

Conception Orientée Objet

Modélisation objet d’un simulateur de robot

Rémy KALOUSTIAN

Vincent RAYBAUD

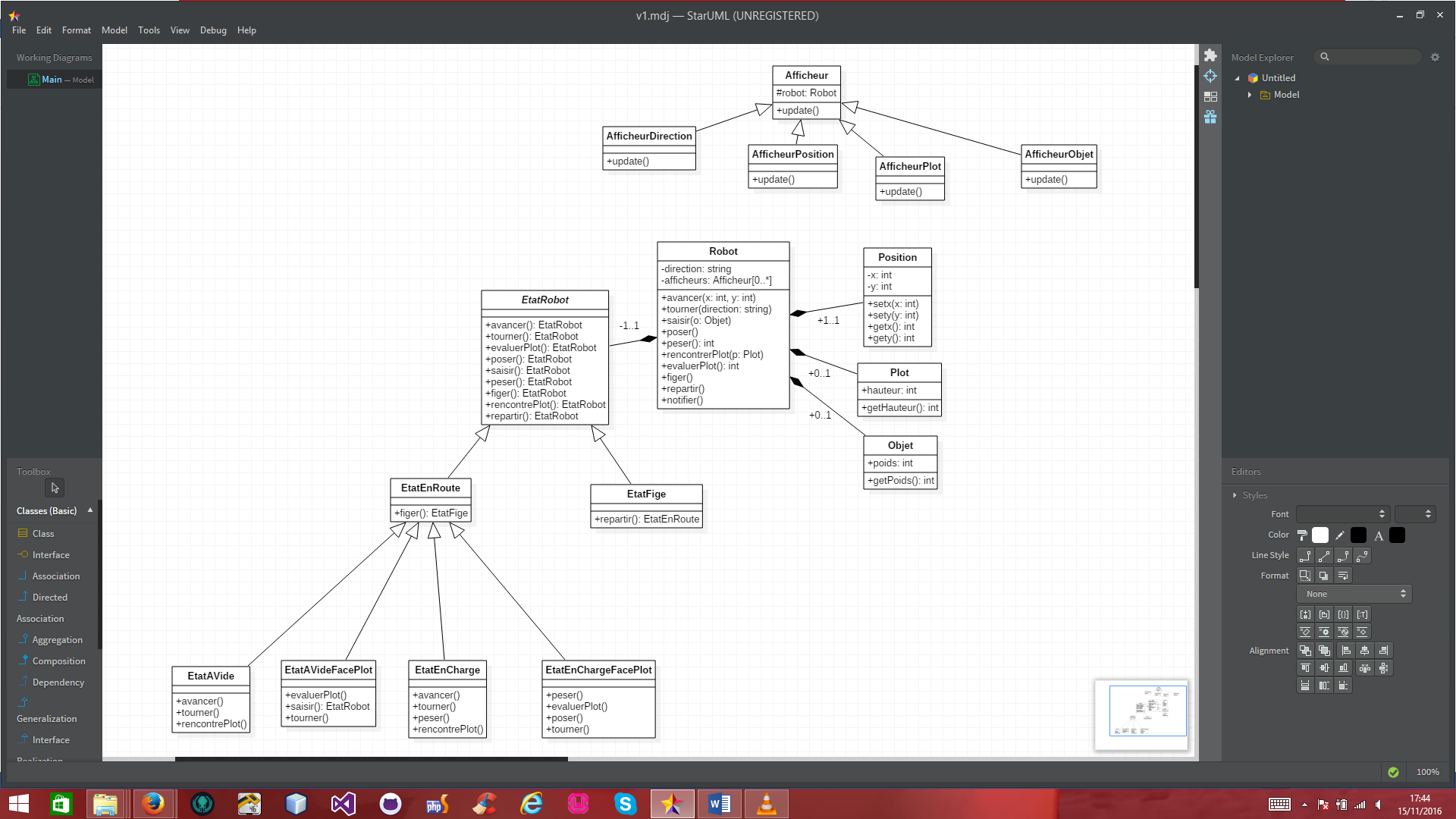
Polytech Nice-Sophia

SI4 – G3 - 2016

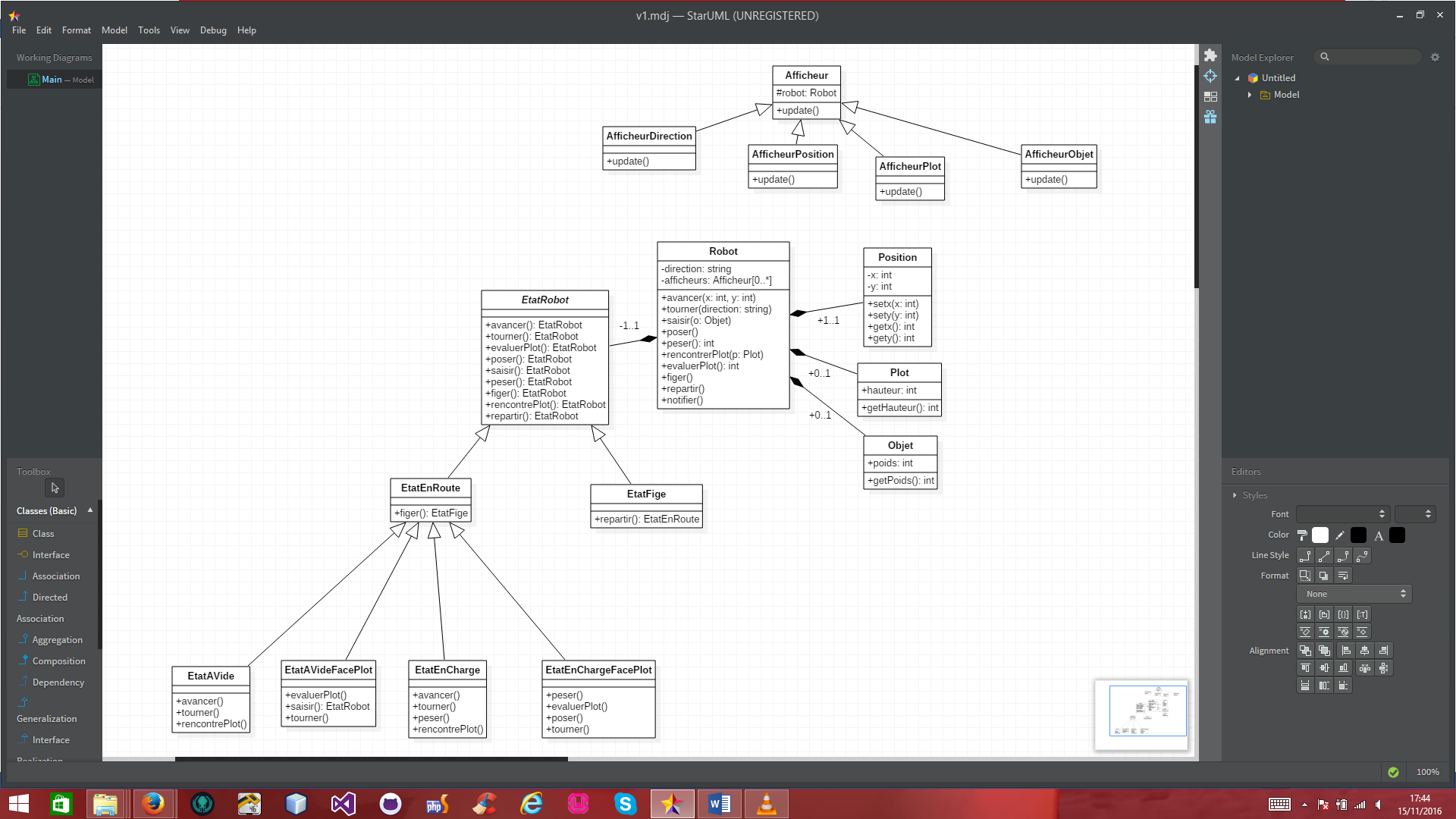
Introduction

Nous disposons d’une classe Robot mise en évidence dans un diagramme de classe basique. Le but de ce projet est d’utiliser cette base, et d’ajouter ou modifier des éléments afin de rajouter des fonctionnalités et améliorer la structure qui nous a été fournie. L’intégration ou la modification de fonctionnalités doit être la plus propre possible afin de minimiser le travail lors de l’extension du projet ou pour l’ajout d’autres fonctionnalités.

# Diagramme de classe général



# L’état du robot est un **État**



Le schéma choisi pour représenter les états du robot est le….

Pourquoi un État ?

Nous avions besin d’un schéma qui pouvant modéliser le fait que le robot puisse changer d’état, et que son état indique quelles actions il peut faire. Ici, le comportement du robot dépend de son état. Ce dernier changera dynamiquement en fonction des actions effectuées par le robot.

Sans ce schéma, nous aurions dû passer par une série de if() pour vérifier l’état du robot. Nous nous serions éloignés du paradigme de la programmation objet dont le but est d’offrir une solution élégante, extensible et modulable.

Le choix du shcéma Etat nous paraissant pertinent car il nous permettait de définir le comportement du Robot, autrement dit, les actions qu’il peut faire, en fonction de son état. Tout cela sans recopier de l’information dans les états eux-mêmes. Le schéma nous permet alors de décider quelles actions sont faisables par le robot, sans avoir à manipuler un grand nombre de données (direction, position,…) et en utilisant la force du polymorphisme.

C’est donc pour sa capacité à dire de manière élégante si un robot peut effetcuer une action que nous avons choisi ce schéma.

Quelles sont les conséquences ?

La première conséquence est le recours obligé au polymorphisme pour les état, afin de n’avoir qu’un seul EtatRobot en attribut de Robot, et de pouvoir ainsi changer d’état à volonté sans jamais ajouter plus d’un seul attribut EtatRobot dans Robot.

De plus, ce schéma nous permet, en plus de bien séparer les états par comportement, de rajouter facilement un nouvel état. Il suffit pour cela de créer ue classe qui hérite d’EtatRobot (et de retourner un EtatRobot dans la méthode déclenchant le changement d’état, voir partie juste après).

Les transitions entre les états sont para illeurs explicites et actionnées par les fonctions des états. Ainsi, on a bien une séparation de responsabilité. Le Robot modifie ses données et enclenche un changement d’état, mais c’est L’EtatRobot qui va rendre réel ce changement d’état (en retournant un EtatRobot).

Un autre conséquence possiblement exploitable est la matérialisation de l’état du Robot sous forme d’objet. Nous pouvons alors imaginer que l’ajout d’attribut ou de fonction relatif searit alors effectué directement dans l’état plutôt que dans le robot. De même, si une autre classe a besoin de l’état du Robot, nous pouvons alors lepasserd irectement en paramètre, sans avoir à transmettre tout le Robot.

Choix particulers et problèmes

Nous avons décidé de faire retourner un EtatRobot pour chaque fonction de la classe EtatRobot. AU départ, seules les méthodes qui permettaientd e changer d’état dans le diagramme d’états-transition de l’énoncé retournaient un EtatRobot. Puis nous avons pensé que l’ajout d’un changement d’état pour des méthodes déjà existantes nécessitait le retour d’un EtatRobot pour toutes les fonctions d’EtatRobot. Ainsi, si il ya un changement d’état, on retourne le nouvel état, sinon, this, pour conserver l’état courant.

Pour matérialiser la possibilité d’effectuer une action ou pas, nous avcons intégré un sytème d’exception. Ce système est couplé à l’utilisation de l’héritage pour les états.

De base toutes les fonctions d’EtatRobot renvoient une UnAuthorizedCallException. Les états qui héritent d’EtatRobot implémentent alors juste les fonctions correspondant aux actions faisables dans cet état. Une fonction non implémentée que l’on appelle sur un état renverra à la version de la classe mère EtatRbot, et lèvera une exception.

Par exemple, l’EtatFige implémente repartir(). L’appel de cette fonction change l’état. EN revanche, EtatFige n’implément pas avancer(). Si on appelle avancer() alors que l’état est EtatFige, le programme exécute avancer() d’EtatRobot, qui renvoie une exception.

Enfin, un problème assez important avec ce schéma est l’ajout d’une classe pour chaque nouvel état. bien que le schéma demande une classe par état, l’augmentation du nombre de classes est pour l’instant moindre comparé au bénéfice qu’il nous apporte, c’est-à-dire, savoir quelles fonctions le robot peut faire en fonction de son état, le tout de manière élégante et extensible.

# L’afficheur est un **Observateur**

