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
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

Baptiste MOUNIER

Gérald OSTER
Isabelle DEBLED-RENNESSON

Remerciements

IUT

+#

+Marraine de stage

LORIA

+COAST

+détails

Table des matières

Remerciements ..... - 3 -
Introduction 7
Sujet 7
Mise en œuvre 7
LORIA 9
Laboratoire 9
Contexte 10
Equipe COAST 10
Etat du projet 10
Blockly 11
Outils, technologies et méthodes 11
Développement 12
Découverte 12
Insertion 12
Sélection 12
Affichage 13
Génération 14
Toolbox 14
Bloc 14
Liaison 16
Stockage 16
Traduction 18
Ouverture 19
Conclusion 20
Bibliographie 21
Sites web 21
Rapport 21
Annexe 1 22
Diagramme de Gantt 22
Organigramme 2014 LORIA 22
Directive de Blockly 22
Annexe 2 23
Service de Blockly 23
Fonction runCode appellé lorsqu'on lance l'execution 23

Annexe 3 24
Code de génération de code en Python 24
Boite d’outils pour l’exercice Moria 24

Introduction

Sujet

Le stage consiste à la réalisation de l'ajout de Blockly au sein de l'application web de PLM. Utilisant les langages Java, Python, Scala et C, Blockly ajoute une couche visuelle afin de facilité la première approche vis-à-vis de la programmation.

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 visuel est 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 par l'équipe COAST du LORIA sous licence publique générale GNU v3.0. L'application est actuellement utilisée en école d'ingénieur, Télécom Nancy, pour une progression rapide des étudiants découvrant la programmation. 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.

Mise en œuvre

Mon travail s'est effectué en 5 grands axes que je développerais dans cet ordre dans ce dossier (ref : 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 employés dans sa conception.

Pour ma contribution, il m'a d'abord fallu insérer Blockly dans le projet, tant au niveau de PLM que webPLM. Simplement pour pouvoir dans un premier temps sélectionner Blockly et 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. 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.

Puis la gestion du stockage du travail réalisé. Utilisé pour la simple sauvegarde de l'avancement dans les exercices, le stockage est aussi indispensable pour réaliser le suivi des étudiants par les enseignants.

Et enfin de 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, même dans ce cas, le portage en anglais et en français de l'application est un véritable plus d'accessibilité.

LORIA

Laboratoire

Le LORIA est le Laboratoire Lorrain de Recherche en Informatique et ses Applications. Créé en 1997, il est membre, avec les deux autres principaux laboratoires de recherche en mathématiques et STIC (Science et Technologie de l'Information et de la Communication) de Lorraine, de la Fédération Charles Hermite. C'est une Unitée 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 1 : Logo LORIA

Figure 2 : Logo Inria

Figure 3 : Logo CNRS

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

Contexte

Equipe COAST

Le stage s'est déroulé au sein de l'équipe COAST du LORIA et plus particulièrement en collaboration direct avec Gérald Oster, enseignant chercheur, et Matthieu Nicolas, ingénieur interne. Egalement avec deux autres stagiaires travaillant également sur PLM mais sur des problématiques différentes, Benjamin Thirion et Theodore Lambolez.

L'équipe, dirigée par François Charoy, fait partie du 3ème département « Réseaux, système et services » et travail 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.

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 utilise Scala, C, Python, mais reste majoritairement 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 et avec sa langue, à choisir entre l'anglais, le français, l'italien, et le portugais.

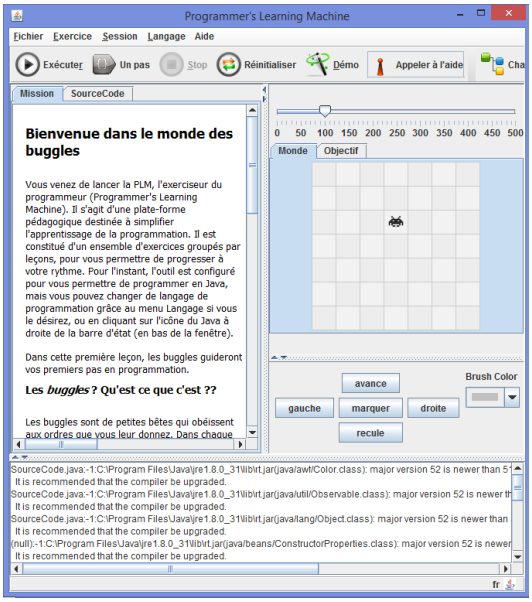


Figure 4 : PLM

Cette année, PLM devient progressivement webPLM et se tient prêt à être utiliser pour la rentrée 2015. Gain d'accessibilité tant pour les utilisations encadrées que libres, webPLM souhaite se séparer de la nécessité d'être téléchargé pour son utilisation et ainsi devenir accessible à conditions d'une connexion internet. Il reprend les mêmes exercices et options que PLM à la différence des langages humains qui sont pour le moment limité à l'anglais ou au français. L'application côté web utilise Scala, HTML, CSS mais reste majoritairement en JavaScript.

Blockly

La librairie Blockly dispose de toute une gamme d'outils et d'options permettant son utilisation dans d'autres projets de même d'une documentation plutôt bien fournie. Une fois en main, il permet une création simple de ses propres blocs tout en utilisant les blocs déjà existant. Blockly peut générer différent langages grâce à ses blocs, du JavaScript, du Python, du Dart et est actuellement en développement pour du PhP. Il dispose également de divers outils notamment sur les traductions, le stockage dont je me suis servis ou modifier pour correspondre au besoin et à l'implémentation actuelle de webPLM. Blockly possède également un grand avantage dans l'apprentissage de la programmation, le fait que les erreurs de syntaxe sont impossible, évitant ainsi de tourner en rond pour un simple oubli.

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.

Figure 6 : GitHub

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.

Le projet est composé de plusieurs langages de programmation. Dans le cadre de mon contribution j'ai utilisé en très grosse majorité 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 les directives d'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 « forward() ; ». On voit également que nous sommes en Java et que le panneau déroulant sur un clique sur le langage nous offre le choix entre Java, C, Scala et Python.

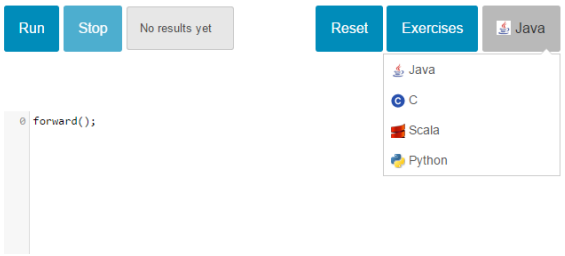


Figure 7 : Affichage de la zone de code et de la sélection du langage avant l'ajout de Blockly

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

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

- Existance d'un fichier nommé Lang*NomLangage*.java qui défini le langage désigné pour la liaison avec le serveur comme les retours d'erreurs.
- Existance d'un fichier *NomExercice*Entity.*extensionLangage* dans la package qui défini 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 complèter par quelques détails des fichiers principaux qui s'occupent de la gestion des langages comme le fichier Game.java qui possède une liste de tous les langages possibles dans laquelle j'ai dû permettre le choix de Blockly ou encore l'ajout d'une icone désignant Blockly pour rester dans le même rendu que pour les autres langages. La figure 8 montre le rendu dans le menu déroullant 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 qu'inutile sans l'éditeur associé, il me faut à présent insérer la vue Blockly à la place de la zone de code. Pour ce faire Blockly propose des outils d'insertion, en taille fixe, dynamique ou via une iframe. Ces trois choix, bien que fonctionnelle dans une structure lambda, ne correspondent pas à l'implémentation nécessaire à webPLM, structuré avec AngularJS. Je me suis donc interessé, après conseil, à une implémentation déjà existante de Blockly via AngularJS partagé sur GitHub.

PLM se sert actuellement de CodeMirror comme interface 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 mettre à l'œuvre 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 sur GitHub. Pour ce faire je me suis servis, comme vous pouvez le voir sur la figure 9, des options « ng-show » et la variable « exercice.ide » du contrôleur d'exercice désignant si je me sers de CodeMirror ou de Blockly pour éditer le code. Si la condition est validée alors la directive s'affiche.

```
20      <ui-codemirror ng-show="exercise.ide === 'codemirror'" ng-
    model="exercise.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="exercise.ide === 'blockly'"></blockly>
```
Figure 9 : Partie traitant le choix de l'ide dans le fichier ide.directive

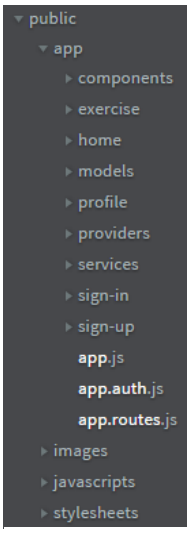


Figure 10 : Partie structure webPLM

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

Afin de distribuer les fonctionnalités à leur place par exemple les services ont été placé 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 à 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 intération 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 résonable étant donné que Blockly intègre deux scrollbars en cas de besoin de plus de place.

Génération

Cette troisième phase de l'ajout de Blockly consistais à liéer la génération de code avec la gestion actuelle de webPLM. C'est dans le fichier exercise.controller.js qu'on défini le fonctionnement principale de webPLM et sa liaison avec PLM.

Tout d'abord je dois regarder dans la fonction appelé lors de l'execution (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 actuel mais dans le cas de Blockly je dois lui demander de générer en Python le code correspondant aux blocs présent dans l'espace de travail. Il possède une fonction « Blockly.Python.workspaceToCode() » qui renvois 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 mis en place le code est envoyé au serveur qui le traite et renvois l'évolution de l'exercice ou les erreurs si il y en a.

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 fabriqué de nouveaux blocs correspondant aux fonctions reconnus par le serveur et nécessaire à Blockly comme les blocs de mouvement « forward », « backward », ou d'interaction avec l'environnement « pickupBaggle », « isFacingWall ».

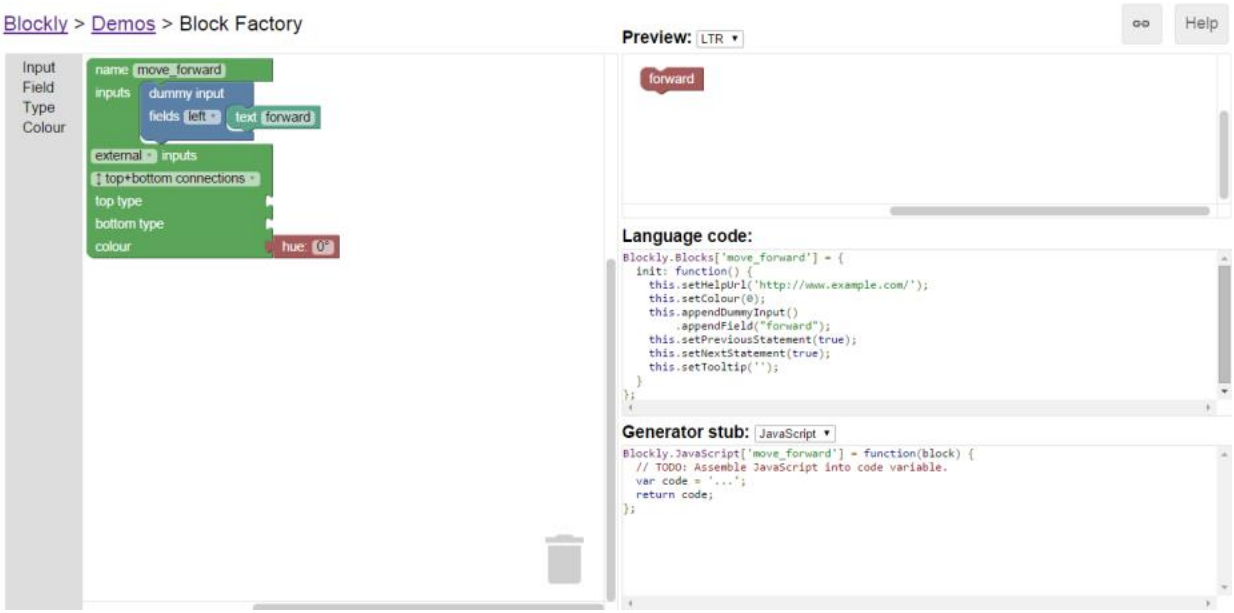


Figure 11 : Aperçu de la création de bloc

La documentation Blockly met à disposition une application de Blockly permettant la création de bloc personnalisé, très utile mais assez long. L'interface ci-dessus montre la création du bloc « forward », dans la partie de gauche on crée le bloc à l'aide d'une interface Blockly, divers choix sont possible, comme la couleur les nom, types et 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 du bloc visuellement puis le code à placer dans la définition des blocs et enfin le code qui détermine la génération de code.

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

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 utile. 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 function reconnu par le serveur et un retour chariot pour l'instruction suivante (réf : Annexe 3 – Code de génération de code en Python).

Liaison

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

Dans notre cas, un fichier json définit deux éléments de la boite d'outils dont un seul est 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 # dans une section « Move » dans la boite d'outils.

Lorsqu'on regarde le fichier de la boite d'outils pour l'exercice Moria (réf : Annexe 3 - Boite 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 boite d'outils. Les champs « name : » désigne 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 boites d'outils.

Stockage

Dans un troisième temps, le problème du stockage et récupération de l'avancement. 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 en repartant à l'endroit où on s'était arrêté.

Pour ce faire j'ai dû modifier les envois et traitements. En effet en y réfléchissant on se rend compte que les langages de base dispose d'un unique code qu'il affiche et envois alors que Blockly en compte deux. Le premier que j'appellerai le code blocs correspond au 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 de manipulation de l'espace de travail en format json mais n'est malheureusement pas totalement fonctionnel, je me suis donc servis de la manipulation de base disponible dans Blockly en format xml.

La première chose à savoir c'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 ».

- « #.code » contient le code de résolution de l'exercice
- « #.workspace » sauvegarde l'espace de travail, permettant la reconstruction des blocks utilisés
- « #.correction » garde la solution pour un exercice réussi
- « #.error » stock 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

Le tout étant soit envoyé sur un dépôt git soit, si aucune connexion internet, stocké dans un dossier « .plm » dans l'espace utilisateur de la machine en local.

Pour effectuer cette réalisation, j'ai dû explorer à nouveau PLM et webPLM et modifier le trajet de l'envoi et de 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 chaine de caractère à partir d'un élément du dom. Il me reste plus qu'à insèrer ce texte dans un nouveau champs de l'objet envoyé au serveur.

Une fois recut, le serveur réparti le contenu de l'objet dans les différents fichiers. La classe PLM en scala s'occupe de cette partie mais elle nécessite la création préallable de ces fichiers qu'elle ne fait que remplir. C'est dans « Exercise.java » que sont générés les fichiers, et on regarde pour le moment le langage qui est demandé afin de déterminer le nombre nécessaire.

Traduction

Le système de traduction actuel est plutôt complexe. Le projet dispose de plusieurs principes de traductions, en cours de mise en commun 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 rélevé très complet au niveau des langues disponibles, plus de 40 actuellements. En revanche, il pose un problème majeur, il fonctionne par rechargement de la page ce qui ne convient pas du tout à l'implémentation de webPLM. Son association dans un système du projet est donc devenu le meilleur 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 chaines de caractères par un appel d'une constante contenant le texte de la langue en cours. Par exemple pour le nom du bloc forward on désire « forward » en anglais et « avance » en français, comme texte sur le bloc, il faut donc remplacer la chaine actuelle par « Blockly.Msg.MOVE_FORWARD_TITLE » qui se chargera d'aller chercher dans le service BlocklyMsg la chaine « forward » si le client demande l'anglais ou « avance » si le client demande le français.

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 mets à jours grace 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 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, chose pas génante étant donné que la génération de code pour l'execution et le stockage est le même quel que soit la langue. Si l'utilisateurchange d'avis en cours de réalisation et trouve le mélange génant, il lui suffit de lancer l'execution pour sauvegarder ses blocs et de recharger la page dans la langue de son choix afin que tous les blocs s'uniformise.

Ouverture

Conclusion

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

Annexe 1

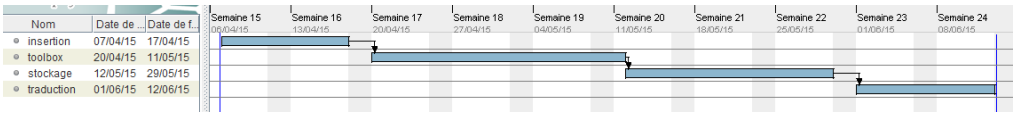
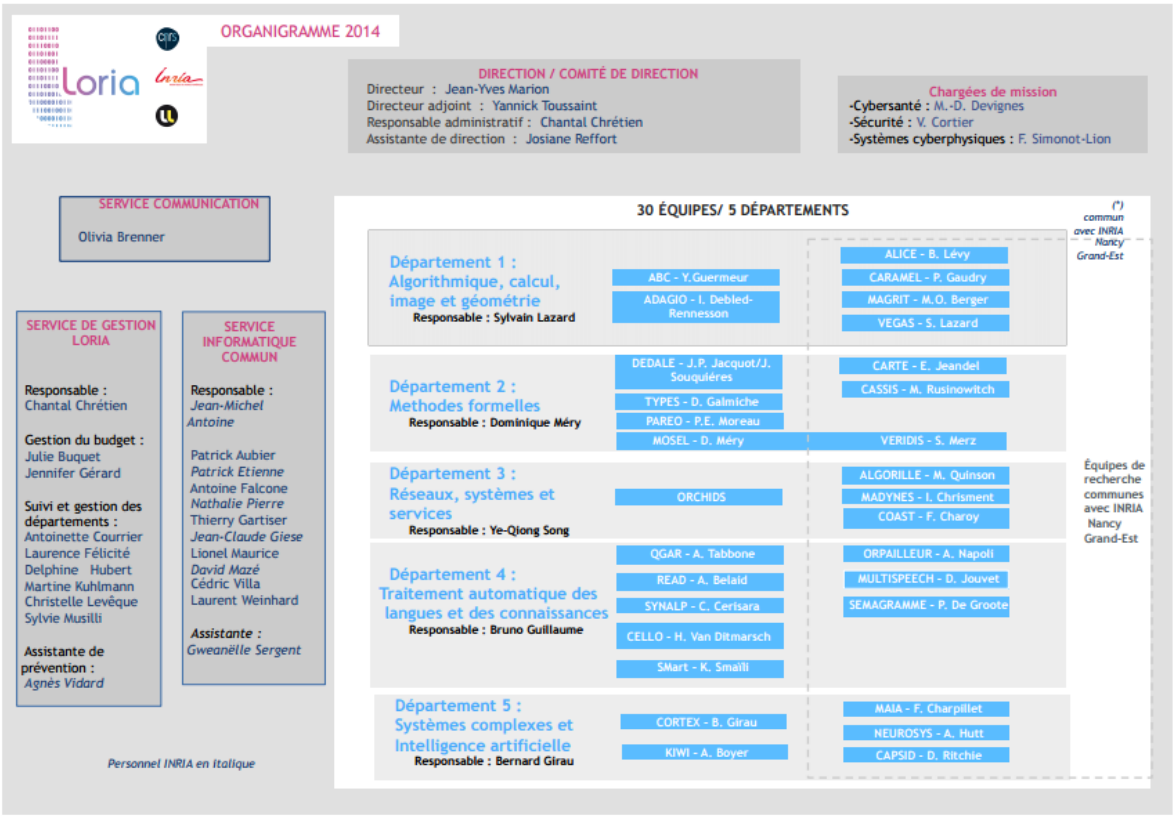


Diagramme de Gantt



Organigramme 2014 LORIA

```
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 })();
```

Directive de Blockly

Annexe 2

```
(function () {
    'use strict';

    angular
        .module("PLMApp")
        .service("blocklyService", blocklyService);

    blocklyService.$inject = ['BlocklyMsg'];

    function blocklyService(BlocklyMsg) {
        var options = {
            path: "/assets/javascripts/blockly/media/",
            trashcan: true,
            toolbox: [{⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}, {⊖}]
        };

        updateMsg();
        console.log(Blockly.Msg);
        var service = {
            getOptions: getOptions,
            setOptions: setOptions,
            updateMsg: updateMsg
        };
        return service;

        function updateMsg() {
            Blockly.Msg = BlocklyMsg.getModel();
        }

        function getOptions() {
            return options;
        };

        function setOptions(opt) {
            options = opt;
        };
    }

})();
```

Service de Blockly

```
        function runCode() {
            var args;

            exercise.updateViewLoop = null;
            exercise.isPlaying = true;
            exercise.worldIDs.map(function (key) {
                reset(key, 'current', false);
            });
            setCurrentWorld('current');
            exercise.tabs.map(function (element) {
                if (element.worldKind === 'current' && element.drawFnct === exercise.drawFnct) {
                    exercise.currentTab = element.tabNumber;
                }
            })

            if (exercise.ide === 'blockly') {
                Blockly.Python.INFINITE_LOOP_TRAP = null;
                exercise.code = Blockly.Python.workspaceToCode();
                var xml = Blockly.Xml.workspaceToDom(Blockly.getMainWorkspace());
                exercise.studentCode = Blockly.Xml.domToText(xml);

                args = {
                    lessonID: exercise.lessonID,
                    exerciseID: exercise.id,
                    code: exercise.code,
                    workspace: exercise.studentCode
                };
            } else {
                args = {
                    lessonID: exercise.lessonID,
                    exerciseID: exercise.id,
                    code: exercise.code
                };
            }
            connection.sendMessage('runExercise', args);
            exercise.isRunning = true;
        }
```

Fonction runCode appellé lorsqu'on lance l'execution

Annexe 3

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

Code de génération de code en Python

```
[{
    "name": "Buggle",
    "blocks": [{
        "type": "buggle_facingWall"
                }]
            }, {
    "name": "Logic",
    "blocks": [{
        "type": "controls_if"
                }, {
        "type": "newlogic_operation"
                }, {
        "type": "logic_negate"
                }]
            }, {
    "name": "Loops",
    "blocks": [{
        "type": "controls_whileUntil"
                }]
            }, {
    "name": "Move",
    "blocks": [{
        "type": "move_forward"
                }, {
        "type": "move_backward"
                }, {
        "type": "turn_right"
                }, {
        "type": "turn_left"
                }, {
        "type": "turn_back"
                }]
            }, {
    "name": "World",
    "blocks": [{
        "type": "world_baggle_ground"
                }, {
        "type": "world_baggle_pickup"
                }, {
        "type": "world_baggle_drop"
                }]
}]
```

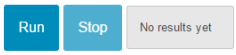
Boite d'outils pour l'exercice Moria

TEMPO IMAGE :

Fig # : Rendu d'un bloc forward()

```
Blockly.Blocks['move_forward'] = {
    init: function () {
        this.setColour(30);
        this.appendDummyInput()
            .appendField(Blockly.Msg.MOVE_FORWARD_TITLE);
        this.setPreviousStatement(true);
        this.setNextStatement(true);
        this.setTooltip(Blockly.Msg.MOVE_FORWARD_TOOLTIP);
    }
};
```

Fig # : Code d'un bloc forward()



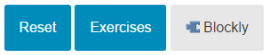


Fig # : Affichage de la zone de code avec Blockly