Rapport de TER :

Automatiser la production logicielle de contrôleurs d'automates

Clément Jehanno - Loïc Mahier - Demetre Phalavandishvili

Année universitaire 2017 - 2018

Master 1 ALMA

UNIVERSITÉ DE NANTES

Remerciements

Avant toutes choses, nous tenons à remercier André Pascal, notre encadrant, pour ses conseils et sa disponibilité. Sans oublier Hugo Brunelière et Mohammed el amin Tebib pour leur aide ainsi que tous le personnel et les enseignants-chercheurs du LS2N que nous avons pu côtoyer durant ce semestre.

Table des matières

Remerciements **1**

Table des matières **2**

Introduction **3**

Etat de l'art **4**

Organisation du projet **5**

Outils utilisés **6**
- Outils collaboratifs 6
- Outils de développement 7

Première partie : développement en Java **8**
- Premier pas avec Lejos 8
- Implémentation du système de porte automatique 9
  - A partir de la spécification UML 9
  - Intégration de Lejos 16
- Réalisation d'une maquette fonctionnelle 20
- Applications 23
  - Application en Java 23
  - Application Mobile 24
- Tests 24

Deuxième partie : transformations de modèles **25**
- Découverte d'ATL 28
- Objectifs avec ATL 29
- Travail réalisé 30
  - Comprendre ATL : reproduire le modèle 30
  - Intégrer les diagrammes états-transitions 32
  - Intégrer des paramètres externes 40

Conclusion **41**

Références **42**

Glossaire **43**

Lexique **44**

Annexe **45**

Introduction

L'objectif de ce projet est d'automatiser la production logicielle : cela consiste dans ce projet exploratoire, à générer du code le plus complet possible à partir d'une spécification UML. Ainsi, nous allons travailler sur cette spécification UML pour la faire évoluer, l'enrichir, mais aussi la simplifier jusqu'à arriver à une spécification à partir de laquelle nous pouvons générer du code. Et nous entendons par là générer davantage que le squelette du code grâce notamment aux diagrammes états-transitions.

Pour ce faire nous allons dans un premier temps, à partir de la spécification, implémenter en Java un système de porte automatique. Aussi, nous souhaitions avoir un visuel, une maquette fonctionnelle. Nous allons donc utiliser un robot LEGO Mindstorm pour représenter le système. C'est pourquoi nous devrons apporter certaines modifications par rapport à la spécification de sorte que cela soit compatible en Lejos. Lejos étant une librairie Java permettant d'utiliser des fonctions compréhensible par le micro-contrôleur du robot LEGO Mindstorm. A cela s'ajoutera une application mobile permettant de contrôler à distance le système de porte automatique.

Cette première étape, focalisée sur de l'implémentation en Java, va nous permettre d'étudier la spécification et d'effectuer des modifications ou des ajouts car celle-ci n'est pas nécessairement complète ou adapté à la librairie Lejos. Il n'y a par exemple aucune information sur les ports à utiliser pour lier les capteurs au micro-contrôleur, en effet nous allons rajouter manuellement ces informations. Mais le fait de commencer par implémenter le système va nous permettre de réfléchir à la spécification UML du système réalisé en Java, et donc de mieux la comprendre.

Ainsi, une fois l'implémentation réalisé, nous aurons la spécification UML initiale et la spécification UML finale de notre cas d'étude. Puisqu'une fois le code fonctionnel, il est assez aisé de produire sa correspondance UML. C'est là que nous entrerons dans le vif du sujet puisque l'idée de ce projet est d'être capable à partir de cette spécification UML initiale d'effectuer des transformations de modèle jusqu'à obtenir le modèle UML final. Nous pourrons donc directement comparer ces deux modèles UML et voir les manques de la spécification initiale.

Dans l'optique d'automatiser le processus, il semble qu'il y ai deux approches : la première consiste à travailler sur des templates avec une génération de code qui va être basée sur un exemple concret et l'objectif sera ainsi de rendre ce template le plus générique possible. La deuxième approche, celle que nous avons choisi, porte sur l'étude des modèles. En effet nous allons chercher avec cette approche à automatiser à partir des modèles. De fait, il nous faut donc savoir comment travailler sur ces modèles et définir ce que nous cherchons à faire.

Pour ce faire nous allons utiliser le langage de transformation de modèle ATL disponible comme plugins sur Eclipse. Ce langage, permettra au travers de règle de transformation, de définir de quelle manières passer du modèle UML de la spécification initiale au modèle UML de la spécification finale. Nous aurons ainsi un certain nombre de modèle intermédiaire qui seront enrichie au fur et à mesure du processus. Cette approche aura donc pour objectif d'enrichir une spécification initiale jusqu'à ce qu'elle soit similaire à du code Java.

Pour finir, l'objectif de ce projet exploratoire est de travailler sur les modèles, et leurs transformations, à l'aide d'ATL. Nous somme conscient que c'est un sujet qui dépasse vraiment notre domaine de compétence, pour l'instant dans tous les cas. Aussi il est peu probable que nous produisions au final l'intégralité du processus de transformation via ATL, mais là n'est pas le but, dans le temps imparti.

Etat de l'art

à compléter

Organisation du projet

Pour avoir une vue globale du projet et une identification précise des différentes tâches à effectuer, nous avons dans un premier temps avec notre encadrant listé toutes les étapes à réaliser en les répertoriant dans différents sprints. En tout, nous avons défini 5 sprints. Chaque sprint constituant une tâche précise, avec une problématique à résoudre. A chaque fin de sprint nous avons décidé d'écrire un compte rendu, un rapport intermédiaire ayant pour but de garder une trace du travail effectuer, mais aussi des problème rencontrer, de nos réflexions dans le but de faciliter l'écriture du rapport final. Voyons à présent dans le détail le contenu de chaque sprint :

- Sprint 1
  - Dans ce sprint nous étions amené à configurer Lejos sur nos ordinateurs et à réaliser différents tests pour comprendre le fonctionnement des différents composants de Lejos. Cela tout en vérifiant que le matériel à notre disposition était fonctionnel.

- Sprint 2 :
  - Dans ce second sprint, l'objectif était d'étudier le modèle UML et les différents diagrammes états-transitions pour au final implémenter le programme Java avec la librairie Lejos. Puis l'idée était de réaliser une maquette du système de porte automatique en LEGO pour d'une part vérifier le fonctionnement du programme et d'autre part avoir une démonstration pour la soutenance..

- Sprint 3 :
  - Le sprint 4 constitue peut être la partie la plus importante du projet, de fait c'est dans ce sprint que nous allons travailler sur l'automatisation de la production logicielle de contrôleurs d'automates en nous basant sur le modèle UML de départ et les diagrammes états-transitions. Et cela en utilisant l'outil ATL.

- Sprint 4 :
  - L'objectif de ce sprint était la découverte de l'environnement de développement Android dans le but de développer une application mobile pour contrôler notre porte a distance en utilisant le protocole Wifi ou le protocole Bluetooth.

- Sprint 5 :
  - Enfin, ce dernier sprint est consacré à la récapitulation du projet et ainsi à la rédaction du rapport et à la préparation de la soutenance.

Outils utilisés

Outils collaboratifs

Pour améliorer l'organisation du projet et le suivi du projet, nous avons utilisé plusieurs outils collaboratif, permettant une bonne gestion et un "versionning" du projet.

Commençons tout d'abord par le logiciel de gestion du versionning du projet : Github. Ce logiciel, très connu pour tout informaticien, se base sur un logiciel de gestion de versions décentralisé nommé Git. Github est ainsi un service d'hébergement, de gestion et de développement de code. Cet outil nous permet de contrôler différentes versions de notre projet et en cas de besoin de revenir à une version antérieur. Mais le plus gros avantage d'un gestionnaire de version est la possibilité d'utiliser différente branche. Cette approche permet de travailler en parallèle sur du code sans avoir de conflit avec les versions sur lesquelles les autres membres du projet travaillent.

Parlons à présent du gestionnaire de projet en ligne Trello. Cet outil nous a été introduit par notre encadrant et permet de créer des sprints, ainsi que différentes sous tâches qui lui sont attribuées. Cela tout en les classifiants de manière à savoir où on en est dans le projet, qu'est ce qui a été fait, qu'est ce qui est à faire, etc. Ainsi nous avons établit 5 sprint comme expliqué plus haut et à chacun nous avons associé des sous tâches. Ensuite, nous avons défini 4 différentes colonnes : To do, In Progress, Review et Done. Ces colonnes permettant de savoir qu'elle était le statut de la tâche en question.

- To Do
    - Cette colonne contient toutes les tâches à réaliser pour chaque sprint, chaque sprint ayant son propre propre code couleur :
        - Sprint 1 : vert
        - Sprint 2 : jaune
        - Sprint 3 : orange
        - Sprint 4 : rouge
        - Sprint 5 : violet

- In Progress
    - Contient les tâches en cours de traitement, celles sur lesquelles quelqu'un travail actuellement.

- Review
    - Contient les tâches effectuées, tâches attendant que nous les regardions ensemble, avec l'encadrant éventuellement et que nous les validions.

- Done

- Contient les tâches terminées et validées.

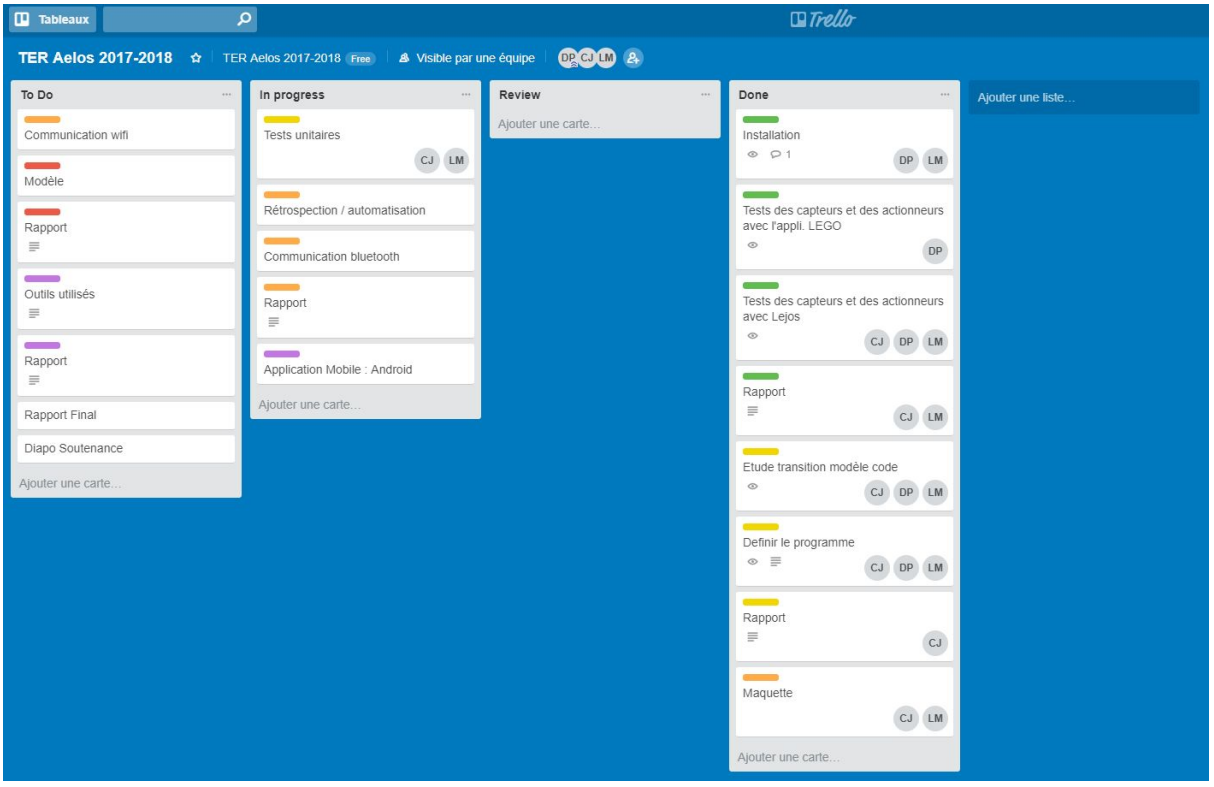


Figure 1 - Aperçu global du projet sous Trello

Voici ci-dessus à quoi ressemble l'organisation du projet sous Trello. On retrouve les 4 colonnes et le code couleur correspondant aux différents sprint. La colonnes de gauche contenant au départs la totalité des tâches à effectuer, toutes les tâches se retrouve ensuite déplacées vers la droite au fur et à mesure de l'avancement du projet avec pour objectif qu'elles finissent toutes dans la colonne la plus à droite, signifiant qu'elles sont finis.

A la fin, parlons d'une plate-forme de communication collaborative *Slack,* que nous avons utilisés pour communiquer avec notre encadrant, en particulier ca a nous permit échanger différents liens ou partager différent article scientifique.

Outils de développement

Pour réaliser ce projet nous avons eu besoin de plusieurs applications. Pour réaliser les sprints 1 et 2 du projet nous avons utilisé *Eclipse* qui est un environnement de développement en Java. Cet environnement de travail est configuré pour le développement des programmes pour *Lejos EV3,* que nous allons aborder plus tard dans le rapport.

Passons au sprint 3, c'est la partie où nous avons commencé à faire de la recherche sur l'automatisation de la production logicielle. Pour ce faire,après lecture de quelques

articles scientifiques et des cours que nous a fournis notre encadrant, nous avons identifié les outils que nous allions utiliser.

Tout d'abord, nous avons besoin d'une version spéciale de l'environnement de développement *Eclipse*, à savoir *Eclipse Modeling Tools,* qui est destiné à la modélisation des projets (voir l'annexe pour la configuration).

Ensuite, il fallait rajouter plusieurs plugins pour que nous puissions représenter les différents diagrammes sous forme numérique. Pour cela, nous avons ajouté à notre environnement de développement le plugin Papyrus. Papyrus est un outil qui fournit un environnement intégré facile à utiliser pour éditer les modèles de type EMF (Eclipse Modeling Framework), il soutient en particulier UML. L'avantage de cet outil est la génération automatique de la représentation d'un diagramme UML dans un fichier de type *XMI* (XML Metadata Interchange), qui est un standard pour l'échange d'informations de métadonnées UML. Au sein de Papyrus ce fichier a une autre extension, UML, mais en réalité il s'agit d'un fichier XMI qui est exploitable par un autre plugin que nous présenterons.

Une fois la représentation sous la forme d'un fichier XMI obtenue, nous avons besoin d'un autre outils, ATL. C'est un langage de transformation de modèle développé à Nantes, par l'équipe AtlanMod et disponible sous la forme d'un plugin Eclipse. Nous reviendrons sur ATL dans la deuxième partie du rapport.

Parlons à présent du sprint 4, qui consiste à réaliser l'application mobile sous Android, application qui permet de communiquer avec la maquette que nous avons réalisé lors du sprint 2. Pour cela nous avons utilisé un environnement de développement, *Android Studio.* Nous reviendrons la dessus à la fin de la premier partie.

Première partie : développement en Java

Avant d'aller plus loin il nous faut bien comprendre le cas d'étude sur lequel nous allons travailler, avec ses problèmes et ses spécificités techniques. C'est pour cela que nous allons commencer par l'implémenter.

Premier pas avec Lejos

Nous avons besoin de la librairie Lejos pour pouvoir programmer un code java fonctionnel sur le micro-contrôleur LEGO Mindstorm EV3. La librairie Lejos est disponible sous la forme d'un plugin Eclipse. Cependant il requiert la version 32-bits d'Eclipse sous Windows avec Java 7 d'installé. Une carte SD est aussi nécessaire, en effet la carte doit être configuré de sorte à ce que le micro-contrôleur LEGO soit compatible avec Lejos (voir Annexe 4 pour voir en détail différente étape de cette configuration).

Une fois le matériel configuré, commençons par tester l'ensemble des pièces LEGO à notre disposition : principalement les actionneurs et les divers capteurs. Cela nous permet de comprendre le fonctionnement de la librairie et notamment d'utiliser plusieurs fonctions utiles pour la suite. De fait, le système de porte automatique comportera des capteurs de contact (touch sensor) pour détecter si la porte est ouverte ou fermé mais aussi un moteur pour l'actionner.

Cette étape étant assez succincte et puisque nous évoquons déjà le système de porte automatique, passons à son implémentation. Vous trouverez en annexe une documentation détaillée pour configurer votre propre environnement Lejos sous Eclipse.

Implémentation du système de porte automatique

Tout commence par la création sous Eclipse d'un projet Lejos. Nous aurions souhaité faire un projet maven pour être sur d'avoir une configuration stable, que ce soit pour nous ou pour une réutilisation, cependant le plugin Lejos rencontre de gros problème de compatibilité avec Maven. En effet créer un projet Lejos ne suffit pas, il faut ensuite le convertir en projet EV3 de sorte à ajouter la librairie EV3 Runtime. Librairie essentielle vis à vis du robot LEGO à notre disposition (modèle EV3). C'est d'ailleurs cette librairie en particulier qui pose problème à Maven, puisque la conversion en projet EV3 ne fonctionne pas.

Ne trouvant pas de solution dans l'immédiat, nous avançons en laissant Maven de coté pour l'instant. Nous pensons y revenir un peu plus tard puisque nous prévoyons d'écrire des tests Junit, tests qui sont simple à effectuer sous Maven.

A partir de la spécification UML

Voici ci-dessous le diagramme de classe UML du système de porte automatique tel qu'il est présenté dans la spécification. Ce dernier est accompagné de plusieurs diagrammes état-transition, en l'occurrence un pour chaque composant excepté la télécommande qui en a deux. C'est sur ce premier diagramme que nous nous basons pour créer le squelette du code.

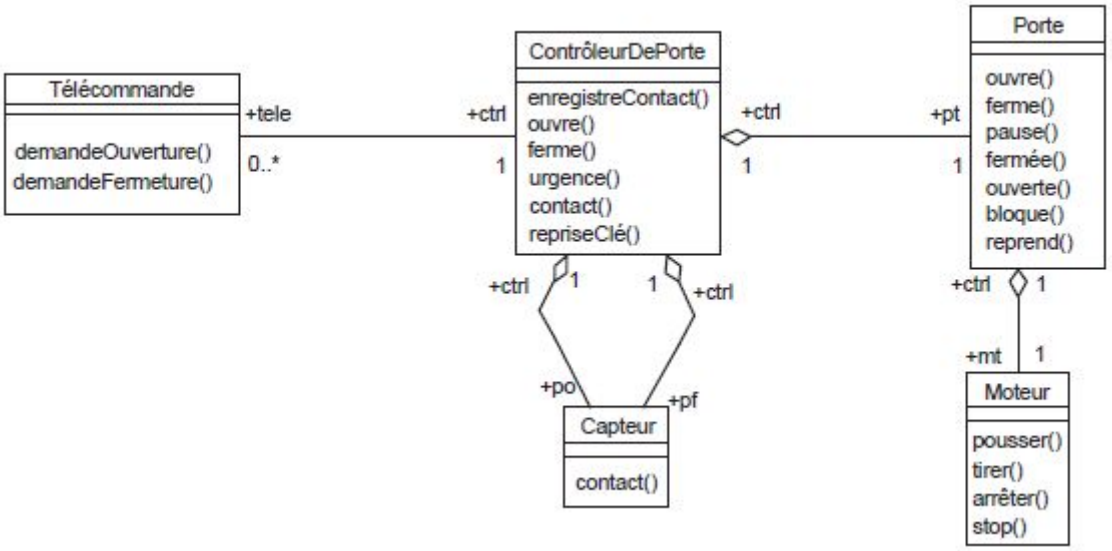


Figure 2 - Diagramme de classe

Nous commençons ainsi par créer le squelette de chacunes des classe, à partir du diagramme de classe, squelette que nous complétons dans un second temps avec les diagrammes états-transitions.

- La Classe ControleurDePorte

Nous créons donc tout d'abord la classe ControleurDePorte, dans laquelle nous pouvons déjà indiquer certains attributs et certaines méthodes à l'aide du diagramme de classe. Nous ajoutons donc 3 attributs *po* et *pf* de type *Capteur* et *pt* de type *Porte*. *po* correspond au capteur de porte ouverte et *pf* au capteur de porte fermée. *pt* représente la porte. Aussi nous ajoutons les méthodes suivantes *enregistrerContact()*, *ouvre()*, *ferme()*, *urgence()*, *contact()* et *repriseCle()* qui pour l'instant restent vides. Nous ajoutons également les "setter" et les "getter" nécessaire.

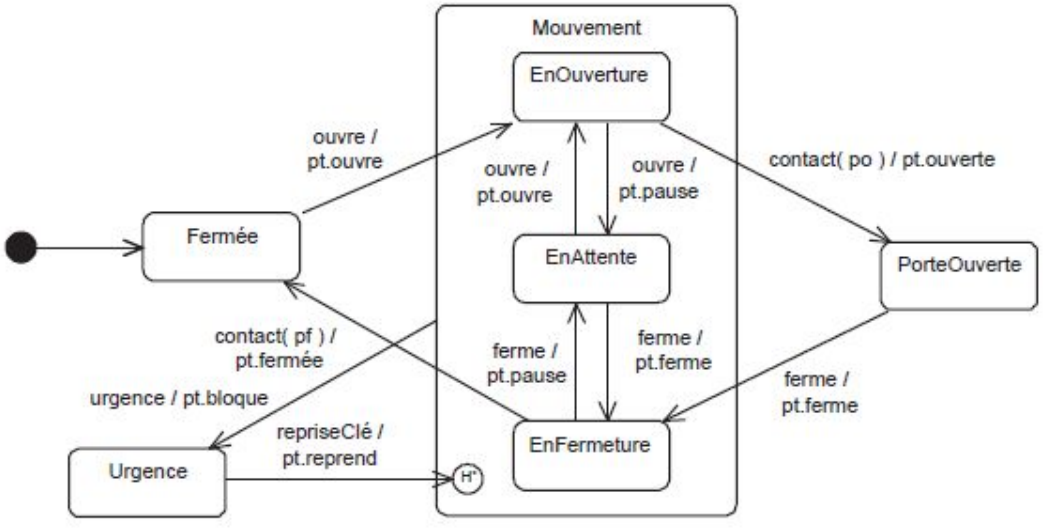


Figure 3 - Diagramme états-transitions du contrôleur

Continuons avec le diagramme états-transitions ci-dessus pour compléter notre classe. Ce dernier nous donne le contenu des méthodes et nous indique les états que nous allons devoir gérer. Avant toute chose, pour ce faire, nous pensions dans un premier temps utiliser un pattern State.

Chaque instance tel que la porte, le contrôleur, ou chaque classe qui nécessite un changement d'état, aurait été une instance du pattern. Un State à proprement parler. Les premiers soucis qui nous sont apparus sur la phase de conception étaient ceux liés au couplage : changer l'état de la porte passe aussi par changer celui du moteur, etc.

Après avoir discuté avec notre encadrant, un problème auquel nous n'avions pas pensé par manque d'expérience est celui de la volatilité. En effet, un pattern State est quelque chose d'assez lourd à mettre en place, pour un exemple aussi simple que le nôtre cela nous faisait créer une dizaine de classes, des transitions, etc. De plus cela risque de s'avérer lent et il s'avère qu'en réalité dans le diagramme de classe nous possédons un contrôleur qui va donc être en mesure de gérer ces changements d'état.

Cela ne simplifiant pas le code, voir même le complexifiant inutilement et le rendant difficilement évolutif nous avons cherché une autre option. Les énumérations semblent être une piste intéressante : en effet, nous pouvons faire exactement ce que nous voulons avec une énumération et des attributs d'états. De fait, ajouter un nouvel état nécessitera uniquement un nouvel élément dans l'énumération et non pas une nouvelle classe. Nous resterons donc sur un principe "KISS" : "Keep it simple, stupid".

Nous créons donc une énumération propre au contrôleur nommée *EnumEtatControleur*, celle-ci contient 6 états : *Fermee*, *EnFermeture, PorteOuverte, EnOuverture*, *EnAttente* et *Urgence*.

```
public enum EnumEtatControleur {
	Fermee,
	EnOuverture,
	PorteOuverte,
	EnFermeture,
	EnAttente,
	Urgence
};
```

Figure 4 - Énumération d'état pour la classe ControleurDePorte

Dans le même temps nous créons 2 attributs dans la classe *ControleurDePorte* : *EtatCourant* et *EtatPrecedant* qui stockent respectivement l'état actuel du contrôleur et son état précédant. Noter que nous devons impérativement stocker l'état précédent à cause du cas *Urgence*, en effet dans ce cas précis nous appellons la méthode *repriseCle()* qui doit permettre au contrôleur de revenir à son état précédent lorsque l'urgence est levée. Ainsi, par défaut, l'attribut *EtatCourant* est *fermee* comme l'indique le schéma, et l'attribut *EtatPrecedent* est nul.

```
public class ControleurDePorte {
	private EnumEtatControleur _etatCourant;
	private EnumEtatControleur _etatPrecedant;
	private Porte _porte;
	private Capteur _po;
	private Capteur _pf;

	public ControleurDePorte(Porte porte, EV3TouchSensor touchOuvert, EV3TouchSensor touchFerme) {
		this._etatPrecedant = null;
		this._etatCourant = EnumEtatControleur.Fermee;
		this._porte = porte;}
```

Figure 5 - Création des attributs d'états de ControleurDePorte

Puis, conformément au diagramme, nous vérifions à l'aide de conditionnelles dans chacune des méthodes que l'état courant permet d'appeler cette même méthode et nous spécifions les états que le contrôleur peut prendre à présent. De fait on ne peut pas passer d'un état *PorteOuverte* à un état *Fermee* sans passer par l'état intermédiaire *EnFermeture*. Cela doit être vérifié dans la méthode *ferme()*.

```
public void ferme() throws Exception {
	if(this._etatCourant.equals(EnumEtatControleur.EnFermeture)) {
		this._etatPrecedant = this._etatCourant;
		this._etatCourant = EnumEtatControleur.EnAttente;
		this._porte.pause();
	}
	else if(this._etatCourant.equals(EnumEtatControleur.EnAttente) ||
	this._etatCourant.equals(EnumEtatControleur.PorteOuverte)) {
		this._etatPrecedant = this._etatCourant;
		this._etatCourant = EnumEtatControleur.EnFermeture;
		this._porte.ferme();
	}
	else throw new Exception("La porte est ferme");
}
```

Figure 6 - Méthode ferme() de ControleurDePorte

Ainsi, comme vous le constatez ci-dessus les conditionnelles permettent de faire respecter le diagramme états-transitions. En effet la méthode ferme s'applique uniquement lorsque le statut est "EnAttente" ou "PorteOuverte", auquel cas l'état actuel passe à "EnFermeture" et l'état précédent est récupéré. La deuxième possibilité est que la porte se fermait déjà et qu'on demande encore de la fermer. Dans ce cas, l'état courant passe à "EnAttente" et la méthode *pause()* est appelée.

- La classe Porte

Nous créons ensuite la classe Porte, toujours à partir du diagramme de classe dans un premier temps. Nous lui ajoutons deux attributs, *ctrl* de type *ControleurDePorte* et *mt* de type *Moteur*, ainsi que 7 méthodes : *ouvre()*, *ferme()*, *pause()*, *fermee()*, *ouverte()*, *bloque()* et *reprend()*. Ces méthodes sont vides mais vont être complété de suite par le diagramme états-transitions ci-dessous. Là encore nous ajoutons les "setter" et les "getter" nécessaire.

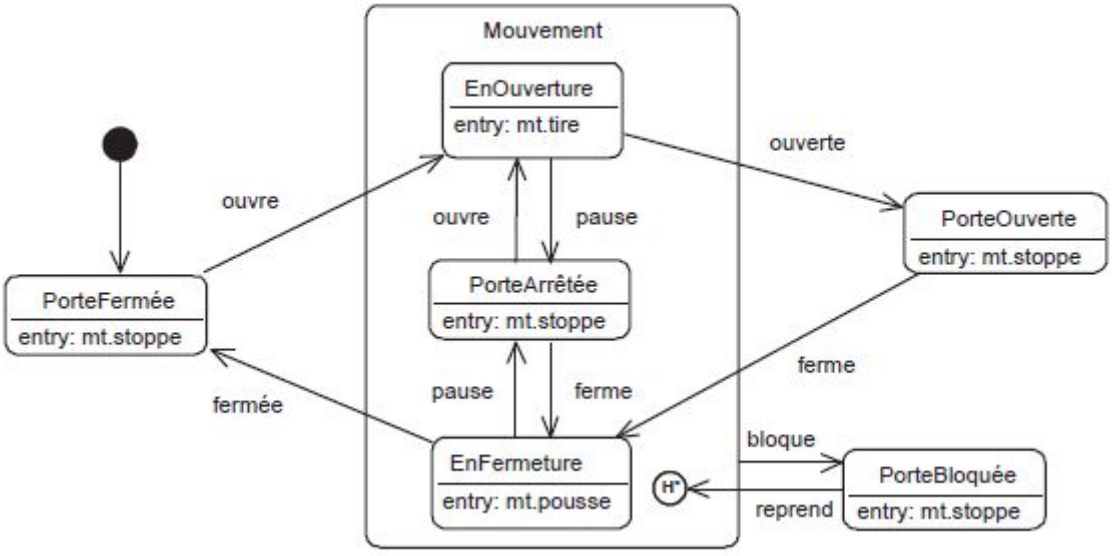


Figure 7 - Diagramme états-transitions de la porte

Complétons la classe *Porte*, pour ce faire la démarche est exactement la même que pour la classe *ControleurDePorte*. En effet nous créons là encore une énumération nommée *EnumEtatPorte* propre à cette classe. Cette énumération contient 6 états : *PorteOuverte*, *PorteFermee*, *PorteBloquee*, *EnOuverture*, *EnFermeture*, *PorteArretee*.

```
public enum EnumEtatPorte {
	PorteOuverte,
	PorteFermee,
	PorteBloquee,
	enOuverture,
	enFermeture,
	PorteArrete,
};
```

Figure 8 - Énumération d'état pour la classe Porte

De même, nous créons deux attributs *EtatCourant* et *EtatPrecedent*, puisque nous devons gérer une situation similaire au cas *Urgence* du contrôleur avec le cas *PorteBloquee* cette fois et la méthode *reprend()*. Enfin nous spécifions les transitions possibles, et ce à quelles conditions.

```
public class Porte {
	private EnumEtatPorte _etatCourant;
	private EnumEtatPorte _etatPrecedent;
	private Moteur _mt;
	private ControleurDePorte _ctrl;

	public Porte(Moteur mt) {
		this._etatCourant = EnumEtatPorte.PorteFermee;
		this._etatPrecedent = this._etatCourant;
		this._mt = mt;
	}

	public void ferme() {
		if (this._etatCourant != EnumEtatPorte.PorteFermee) {
			this._etatCourant = EnumEtatPorte.enFermeture;
			this._etatPrecedent = this._etatCourant;
			_mt.pousser();
		}
	}
```

Figure 9 - Création des attributs et de la méthode ferme() de la classe Porte

- La classe Moteur

Conformément au diagramme de classe nous ajoutons un attribut à la classe *Moteur*, *ctrl* de type *ControleurDePorte*. Puis quatre méthodes, à savoir : *pousser()*, *tirer()*, *arreter()* et *stop()* ainsi que les "setter" et "getter" nécessaire.

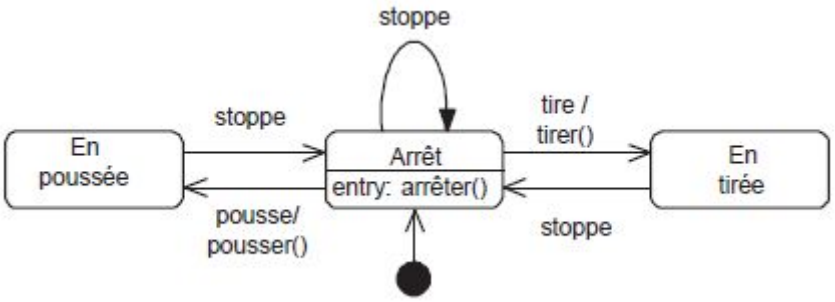


Figure 10 - Diagramme états-transitions du moteur

Pour la classe *Moteur* le diagramme états-transitions est plus simple. Ainsi, nous créons d'abord une énumération nommée *EnumEtatMoteur* contenant 3 états : *EnPoussee*, *Arret* et *EnTiree*. Puis nous créons un seul attribut *EtatCourant*. En effet il n'y a là pas de cas ou nous avons besoins de connaître l'état précédant contrairement au deux précédent. Aussi nous spécifions encore les transitions possibles et cela sous quelles conditions.

```
public enum EnumEtatMoteur {
	Enpousee,
	Entiree,
	Arret,
};
```

Figure 11 - Énumération d'état pour la classe Moteur

- La classe Capteur

Vient ensuite la classe *Capteur*, nous créons là un unique attribut *ctrl* de type *ControleurDePorte* ainsi qu'une énumérations nommée *EnumCapteurType*. Celle-ci contient les deux types de capteur nécessaire au contrôleur pour différencier une porte ouverte d'une porte fermée, à savoir *capteurPourOuverture* et *capteurPourFermeture*. Enfin nous ajoutons les "setter" et les "getter" nécessaire.

```
public enum EnumCapteurType {
	capteurPourOuverture,
	capteurPourFermeture,
};
```

Figure 12 - Énumération des types de capteur disponible pour la classe Capteur

Le diagramme états-transitions de la classe Moteur est assez simple là encore et n'apporte pas vraiment de nouveauté à notre code. La difficulté avec cette classe est davantage de gérer l'aspect passif du capteur de contact (touch sensor) à distinguer donc du capteur actif. Un capteur actif tel qu'un capteur infrarouge par exemple reçoit et émet de l'information en permanence. Là où un capteur passif tel que le capteur de contact n'envoie de l'information que lorsque qu'il y a un changement : lorsqu'il y a contact ou lorsqu'il n'y a plus de contact. Le problème est donc d'envoyer au contrôleur l'information que la porte est fermée ou ouverte.

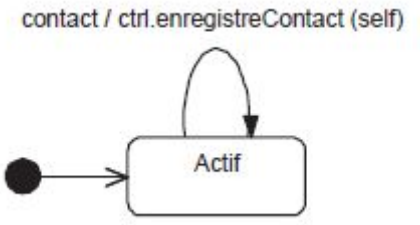


Figure 13 - Diagramme états-transitions du capteur

Pour ce faire nous choisissons d'utiliser la classe *Thread*. En effet la classe *Capteur* hérite de la classe *Thread*. Cet héritage nous permet d'exécuter en parallèle la fonction qui détecte le changement d'état du capteur physique et celle qui avertit le contrôleur pour qu'il réalise les actions correspondant à l'activation d'un capteur pour la porte fermée ou la porte ouverte.

- La classe *Telecommande*

Finissons avec la classe *Telecommande* à laquelle nous ajoutons un attribut *ctrl* de type *ControleurDePorte* avec les "getter" et "setter" associés ainsi que deux méthodes : *demandeOuverture()* et *demandeFermeture()*.

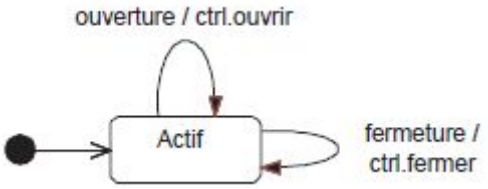


Figure 15 - Diagramme états-transitions de la télécommande

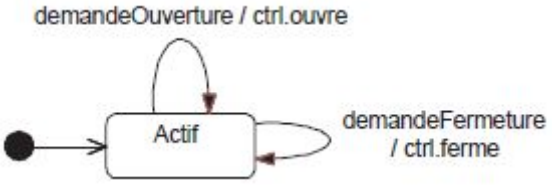


Figure 16 - Diagramme états-transitions de la télécommande (amélioré)

Deux diagrammes états-transitions sont proposés pour cette classe, nous décidons d'utiliser le deuxième. En effet la télécommande dans notre esprit, ne décide pas d'une tâche à effectuer. Celle-ci doit demander au contrôleur de l'effectuer et ce dernier l'effectue si il estime que c'est possible. Mais c'est à lui de décider, de contrôler.

Intégration de Lejos

Il faut maintenant ajouter les éléments de la librairie Lejos qui vont nous permettre de faire fonctionner la maquette que nous allons vous présenter plus loin. Nous allons donc rajouter du Lejos dans deux classes en particulier, les classes *Capteur* et *Moteur*. En effet ces deux classes correspondent à des éléments du robot LEGO pour lesquels nous devons surcharger notre code. Nous devrons ensuite dans les autres classes, notamment dans les classes *ControleurDePorte* et *Main* ajouter du Lejos de manière à instancier le moteur, les capteurs et le micro-contrôleur du robot LEGO.

- La classe Capteur

Ainsi commençons par la classe Capteur. Nous avons uniquement besoin d'y créer un attribut de type *EV3TouchSensor* et de l'instancier dans le constructeur. Voyons de plus près comment est instancié le capteur :

```
public class Capteur extends Thread {

	private ControleurDePorte _ctrl;
	private EnumCapteurType _typeCapteur;
	private EV3TouchSensor _touchSensor;

Capteur(EnumCapteurType typeCaptTactile, EV3TouchSensor touchSensor) {
		this._touchSensor = touchSensor;
		this._typeCapteur = typeCaptTactile;
}
```

Figure 17 - Instanciation du EV3TouchSensor dans la classe Capteur

Maintenant intéressons nous nous davantage au fonctionnement de la classe Capteur. Comme nous l'avons dit plus haut, cette classe hérite de la classe *Thread,* ce qui nous permet d'avoir des capteurs qui s'exécutent en "parallèle".

```
void contact() {
	while (true) {
		int size = this._touchSensor.sampleSize();
		float[] sample = new float[size];
		this._touchSensor.fetchSample(sample,0);
		if (sample[0] == 1.0) {
			this._ctrl.enregristreContact(this);
		}
	System.out.println(this._ctrl.getPorte().getMt().getEtatMoteur().toString());
	}
}
```

Figure 18 - Intégration de Lejos dans la fonction contact()

Sur la figure 10, nous observons le fonctionnement du capteur *EV3TouchSensor* qui en général possède deux états : contact (1) et pas contact (0). Dans notre cas nous ne nous intéressons qu'à l'état 1 comme on le constate ci-dessus. Au moment du contact avec la porte, le capteur fait appel à la fonction *enregistreContact()* de la classe *ControleurDePorte.* Ensuite c'est le contrôleur de la porte qui interprète ce contact.

- La classe Moteur

Passons à la classe Moteur. Nous ajoutons là un attribut de type *RegulatedMotor*, pour spécifier le type de moteur utilisé (voir Figure 10). Aussi, dans chacune des méthodes propres à la classe *Moteur* comme *pousser()*, *tirer()*, *arreter()* et *stop()*, il faut donner les instructions en Lejos.

```
public class Moteur {

	private EnumEtatMoteur _etat;
	private RegulatedMotor _mA;
	private int _vitesse;

	public Moteur(RegulatedMotor mA,int vitesse) {
		this._mA = mA;
		this._vitesse = vitesse;
		this._mA.setSpeed(_vitesse);
	}
```

Figure 19 - Instanciation du RegulatedMotor de la classe moteur

Prenons ensuite pour exemple la fonction *tirer()* de la classe *Moteur*, cette fonction est utilisée pour l'ouverture de la porte. Pour réaliser ceci, nous utilisons les méthodes présentes dans la classe *RegulatedMotor* : *setSpeed()* et *forward()*. La méthode *setSpeed()* permet définir la vitesse de rotation du moteur LEGO (ici vitesse = 20) et la méthode *forward()* permet au moteur de commencer à tourner en avance.

```
	public void tirer() {
		_mA.forward();
		setEtat(EnumEtatMoteur.Entiree);
	}
```

Figure 20 - Intégration de Lejos dans la fonction tirer()

- La classe ControleurDePorte

Voyons à présent comment nous avons instancié les capteur dans la classe *ControleurDePorte*. Nous avons là uniquement appelé le constructeur de la classe *Capteur*, en passant en paramètre des *EV3TouchSensor* comme requis ainsi que le type du capteur en question. Pour rappel il y a deux types de capteur de définis dans une énumération, à savoir *capteurPourOouverture* et *capteurPourFermeture*. Voyez plus en détail ci-dessous :

```
public ControleurDePorte(Porte porte, EV3TouchSensor touchOuvert, EV3TouchSensor
touchFerme) {
      this._etatPrecedant = null;
      this._etatCourant = EnumEtatControleur.Fermee;
      this._porte = porte;
      //initialisation des capteurs
      this._po = new Capteur(EnumCapteurType.capteurPourOuverture, touchOuvert);
      this._po.set_ctrl(this);
      this._pf = new Capteur(EnumCapteurType.capteurPourFermeture, touchFerme);
      this._pf.set_ctrl(this);
}
```

Figure 21 - Instanciation des capteurs dans la classe ControleurDePorte

- Le programme principal

Enfin, parlons de la classe *Main*, le programme principal. Nous devons tout d'abord instancier le micro-contrôleur du robot LEGO (il s'agit de la première ligne ci-dessous), puis instancier les deux capteurs et le moteur en indiquant le port sur lesquels ces derniers sont connectés au micro-contrôleur LEGO.

```
EV3 ev3 = (EV3) BrickFinder.getDefault();

EV3TouchSensor portOuverte1 =  new EV3TouchSensor(SensorPort.S2);
EV3TouchSensor portFermet1 =  new EV3TouchSensor(SensorPort.S1);

EV3TouchSensor portOuverte2 =  new EV3TouchSensor(SensorPort.S4);
EV3TouchSensor portFermet2 =  new EV3TouchSensor(SensorPort.S3);

Moteur mt1 = new Moteur(new EV3LargeRegulatedMotor(MotorPort.A),20);
Moteur mt2 = new Moteur(new EV3LargeRegulatedMotor(MotorPort.B),20);

Porte pt1 = new Porte(mt1);
Porte pt2 = new Porte(mt2);

final ControleurDePorte cp1 = new ControleurDePorte(pt1,portOuverte1, portFermet1);
final ControleurDePorte cp2 = new ControleurDePorte(pt2,portOuverte2, portFermet2);

Telecommande2ctrl teleCommande = new Telecommande2ctrl(cp1, cp2);
```

Figure 22 -Instanciation des classes dans le programme Principal

Dans notre cas, comme vous pourrez le constater sur la maquette ci-dessous, nous avons deux portes. Cela signifie que nous avons dans le main deux instanciations de *ControleurDePorte*, puisqu'un contrôleur ne contrôle qu'une porte. Avec donc deux instanciations de *Porte*, une pour chaque contrôleur, puis deux instanciations de *Moteur*, toujours une pour chaque contrôleur et enfin 4 instanciations de *Capteur*, deux pour chaque contrôleur (un capteur d'ouverture et un capteur de fermeture).

Réalisation d'une maquette fonctionnelle

Une fois le code écrit et avant de passer aux tests sous Junit nous décidons de construire la maquette pour plusieurs raisons : tout d'abord pour faire tourner le programme et s'assurer qu'il fonctionne bien, que tous les cas soient pris en compte et que les changement d'états prévus par les diagrammes états-transitions soient respectés ; ensuite en prévision de la soutenance où une petite démonstration à l'aide d'un robot LEGO amènera de l'originalité et aidera à la compréhension et à la visibilité de notre travail ; et enfin parce que cela nous apporte un peu d'amusement qu'il ne faut pas négliger dans un projet collectif.

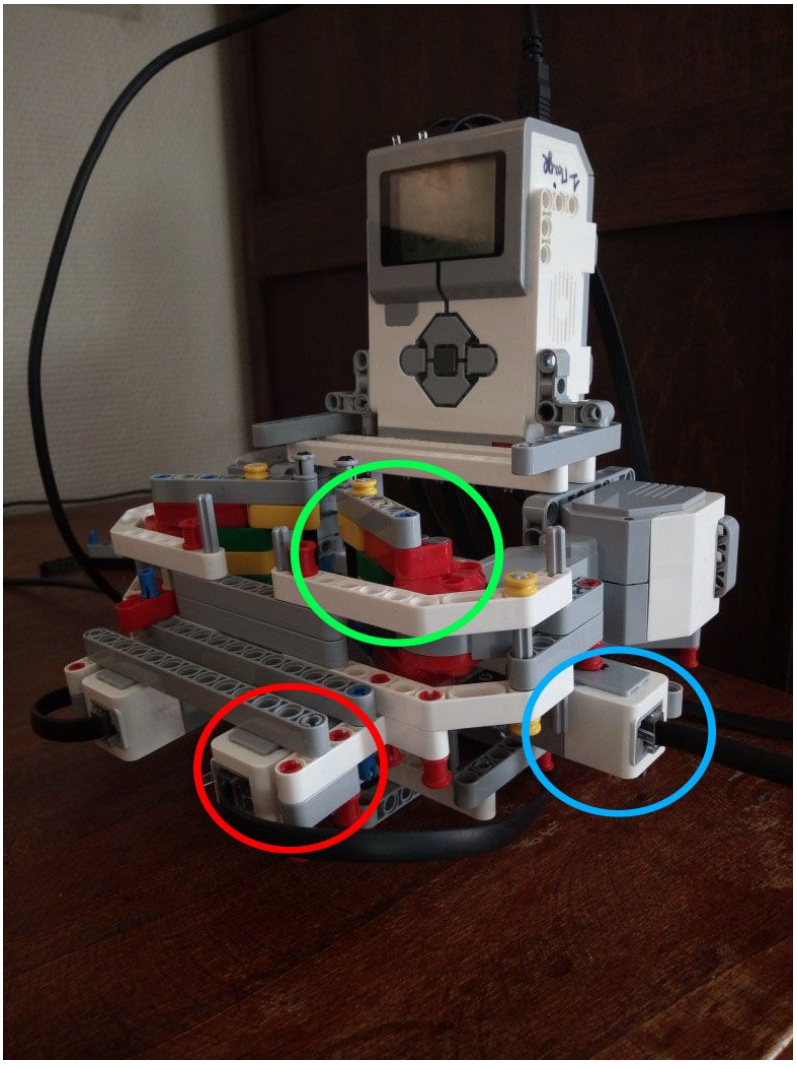


Figure 23 - Maquette de Lego

La maquette est symétrique, elle est constitué en bleu d'un capteur qui nous permet de savoir si la porte est ouverte, en rouge d'un capteur pour savoir si la porte est fermée. En vert nous avons la porte. Les capteurs sont des "Touch sensor" qui vont s'activer s'ils détectent une pression. Nous avons aussi un actionneur qui va faire bouger cette porte :

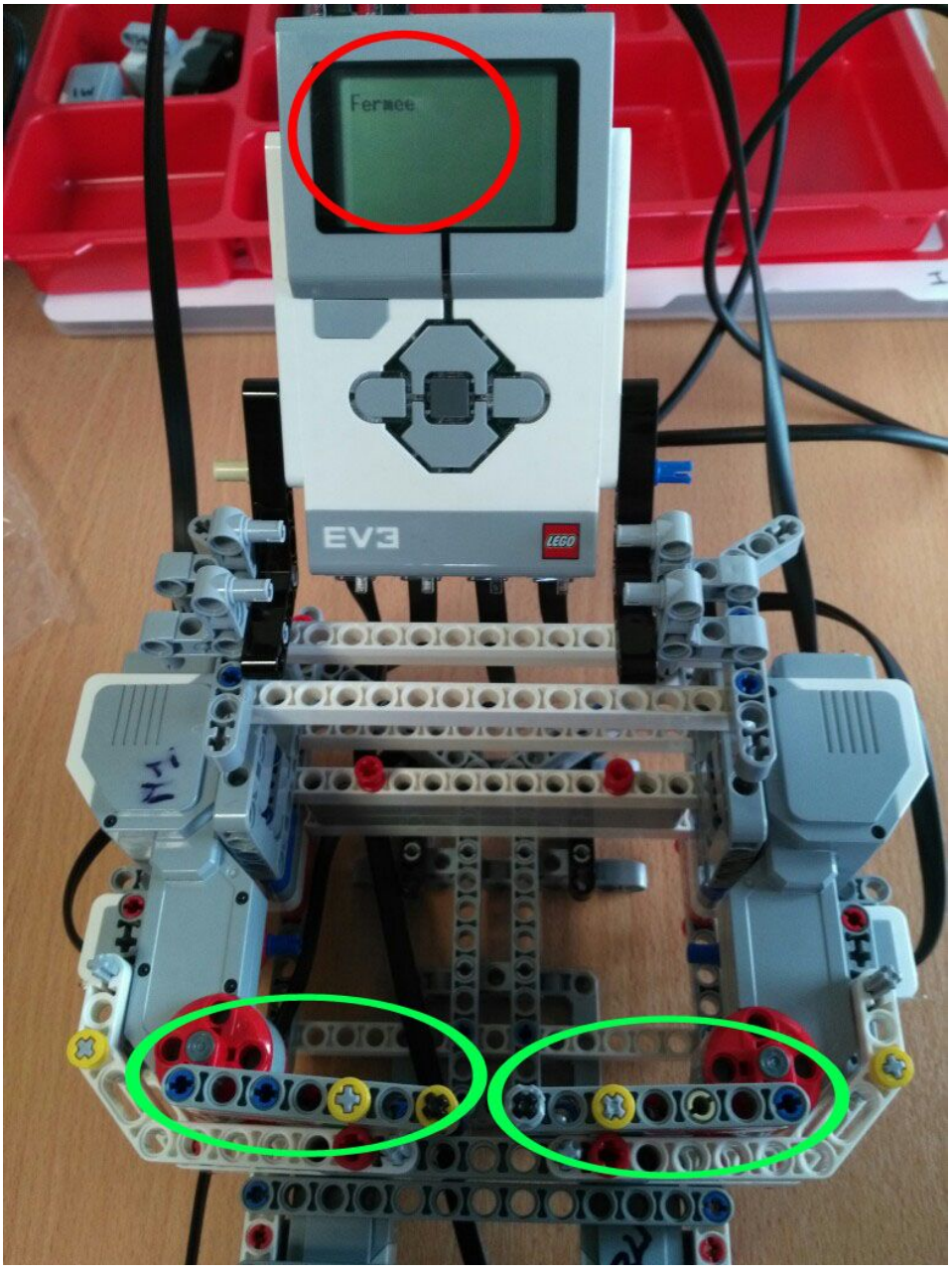


Figure 24 - Maquette portes fermées

Nous avons donc un aperçu des portes fermées en vert avec le contrôleur qui nous indique le bon état. Même chose pour les portes ouvertes :

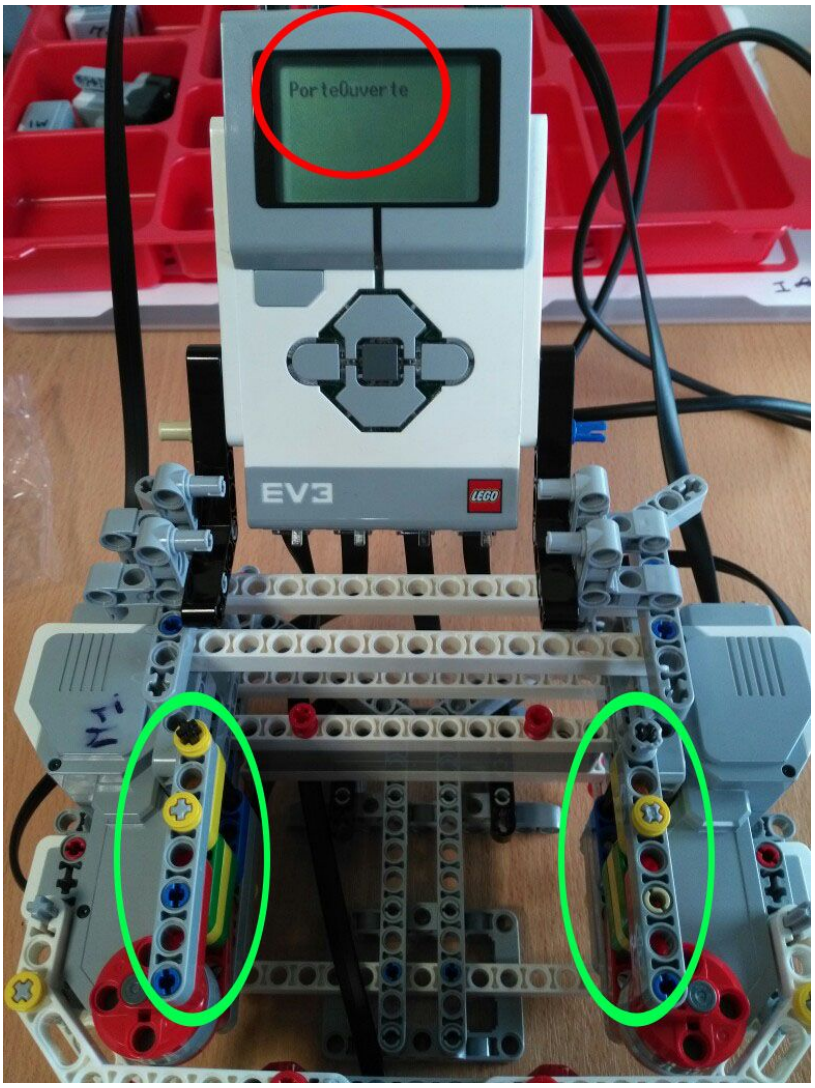


Figure 25 - Maquette portes ouvertes

Pour conclure sur cette maquette, elle n'est pas très esthétique. Cependant notre but premier est qu'elle soit fonctionnelle. D'autant que les LEGO à notre disposition ne sont pas destinés à faire un système de porte automatique mais une voiture. L'essentiel est donc de représenter une porte et de placer les capteurs de manière adéquates et de manière à ce qu'ils ne bougent pas, d'où le renforcement à ces endroits.

Une fois la maquette réalisée nous pouvons avancer sur le développement et commencer à travailler sur des applications qui vont nous permettre de contrôler cette porte à distance.

Applications

Application en Java

Dans le sprint 4 de l'organisation du projet, on devrait faire une application mobile sous Android, mais en attendant que nous développerons cet application nous avons préparé le code du code Lejos EV3 brick pour contrôler la maquette à distance avec un réseau filaire. Pour cela nous avons réalisé une application en java avec une interface graphique composée de deux boutons : Ouverture de la porte et Fermeture de la porte :

Figure 26 - Capture d'écran de l'application JAVA

Pour réaliser cette communication filaire avec la brick LEGO EV3 et l'ordinateur, nous avons besoin d'utiliser la classe *Socket* que nous avons configuré différemment sur les deux plateformes.

Parlons tout d'abord de l'implémentation de cette application, nous la configurons comme un client qui envoie des informations au serveur (dans ce cas la brick LEGO EV3). Pour cela nous utilisons la classe *Socket* en utilisant l'adresse IP de la brick (ici adresse IP par défaut : 10.0.0.1) et le port 5555. En fonction du bouton pressé sur l'application, nous envoyons la commande soit d'ouverture de la porte, soit de fermeture de la porte.

Parlons maintenant de la brick LEGO EV3 que nous configurons comme un serveur. Pour cela, nous avons introduit une classe *SocketThread*. C'est une classe qui hérite de la classe *Thread*, et qui permet donc de lancer cette classe dans un thread. Dans cette classe, nous configurons le serveur en utilisant le socket, qui écoute les informations reçus sur le port 5555. Une fois qu'une commande est émise, cette classe se charge d'interpréter la commande reçu pour appeler soit la méthode *demandeOuverture()*, soit *demandeFermeture')* de la classe *Telecommande*.

Application Mobile

Comme nous avons expliqué dans la partie de l'organisation du projet, le sprint 4 est destiné au développement l'application mobile sous Android, pour contrôler notre maquette à partir d'une mobile multifonctionnelle. Pour développer l'application mobile sous Android nous avons repris même idée que pour l'application Java vu dans la partie précédente avec une seule différence que nous communiquons avec la maquette via un bluetooth ou Wifi et non pas avec la câble USB.

Pour développer ce projet nous avons utilisé l'environnement de développement *Android Studio* qui nous a permis facilement développer l'interface graphique (voir figure 27). Ensuite nous avons commencé à implémenter la fonctionnalité suivante : la communication de mobile avec le brick EV3 en utilisant le réseau sans fil *Bluetooth*.

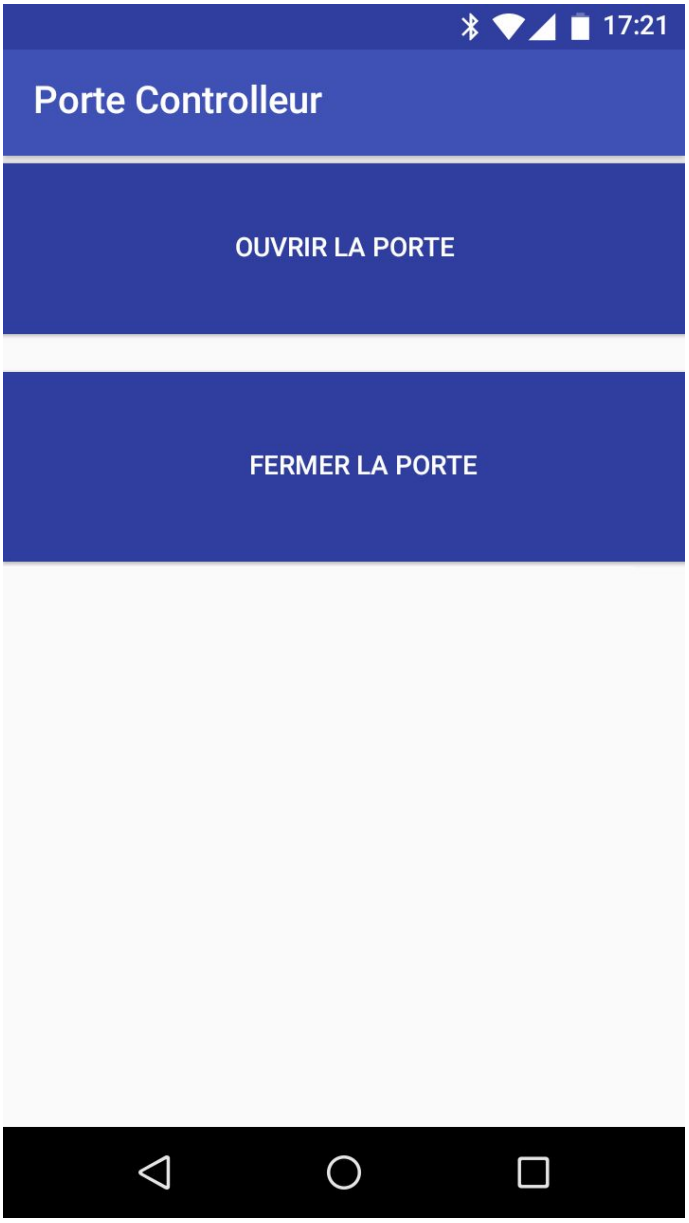


Figure 27 - Interface de l'application mobile

a completer

Tests

Comme mentionné plus haut dans le rapport, nous avons essayé tout au long de la phase d'implémentation d'écrire des tests, notamment unitaire. Cependant, cela ne s'est pas déroulé comme prévu, revenons donc sur quelques uns des tests que nous avons écrit puis nous parlerons de pourquoi ils ont été plus difficiles à mettre en place que prévu.

Les tests qui nous intéressaient étaient ceux concernant nos états. Il s'agissait de vérifier simplement que la porte était dans le bon Etat si on effectuait la bonne transition.
Par exemple nous avons ici le test de l'ouverture et de la fermeture de nos portes :

```
@Test
public void testOuvre(){
	// pas besoin de changer l'état car la porte est fermée donc on peut l'ouvrir
	p.ouvre();
	assertEquals(p.getEtatCourant(), EnumEtatPorte.enOuverture);
}

@Test
public void testFerme(){
	// besoin de changer l'état car la porte est fermée
	p.setEtatCourant(EnumEtatPorte.PorteOuverte); //met en ouverture
	p.ferme();
	assertEquals(p.getEtatCourant(), EnumEtatPorte.enFermeture);
}
```

Figure 28 - Extrait de tests java

Nos tests bien que basiques étaient censés simuler des actions permettant de passer d'un état à l'autre, cependant nous avons rencontré un certain nombre de problème à les effectuer, ce qui est principalement dû nous le pensons à la librairie Lejos.

En effet, nous l'évoquions au début de la phase d'implémentation, nous n'avons pas pu configurer notre projet sous Maven pour cause d'incompatibilité avec Lejos à première vue. Ce qui est regrettable tant cela aurait simplifier nos test, de fait nous sommes habitué à lancer nos tests sous Maven. Nous avons donc essayé de les effectuer directement, avec Junit mais toujours sans succès. Aussi, faire des tests étant à la fois utile et nécessaire mais ne répondant ni à l'objectif premier ni même à l'objectif secondaire du TER, nous avons choisi de les laisser de côté et d'avancer vers la partie transformation de modèle.

Nous avons donc une implémentation concrète de notre modèle avec une compréhension plus globale du système et de son fonctionnement, nous avons pensé l'implémentation et donc eu des problèmes à traiter, problèmes qui vont probablement nous servir pour la suite. En effet, une fois la maquette mise en place il est temps de passer à la partie plus théorique du sujet : la transformation de modèles.

Deuxième partie : transformations de modèles

Comme nous l'évoquions dans l'introduction, maintenant que nous avons un programme fonctionnel nous allons travailler sur les modèles. Ce que l'on constate dans un premier temps, c'est que si l'on essaie de générer du code à partir de la spécification fournit via un outil de génération de code existant, alors on obtient un squelette vraiment très pauvre. En effet, les diagrammes états-transitions ne sont pas pris en compte.

Par exemple nous sommes parfaitement capables de générer le code du modèle de base (voir Annexe 1). Avec Acceleo par exemple, nous avons de la génération automatique qui nous donne ce genre de squelette :

```
// Start of user code (user defined imports)

// End of user code

/**
 * Description of Capteur.
 *
 * @author Clément
 */
public class Capteur {
	// Start of user code (user defined attributes for Capteur)

	// End of user code

	/**
	 * The constructor.
	 */
	public Capteur() {
		// Start of user code constructor for Capteur)
		super();
		// End of user code
	}

	/**
	 * Description of the method contact.
	 */
	public void contact() {
		// Start of user code for method contact
		// End of user code
	}

	// Start of user code (user defined methods for Capteur)

	// End of user code

}
```

Figure 29 - Extrait de code pauvre fourni par Acceleo

Ce que l'on constate également, c'est que nous avons pris énormément de liberté sur certains éléments du programme. En effet, la spécification donne des informations pour faire le squelette du programme, voir même davantage avec les diagrammes

états-transitions, mais de nombreuses informations sont manquantes, notamment en ce qui concerne la librairie Lejos.

Par exemple, dans le code de la figure 22 présenté plus haut (copiée ci-dessous) nous attribuons nous-même les moteurs et les capteurs sur les bons ports car nous avons configuré l'EV3 et donc nous savons comment cela se passe.

```
EV3 ev3 = (EV3) BrickFinder.getDefault();

EV3TouchSensor portOuverte1 =  new EV3TouchSensor(SensorPort.S2);
EV3TouchSensor portFermet1 =  new EV3TouchSensor(SensorPort.S1);

EV3TouchSensor portOuverte2 =  new EV3TouchSensor(SensorPort.S4);
EV3TouchSensor portFermet2 =  new EV3TouchSensor(SensorPort.S3);

Moteur mt1 = new Moteur(new EV3LargeRegulatedMotor(MotorPort.A),20);
Moteur mt2 = new Moteur(new EV3LargeRegulatedMotor(MotorPort.B),20);

Porte pt1 = new Porte(mt1);
Porte pt2 = new Porte(mt2);

final ControleurDePorte cp1 = new ControleurDePorte(pt1,portOuverte1, portFermet1);
final ControleurDePorte cp2 = new ControleurDePorte(pt2,portOuverte2, portFermet2);

Telecommande2ctrl teleCommande = new Telecommande2ctrl(cp1, cp2);
```

Figure 22 (bis) - Instanciation des classes dans le programme Principal

Nous avons aussi décidé de fixer la vitesse des moteurs à 20 (cf Figure 22 ci-dessus) car cela nous semblait bien pour nos tests, mais ce genre d'information n'est pas une information fournie par le diagramme de classe, il faut la récupérer ailleurs.

Comme nous venons de le voir nous avons donc deux points bloquants : les diagrammes états-transitions et les paramètres spécifiques à Lejos. Il serait donc intéressant de prendre notre diagramme de classe initial et de le compléter à l'aide des diagrammes états-transitions. Voir même de le simplifier pour s'approcher du langage Java utilisé. Aussi, ajouter des paramètres Lejos à notre modèle UML aiderait énormément. C'est là qu'entre en jeu un outil tel que ATL, permettant de transformer des modèles, comme notre diagramme de classe par exemple.

D'une manière plus générale et pour reprendre les idées introduites au début de ce rapport, il faut bien comprendre que les modèles fournis par les outils existants ne sont pas en adéquation complète avec ce que nous souhaitons faire. Ils sont pour la plupart incomplets car ils n'arrivent pas à introduire des paramètres externes ou bien les diagrammes états-transitions.

L'idée de ce travail est de comprendre quels sont les problèmes liés à la génération automatique de code et d'essayer d'apporter notre brique à l'édifice afin de compléter les choses déjà existantes. Notamment en ce qui concerne des intégrations particulières de paramètres externes tel que des DET ou des informations spécifiques à un projet (un port spécifique sur lequel se connecter, une librairie ou un module précis, etc.)

Découverte d'ATL

Avant d'introduire ATL il faut prendre un peu de recul sur le projet. Comme dit précédemment nous cherchons à comprendre quels sont les problèmes liés à la génération automatique de code. Des outils sont existants, tel que QVT qui est un standard de l'OMG mais nous avons rapidement été guidés vers ATL par notre encadrant.

Comme dit plus haut, ATL est un langage de transformation de modèle développé à Nantes, par l'équipe AtlanMod et disponible sous la forme d'un plugin Eclipse. Plus précisément, ATL permet de faire des transformations au niveau des modèles. C'est un outil assez complexe mais très puissant nous allons donc revenir sur le fonctionnement de base d'ATL.

Nous travaillons sur des modèles, dans notre cas il s'agit de modèles UML. Ces modèles sont définis par des méta-modèles. Pour faire une analogie avec quelque chose qui est peut-être plus familier prenons l'exemple d'un langage. Un langage est défini par une grammaire, un modèle est défini par un méta-modèle.

Les métamodèles vont définir nos modèles d'entrée mais aussi nos modèles de sortie. Ensuite ATL intervient, nous avons donc d'une part le méta-modèle (MM) source avec le modèle source et d'autre part le métamodèle cible, mais il faut produire le modèle cible : c'est là qu'intervient ATL. Pour résumer voici un schéma explicatif du système :

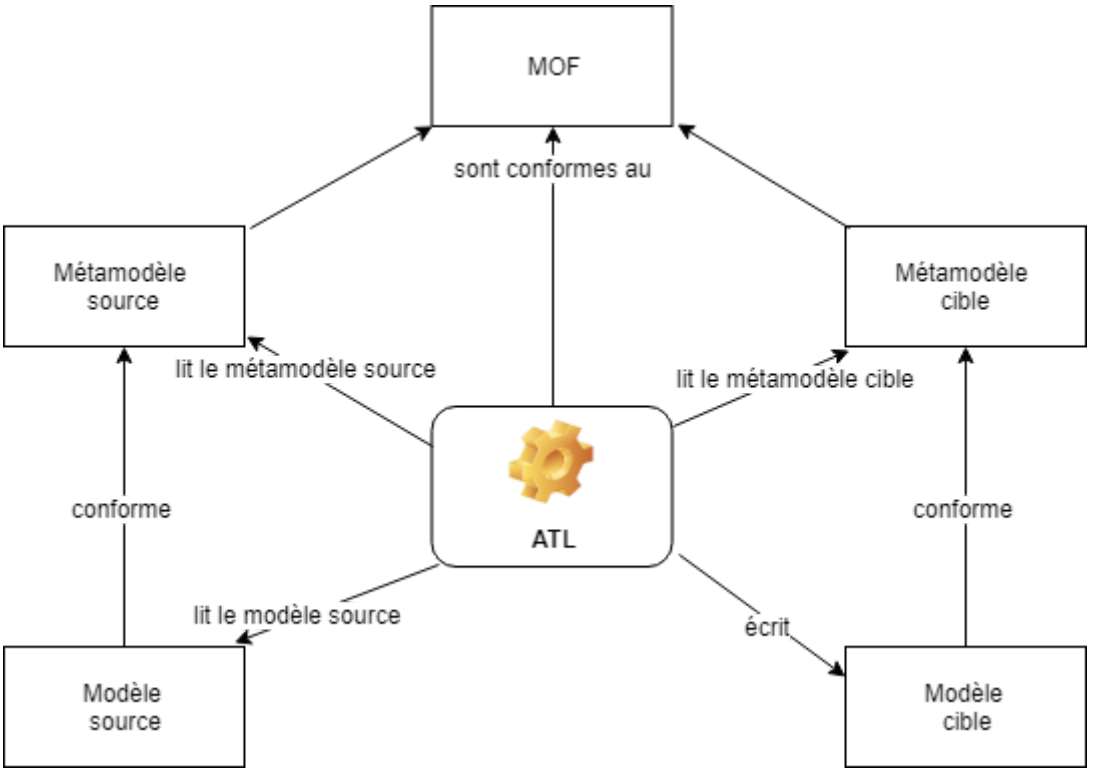


Figure 30 - Fonctionnement d'une transformation ATL

Dans la figure 30, nous voyons le schéma du fonctionnement de la transformation ATL appliqué au modèle source pour produire le modèle cible. Commençons par le noeud MOF, *Meta-Object Facility*, c'est un standard de l'*Object Management Group* (OMG) qui permet la représentation et la manipulation des MM. Dans l'architecture de modélisation, le standard MOF se situe au niveau M3 parmi les 4 couches existantes. Dans ce projet, notre travail se base sur les niveaux M1 et M2 (voir figure 31).

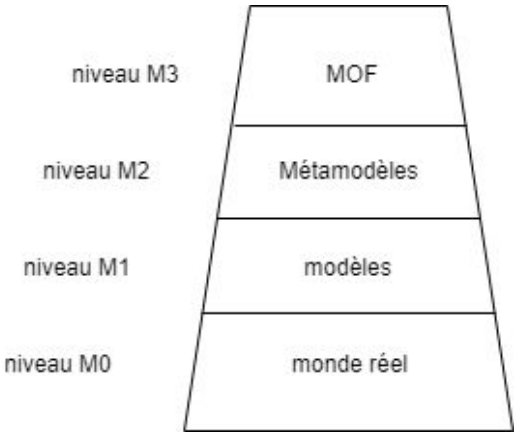


Figure 31 - Architecture de modélisation

Revenons sur la figure 31, dans notre cas le modèle source correspond au diagramme de classe UML de la figure 2 que nous avons représenté sous la forme de *XMI* a l'aide de plugin *Papyrus* disponible sous *Eclipse*. Au début notre modèle source est le diagramme de classe UML, qui est conforme au méta-modèle source qui représente UML de version 2.5.

Ce méta-modèle UML contient toutes les informations sur l'organisation des différents diagrammes UML, en particulier les diagrammes de classes et les diagrammes états-transitions. Dans notre cas les méta-modèle source et cible sont identiques, parce que à chaque transformations nous ajoutons plus de détail à notre modèle.

La règle ATL qui se trouve au milieu du schéma prend en entrée deux méta-modèle (source et cible) et le modèle source sur lequel nous appliquons la transformation. Nous allons voir plus en détail l'organisation d'une règle ATL dans la partie suivante.

Objectifs avec ATL

En entrée nous allons donc fournir des modèles UML : un diagramme de classe et des diagrammes états-transitions (DET) présentés plus haut. L'objectif en partant de ces diagrammes est de réussir à faire évoluer notre modèle théorique UML au modèle proche du code que nous avons implémenté. Nous partons donc du modèle de base [1] afin d'arriver vers un modèle plus complexe qui est la représentation concrète de notre code [2] .

Comme nous pouvons le voir il y a une énorme marge entre le modèle d'entrée fourni et le modèle concret obtenu par le code. Il faut prendre du recul sur le modèle puisqu'ici nous travaillons sur un cas concret, le travail que nous réalisons sur ATL est quelque chose de complexe puisqu'il ne doit pas s'attacher au modèle mais bien au méta-modèle sur la transformation. Par exemple un de nos problèmes est d'intégrer un diagramme états-transitions en le transformant en énumérations et ce quelque soit les classes, le nom de la classe, etc. Dans la section suivante nous allons revenir sur nos travaux et voir quels ont étés les différents problèmes rencontrés.

Travail réalisé

Dans cette section nous allons dresser un bilan de nos travaux et nous allons voir jusqu'où nous avons pu aller sur les transformations de modèles.

Comprendre ATL : reproduire le modèle

Notre premier travail a été de comprendre comment fonctionne ATL, pour commencer nous avons donc essayé de reproduire le diagramme de classes fournit en entrée. Cette transformation nous a pris beaucoup de temps puisque c'était la première et qu'il fallait donc appréhender l'outil qu'est ATL. Au début nous avons essayé de prendre les éléments de ATL et de les recopier un par un, ce qui nous donne ce genre de règles :

```
create OUT : MM1 from IN : MM;

rule CopieColleClasse {
	from
		m1 : MM!Class
	to
		m2 : MM1!Class (
			name <- m1.name,
			ownedOperation <- m1.ownedOperation,
			ownedAttribute <- m1.ownedAttribute
		)
}
rule CopieOps {
	from
		m1 : MM!Operation
	to
		m2 : MM1!Operation (
			name <- m1.name
		)
}
```

Figure 32 - Extrait de règles ATL

```
rule CopieProperty {
	from
		s : MM!LiteralInteger
	to
		m2 : MM1!LiteralInteger (
	__xmiID__ <- s.__xmiID__,
	name <- s.name,
	visibility <- s.visibility,
	isLeaf <- s.isLeaf,
	isStatic <- s.isStatic,
	isOrdered <- s.isOrdered,
	isUnique <- s.isUnique,
	isReadOnly <- s.isReadOnly,
	isDerived <- s.isDerived,
	isDerivedUnion <- s.isDerivedUnion,
	aggregation <- s.aggregation,
	eAnnotations <- s.eAnnotations,
	ownedComment <- s.ownedComment,
	clientDependency <- s.clientDependency,
	nameExpression <- s.nameExpression,
	type <- s.type,
	upperValue <- s.upperValue,
	lowerValue <- s.lowerValue,
	templateParameter <- s.templateParameter,
	deployment <- s.deployment,
	redefinedProperty <- s.redefinedProperty,
	defaultValue <- s.defaultValue,
	subsettedProperty <- s.subsettedProperty,
	association <- s.association,
	qualifier <- s.qualifier)
}
```

Figure 32 bis - Extrait de règles ATL

Comme nous pouvons le voir avec ce bref extrait de règles nous avons à faire à des règles compliquées qui nous demandent de connaître et de spécifier tous les paramètres de

chaque classe, chaque attribut, d'une manière globale, de chaque élément décrit pas le modèle UML. Nous nous sommes très vites rendus compte que c'était ingérable de procéder ainsi et que en plus de cela, il nous faut tout reprendre dès lors qu'on oublie un nouvel élément par exemple.

Comme nous l'avons dit nous avons besoin de transformations qui doivent absolument être généralistes et ne pas s'attacher au modèle. Autrement dit, si nous travaillons sur notre transformation pour notre modèle précis mais que par exemple il ne contient pas de composition, si on fournit en entrée un modèle qui contient une composition alors notre transformation n'est plus viable.

Après des recherches et des échanges avec Hugo Brunelière, ce dernier nous a parlé du mode "refining de ATL" qui nous permet de recopier le modèle. Outre cela, le mode "refining" nous permet aussi de modifier quelques éléments, par exemple il nous permet de modifier les éléments de type classe, ainsi les règles deviennent plus simples :

```
-- @nsURI MM=uri:http://www.eclipse.org/uml2/5.0.0/UML

create OUT : MM refining IN : MM;

rule Class2Class {
    from
        m1 : MM!Class (
            )
    to
        m2 : MM!Class (
            name <- 'test'.concat(m1.name)
         )
}
```

Figure 33 - Extrait de code ATL

La principale différence se fait au niveau du mot clé "refining" qui signifie à ATL que nous voulons recopier le modèle, dans l'exemple ci-dessus nous ne faisons qu'une modification du nom des classes. De plus, sur l'extrait de code ATL, nous voyons que le mode refining de ATL prend en compte un métamodèle source (IN : MM) et métamodèle cible (Create OUT : MM).

Nous arrivons donc à prendre le diagramme de classes tel qu'il est donné en entrée et à le recopier. Désormais il faut rajouter d'autres informations afin de se rapprocher du modèle du code. Nous avons donc décidé de commencer par intégrer les diagrammes état transition.

Intégrer les diagrammes états-transitions

Les DET sont une partie complexe et très importante du problème puisqu'ils viennent compléter le diagramme de classes. Ainsi, si on trouve un moyen de les intégrer ne serait-ce qu'au squelette, pour des diagrammes très complexes cela ferait gagner beaucoup de

temps aux développeurs. Sur certains modèles on pourrait peut-être générer des classes sous la forme d'un State pattern mais dans ce cas présent nous travaillons sur des énumérations.

Le premier souci est qu'il ne faut pas s'attacher à la classe. Prenons l'exemple d'un DET qui va définir les états d'un moteur. On l'appelle *StateCharteMoteur* et on veut qu'il opère sur la classe *Moteur* sauf que lors de travaux à l'étranger nos collègues nous donnent un diagramme de classe avec une classe *Motor*, nous avons donc un souci.

Il faut donc définir une convention et elle se doit d'être plus généraliste qu'un simple nom de classe. Par exemple admettons que l'on ai des actionneurs mais que ces actionneurs aient des noms de classe "capteur infrarouge" ou "détecteur de mouvement" on ne peut pas se baser sur le nom des classes puisqu'ils sont tous différents. L'idéal est de trouver un attribut à définir dans le modèle UML.

Pour les diagrammes de classe nous pouvons définir un *Stereotype* dans le modèle UML, pour les DET il s'agira d'un "Use case" qui sera définit dans l'énumération. Ensuite il faut intégrer le DET au diagramme de classe. Pour ce faire nous ne pouvons nous passer d'une intervention extérieure.

Nous allons reprendre nos transformations qui sont au nombre de 5 afin de comprendre le processus complet d'évolution du modèle mais avant de rentrer dans l'aspect plus technique voici un schéma explicatif.

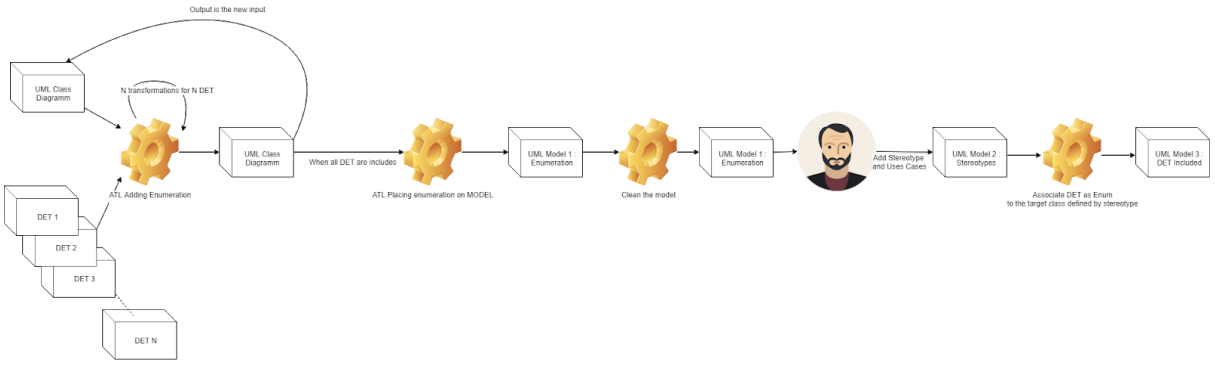


Figure 34 - Processus de transformation

Pour plus de clarté le schéma est disponible en annexe [3]. Nos premières transformations (au nombre de deux actuellement) sont celles de transformations du modèle en y intégrant les DET sous forme d'énumération :

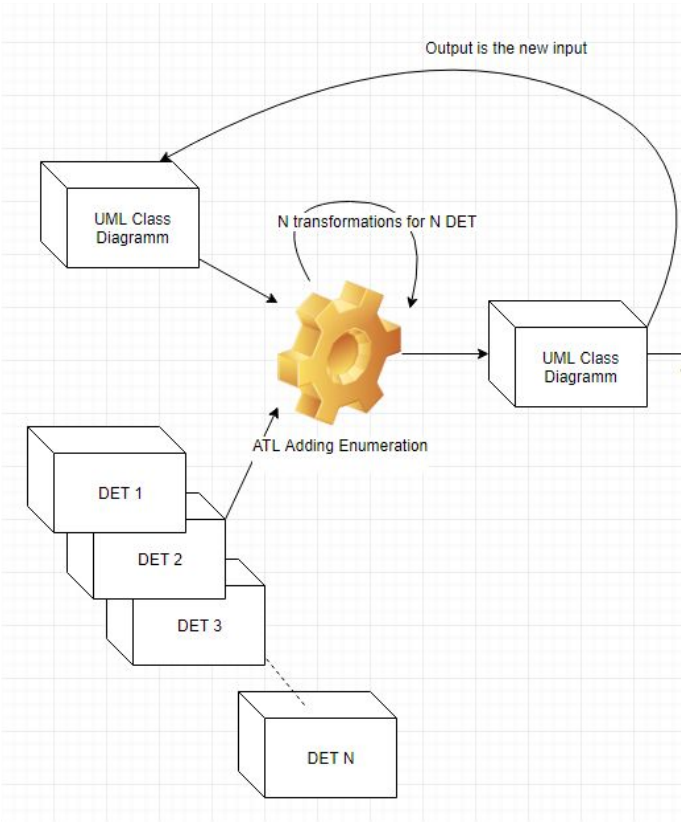


Figure 35 - Premières Transformations

Sur ce chapitre nous allons revenir sur les transformations que nous effectuons. Nous partons donc du modèle de base (voir Annexe 1) pour arriver au modèle du code. Le premier point va être d'ajouter des énumérations à ce premier modèle. Pour visualiser notre UML nous utilisons le modèle UML Model Editor de Eclipse qui présente le modèle sous une forme arborescente ce qui nous permet de suivre plus précisément qu'un diagramme. Nous pouvons donc observer notre modèle actuel :

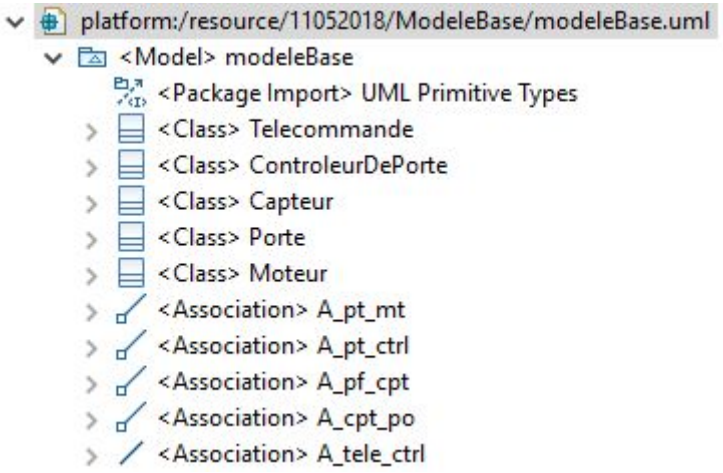


Figure 36 - Modèle de base en UML

D'autre part nous avons notre diagramme états-transitions de la classe porte :

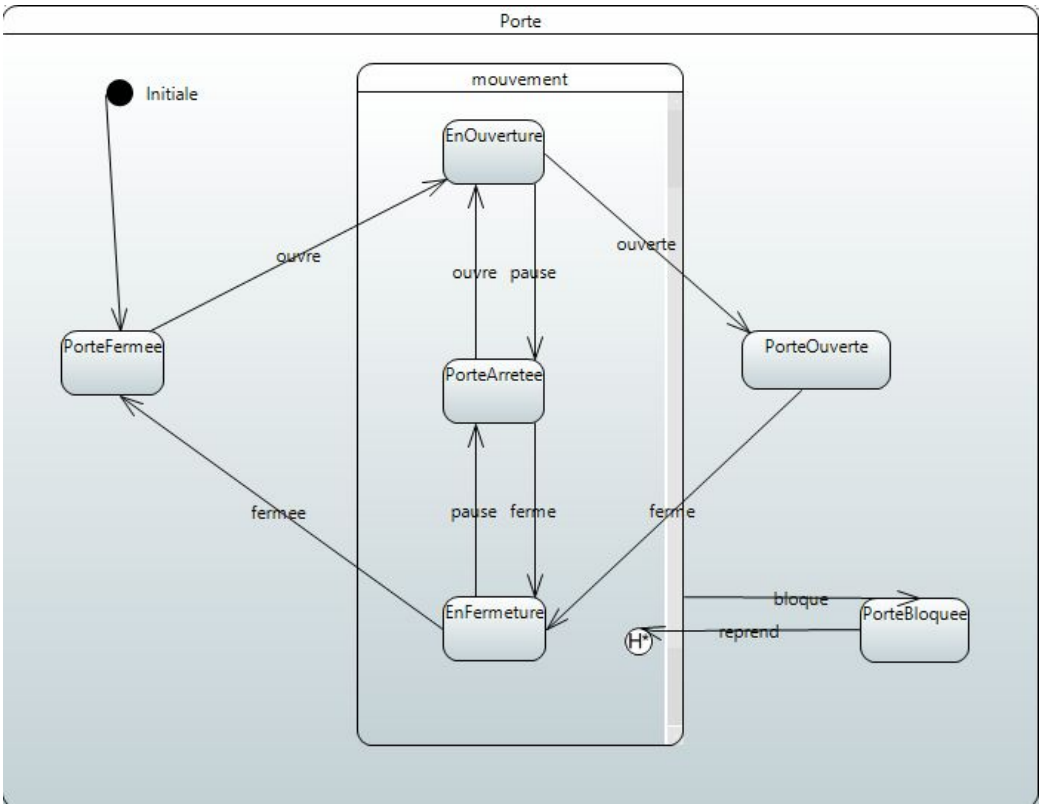


Figure 37 - Diagramme état transition de la porte

Qui est aussi exploitable sous sa forme UML :

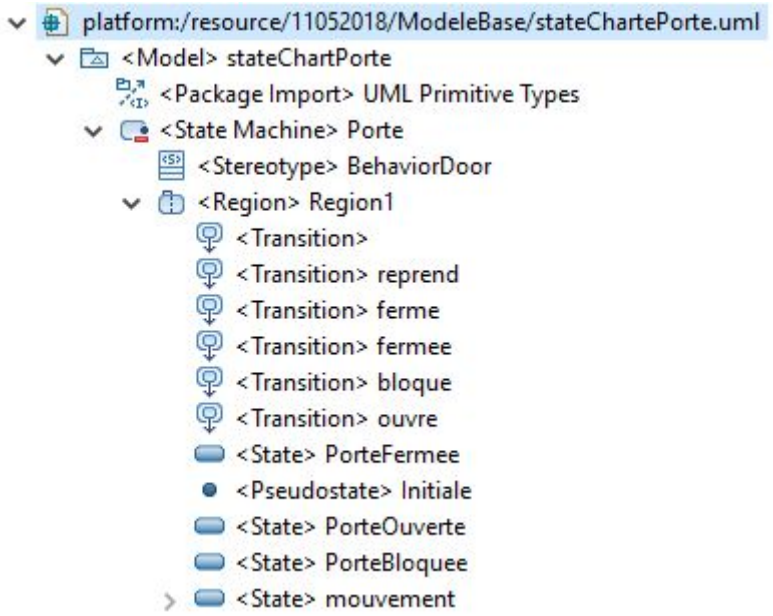


Figure 38 - DET de la porte sous forme UML

Il faut donc l'intégrer au diagramme de classe.

Dans cette étape on parcours les DET et on va convertir chaque état comme un champ "EnumerationLiteral" qui sera inclus dans une seule "Enumeration". Si on parcourt un DET nommé "StateCharteControleur" on va créer une énumération "EnumControleur" et tous les états contenu dans le DET vont se transformer comme des valeurs de l'énumération.

Pour effectuer cette intégration, il faut diviser nos règles ATL en plusieurs parties : Commençons par créer une Enumeration du nom du StateChart :

```
helper def : getOneEnumLit :
    Sequence(MM!EnumerationLiteral) =
        MM1!State.allInstances();


rule Stat2Enum{
    from
        etat : MM1!StateMachine
    to
        enum2 : MM!Enumeration(
            name <- 'enum'.concat(etat.name.toString()),
            ownedLiteral <- thisModule.getOneEnumLit
        )
}
```

Figure 39 - Règles de transformation des Énumérations

Cette règle prend tous les "StateMachine" et les transforme en "Enumeration", en copiant leur Nom concaténé avec le nom de l'état, ce qui va créer une énumération du nom "enumPorte" pour le statecharte de la Porte.

Cependant, cette règle pour s'effectuer utilise un helper (ce qui équivaut à une fonction en ATL) qui va nous permettre de remplir les "ownedLiteral" de l'énumération car au départ l'énumération est vide. Pour la remplir il va falloir donner les différentes valeurs de l'énumération possibles. Ces valeurs sont en réalité le nom de nos Etats.

Hors, il est impossible d'inclure nos Etats tel quel, il faut pour cela passer des EnumerationLiteral, qui seront directement créés à partir de nos Etats, il faut donc les construire :

```
rule stateToEnumLit{
    from
        etat : MM1!State
    to
        m2 : MM!EnumerationLiteral (
            name <- etat.name
        )
}
```

Figure 40 - Règle de transformation des EnumerationsLiteral

Grâce à cette règle nous parcourons nos Etats pour créer des EnumerationLiteral qui vont les représenter. A priori, ATL va commencer par appliquer cette règle et créer nos EnumerationLiteral, ensuite il va appliquer les règles pour créer l'Enumeration et la remplir correctement. Une chose est sûre, on ne peut pas remplir nos Enumerations comme nous l'avons fait si nous n'avons pas créé au préalable ces EnumerationLiteral.

La transformation nous permet donc d'intégrer un diagramme état-transition, il faut donc la répéter autant de fois qu'il y a de DET. Une fois qu'ils sont tous ajoutés au modèle, on les replace car l'ajout ne se fait pas proprement et les énumérations ne sont pas présentes au bon endroit :

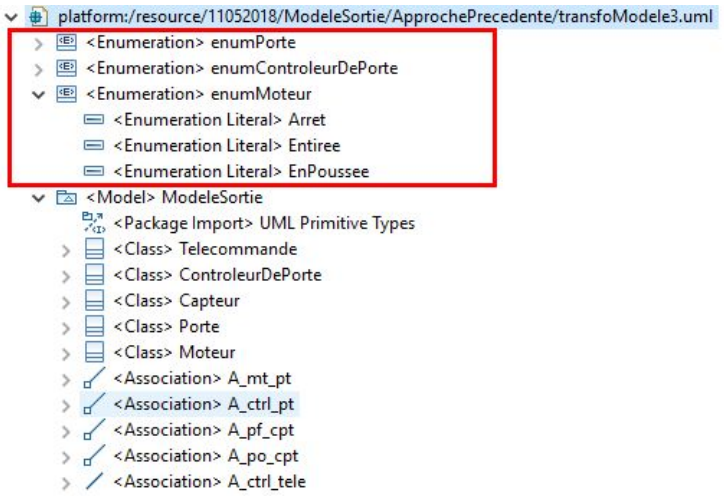


Figure 41 - Enumeration mal placées

Pour les déplacer nous procédons en deux temps : premièrement nous copions les énumérations et leur contenu, mais en procédant ainsi nous n'arrivons pas à enlever les énumération de là où elles sont, nous arrivons à déplacer le contenu de l'énumération mais pas à supprimer la première énumération. Cela semble être dû au mode refining de ATL qui permettrait soit de copier, soit de supprimer, mais pas les deux en même temps. Il faudra donc dans un second temps les supprimer.

Pour copier les énumérations voici comment nous procédons :

```
create OUT : MM refining IN : MM;

rule CopieModel {
	from
		m1 : MM!Model,enum : MM!Enumeration
	to
		m2 : MM!Model	(
			name <- 'ModeleSortie',
			packageImport <- m1.packageImport,
			packagedElement <- m1.packagedElement.including(enumAdd)
		),
		enumAdd : MM!Enumeration (
			name <- enum.name,
			ownedLiteral <- enum.ownedLiteral
		)
}
```

Figure 42 - Règle de copie des éléments

Comme nous pouvons le noter nous avons ici le mot clé "refining" qui joue le même rôle que présenté en introduction à ATL. Dans notre règle nous créons un modèle de sortie dans lequel on ajoute les énumérations. Ce qui nous donne ce résultat-ci :

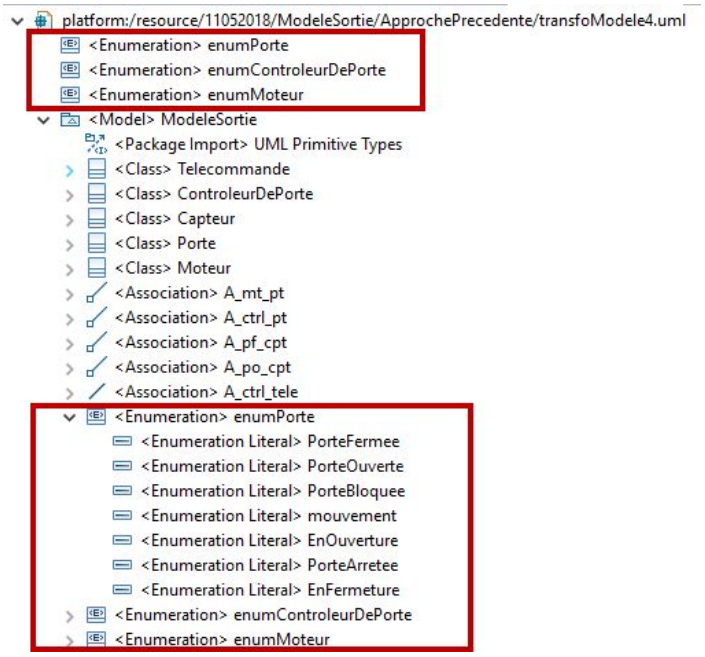


Figure 43 - Résultat de la copie

Comme présent dans les encadrés nous voyons bien que les énumérations complètes ont étés déplacées au bon endroit mais que leur squelette vide est toujours présent dans la définition du modèle. Pour le supprimer nous allons à nouveau utiliser le mode refining et sélectionner les énumérations vides de sorte à les supprimer :

```
create OUT : MM refining IN : MM;

rule clean {
	from
		enum : MM!Enumeration (enum.ownedLiteral->isEmpty())
	to
		drop
}
```

Figure 44 - Suppression des énumérations mal placées

Ici nous sélectionnons les énumération qui n'ont pas d'ownedOperation, autrement dit, les énumérations sans littéraux. Suite à nos transformations et nos suppressions nous obtenons un modèle qui contient nos énumérations dans le bon modèle :

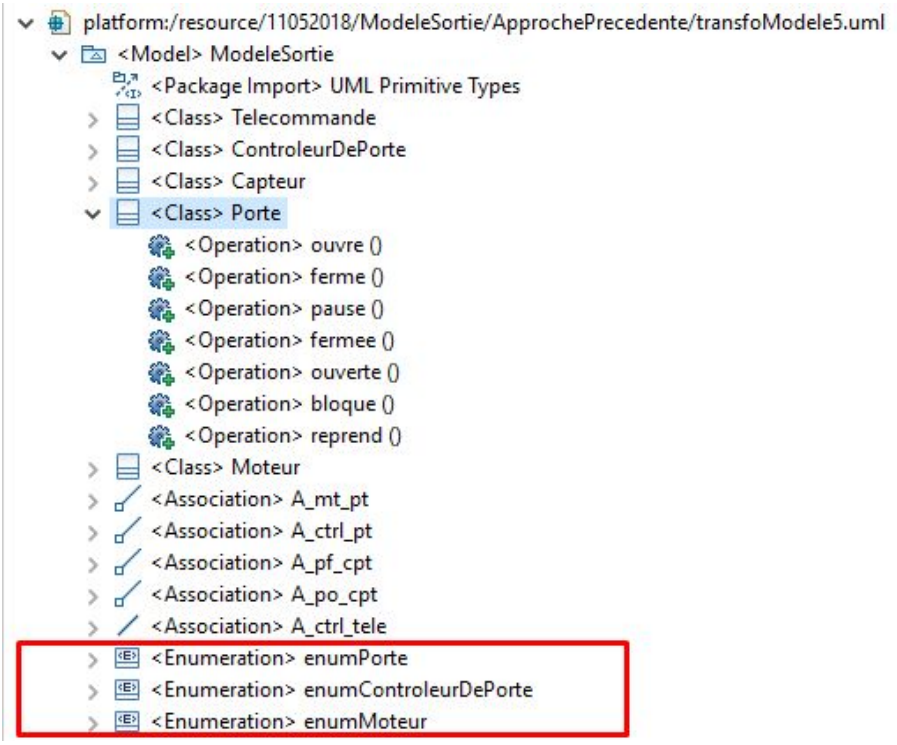


Figure 45 - Représentation UML de notre modèle

Cependant, comme nous pouvons le voir, les énumérations ne sont pas associées aux bonnes classes. Par exemple l'énumération de la porte doit être associé à la porte. C'est la dernière transformation :

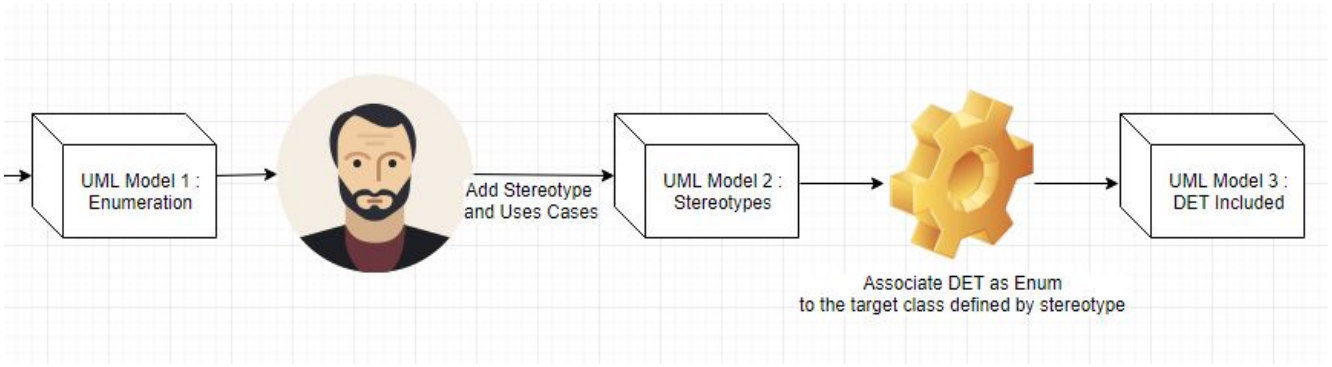


Figure 46 - Ajout Stéréotype et Use case

Cette transformation nécessite une intervention extérieure car ATL ne sait pas à quelle classe il doit associer l'énumération adéquate. Ainsi nous avons besoin de définir nous-même cette relation. Pour cela nous allons rajouter des Stéréotypes dans les classes et des Use Case dans les énumérations :

Figure 47 - Extrait UML représentant les Stereotypes et les Use Case

Désormais, ATL sait à quelle classe on doit associer l'énumération. Nous appliquons donc notre dernière transformation et arrivons à un modèle complet avec les énumérations sur les bonnes classes :

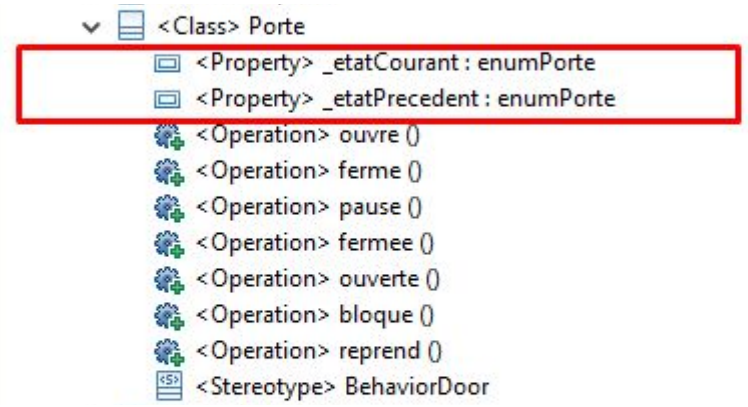


Figure 48 - Extrait UML des énumérations

Intégrer des paramètres externes

à compléter

Conclusion

Cette introduction à la recherche nous a amené sur un domaine que nous ne connaissions pas du tout avec une problématique très intéressante et qui ouvrirait beaucoup de possibilités dans le domaine de la génération automatique de logiciels.

Partant de la spécification UML nous avons implémenté un robot Lejos et fournit une maquette fonctionnelle répondant à la spécification technique qui nous a été donné. A partir de ce simple diagramme de classe et en y ajoutant des diagrammes états-transitions nous avons réussi à automatiser un processus d'intégration de ceux-ci. Deux applications ont été développées, disponible sur pc et sur mobile.

Nous avons eu de nombreux problèmes techniques comme le besoin d'appréhender la librairie Lejos. Nous n'avions aucune connaissances sur l'ingénierie des modèles que cela soit sur les définitions même ou bien sur les transformations. Nous avons dû tout apprendre en partant de zéro en un temps restreint. ATL a été compliqué à appréhender et nous avons été très bien encadré et dirigé vers les bonnes personnes pour apprendre de la meilleure des manières.

Nos travaux nous ont aussi permis d'apprendre à travailler en groupe sur un projet qui est à notre échelle, un projet à long terme. Nous avons travaillé à 3 pendant près de 6 mois sur le même projet, tout le temps et cela la première fois. Cela a donc renforcé notre travail d'équipe et nous avons appris à gérer les tensions sur un projet important.

Nous avons appris à travailler de manière agile, chose qui nous n'avions jamais faite auparavant. En passant par une organisation qui à la base était bien définie en sprints, nous avons finalement dû apprendre à nous adapter et à évoluer au cours du temps, du fait de nos besoins par rapport au projet mais aussi à cause des obstacles rencontrés.

Ce projet exploratoire a été très complet et touche bien des horizons. Passant par de la robotique et du java à l'ingénierie des modèles et la rétro-ingénierie de leur conception, en finissant par du développement mobile. Bien souvent nous nous sommes senti dépassés par la difficulté technique du projet et par un manque de connaissances dans certains domaines. Mais cela a aussi été très formateur puisque nous avons élargi très rapidement notre champ de vision et nos connaissances. Nous avons exploré de nombreux domaines et travaillé avec des personnes qui auront contribué à notre apprentissage.

Le résultat n'est malheureusement pas celui escompté. Bien que nous ayons avancé sur la problématique, nous aurions aimé aller plus loin sur les transformations de modèles pour peut-être avoir une génération automatique de code plus complète. Certains point nous ont beaucoup ralentis et la tâche s'est avérée bien plus complexe que ce que nous envisagions il y a quelques mois.

Les possibilités d'évolution du projet sont nombreuses : ajouter des paramètres externes, étudier une transformation plus complète de code intégrant automatiquement du Lejos, créer des transitions automatiques qui suivent les diagrammes états-transitions, etc.

Cependant, au delà du résultat final, nous aurons au cours de ce projet exploratoire énormément appris sur beaucoup de domaines tel que l'ingénierie des modèles ou encore sur les processus de gestion de projet. Nous aurons découvert des nouveaux langages, discuté avec de nouvelles personnes et travaillé en collaboration avec des chercheurs.

Références

Github de notre projet - https://github.com/demeph/TER-2017-2018

ATL Documentation - http://www.eclipse.org/atl/

Lejos EV3 Mindstorms - http://www.lejos.org/ev3.php

Mapping UML Statecharts to Java Code, Iftikhar Azim Niaz and Jiro Tanaka, Institute of Information Sciences and Electronics University of Tsukuba

Modeling Java Threads in UML, 2-97, ,Martin Schader, Axel Korthaus, Lehrstuhl für Wirtschaftsinformatik III, Universität Mannheim, Germany

Exploration of UML State Machine implementations in Java, Morten Olav Hansen, Master thesis, University Of Oslo, Department of Informatics, February 15, 2011

Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets : application au formalisme DEVS, Thèse de Stéphane Garredu, Université de Corse-Pascal Paoli

Modeling and Analysis of Exception Handling by Using UML Statecharts Gergely Pintér and István Majzik Department of Measurement and Information Systems Budapest University of Technology and Economics, Hungary

Glossaire

ATL - ATLAS Transformation Language

DET - Diagramme Etat Transition

LS2N - Laboratoire des Sciences du Numérique de Nantes

OMG - L'Object Management Group (OMG) est un consortium américain à but non lucratif créé en 1989 dont l’objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes.

QVT - Standard informatique défini par l'OMG. Il s'agit d'un langage standardisé pour exprimer des transformations de modèle

TER - Travail En Recherche

UML - Unified Modeling Language

Lexique

ATL - ATLAS Transformation Language est un langage de transformation de modèles plus ou moins inspiré par le standard QVT de l'Object Management Group. Il est disponible en tant que plugin dans le projet Eclipse.

Diagramme états-transitions - Un diagramme états-transitions est un schéma utilisé en génie logiciel pour représenter des automates déterministes. Il fait partie du modèle UML et s'inspire principalement du formalisme des statecharts et rappelle les grafcets des automates. S'ils ne permettent pas de comprendre globalement le fonctionnement du système, ils sont directement transposables en algorithme. Tous les automates d'un système s'exécutent parallèlement et peuvent donc changer d'état de façon indépendante.

Diagramme de classe - Le diagramme de classes est un schéma utilisé en génie logiciel pour présenter les classes et les interfaces des systèmes ainsi que les différentes relations entre celles-ci. Ce diagramme fait partie de la partie statique d'UML car il fait abstraction des aspects temporels et dynamiques.

Eclipse - Eclipse est un projet, décliné et organisé en un ensemble de sous-projets de développements logiciels, de la fondation Eclipse visant à développer un environnement de production de logiciels libre qui soit extensible, universel et polyvalent, en s'appuyant principalement sur Java.

Java - Java est un langage de programmation orienté objet.

Lejos - Lejos est une librairie disponible sous la forme d’un plugin sous Eclipse permettant de développer un programme fonctionnant sur un robot LEGO (firmware).

UML - UML est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système.

Annexe

Annexe 1 - Modèle de base

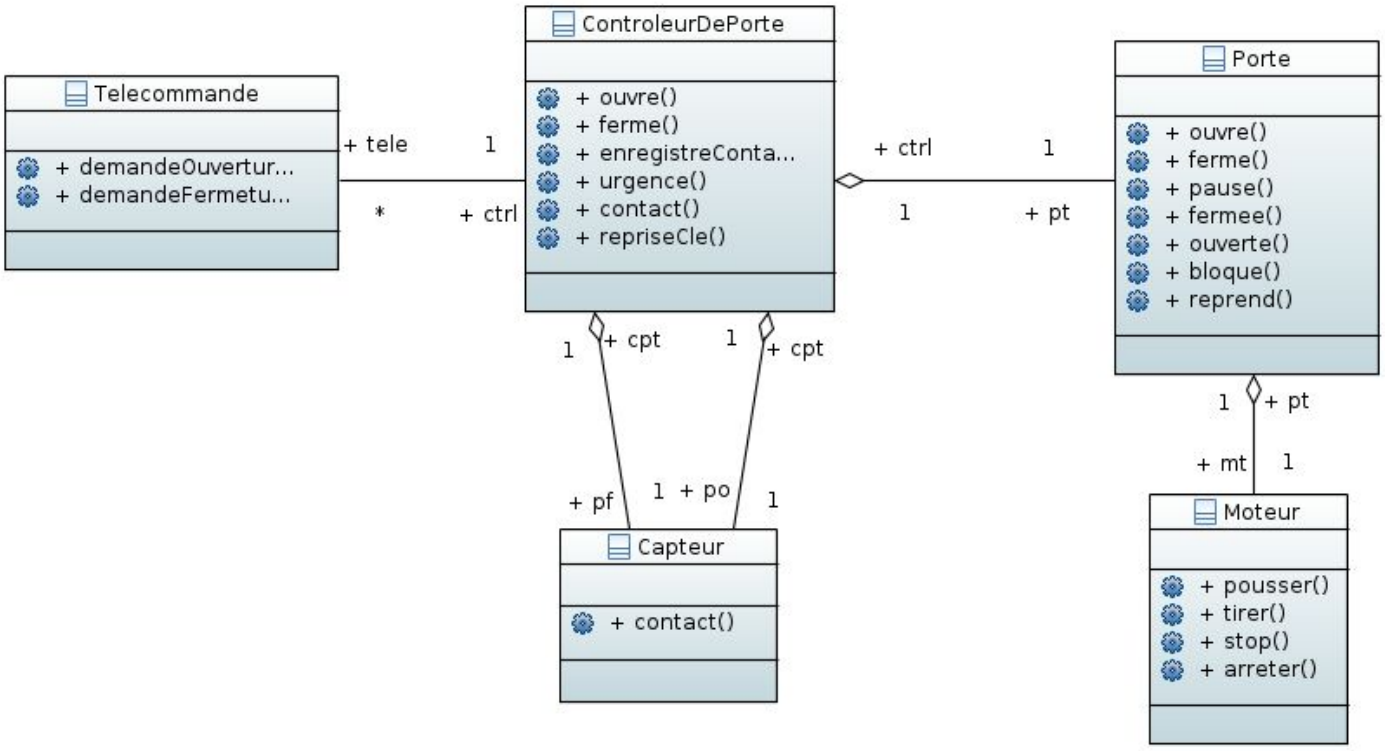
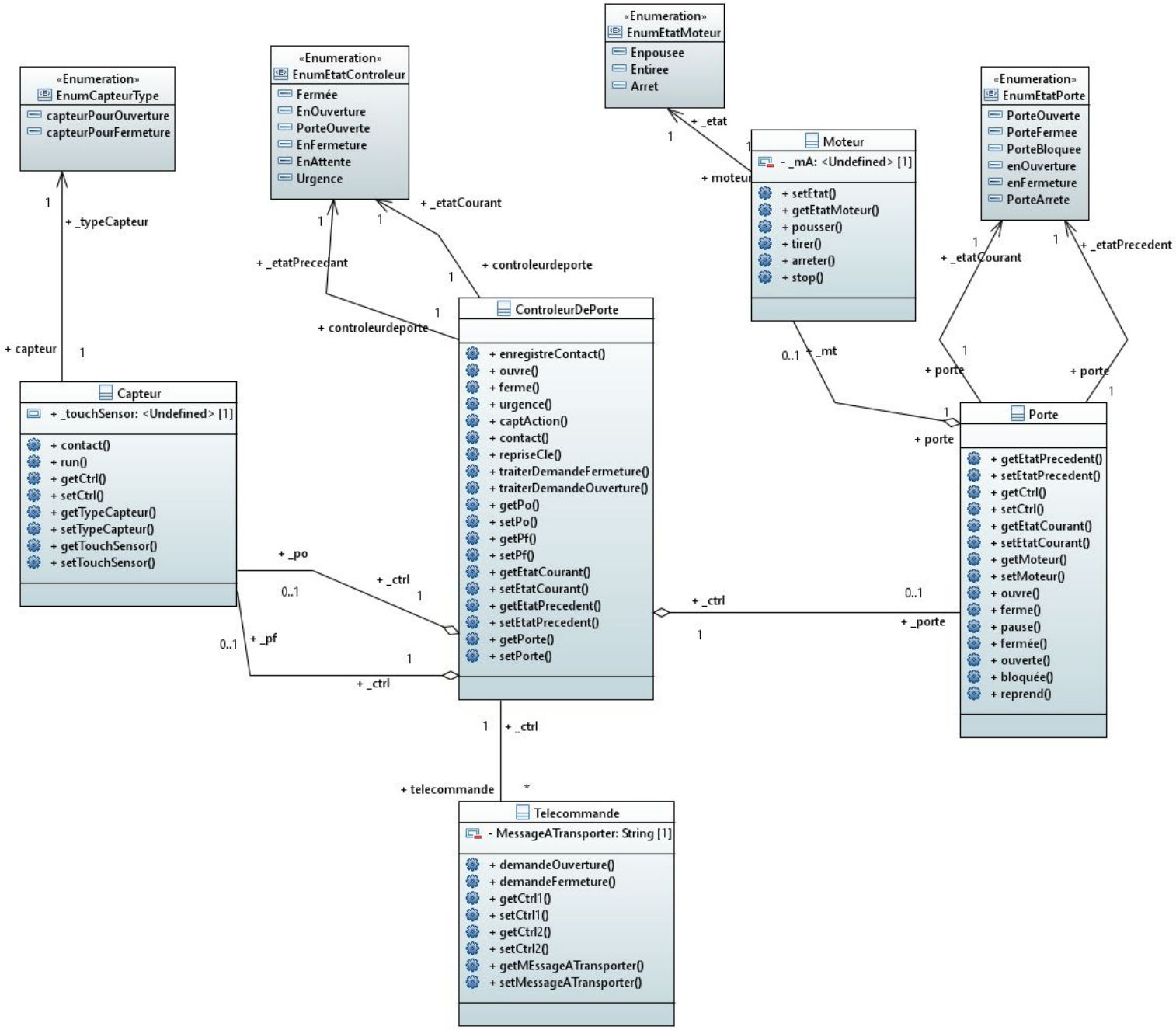
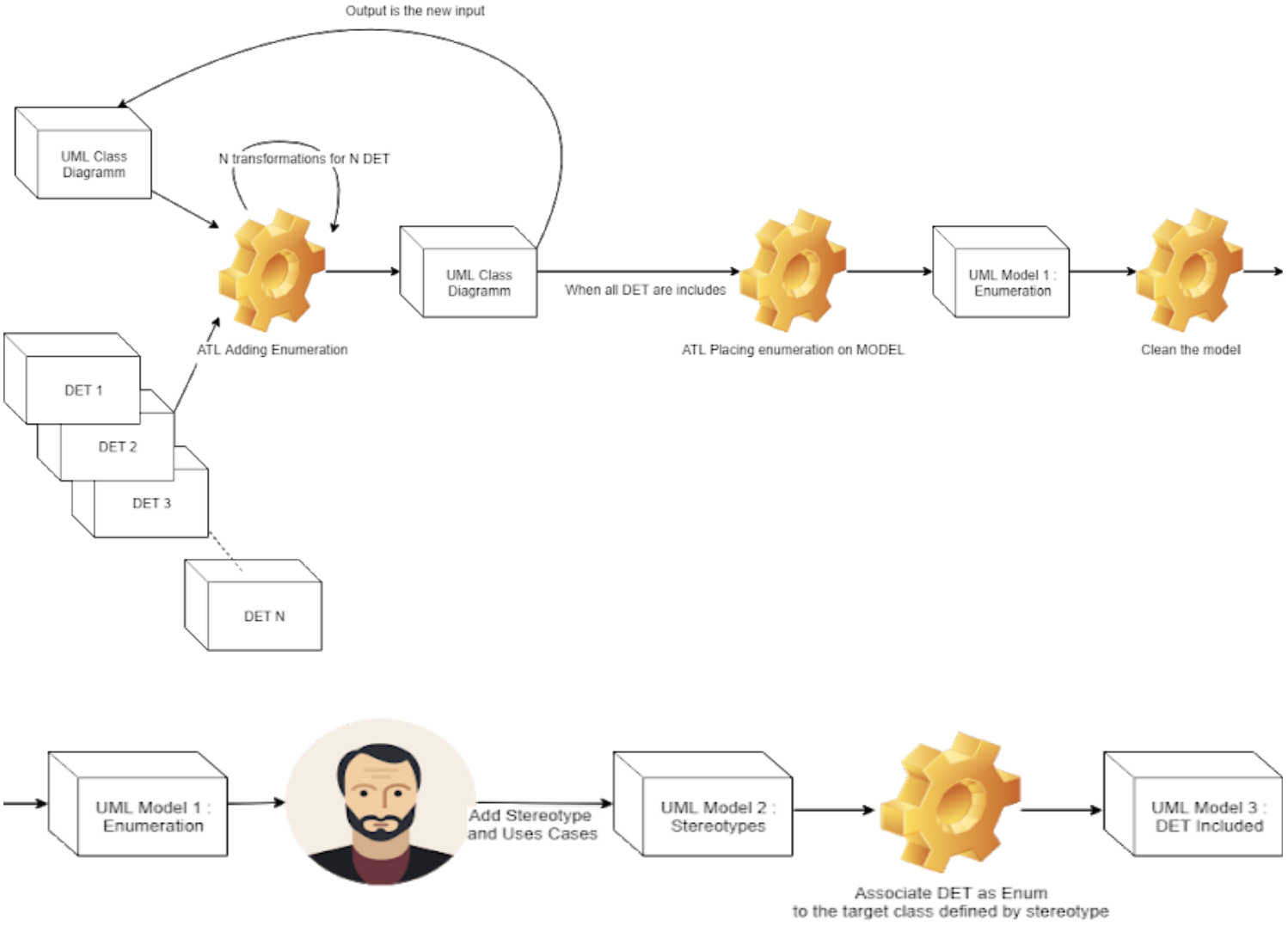


Figure 1 - ...

Annexe 2 - Modèle complet



Annexe 3 - Processus de transformation des modèles

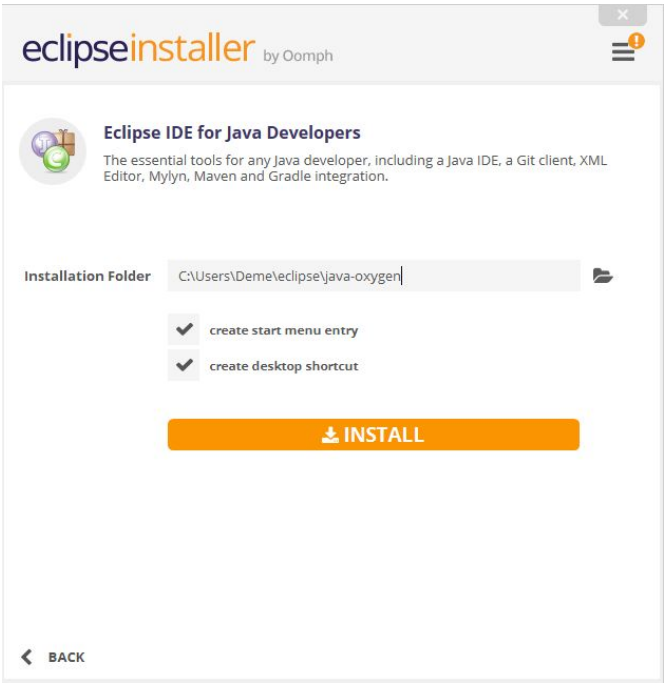


Annexe 4 - Configurer Lejos sous Windows

Dans cette partie d'annexe, nous allons faire le tutoriel de comment configurer Lejos sur le système d'exploitation WIndows, plus précisément Windows 10. Pour commencer la configuration nous avons besoin à installer la version *Java* 8 32 bits pour utiliser le plugin Lejos. Pour cela sur notre page de github dans le dossier de configuration nous avons mis en disposition le fichier d'installation de cette version de *Java*. Une fois java 8 installer, nous installons la version 32 bit de l'environnement de développement *Eclipse*, pour cela nous utilisons fichier d'installation *Eclipse*. Voici les étapes à suivre :

1) Une fois fichier d'installation lancer, on choisit "*Eclipse IDE for Java Developpers*"

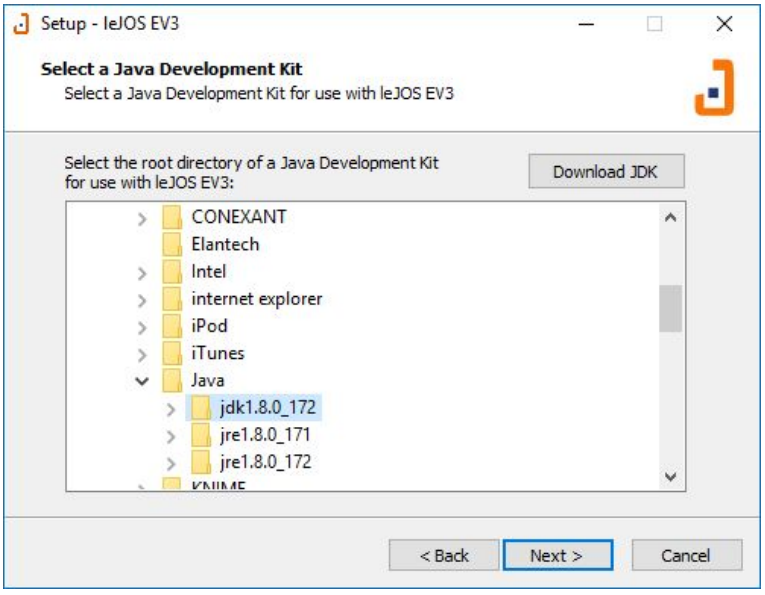
2) Ensuite on choisie le dossier dans laquelle on veut installer eclipse et on appuie sur le bouton Install



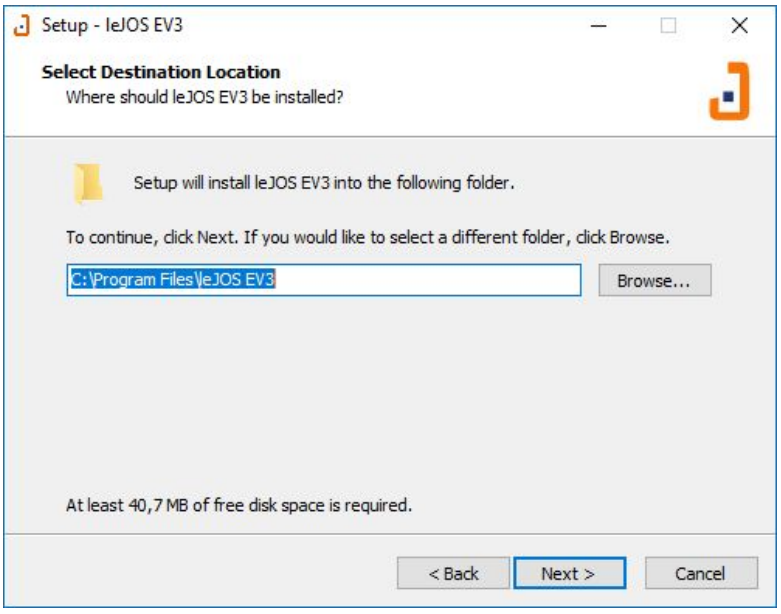
Installation de Lejos

Une fois eclipse installer, nous passons à l'étape prochaine qui consiste à installer les librairies de lejos. Pour cela on va dans notre répertoire *Github* dans le dossier *Configuration* se trouve le fichier d'installation de lejos : "leJOS_EV3_0.9.1-beta.exe". Une fois téléchargé et lancés nous avons quelque étapes à passer pour l'installer :

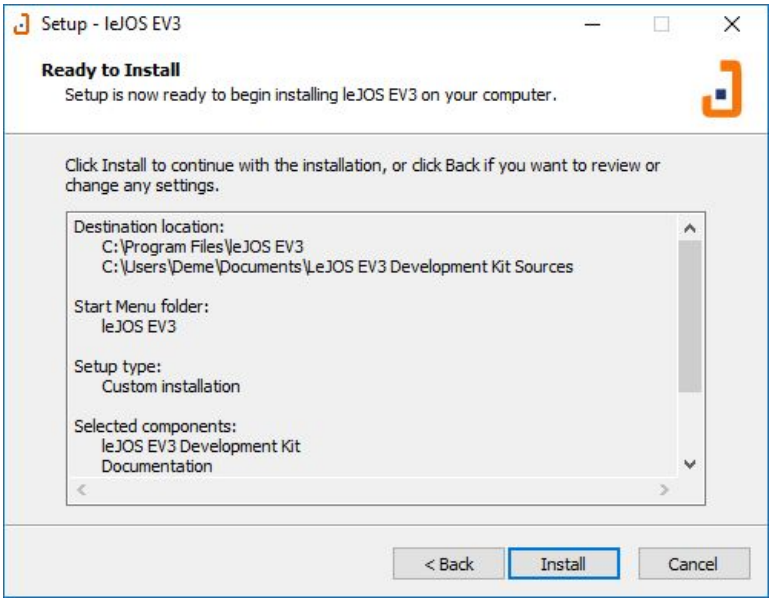
1) Tout d'abord on doit choisir la version de JDK de JAVA comme ceci :



2) Ensuite on choisit le dossier ou on met la librairie de Lejos EV3

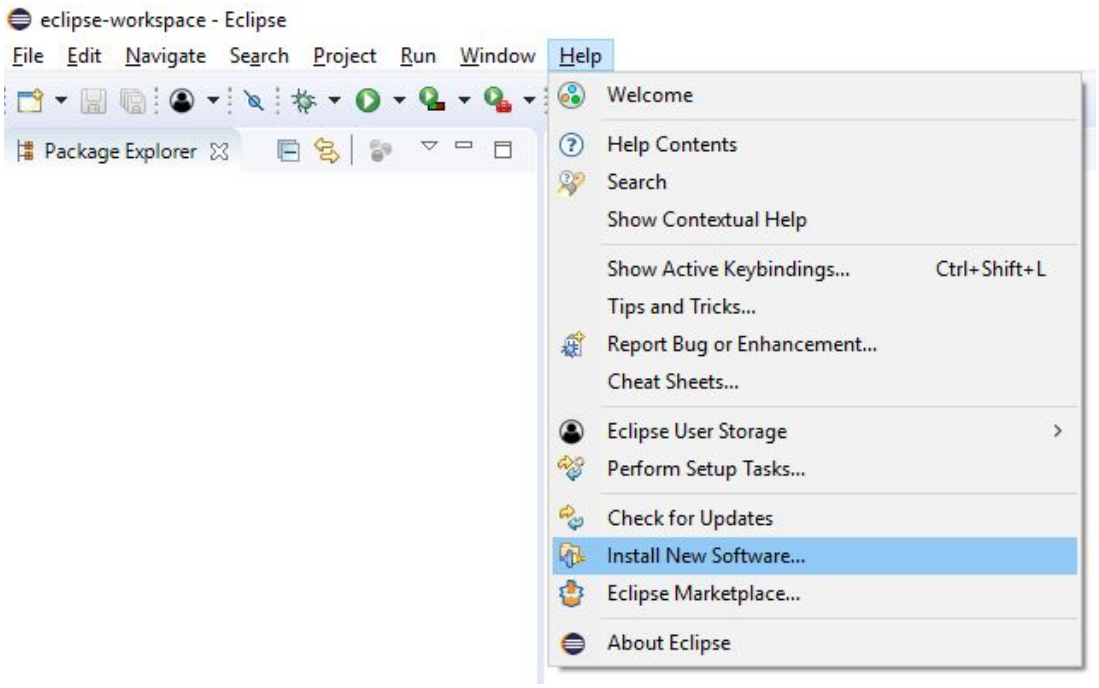


3) Ensuite après plusieurs next, on arrive a la page d'installation de Lejos EV3, on appuie sur "Install" :



Installation de plugin *Lejos* dans *Eclipse*

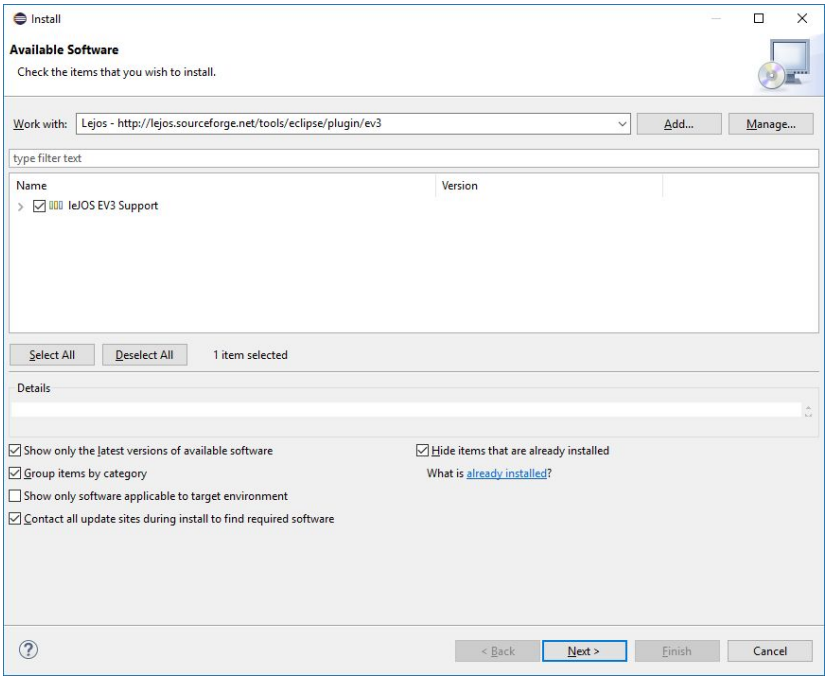
Pour installer le plugin eclipse dans Eclipse premièrement il faut aller a l'onglet Help(ou Aide en fr) et sélectionner *Install New Software*



Ensuite dans la fenêtre ouverte nous copions l'adresse suivante dans le champs *Work With* et on appuie sur le bouton *Add*.

Le lien a copier : http://lejos.sourceforge.net/tools/eclipse/plugin/ev3

Puis, après quelques instant, ca apparait *Lejos EV3 Support*, on coche ce champ et on appui sur *Next* plusieur fois et au final nous acceptons tout les conditions d'installation.



Une fois l'ajout de plugin terminée, il faudra redémarrer Eclipse pour que l'installation soit valide.