

Génération de code à partir d'architecture logicielle



Année universitaire : 2017-2018
Formation : Licence 3 Informatique
Nom : Gayraud
Prénom : Matthieu

Sommaire

Remerciement.....	3
Introduction.....	4
Objet du stage, contexte et enjeux.....	4
Présentation de l'université et de l'équipe :.....	5
Présentation du contexte et du projet global:.....	5
Concepts liés à la modélisation.....	6
Présentation et installation des outils.....	7
Outils pour la modélisation.....	7
Outils pour le robot lego-mindstorms-ev3.....	7
Création et utilisation des outils pour la modélisation.....	9
Création de la grammaire.....	10
Règles de vérification et modélisation.....	13
<i>Vérification de la validité d'un assemblage.....</i>	<i>13</i>
<i>Vérification de l'unicité des noms.....</i>	<i>14</i>
<i>Validation de la vérification.....</i>	<i>14</i>
<i>Conclusion sur la création des outils.....</i>	<i>14</i>
Java LeJOS.....	15
Des exemples au modèles.....	15
Retour sur la grammaire.....	16
Bibliographie et sitographie.....	17

Remerciement

Je tiens à remercier toutes les personnes m'ayant aidé au cours de ce stage.

Tout d'abord j'adresse mes remerciements à mon maître de stage Gilles ARDOUREL pour son accueil au ainsi que pour ses conseils et explications.

Je remercie également l'ensemble de l'équipe AeLoS pour leur accueil lors de la première réunion à laquelle j'ai pu assister, et en particulier, Pascal ANDRE, pour ses conseils sur la dernière partie de mon stage.

Introduction

Objet du stage, contexte et enjeux

Durant ce stage je me suis intéressé à la génération de code à partir d'architecture logicielle. Pour pouvoir générer du code de manière automatique il est nécessaire d'avoir un modèle qui sert de patron. Il est par conséquent indispensable d'avoir un ensemble de règles permettant de construire un modèle de la façon la plus rigoureuse possible, cela permettant de garder une syntaxe similaire entre différents modèles même si ces derniers sont créés par des utilisateurs différents. On peut donc déjà voir les premières étapes nécessaires à la génération de code, et la nécessité de devoir créer des outils permettant d'y arriver.

Afin de ne pas avoir à remonter encore plus haut et devoir créer un langage permettant de créer un langage permettant de modéliser, on utilisera un plug-in d'un environnement de développement. Il est bien évident qu'il est nécessaire d'avoir une limite pour les langages définissant d'autres langages et on s'aperçoit donc rapidement qu'il est indispensable d'avoir un langage se définissant lui-même.

Sachant cela, j'ai d'abord commencé à lire un article définissant les concepts liés à la modélisation. Une fois ces concepts compris et assimilés j'ai commencé la création d'une grammaire pour un langage définissant les modèles, ce que je présenterais dans la partie « Création de la grammaire »

Ensuite, je me suis intéressé à la modélisation, avec un exemple simple et fictif, ce qui m'a permis d'effectuer des tests ainsi que de me rendre compte que les règles de grammaires seules ne suffisent pas pour assurer la cohérence du modèle. Nous verrons cela en détail dans la partie « Règles de vérification et modélisation »

Une fois la création du langage permettant la modélisation terminée, je me suis intéressé à Java LeJOS, un firmware java permettant de programmer des petits robots en lego. L'objectif étant à partir d'un modèle, de générer du code permettant d'utiliser un robot mindstorms-ev3.

Présentation de l'université et de l'équipe :

Ce stage se déroule à l'université des sciences et des techniques de Nantes, dans le Laboratoire des Sciences du Numérique de Nantes (LS2N), une unité de recherche créée en janvier 2017 suite à la fusion du Laboratoire Informatique de Nantes Atlantique (LINA) et de l'Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)^[1]

Mon stage a été effectué au sein de l'équipe AeLoS (Architectures et Logiciels Surs) à l'université des Sciences et des Techniques de Nantes. Cette équipe a comme principal objet d'étude le logiciel et cherche à garantir leur correction. Les différentes difficultés de ces recherches sont abordés suivant plusieurs thématiques qui gravitent autour des modèles afin de couvrir les différentes couches d'abstractions constituant un logiciel.

Présentation du contexte et du projet :

Ce projet permet, en plus de travailler sur la génération de code, d'introduire des éléments de programmation par composant.

Concepts liés à la modélisation

Tout d'abord mon travail a consisté à lire un article de recherche où sont définis de manière rigoureuse les concepts liés à la modélisation, aux méta-modèles et à l'ingénierie logicielle dirigée par les modèles. Les définitions que nous utiliseront sont celles issues de l'article de A.R.d.Silva « Model-driven engineering: A survey supported by the unified conceptual model »

D'après l'article, un modèle est un système aidant à définir et à répondre aux questions d'un système étudié sans avoir à le considérer. Un méta-modèle est un modèle qui définit la structure d'un langage de modélisation. On peut résumer de manière simplifier ces notions avec le schéma suivant :

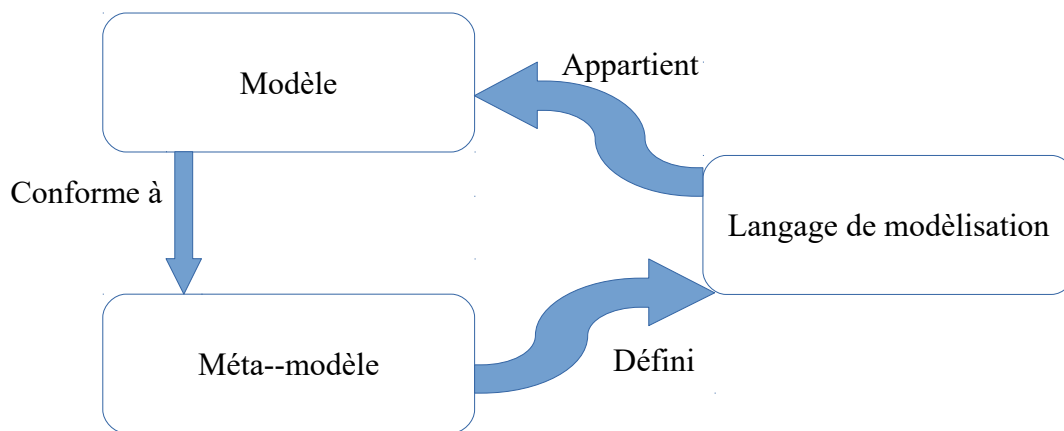


Fig 1 : Relation entre méta-modèle, modèle et langage de modélisation

L'utilisation de modèles dans la programmation n'est pas récente cependant ils étaient alors considérés comme des artefacts annexes, principalement utilisés pour de la documentation. Cependant, depuis une dizaine d'année une nouvelle approche concernant l'utilisation des modèles est apparu, ces derniers occupent dans certains cas une place importante lors du développement, on parle alors d'ingénierie logicielle dirigée par les modèles.^[2]

L'article permet donc de mieux comprendre l'intérêt de la modélisation ainsi que la notion de hiérarchie y étant lié en définissant les différentes notions abordées. Cela permet de répondre à de nombreuses questions notamment « Comment utiliser les modèles dans le contexte d'un développement logiciel » .

En fonction des besoins et du domaine dans lequel un utilisateur travaille les besoins lors du développement d'un logiciel ne seront pas les mêmes

Présentation et installation des outils

Outils pour la modélisation

Afin de mener ce projet à bien j'ai utilisé l'environnement de développement Eclipse^[3] sur lequel le plug-in Xtext^[4] est disponible. Il est aussi possible d'utiliser Xtext sur un navigateur, cependant on ne pourrait pas en exploiter toutes les fonctionnalités, par exemple, le débogage sur navigateur n'est pas possible.

Il faut donc, une fois l'installation d'Eclipse terminée, installer Xtext à l'aide de la fonction « installer de nouveaux logiciels ». Il n'y a pas besoin d'effectuer de manipulations supplémentaires, l'environnement de développement qui se charge de l'installation des plug-in.

Xtext permet de créer des langages de programmations ainsi que des langages spécifique à un domaine. Cela permet donc la création de langage très précis lié à un besoin particulier, l'intérêt étant d'éviter à l'utilisateur d'avoir à apprendre un nouveau langage complexe.

En effet, comme le langage est créer en fonction des besoins de l'utilisateur et du domaine dans lequel il travaille, il n'est pas nécessaire d'avoir un outil complexe constitué de nombreuses bibliothèques qui ne seront jamais utile dans un contexte précis.

On peut donc ici voir le lien avec les notions abordées dans le point précédent, Xtext permet de définir un méta-modèle afin de créer un langage de modélisation et fait est donc un méta-méta-modèle.

Outils pour le robot lego-mindstorms-ev3

C'est aussi en utilisant Eclipse que j'ai créer un projet en Java, pour faire fonctionner le robot mindstorms-ev3, pour des programmes simples il est aussi possible d'utilisé le logiciel fourni avec les lego-mindstorms-ev3, appelé LEGO MINDSTORM Education EV3. Ce logiciel permet à des utilisateurs n'ayant que très peu de connaissances en informatique de créer des programmes simple pour un robot, on peut donc voir ce logiciel comme une variante, bien plus complexe, du travail effectué sous Xtext. A l'aide de différents blocs on peut créer un modèle donnant une suite d'instructions à un robot, ce qui est plus simple à comprendre qu'une suite d'instruction textuel comme ce qu'on pourrait faire grâce à la modélisation sur Xtext.

En fonction des règles de modélisation il est possible de créer des modèles assez simple permettant de générer un code assez complexe, dans ce cas le travail sera plus long lorsqu'on fixera les différentes règles de génération, mais la création de la grammaire sera plus simple.

Les projets que l'on peut créer grâce au logiciel proposé par Lego peuvent donc se rapprocher davantage d'un programme, que la modélisation faite avec Xtext.

Bien que représenté sous forme de bloc, les instructions que l'on peut donner au robot se rapproche énormément d'un programme écrit en ligne de code, on y retrouve par exemple les conditionnelles, de plus un bloc est généralement l'équivalent d'une fonction, dont on donnera les paramètres.

Ci-dessous un exemple de programme avec le logiciel MINDSTORM, la suite d'instruction permet de faire avancer le robot en ligne droite.



Fig 2 : Exemple de programme avec le logiciel MINDSTORM

On peut donc distinguer 6 blocs, cependant, dans cet exemple il n'y a que 3 blocs différents. Le premier bloc permet d'exécuter le programme. Les blocs verts permettent de contrôler les moteurs B et C, chacun correspondant à une roue, à l'aide de plusieurs paramètres, le premier pour la puissance à gauche, le second pour la puissance à droite, on peut ensuite donner le nombre de rotation, l'angle de rotation ou encore le temps de rotation, chacune de ces options est représenté par l'un des blocs vert. Le bloc orange avec le sablier permet d'effectuer une pause, la durée de la pause étant donnée en seconde.

Avec Xtext l'objectif est de générer des méthodes équivalentes à ces blocs d'instructions pour que le robot effectue une action donnée.

Bien que l'objectif final du projet soit la génération de code, j'ai choisis de faire en sorte que la syntaxe utilisé dans le modèle soit relativement proche de celle du java, sans que cela ne gêne une personne étant habituée au java, qui verra les nombreuses différences syntaxique lors de la modélisation, ou bien un utilisateur n'ayant aucune connaissances concernant ce langage.

Création et utilisation des outils pour la modélisation

Lorsque l'on crée le projet Xtext, 5 dossiers sont créés automatiquement, (voir fig.3) On ne s'intéressera seulement aux dossiers « com.project.foo » et « com.project.foo.tests ».

La majeure partie du contenu de ces dossiers est générée automatiquement par Xtext, dès qu'on demande l'exécution de la grammaire, ce qui doit être fait après chaque modification.

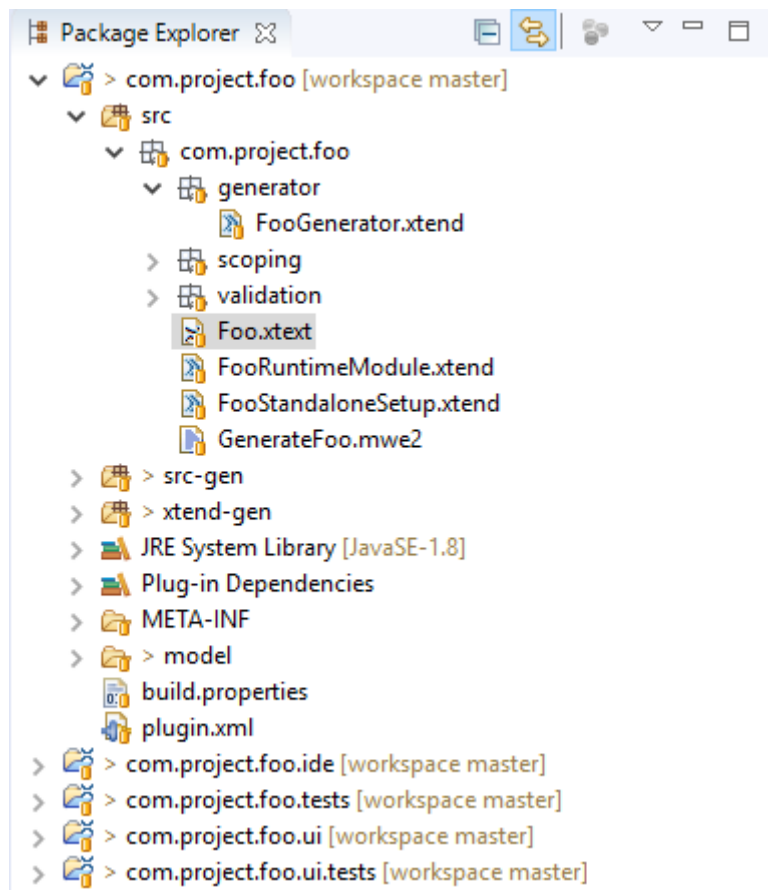


Fig 3 : Hiérarchie du projet

Le fichier *Foo.xtext* contient la grammaire qui sera utilisé pour la modélisation. (Voir partie « Création de la grammaire »)

Les fichiers présent dans les packages generator, scoping et validation sont à modifier après avoir créée la grammaire. Ils permettent respectivement de fixer, les règles de génération de code, les règles de scope, et les règles de validation.

La génération de code ne peut se faire qu'une fois un ensemble de règle fixé, c'est à dire que si dans le modèle apparaît « A » le générateur doit savoir quel partie de code créer.

Le scope, permet de donner limiter ou non la portée d'une information dans le modèle. Par défaut la portée d'un élément est celle de son conteneur, c'est à dire que si dans l'objet A se trouve un élément B alors il ne sera pas possible d'utiliser le même élément B dans le conteneur A'

Avec Xtext, si on ne modifie que le fichier contenu dans le package scoping la visibilité d'un élément s'arrête au fichier, lorsqu'il y a plusieurs fichiers il est nécessaire de faire des imports. On peut donc voir que certaines règles de la grammaire seront liés à la façon dont Xtext gère la visibilité des éléments.

Le package validation contient un fichier appelé *Validator* c'est dans ce fichier que l'on peut rajouter des règles en complément de celles de la grammaire. (Voir partie « Règle de vérification et modélisation »)

Création de la grammaire

La création de la grammaire est la première étape pour pouvoir créer un langage de modélisation.

Lors de la création de la grammaire il est nécessaire d'avoir à l'esprit l'objectif de la modélisation afin de créer un ensemble de règle le plus petit possible pour en simplifier la compréhension par l'utilisateur. Cependant, si la grammaire est trop simplifiée, c'est la génération de code qui deviendra bien plus complexe. Il est donc nécessaire de trouver un certain équilibre lors de la création du langage. De plus il est souvent nécessaire de modifier la grammaire en cours de développement, elle doit donc être relativement expressive afin de pouvoir être modifiable facilement.

Toutes les règles de la grammaire ont une structure similaire, elles sont composées d'un nom et de plusieurs éléments. On distingue deux types de règles, les règles terminales et les non-terminales que l'on notera respectivement RT et RNT.

Une règle est dite non-terminale si certains éléments qui la constitue sont d'autres règles. Une règle est dite terminale si on utilise seulement des éléments terminaux, comme des chaînes de caractères.

Pour faciliter la distinction entre ces deux types de règles Xtext affiche le nom des RNT en bleu et des RT en noir.(Voir fig. 4)

Une règle peut être composée de plusieurs éléments du même type, c'est à dire issue de la même règle. On peut utiliser « * » si l'objet peut être présent aucune fois, une fois ou plusieurs fois, ou « + » si l'élément doit être présent au moins une fois.



```

1 grammar com.project.foo.Foo with org.eclipse.xtext.common.Terminals
2
3 generate foo "http://www.project.com/foo/Foo"
4
5 //Concerne les modeles
6 DomainModel:
7     model=Model;
8
9 /*
10  * Structure d'un modele, correspond
11  * à un paquet pouvant contenir plusieurs
12  * composants et assemblages
13  */
14 Model:
15     'package' name=QualifiedName '{'
16     (imports+=Import)*
17     (components+=Component)*
18     (assembly+=Assembly)*
19     '}' ;
20
21
22 QualifiedName:
23     ID ('.' ID)*;
24
25 Import:
26     'import' importedNamespace=QualifiedNameWithWildcard;
27
28 QualifiedNameWithWildcard:
29     QualifiedName '.*'?;
30

```

Fig 4 : Morceau de grammaire

Dans la capture d'écran ci-dessus on peut voir une partie de la grammaire créée. J'ai fait le choix d'autoriser un seul modèle par fichier, ce qui est visible avec la règle DomainModel, qui est la racine d'un fichier. La règle appelé Model correspond donc à un unique fichier et par conséquent n'est pas obligatoirement un modèle complet. En effet, grâce aux import il est possible de séparer plusieurs composants, ce qui améliore la lisibilité du modèle. Cependant pour tout modèle il existe un équivalent que l'on peut écrire sur un unique fichier.

(Voir annexe A, pour la grammaire)

Un assemblage étant composé de plusieurs composants j'ai fait le choix de déclarer l'ensemble des composants avant les assemblages par soucis de lisibilité.

Chaque ensemble de composant et d'assemblage présent dans un fichier est contenu dans un paquet, si deux fichiers appartiennent au même paquet il n'est pas utile de faire d'import, en effet, la séparation en plusieurs fichiers est une fois encore utile pour simplifier la lecture et hiérarchiser les informations, mais l'ensemble du contenu de fichiers présent dans un même paquet pourrait être contenu dans un unique fichier.

Ces premières règles permettent donc d'avoir la structure générale du modèle, c'est à dire que chaque fichier utilisé pour la modélisation suivra cette structure. C'est à dire : nom du paquet auquel le fichier appartient, puis des import, suivi des composants et des assemblages s'il y en a.

Une fois ces règles écrites il est important de savoir ce qui constitue un import, un composant ou encore un assemblage.

Comme dit précédemment, un import permet d'avoir accès à un composant présent dans un autre fichier. On peut donc en déduire la règle Import.

Un composant est constitué d'une liste de services qu'il fournit et d'une liste de services requis, cette dernière pouvant être vide. Un service correspond donc à une méthode dont on doit donner la signature. Il est donc nécessaire de faire la distinction entre les services requis et des services fournis, ce qui signifie qu'il y a besoin de créer deux nouvelles règles qui seront des éléments de la règle Component.

Un assemblage est constitué d'un ensemble de composants, pour qu'un assemblage soit correcte il faut que chaque service requis par un des composants soit lié à un service fourni par un autre composant. La règle Assembly est constituée de plusieurs éléments, d'abord on liste les composants qui forment l'assemblage puis on donne l'ensemble des liens entre services requis et services fournis.

Cependant, la grammaire toute seule ne suffit pas à créer un langage cohérent et non ambiguë. Il est donc nécessaire de rajouter un ensemble de règles de validation qui permettront de compléter cette grammaire. Une règle de validation permet, par exemple, de faire en sorte que l'utilisateur respecte certaines conventions comme le fait de commencer le nom d'un composant ou d'un assemblage par une majuscule, ce qui permet de tout de suite identifier le type de l'objet lors de la lecture du modèle.

Règles de vérification et modélisation

La création de règle de validation a été la partie la plus complexe, c'est seulement en modélisant et en affinant la modélisation qu'on se rend compte que d'autres vérifications sont nécessaires, ces dernières étant parfois associées aux premières règles de validations créées.

Les méthodes de vérifications ainsi que les test de ces méthodes sont écrits en Java. Cependant, un utilisateur, devra simplement utiliser les règles qui lui sont proposées par le créateur du langage pour modéliser l'ensemble des assemblages sur lesquels il travaille, il n'y a donc pas de connaissances préalable à avoir en Java, ni même besoin d'avoir l'habitude de Xtext, ce qui est l'un des principaux intérêts de ce plug-in.

Les règles de vérification sont exécutés lors de la modélisation, ce qui permet de voir les erreurs directement.

Vérification de la validité d'un assemblage

Lorsqu'on créer un assemblage il est nécessaire que chaque services requis par un composant soient satisfait. (Voir «Création de la grammaire»)

Il est donc nécessaire de vérifier que l'ensemble des services requis par les composants constituant l'assemblage soient satisfait. Pour simplifier la lecture j'ai créer plusieurs règles permettant cette vérification. Dans un premier temps on vérifie que chaque liaison est valide, pour cela on compare un à un les paramètres du service fourni et du service requis présent dans la liaison. Une erreur sera générée si le type de retour des méthodes n'est pas le même, si le nombre de paramètre n'est pas identique, ou bien s'il existe au moins un paramètre ayant un type différent. Cette méthode s'appellera *checkBindingIsValid*.

La vérification des liaison une à une ne suffit pas pour assurer qu'un assemblage est correct. En effet, on pourrait avoir un cas où toutes les liaisons présentes sont correcte mais un service requis d'un composant n'est fourni. Il faut donc aussi vérifier que pour chaque instance des composants présent dans l'assemblage tous les services requis sont liés. Il faut donc rajouter une autre méthode afin d'effectuer cette vérification que l'on appellera *checkAssemblyIsCorrect*. Cette règle de vérification prend en paramètre un assemblage et vérifie que chaque service requis de chaque composant constituant l'assemblage est correctement lié.

Vérification de l'unicité des noms

Il est important d'éviter toute ambiguïté pour que la génération de code fonctionne correctement, pour cela il est nécessaire de ne pas avoir deux éléments possédant le même nom. De plus cela simplifie la lisibilité du modèle. J'ai donc créé un ensemble de méthode permettant de vérifier l'unicité des noms des composants et de leurs attributs, des assemblages et ainsi que des services. Si le type des éléments est différent ils peuvent avoir le même nom car le type sera connu lors de la génération.

Pour vérifier l'unicité la méthode la plus simple est d'utiliser une fonction proposée par Xtext qui permet d'obtenir la liste de tous les éléments d'un fichier pour un type donné. Il suffit ensuite de simplement vérifier qu'un élément n'est pas présent dans la liste pour savoir si le nom est unique ou non. Cette méthode est néanmoins coûteuse, de manière brute cela implique que pour chaque ajout d'un élément on est obligé de le comparer à tous les éléments du même types déjà présent dans le modèle. Stocker les noms lors de leur ajout dans une table de hashage aurait été une meilleure solution, cependant je n'ai pas trouvé de méthodes fonctionnant de cette façon.

Validation de la vérification

Il est en plus important de fournir des jeux de test afin de prouver que les règles de vérification fonctionnent, cependant il n'est pas possible de faire un ensemble de test exhaustif. On cherche donc à tester des cas limites du modèle pour voir le fonctionnement de ces règles lorsqu'il y a des erreurs dans la modélisation. J'ai donc créé une méthode de test pour chaque règle de validation créée. Il est nécessaire de tester chaque méthode indépendamment afin d'en assurer le bon fonctionnement, car ce sont les briques de bases du programmes. Il faut donc chercher à tester les choses les plus élémentaires pour que l'ensemble de la modélisation fonctionne.

Conclusion sur la création des outils

Une fois les différents outils permettant de modéliser des assemblages et leurs composants implémenter, on peut commencer à s'intéresser aux programmes que l'on cherche à générer de manière automatique. La modélisation ne peut être faite qu'après cette étude.

Java LeJOS

Dans cette partie on s'intéresse à la programmation des robots lego-mindstorms-ev3. Comme indiqué précédemment, les programmes que les robots utilisent peuvent être codés en java, ou bien grâce au logiciel MINDSTORM. Ces robots étant constitués de différentes pièces, tels que des capteurs ou des moteurs, on peut voir le lien avec les composants. Il s'agit dans le cas du robot de composant matériel, lors de la modélisation on s'intéressera à des composants logiciels.

Des exemples au modèles

Dans un premier temps j'ai commencé par regarder de nombreux exemples de code permettant de faire fonctionner des composants du robot, tels que les moteurs ou un capteur de proximité.

Les exemples commencent toujours de la même façon, les différents composants nécessaires pour que le robot exécute le programme sont déclarés, par exemple dans le cas où l'on veut faire déplacer le robot il est impératif d'initialiser les moteurs et leur donner une vitesse.

Le principal défaut des différents exemples est que la totalité du code se trouve dans la fonction main d'une classe Java. Il est donc nécessaire d'extraire ce qui est lié à l'architecture logicielle de ce code, par exemple la déclaration d'un moteur. (voir fig 5)

```
// create two motor objects to control the motors.
UnregulatedMotor motorA = new UnregulatedMotor (MotorPort.A);
UnregulatedMotor motorB = new UnregulatedMotor (MotorPort.B);

// set motors to different power levels. Adjust to get a circle.
motorA.setPower(70);
motorB.setPower(30);
```

Fig 5 : Déclaration des moteurs et de leur puissance ^[5]

Dans l'extrait de code ci-dessus, on peut voir que deux moteurs sont déclarés ainsi que l'utilisation de leur port, il est donc impératif de laisser à l'utilisateur le choix du port qu'il utilisera lors de la modélisation.

La méthode `setPower(.)` permet d'ajuster la puissance d'un moteur, c'est donc un service fourni par le composant moteur.

Avec la grammaire précédemment créée le contenu des services était inexistant, et un service était seulement défini par une signature. Pour pouvoir modéliser les différents services il est donc nécessaire de rajouter la possibilité d'avoir des boucles qui permettent plusieurs appels à un autre service.

Retour sur la grammaire

Suite à l'étude de différents exemples pour Java LeJOS, la grammaire que j'ai créée au début du stage s'est avérée incomplète. En effet, il est possible qu'un composant soit constitué d'un assemblage ou de plusieurs assemblage. De plus dans la modélisation on ne peut pas toujours faire un lien direct entre les composants matériel et les composants logiciels.

J'ai donc rajouté dans la grammaire précédente des éléments dans la règle Component. Avec le mot clé «refines» on peut indiquer le composant logiciel qui sera lié au composant matériel. Après la déclaration des différents services requis et proposés on peut déclarer un ou plusieurs assemblages appartenant au composant.

Bibliographie et sitographie

[1]LS2 N [En ligne]. [consulté le 15 mai 2018]. Disponible sur : <https://www.ls2n.fr/>

[2]A.R.d. Silva, [Model-driven engineering: A survey supported by the unified conceptual model](#)*Comput Lang Syst Struct*, 43 (Supplement C) (2015), pp. 139-155

[3]Eclipse [En ligne]. [consulté le 16 mai 2018]. Disponible sur : <https://www.eclipse.org/>

[4]Eclipse Xtext [En ligne]. [consulté le 18 mai 2018]. Disponible sur : <https://www.eclipse.org/Xtext/>

[5]STEMRobotics [En ligne]. [consulté le 29 mai 2018]. Disponible sur : <http://stemrobotics.cs.pdx.edu/node/4766>