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

Introduction

L'objectif de ce projet est d'automatiser la production logicielle : cela consiste dans le cas présent 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.

Cette première étape nous va nous permettre d'étudier la spécification et d'effectuer des modifications ou des ajouts car la spécification 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 réaliser la spécification UML du système réalisé en Java Lejos.

Ainsi, nous aurons la spécification UML initiale et la spécification UML finale. C'est là que l'on va entrer dans le vif du sujet : 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. Pour ce faire nous allons utiliser le langage de transformation de modèle ATL disponible comme plugins sur Eclipse.

Première partie : développement en Java

Premier pas avec Lejos

Comme souligné dans l'introduction, 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.

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.

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.

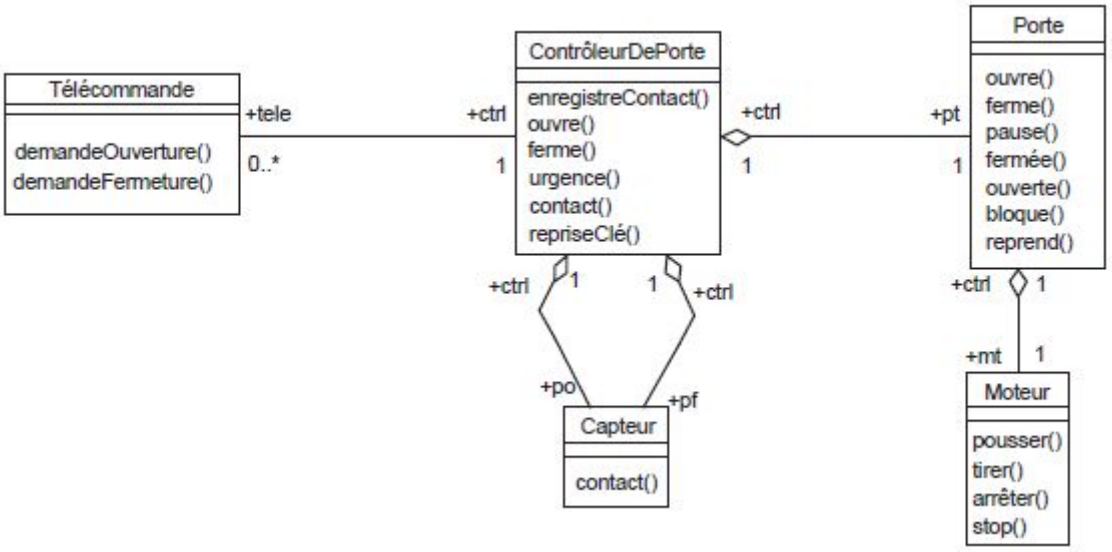


Diagramme de classe

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.

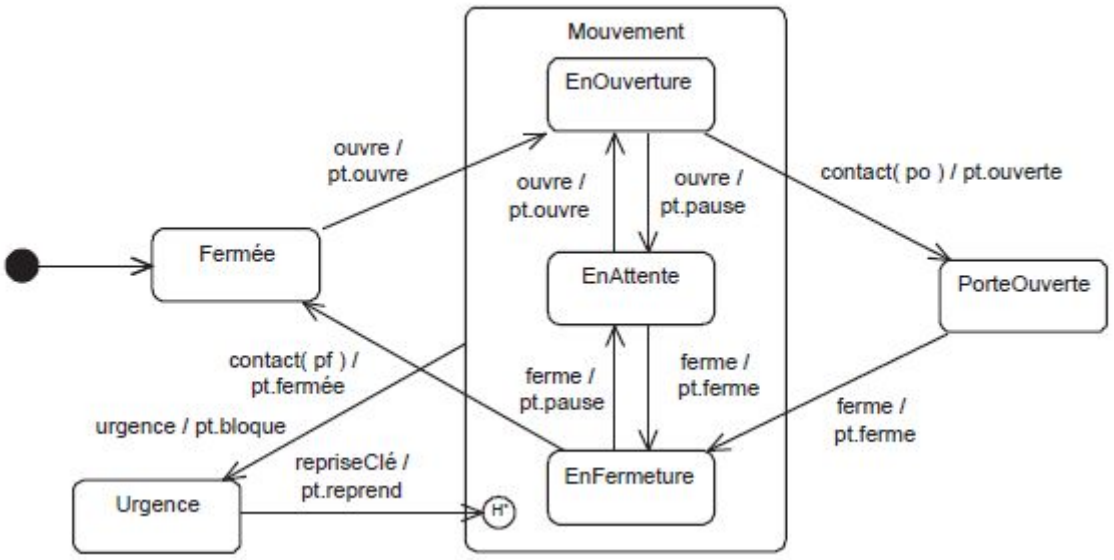


Diagramme états-transitions du contrôleur

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 à utiliser un pattern State, mais après réflexion et discussions nous estimons que cela ne simplifierait pas le code et ne le rendrait pas plus évolutif. 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 créons donc une énumération propre au contrôleur nommée *EnumEtatControleur*, celle-ci contient 6 états : *Fermee*, *EnFermeture, PorteOuverte, EnOuverture*, *EnAttente* et *Urgence*. 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.

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()*.

Les valeurs d'*EtatCourant* et d'*EtatPrecedent* sont changés à l'aide de "setter", comme il se doit en programmation orienté objet. Des "getter" sont également ajouté, nous ne nous attarderons pas dessus.

- 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.

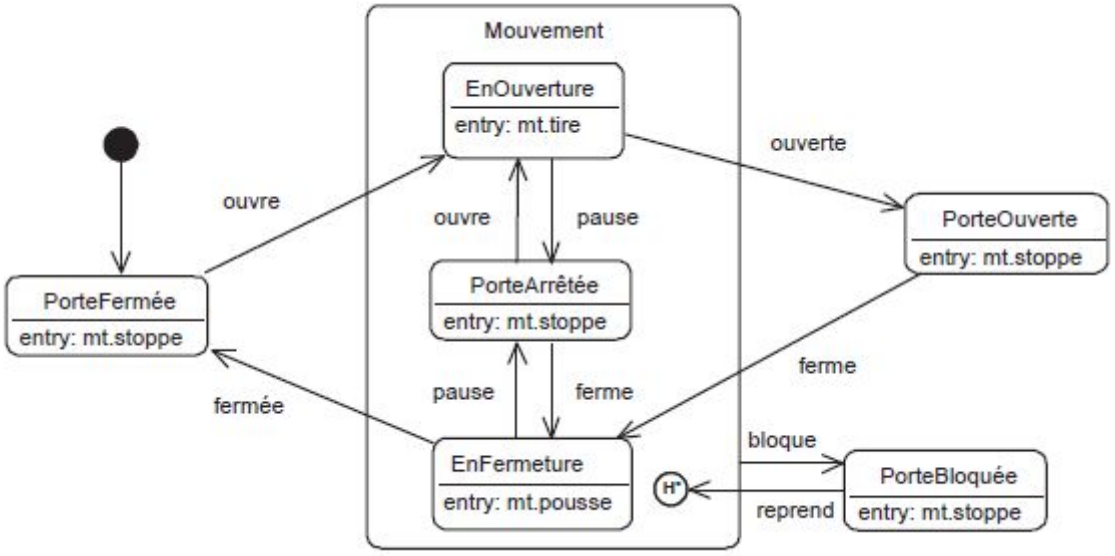


Diagramme états-transitions de la porte

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*. 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.

- 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.

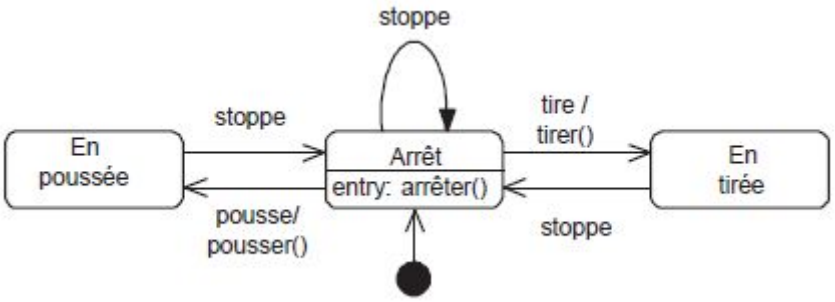


Diagramme états-transitions du moteur

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.

- 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.

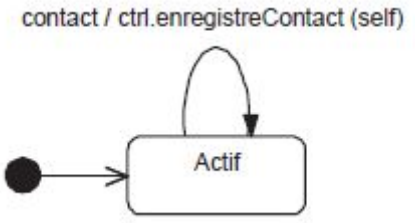


Diagramme états-transitions du capteur

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.

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()*.

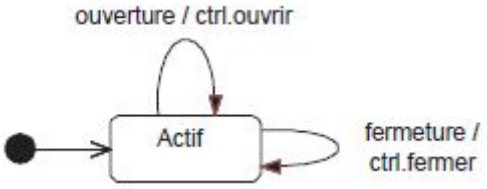


Diagramme états-transitions de la télécommande

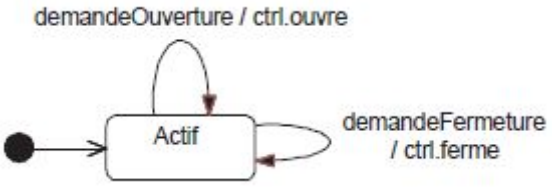


Diagramme états-transitions de la télécommande (amélioré)

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.

En ajoutant à présent du 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 le 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. En effet il n'y a rien de plus à indiquer, ni besoin de paramètre supplémentaire. Seulement de préciser à quel type de capteur le micro-contrôleur LEGO à affaire.

- La classe Moteur

Passons à la classe Moteur. Nous ajoutons là un attribut de type *RegulatedMotor*, pour spécifier le type de moteur utilisé. Ce qui est important puisqu'il y en a plusieurs. Aussi, dans chacune des méthodes propres à la classe *Moteur* comme *pousser()*, *tirer()*, *arreter()* et *stop()*, il faut donner les instructions en Lejos. Par exemple, nous ajoutons dans les méthodes *tirer()* et *pousser()* la vitesse du moteur à l'aide de la méthode *setSpeed()* fournit par Lejos.

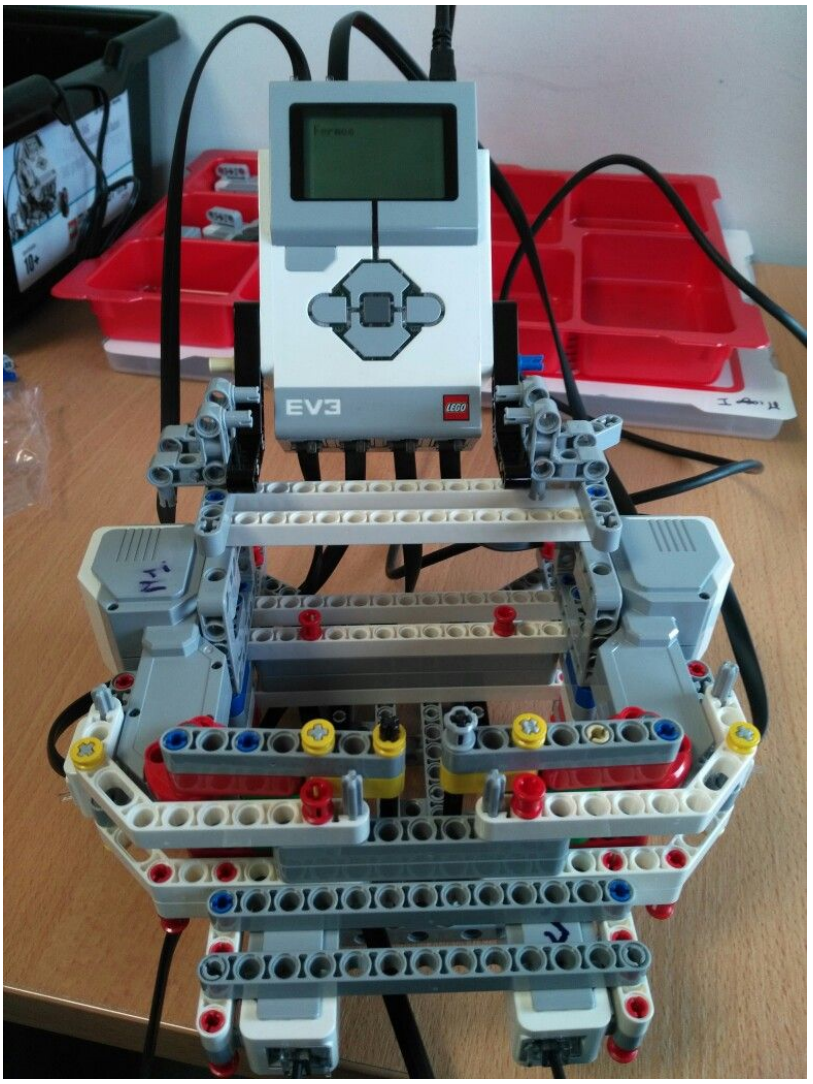
- Le programme principal

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

Dans notre cas, comme vous le constaterez sur la maquette ci-dessous, nous avons deux portes. Cela signifie que nous avons dans le main deux instanciations de *ControleurDePorte*, puisque 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.

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.



La maquette ci-dessus est ainsi constitué de deux portes, nécessitant donc deux capteurs de contact chacunes (touch sensor). Un capteur de contact sur le côté pour tester si la porte est ouverte et un capteur de contact devant pour tester si la porte est fermé. Il y a également deux moteurs pour actionner respectivement les deux portes.

La maquette 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.

Application externes

Application en Java

Pour contrôler la maquette à distance avec un réseau filaire, nous avons réalisé une application en java avec une interface graphique composée de deux boutons : Ouverture de la porte et Fermeture de la porte :

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 android

à compléter

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. Nous avons cependant 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.

Deuxième partie : transformation de modèle

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.

Ce que l'on constate également, c'est que nous avons pris énormément de liberté sur de nombreux 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.

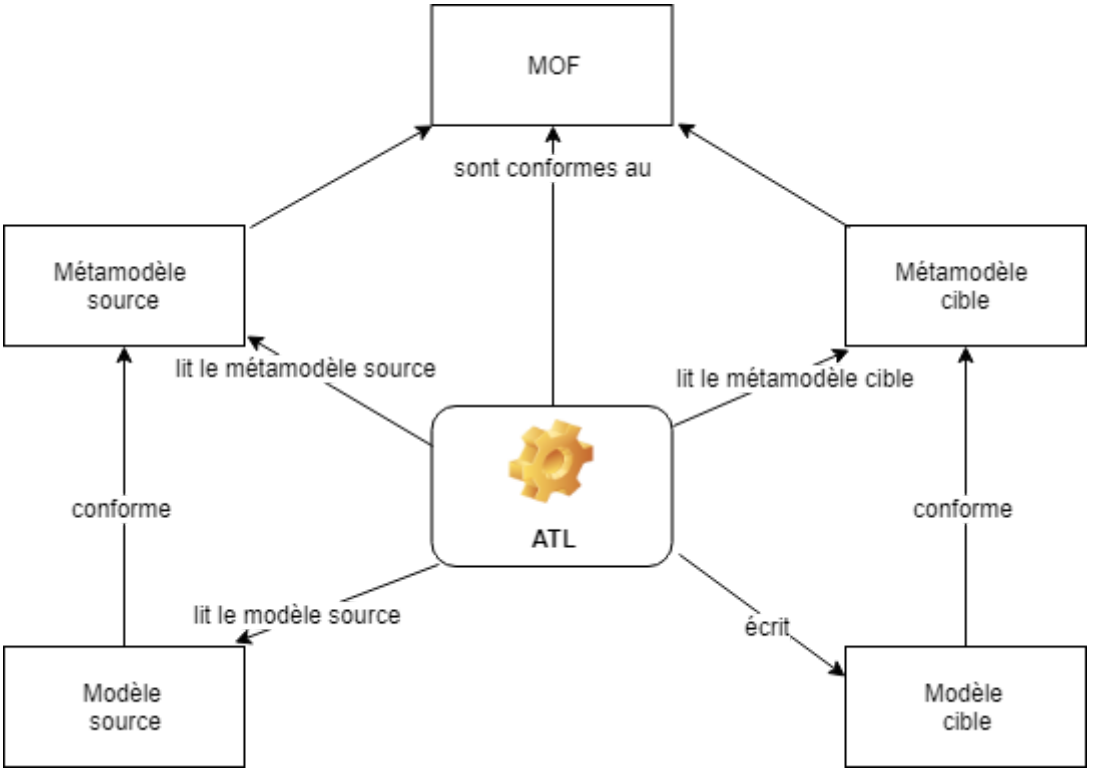
Il serait donc intéressant de prendre notre diagramme de classe initial et de le compléter à l'aide des diagrammes états-transitions. Voir même de le simplifier pour s'approcher du langage Java utilisé. Aussi, ajouté des paramètres Lejos à notre modèle UML aiderait énormément. C'est là qu'entre en jeu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

Découverte d'ATL

ATL est un langage de transformation de modèle développé à Nantes, par l'équipe AtlanMod et disponible sous la forme d'un plugin Eclipse. 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 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 :



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

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
		)
}
```

```
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)
}
```

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 :

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

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

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.

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.

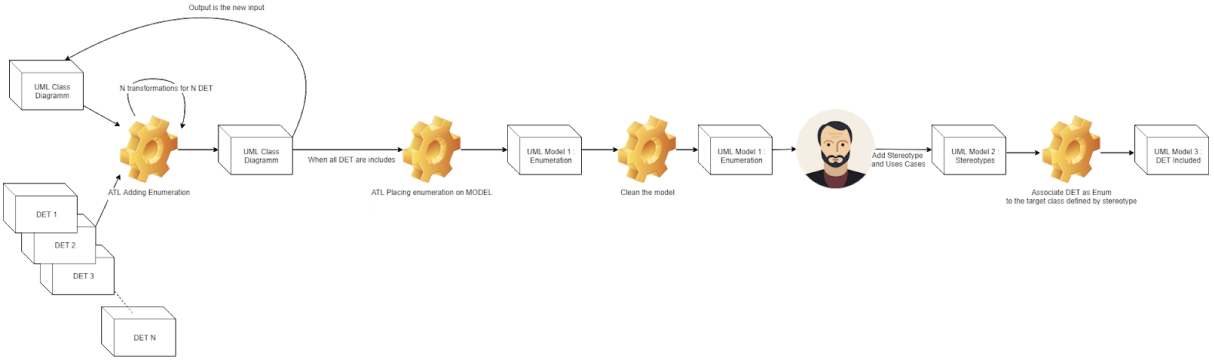
Le premier souci est qu'il ne faut pas s'attacher à la classe. Prenons l'exemple d'un DET qui va définir les états d'un moteur. On l'appelle *StateCharteMoteur* et on veut qu'il opère sur la classe *Moteur* sauf que lors de travaux à l'étranger nos collègues nous donnent un diagramme de classe avec une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

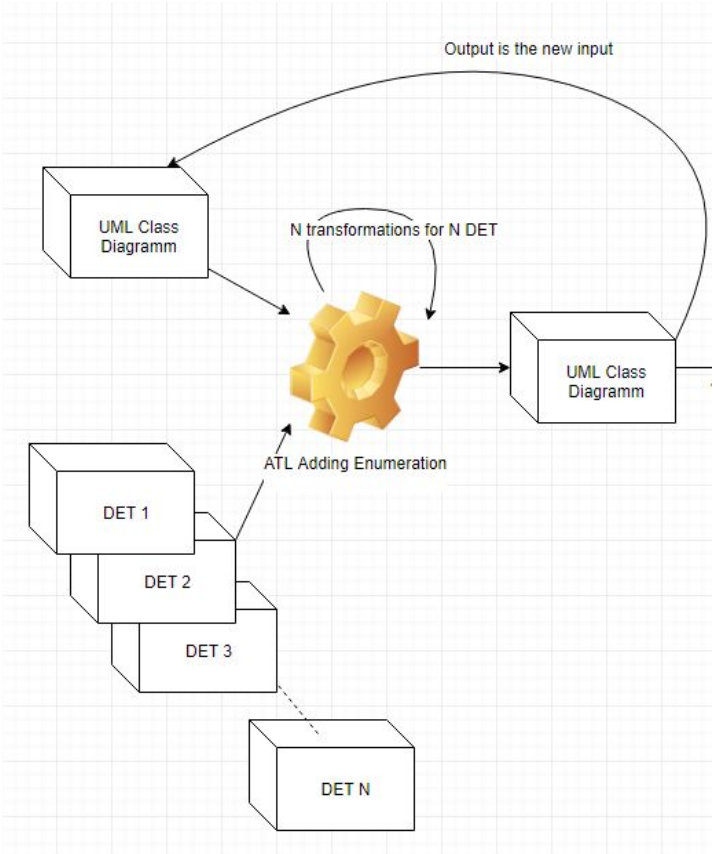
Nous allons reprendre nos transformations qui sont au nombre de 5 afin de comprendre le processus complet d'évolution du modèle mais avant de rentrer dans l'aspect plus technique voici un schéma explicatif.



Processus de transformation

Pour plus de clarté le schéma est disponible en annexe [3].

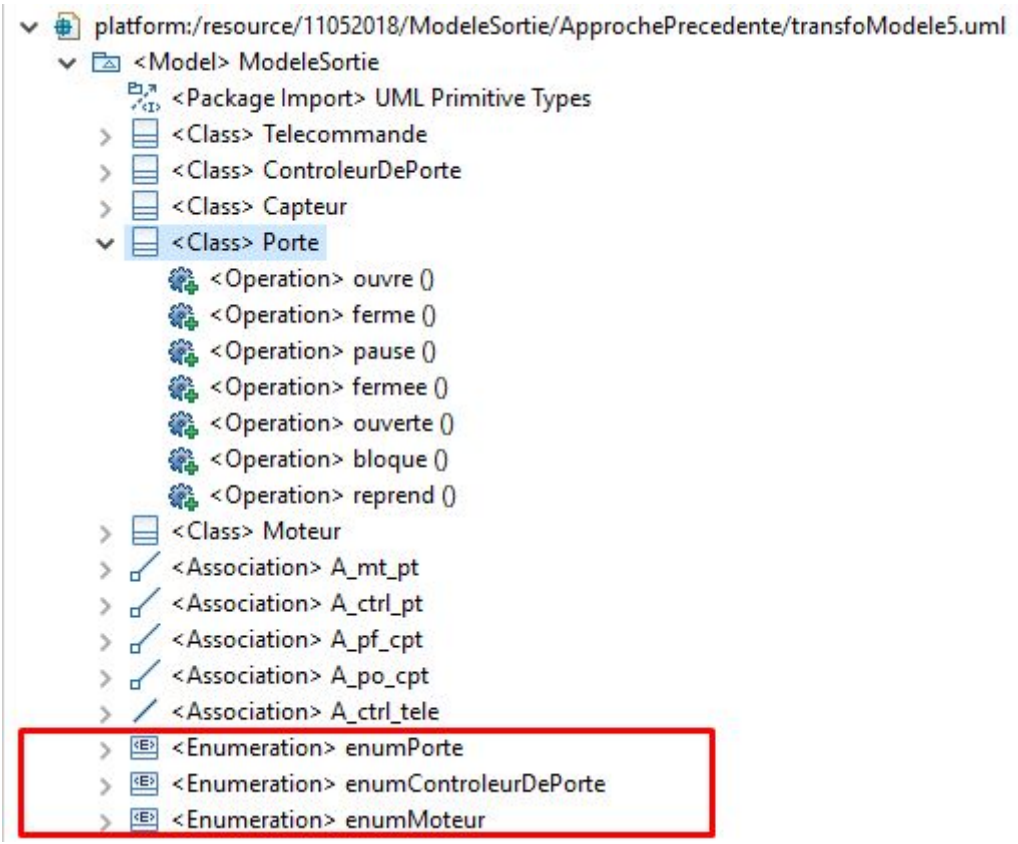
Nos premières transformations (au nombre de deux actuellement) sont celles de transformations du modèle en y intégrant les DET sous forme d'énumération :



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.

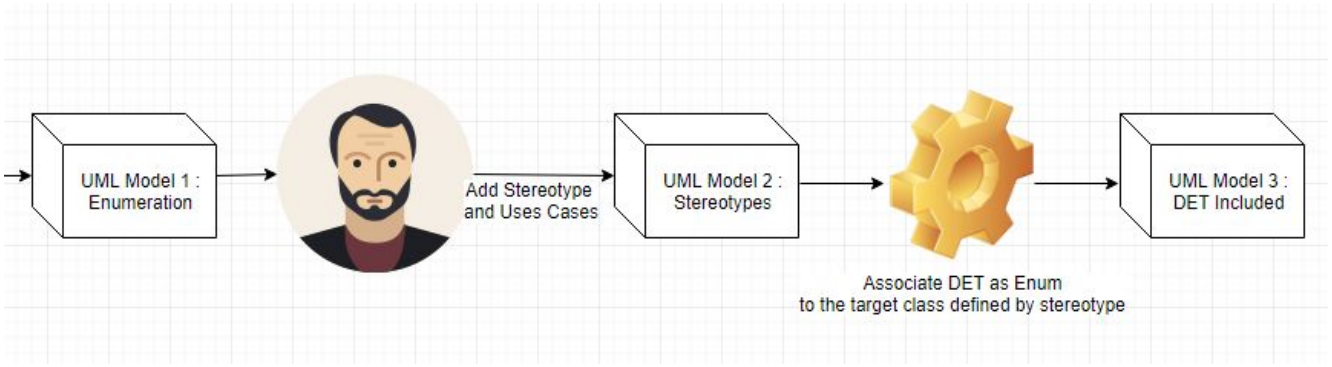
On répète la transformation autant de fois qu'il y a de DET, une fois qu'ils sont tous rajoutés au modèle, on le nettoie car l'ajout ne se fait pas proprement et les énumérations ne sont pas présentes au bon endroit. Ainsi il faut les déplacer au sein du diagramme de classe. Concrètement elles sont ajoutées dans le fichier mais pas dans le modèle qui contient le diagramme de classe.

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. Suite à notre transformation nous obtenons un modèle qui contient nos énumérations dans le bon modèle :

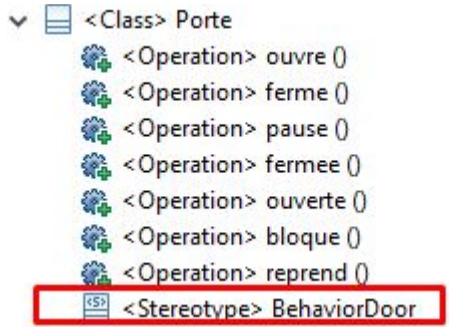


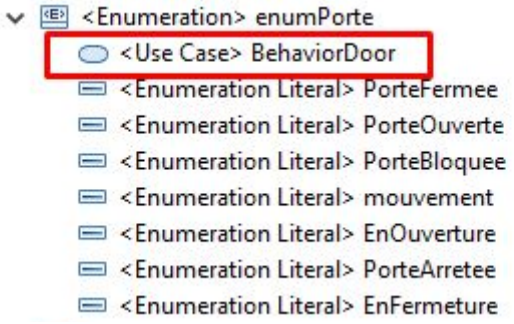
Représentation UML de notre modèle

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 :

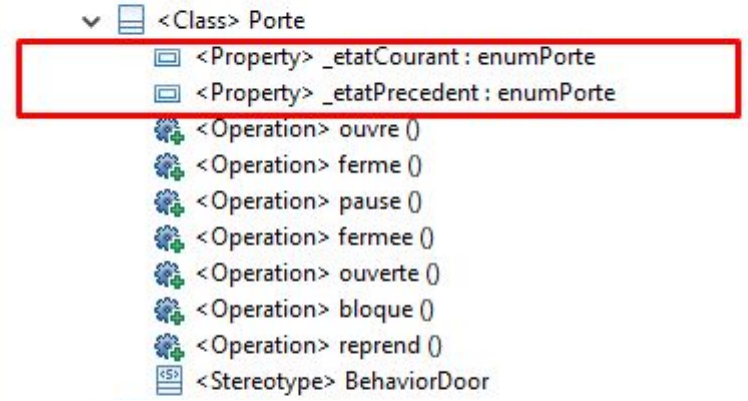


Cette transformation nécessite une intervention extérieure car ATL ne sait pas à quelle classe il doit associer l'énumération adéquate. Ainsi nous avons besoin de définir nous-même cette relation. Pour cela nous allons rajouter des Stéréotypes dans les classes et des Use Case dans les énumérations :





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 :



Intégrer des paramètres externes

à compléter

Conclusion

à compléter

Référence

https://github.com/demeph/TER-2017-2018

http://www.eclipse.org/atl/

http://www.lejos.org/

http://www.lejos.org/ev3.php

Glossaire

ATL - ATLAS Transformation Language

LS2N - Laboratoire des Sciences du Numérique de Nantes

TER - Travail En Recherche

UML - Unified Modeling Language

DET - Diagramme Etat Transition

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.

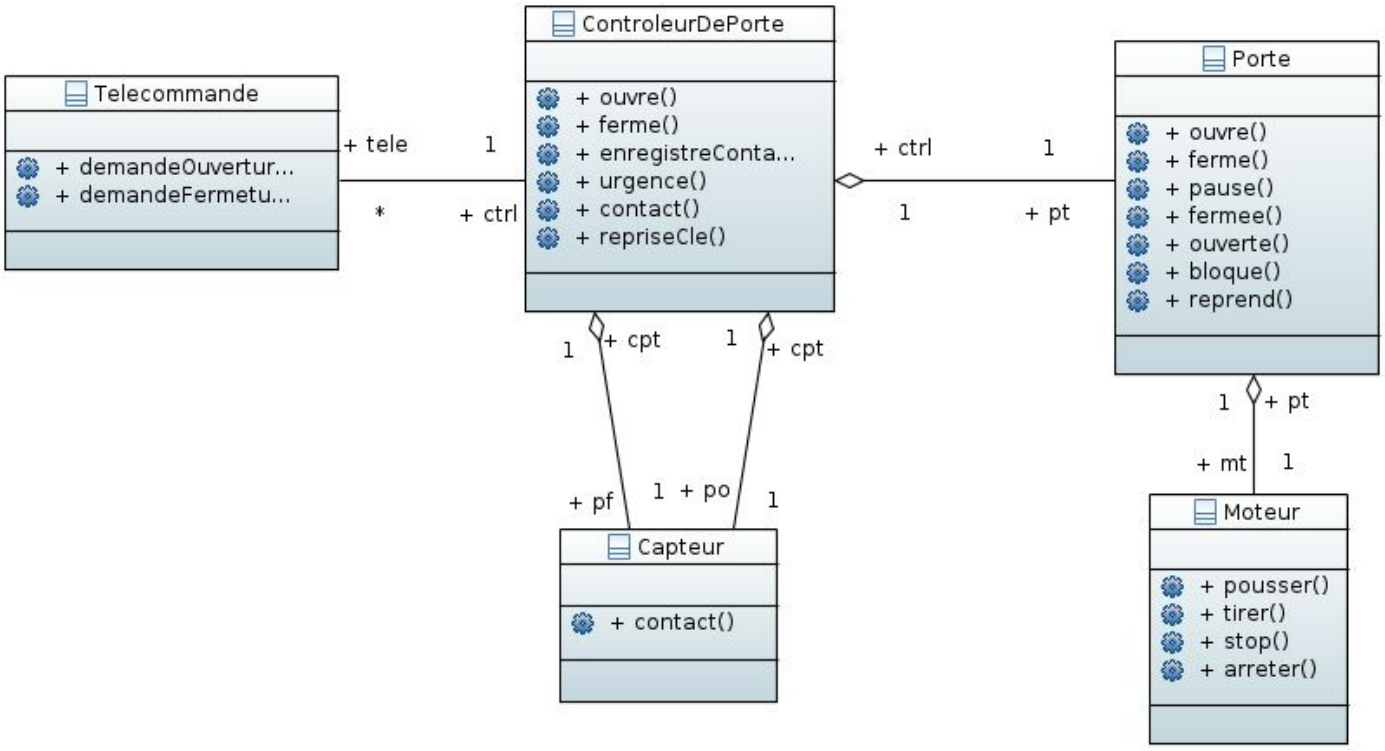
Java - Java est un langage de programmation orienté objet.

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).

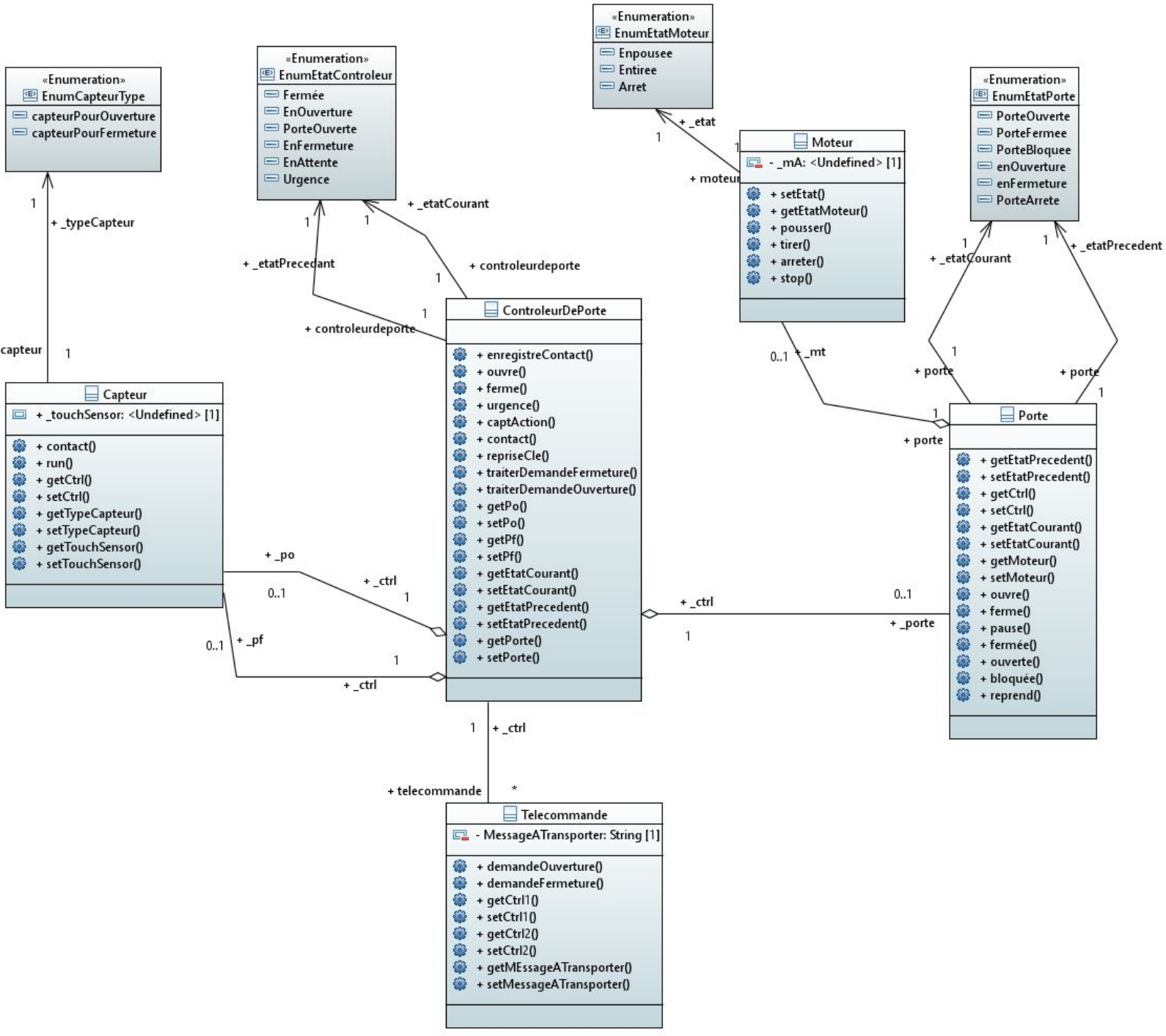
UML - UML est un langage de modélisation graphique à base de pictogrammes conçu pour fournir une méthode normalisée pour visualiser la conception d'un système.

Annexe

[1] Modèle de base



[2] Modèle complet



[3] Processus de transformation des modèles