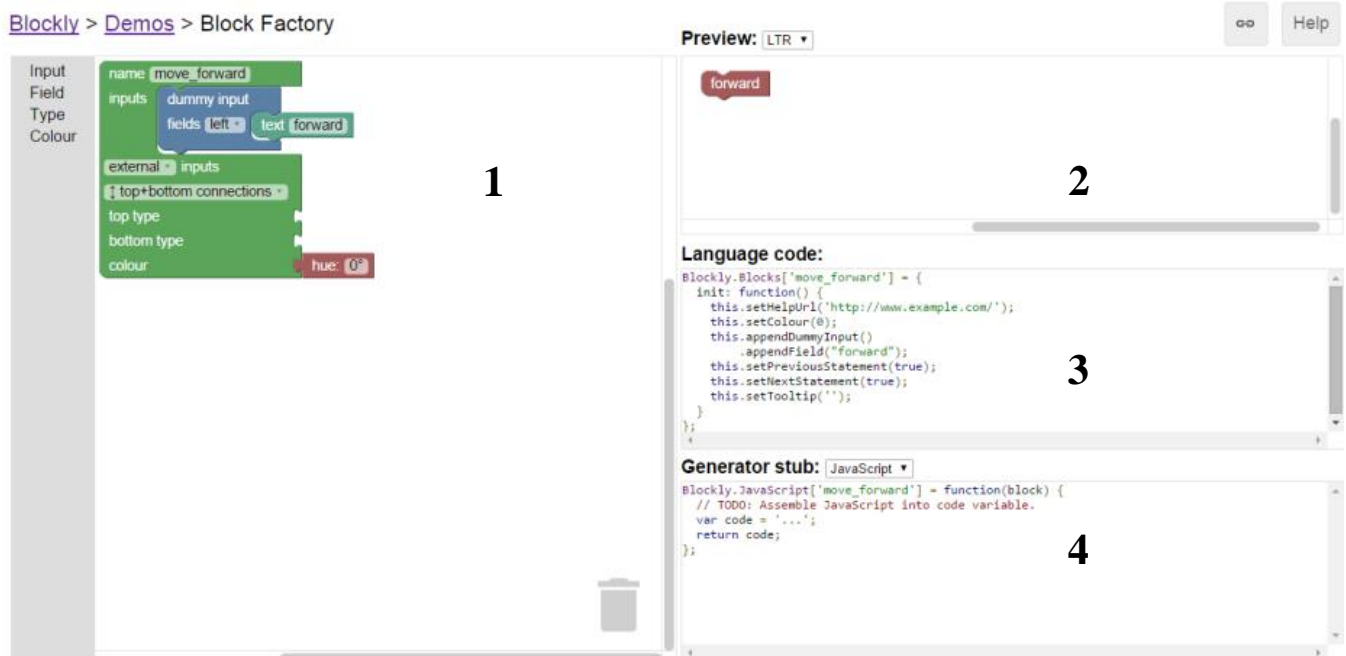


Figure 11 : Aperçu de la création de bloc

La documentation Blockly met à disposition une mise en application de Blockly permettant la création de bloc personnalisé, très utile mais assez longue. L'interface ci-dessus montre la création du bloc « forward ». Dans la partie de gauche (1), on crée le bloc à l'aide d'une interface Blockly, divers choix sont possibles, comme la couleur, le nom, les types et le nombre des paramètres mais également la valeur de retour ou les connections avec l'instruction précédente ou suivante. La partie de droite se décompose en trois zones. Tout d'abord nous avons l'aperçu (2) du bloc visuellement, puis le code qui crée le bloc (3), et enfin le squelette qui abritera la définition de la génération du code (4).

Je l'ai utilisé pour faire le premier bloc et comprendre le fonctionnement d'un bloc afin de réaliser les suivants directement dans le code.

Le fonctionnement est le suivant (réf : Annexe 3 – Code de définition du bloc) :

- « This.setColour(0); » Colorie le bloc de la couleur désirée
- « this.appendDummyInput() » Ajout une entrée du type « Dummy » servant uniquement à placer du texte sur le bloc. D'autres entrées pour des valeurs sont possibles
- « .appendField(chaine); » Ajoute un champ de texte.
- « this.setPreviousStatement(true); » Permet une liaison avec un bloc avant.
- « this.setNextStatement(true); » Permet une liaison avec un bloc après.

- « `this.setTooltip(chaine);` » Définit un texte qui sera afficher si on laisse le curseur sur le bloc pour avoir des informations complémentaires.

La seconde partie de la création d'un bloc est dans le code qu'il doit générer dans chaque langage. Pour notre cas, je me suis limité à la génération en Python étant donné que les autres langages ne nous sont pas utiles. Le principe est relativement simple, on prend le squelette donné par la documentation et on ajoute dans la variable « code » ce qu'on souhaite. Pour le bloc « forward » il suffit donc de mettre dans la variable le code « `forward()\n` » permettant ainsi d'avoir l'appel de fonction reconnu par le serveur et un retour chariot pour l'instruction suivante (réf : Annexe 3 – Code de génération de code en Python).

Liaison

Une fois les blocs créés, il me faut à présent lier la boîte d'outils à l'exercice en cours. Pour ce faire, je dois créer des fichiers dans PLM, pour chaque exercice, contenant la boîte d'outils associée. Je les ai placés au même niveau que les fichiers qui définissent les exercices et ils stockent les boîtes d'outils au format JSON. Nous avons fait le choix de ne pas rendre ce fichier obligatoire, en cas d'oubli ou de suppression involontaire. Donc j'ai placé dans le service de Blockly une boîte d'outils par défaut possédant tous les blocs utiles à l'univers Welcome.

Dans notre cas, un fichier JSON définit deux éléments de la boîte d'outils dont un seul est

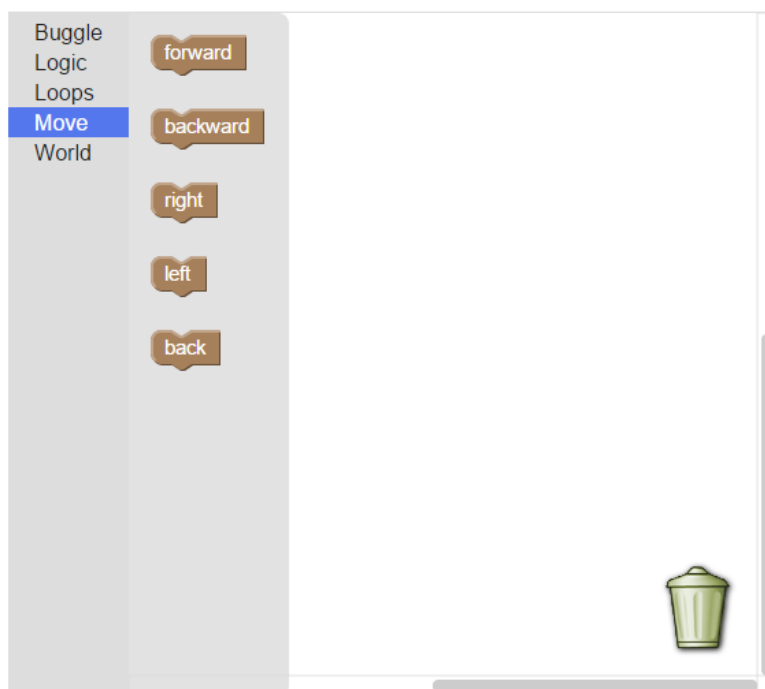


Figure 12 : Blockly avec boîte d'outils

indispensable tandis que l'autre permet un meilleur rendu. Tout d'abord, il donne le type du bloc (qui peut être comparé à son identifiant), « `move_forward` » dans le cas du bloc « forward ». Vu que c'est le client qui possède la librairie Blockly dont la définition des blocs, il est capable de construire le bloc et son fonctionnement uniquement avec ce type. C'est l'élément indispensable.

Ensuite, on peut choisir de trier ces blocs dans des catégories. Par exemple, j'ai décidé de placer tous les blocs qui définissent un mouvement pour le Buggle dans une section « Move » dans la boîte d'outils. Ce tri est très utile dans notre cas car nous avons besoin d'un nombre plutôt conséquent de blocs et les avoir tous dans la boîte d'outils les uns après les autres serait gênant au niveau place et recherche.

Lorsqu'on regarde le fichier de la boîte d'outils pour l'exercice Moria on reconnaît d'une part la structure des fichiers JSON mais on peut également voir facilement la structure de la boîte d'outils. Les champs « name : » désignent le nom de la section et dans le champ « blocs » se trouve tous les blocs qui iront dans cette section. Cette solution reste relativement simple pour ajouter ou modifier des boîtes d'outils ou des blocs dans ces dernières.

```

1  [
2    {
3      "name": "Buggle",
4      "blocks": [
5        {
6          "type": "buggle_facingWall"
7        }
8      ],
9      "name": "Logic",
10     "blocks": [
11       {
12         "type": "controls_if"
13       }, {
14         "type": "newLogic_operation"
15       }, {
16         "type": "logic_negate"
17       }
18     ],
19     "name": "Loops",
20     "blocks": [
21       {
22         "type": "controls_whileUntil"
23       }
24     ],
25     "name": "Move",
26     "blocks": [
27       {
28         "type": "move_forward"
29       }, {
30         "type": "move_backward"
31       }, {
32         "type": "turn_right"
33       }, {
34         "type": "turn_left"
35       }, {
36         "type": "turn_back"
37       }
38     ],
39     "name": "World",
40     "blocks": [
41       {
42         "type": "world_buggle_ground"
43       }, {
44         "type": "world_buggle_pickup"
45       }, {
46         "type": "world_buggle_drop"
47       }
48     ]
49   ]
50 ]

```

Figure 13 : Boîte d'outils pour l'exercice Moria

Stockage

A ce stade, nous pouvons utiliser les blocs, envoyer le code et recevoir la réponse, faire un exercice en somme. Il manque, par rapport aux autres langages de programmation, le fait de sauvegarder son avancement à chaque exécution pour pouvoir quitter et revenir à l'endroit où on s'était arrêté.

Pour ce faire, j'ai dû modifier les envois et les traitements entre le client et le serveur. En effet, en y réfléchissant, on se rend compte que les langages de base disposent d'un unique code qu'il affiche et envoie alors que Blockly en compte deux. Le premier est le code que Blockly possède dans son espace de travail et qui est traduit par l'affichage des blocs. Le second en revanche est le code qui est généré à l'exécution. Nous avons donc besoin à présent de manipuler ces deux codes afin de les envoyer au serveur et qu'il soit capable de les traiter tous les deux.

Le projet AngularJS sur Blockly possède un début d'implémentation sur la manipulation de l'espace de travail en format JSON mais n'est malheureusement pas totalement fonctionnel, je me suis donc servi de la manipulation de base disponible dans Blockly en format XML.

La première chose à connaître est le système de sauvegarde. En effet le serveur garde diverses informations sur un total de six fichiers pour Blockly et cinq pour les autres langages, dont

tous les noms sont construits suivant la même structure :
 « *nomUnivers.lessons.nomUnivers.typeExercice.nomExercice.extensionLangage.typeContenu* ».
 Je vais pour la suite prendre pour exemple l'exercice Moria réalisé sous Blockly dont les fichiers de sauvegarde possède tous une partie commune, jusqu'au type du contenu, soit « *welcome.lessons.welcome.summative.Moria.blockly* » traduit dans la suite par « # ».

- « #.code » contient le code de résolution de l'exercice
- « #.workspace » sauvegarde l'espace de travail
- « #.correction » garde la solution pour un exercice réussi
- « #.error » stocke les erreurs retournées par le serveur
- « #.done » sait si l'utilisateur a réussi l'exercice par le passé
- « #.mission » affiche l'énoncé de l'exercice






 welcome.lessons.welcome.environment.Environment.blockly.code	12/06/2015 13:58	Fichier CODE	1 Ko
 welcome.lessons.welcome.environment.Environment.blockly.correction	12/06/2015 13:58	Fichier CORRECTION	1 Ko
 welcome.lessons.welcome.environment.Environment.blockly.error	12/06/2015 13:58	Fichier ERROR	1 Ko
 welcome.lessons.welcome.environment.Environment.blockly.mission	12/06/2015 13:58	Fichier MISSION	3 Ko
 welcome.lessons.welcome.environment.Environment.blockly.workspace	12/06/2015 13:58	Fichier WORKSPACE	1 Ko

Figure 15 : fichiers d'une session pour un exercice

Le tout est soit envoyé au serveur soit, s'il n'y a aucune connexion internet, stocké dans un dossier « .plm » dans l'espace utilisateur de la machine en local. C'est ensuite le serveur qui se charge, suivant les préférences de l'utilisateur, d'envoyer le code sur un dépôt git GitHub.

Pour effectuer cette réalisation, j'ai dû modifier le trajet de l'envoi et de la prise en charge. Tout d'abord, il a fallu, dans la fonction runCode (réf : Annexe 2 – Fonction runCode) de exercise.controller.js, extraire le contenu de l'espace de travail en XML avec les fonctions de Blockly.Xml.workspaceToDom et Blockly.Xml.domToText. La première prend le contenu de l'espace de travail et le transforme en un élément du DOM permettant ainsi l'utilisation de la seconde qui crée une chaîne de caractère à partir d'un élément du DOM. Il me reste plus qu'à insérer ce texte dans un nouveau champ de l'objet envoyé au serveur.

```
2015-06-12 14:07:03.802 <xml exercise.controller.js:480
xmlns="http://www.w3.org/1999/xhtml"><block type="move_forward" id="6" x="36" y="25">
<next><block type="move_backward" id="12"></block></next></block></xml>
```

Figure 14 : XML généré pour un bloc forward suivi d'un bloc backward

Une fois reçu, le serveur répartit le contenu de l'objet dans les différents fichiers. La classe PLM en scala s'occupe de cette partie mais elle nécessite la création préalable de ces fichiers qu'elle ne fait que remplir. C'est dans « Exercise.java » que sont générés les fichiers, et j'ai

ajouté une condition sous laquelle on crée deux fichiers pour Blockly et un pour les autres langages. De ce fait, seul les exécutions en Blockly provoqueront la création d'un fichier en « #.workspace ».

Traduction

Le système de traduction actuel est plutôt complexe. Le projet dispose de plusieurs mécanismes de traductions, qui seront mis en commun prochainement. Et Blockly possède le sien également. L'idéal est de centraliser ces trois systèmes ou du moins d'ajouter celui des blocs à l'un ou l'autre existant.

Je me suis penché avant tout sur le fonctionnement de celui de Blockly qui s'est révélé très complet au niveau des langues disponibles, plus de 40 actuellement. En revanche, il pose un problème majeur, il se sert d'un rechargement complet de la page afin de prendre en compte le changement de langue. Ce système ne convient pas du tout à l'implémentation du projet. Son association dans un mécanisme de traduction existant du projet est donc devenue l'unique choix.

WebPLM se sert de gettext pour la traduction. J'ai donc implémenté une solution pour y associer Blockly également. Le principe est simple, remplacer toutes les chaînes de caractères par un appel d'une constante contenant le texte de la langue en cours par l'appel « Blockly.Msg.MOVE_FORWARD_TITLE ». A chaque changement de langue, toutes les variables sont régénérées avec le contenu dans la bonne traduction grâce à gettext qui regarde le langage et la chaîne qu'on lui donne et qui retourne la traduction. PLM se sert de l'anglais comme référence, c'est-à-dire que si aucune traduction n'a été trouvée il renvoie l'état de base de la variable qui est en anglais. Pour ma part, je n'ai pas modifié ce comportement déjà fonctionnel. J'ai ajouté un service BlocklyMsg regroupant toutes les variables de traductions utilisées par Blockly et permettant la liaison avec gettext, et réalisé une première traduction, dans le dictionnaire, des mots que j'ai ajouté et je me suis servi des traductions de Blockly pour les blocs de base.

En ce qui concerne le changement de langue en cours de réalisation d'un exercice, le fonctionnement est particulier. En effet, toute la page se met à jour grâce au système de gettext sauf en ce qui concerne Blockly. Lors de la notification du changement du langage, on ne recharge pas la page, mais en revanche on peut actualiser la

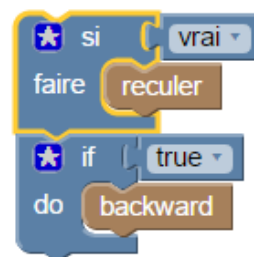


Figure 16 : Blockly avec du français et de l'anglais

boite d'outils. L'espace de travail n'est par contre pas remis à jours, il se peut alors que l'utilisateur se retrouve avec des blocs anglais et français, mais cela n'est pas gênant étant donné que la génération de code pour l'exécution et le stockage est la même quel que soit la langue. Si l'utilisateur change d'avis en cours de réalisation et trouve le mélange perturbant, il lui suffit de lancer l'exécution pour sauvegarder ses blocs et de recharger la page dans la langue de son choix afin que tous les blocs s'uniformisent.

Conclusion

Je suis parti d'un projet en cours de réalisation, le portage d'une application en client lourd à un service web. Mon travail consistait à ajouter une fonctionnalité. L'usage de Blockly, un éditeur visuel de code parmi les langages disponibles pour l'exerciseur.

Ma contribution s'est déroulée en cinq étapes majeures. Une phase de découverte pour m'adapter à mon environnement de travail : projet et langages. Puis l'ajout de l'accès de la nouvelle fonctionnalité. Suivi de la création de l'environnement de cette dernière. Et de la mise en place du fonctionnement derrière celle-ci. Enfin d'une touche plus cosmétique mais importante.

Je suis donc arrivé à un exerciseur où Blockly est fonctionnel pour la réalisation des exercices du premier univers, mais aussi dont l'ajout pour les univers suivants ne posera probablement pas de problèmes à l'avenir.

La traduction des noms de sections serait un atout, déjà sur le plan de l'uniformisation de la fonctionnalité. Un effet de coloration du bloc provoquant l'effet de l'aperçu de l'exercice serait une très profitable amélioration mais risque de poser quelques problèmes, notamment vu que l'aperçu ne verrouille pas l'éditeur de code. Une mise en place de boîte d'outils par pallier éviterait ainsi de surcharger la boîte d'outils de bloc des autres univers lorsqu'on oublie de définir la boîte pour l'exercice en cours. Et pour finir ou plutôt pour commencer, une batterie de tests serait des plus judicieuses afin d'éviter tout développement régressif.

Sur le plan personnel, ce stage m'a beaucoup apporté. Tout d'abord le fait de prendre un projet en cours est plutôt déroutant, nous qui avons plus l'habitude de les réaliser de A à Z. J'ai beaucoup apprécié la façon de travailler, principalement les discussions qui ont eu lieu sur tel ou tel fonctionnalité, leur utilité, leurs possibles implémentations, leurs possibles limites mais également celles sur les problèmes rencontrés afin de les résoudre de manière efficace et permanente. Et ceci, même entre des personnes qui ne travaillent pas sur la même partie du projet. En revanche j'ai aussi découvert le « monde extérieur » du développement dont les projets ou aides dont on se sert et qui se trouvent sur internet ne sont pas toujours au point et complet. Par exemple, le projet AngularJS-Blockly, bien que très utile lors du début de mon stage, s'est révélé incomplet et mis de côté temporairement de la part de son auteur. De ce faite, les documentations n'étaient pas à jour et certaines fonctions n'aboutissaient à rien.

Bibliographie

Sites web

Code School, 7 avril, <https://www.codeschool.com>

LORIA, <http://www.loria.fr/les-actus>

PLM, <http://www.loria.fr/~quinson/Teaching/PLM>

Blockly, <https://developers.google.com/blockly>

GitHub, <https://github.com/BaptisteMounier>

AngularJS, <https://github.com/cdjackson/angular-blockly>

CodeMirror, <https://codemirror.net>

Rapport

Mounier Loïc, Développeur Web, 25 octobre 2010 au 15 janvier 2011, Bibliothèque Universitaire IUT Nancy-Charlemagne, DUT Informatique, Université de Lorraine, 46 pages

Table des annexes

Annexe 1 :

Diagramme de Gantt

Organigramme 2014 LORIA

Directive de Blockly

Annexe 2 :

Service de Blockly

Fonction runCode

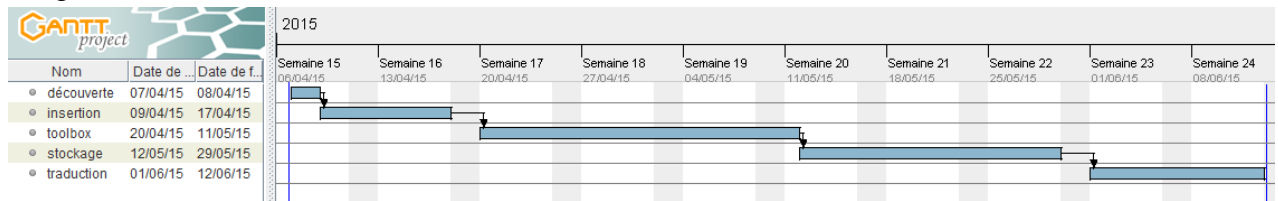
Annexe 3 :

Génération de code en Python

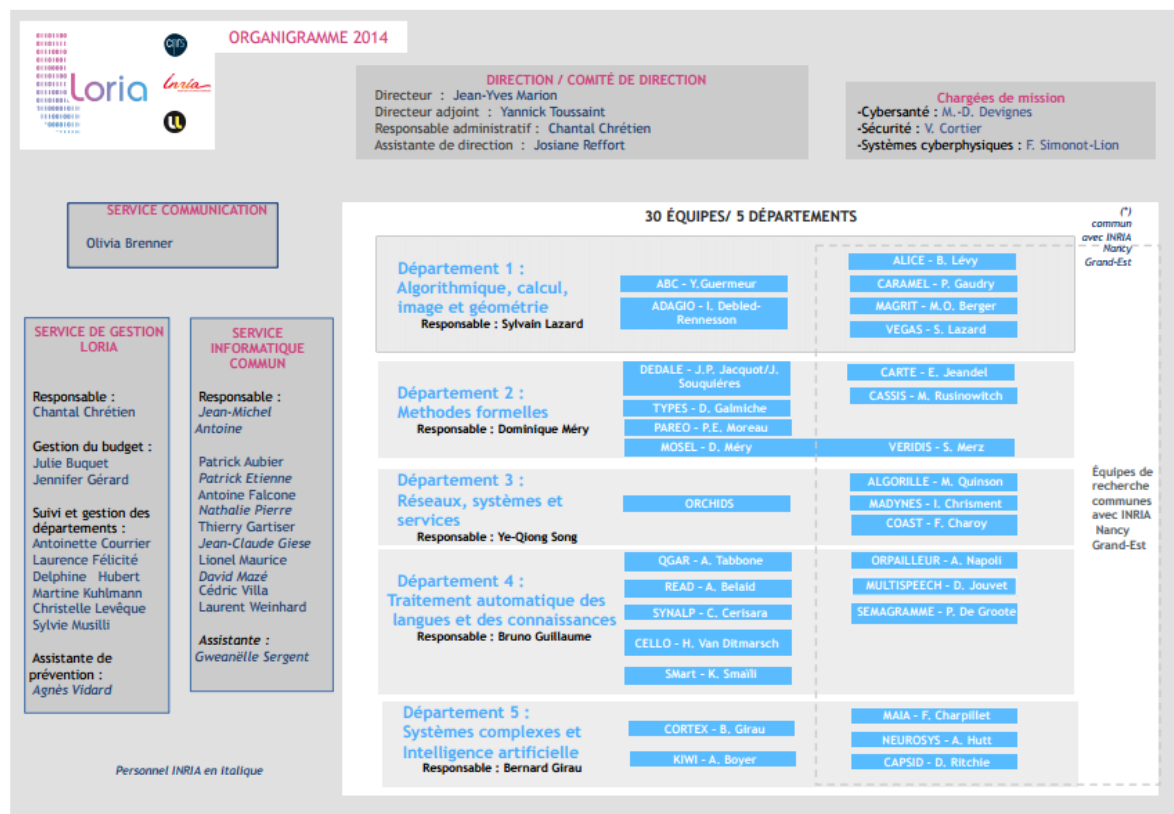
Définition du bloc forward

Annexe 1

Diagramme de Gantt



Organigramme 2014 LORIA



Directive de Blockly

```
1 ▼ (function () {  
2     'use strict';  
3  
4     angular  
5     .module("PLMApp")  
6     .directive('blockly', function ($window, $timeout, $rootScope, blocklyService) {  
7         return {  
8             restrict: 'E',  
9             scope: { // Isolate scope  
10             },  
11             templateUrl: '/assets/app/components/blockly.directive.html',  
12             link: function ($scope, element, attrs) {  
13                 var options = blocklyService.getOptions();  
14                 Blockly.inject(element.children()[0], options);  
15             }  
16         };  
17     });  
18  
19 })();
```


Annexe 2

Service de Blockly

```
1 (function () {
2   'use strict';
3
4   angular
5     .module("PLMApp")
6     .service("blocklyService", blocklyService);
7
8   blocklyService.$inject = ['BlocklyMsg'];
9
10  function blocklyService(BlocklyMsg) {
11    var options = {
12      path: "/assets/javascripts/blockly/media/",
13      trashcan: true,
14      toolbox: [{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}]
15    };
16
17    updateMsg();
18    console.log(Blockly.Msg);
19    var service = {
20      getOptions: getOptions,
21      setOptions: setOptions,
22      updateMsg: updateMsg
23    };
24    return service;
25
26    function updateMsg() {
27      Blockly.Msg = BlocklyMsg.getModel();
28    }
29
30    function getOptions() {
31      return options;
32    }
33
34    function setOptions(opt) {
35      options = opt;
36    }
37  }
38})();
```

Fonction runCode

```
456 function runCode() {
457   var args;
458
459   exercise.updateViewLoop = null;
460   exercise.isPlaying = true;
461   exercise.worldIDs.map(function (key) {
462     reset(key, 'current', false);
463   });
464   setCurrentWorld('current');
465   exercise.tabs.map(function (element) {
466     if (element.worldKind === 'current' && element.drawFnct === exercise.drawFnct) {
467       exercise.currentTab = element.tabNumber;
468     }
469   })
470
471   if (exercise.ide === 'blockly') {
472     Blockly.Python.INFINITE_LOOP_TRAP = null;
473     exercise.code = Blockly.Python.workspaceToCode();
474     var xml = Blockly.Xml.workspaceToDom(Blockly.getMainWorkspace());
475     exercise.studentCode = Blockly.Xml.domToText(xml);
476
477     args = {
478       lessonID: exercise.lessonID,
479       exerciseID: exercise.id,
480       code: exercise.code,
481       workspace: exercise.studentCode
482     };
483   } else {
484     args = {
485       lessonID: exercise.lessonID,
486       exerciseID: exercise.id,
487       code: exercise.code
488     };
489   }
490   connection.sendMessage('runExercise', args);
491   exercise.isRunning = true;
492 }
493
```

Annexe 3

Génération de code en Python

```
1 ▼ Blockly.Python['move_forward'] = function (block) {  
2   var code = 'forward()\n';  
3   return code;  
4 };
```

Définition du block forward

```
1 ▼ Blockly.Blocks['move_forward'] = {  
2 ▼   init: function () {  
3     this.setColour(30);  
4     this.appendDummyInput()  
5       .appendField(Blockly.Msg.MOVE_FORWARD_TITLE);  
6     this.setPreviousStatement(true);  
7     this.setNextStatement(true);  
8     this.setTooltip(Blockly.Msg.MOVE_FORWARD_TOOLTIP);  
9   }  
10 };
```