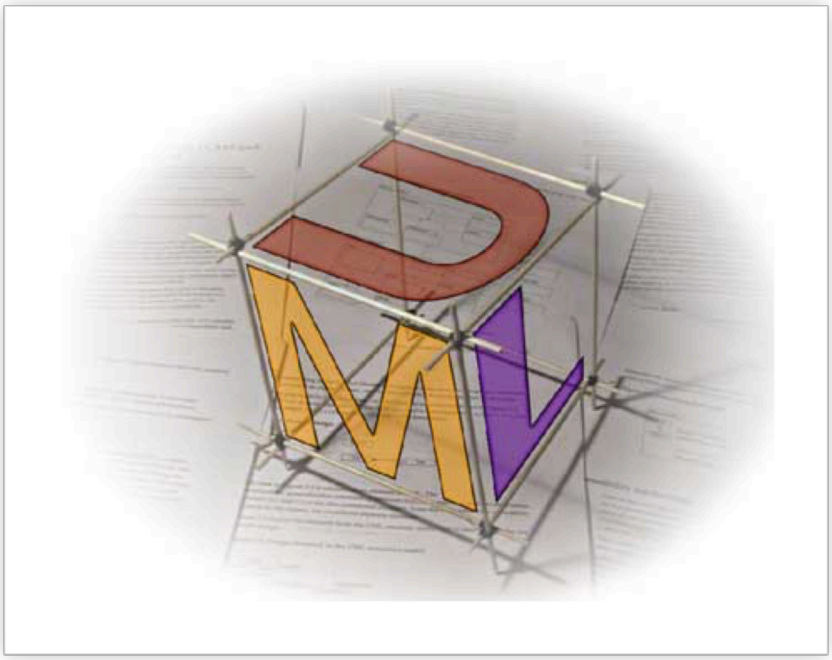


Modélisation objet d'un simulateur de robot

DUPANLOUP Rémy, RAHBARI Guillaume
UML | 25 novembre 2014

Table des matières

Introduction

Nous avons, pour ce projet, réaliser une modélisation d'un simulateur de robot.

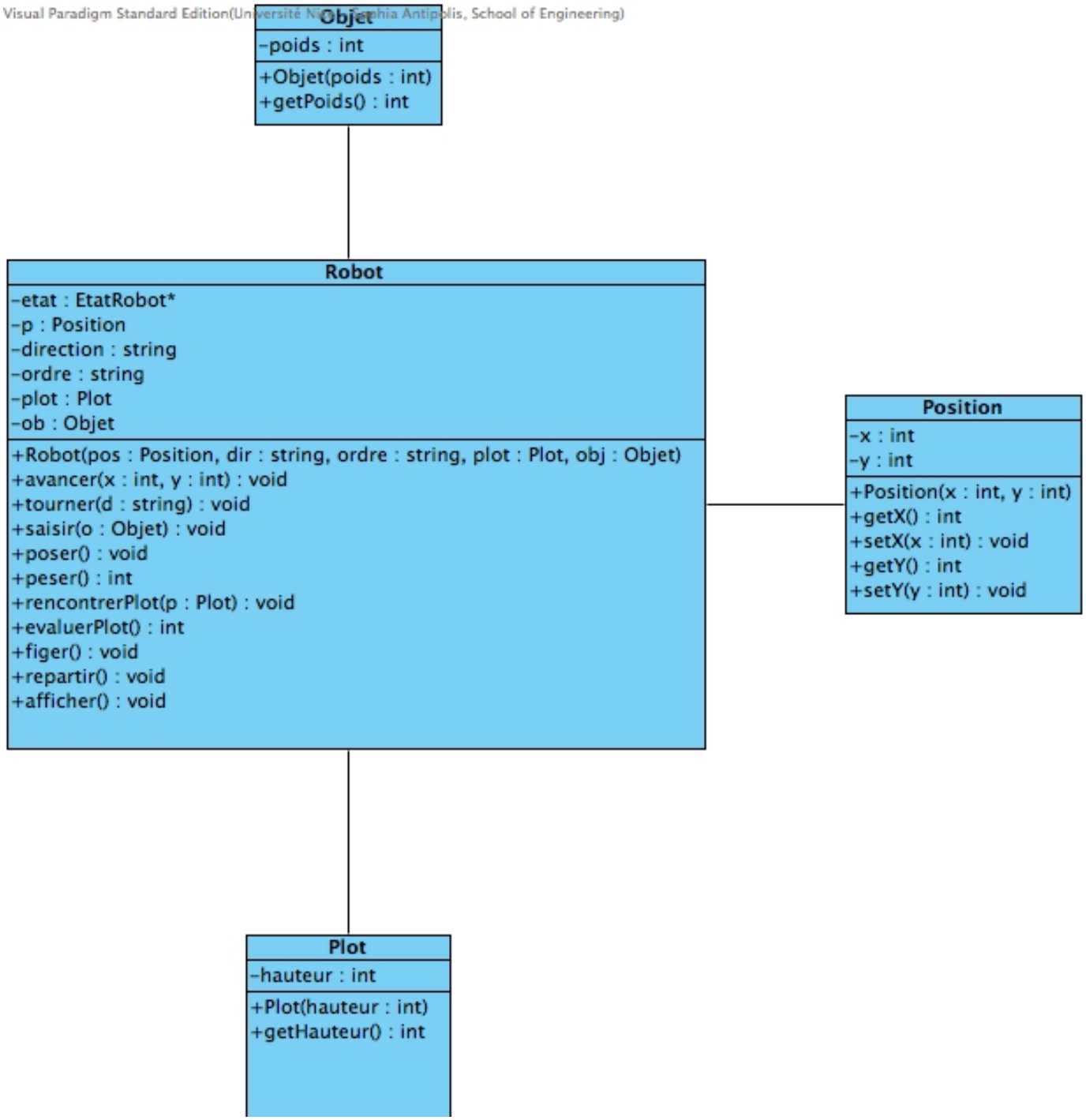
Pour ce faire nous avons mis en place plusieurs patrons de conception qui sont :

- Le patron de conception « état ».
- Le patron de conception « singleton ».
- Le patron de conception « observateur ».

Dans la suite du rapport nous ne détaillerons pas le code de chaque méthode de chaque classe pour la simple raison que cela serait trop lourd à écrire et à lire. Nous nous contenterons donc de quelques méthodes que nous considérons des méthodes « types » (ou « clés »). Nous prendrons notamment l'exemple de la méthode tourner du robot, notifier et afficher.

Robot

DIAGRAMME DE CLASSE



PSEUDO CODE

On suppose que notre robot possède en paramètre un pointeur sur un état qui lui permet de savoir dans quel état il se trouve, un paramètre direction lui indiquant sa direction, un paramètre ordre lui indiquant l'ordre reçu, et un paramètre plot indiquant le Plot.

Voici le pseudo code de la méthode tourner de la classe Robot :

void tourner (string dir) {

Si pas d'erreur alors :

    etat reçoit etat.tourner() ;

    direction reçoit dir ;

    ordre reçoit « tourner »+ dir ;

    on réinitialise plot à 0;

    on appelle la fonction notifier() de la classe Observable ;

Sinon on renvoie l'erreur

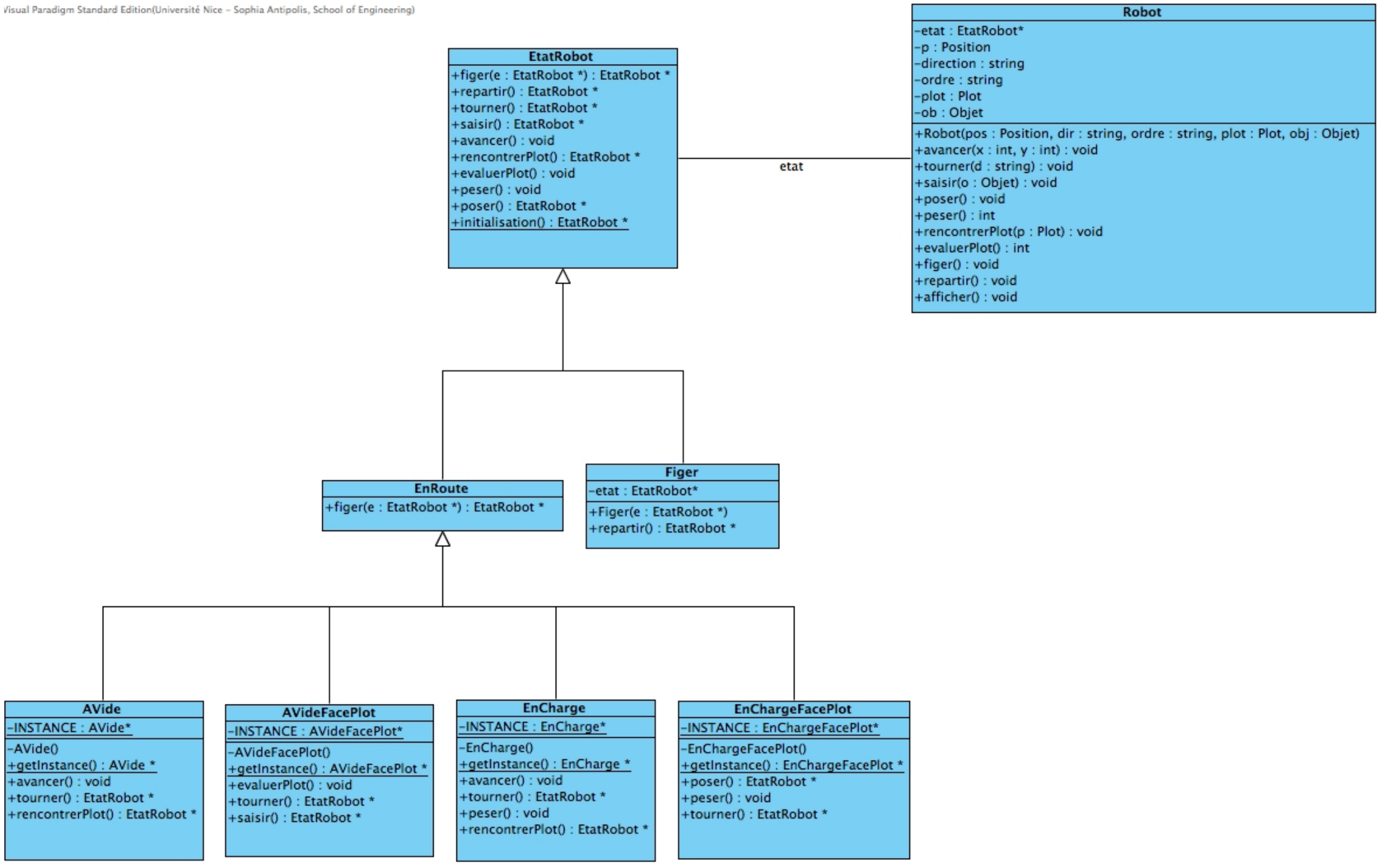
}

Dans cette fonction l'erreur serait remonter par la méthode tourner() dans le cas ou celle-ci serait appelé alors que le robot est dans l'état figé.

Schéma état

Nous avons choisis ce patron de conception car celui-ci était tout désigné pour gérer le changement d'états du robot. Cela permet au robot de passer aisément d'un état à un autre. Le principal problème auquel nous avons dû faire face était l'utilisation concrète du polymorphisme en POO, concept que l'on n'avait pas encore mis beaucoup en pratique en C++ et sur lequel quelques lacunes se sont faites ressentir au début.

DIAGRAMME DE CLASSE



PSEUDO CODE DE LA METHODE TOURNER DANS LA CLASSE ROBOT

Cette méthode n'est appelée que lorsque le robot se trouve dans un état ne possédant pas la méthode tourner (par exemple dans l'état Figer).

```
EtatRobot* tourner() {
        Lève une exception indiquant que l'on a pas le droit de faire ce mouvement ;
}
```

PSEUDO CODE DE LA METHODE TOURNER DANS LA CLASSE AVIDEFACEPLOT

```
EtatRobot* tourner() {
        Renvoie la variable static de la classe AVide qui correspond à l'unique
        instance de la classe AVide.
}
```

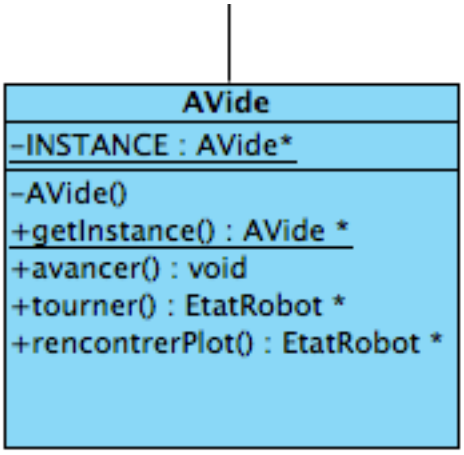
Grâce au polymorphisme du langage objet, le compilateur va toujours chercher la méthode la plus basse dans le diagramme de classe. C'est pour cette raison que nous pouvons lever l'erreur dans une classe mère et définir l'action a réaliser seulement dans les classes filles où nous voulons que cette action soit réalisée.

Nous pouvons également dans une classe fille laisser une méthode vide qui a pour seul but de ne pas renvoyer l'erreur de sa classe mère. Ainsi lorsqu'elle sera appelé a l'intérieur d'un try/catch aucune exception ne sera levée.

Schéma singleton

Nous avons choisis ce patron de conception car celui-ci était tout désigné pour éviter de dupliquer les différents états, nous ne voulions qu'une unique instance de chacune des classes états. Cela permet d'éviter de créer une nouvelle instance de classe à chaque fois que le robot change d'état. Le principal problème auquel nous avons fait face était la définition d'un singleton en C++, chose qui ne paraissait pas aisé puisqu'il faut utiliser des pointeurs, des variables static, et des constructeurs privés (chose que l'on fait rarement).

DIAGRAMME DE CLASSE



PSEUDO CODE

On suppose que l'on a une variable static nommée Instance.

On commence par initialiser cette variable :

Static AVide* Instance = new AVide() ;

On a un constructeur privé.

Et une méthode static nommée getInstance permettant de récupérer la variable Instance :

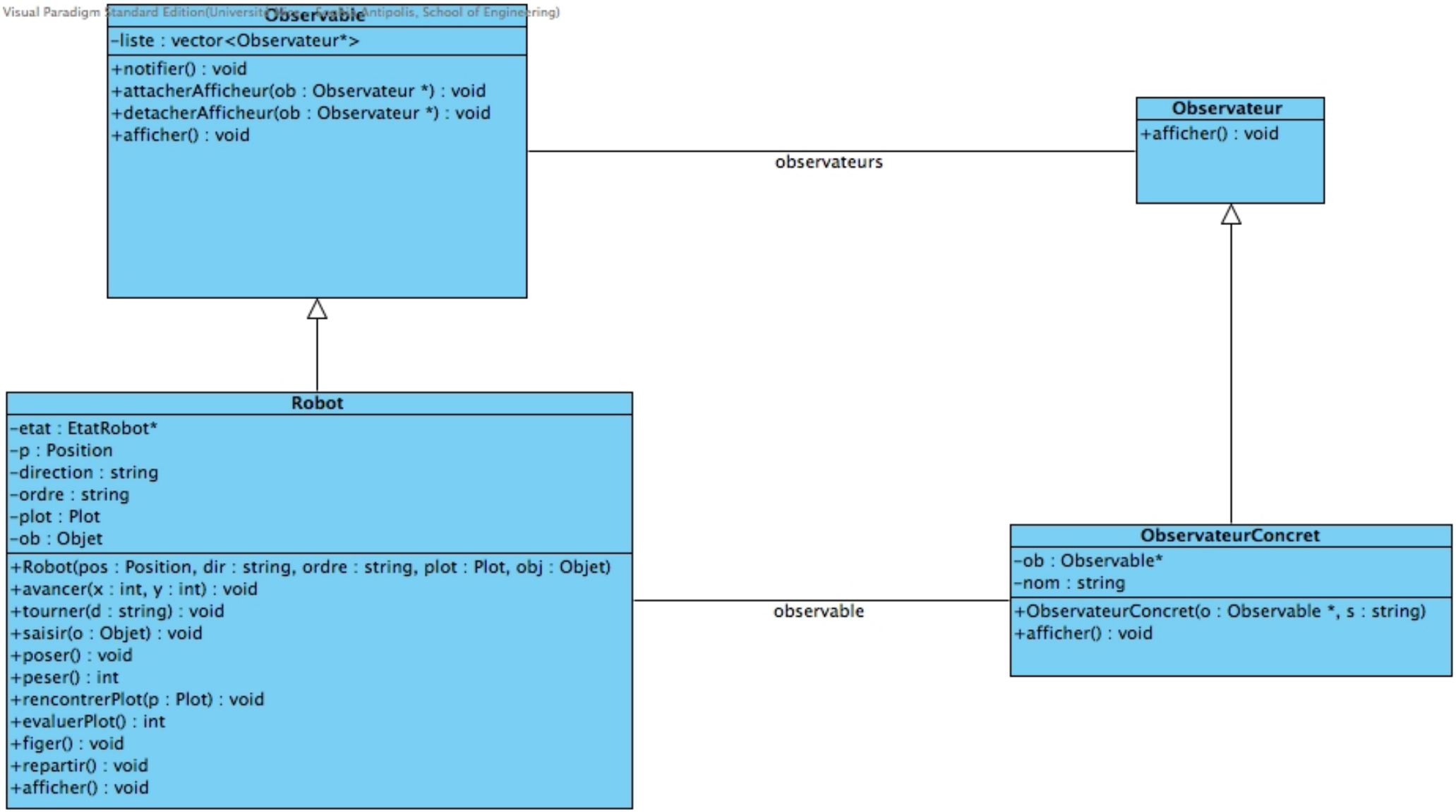
AVide* getInstance() {Renvoie Instance ;}

Schéma observateur

Nous avons choisis ce patron de conception car il permet de rendre l'affichage du robot totalement indépendant de celui-ci. De plus tous les ObservateurConcret héritants d'Observateur peuvent afficher n'importe quel objet d'une classe héritant d'Observable. Cela permet de une réutilisation et une maintenance très facile de l'affichage, sans modifier la classe Robot. Dans le main, on construit un Observateur sur un objet Observable (dans ce cas précis, notre Robot), ensuite on indique à notre robot qu'il va être affiché par celui-ci avec la méthode attacherAfficheur(). La méthode notifier() appelée après chaque modification du robot avertira l'observateur que le robot a changé d'état et celui-ci affichera le nouvel état récupéré.

La principale difficulté rencontrée pour la mise en place de ce schéma réside dans la compréhension de celui-ci. Une fois celui-ci bien compris et la maitrise des pointeurs acquises grâce au schéma singleton, ce schéma observateur se montre être un outils très puissant pour la POO.

DIAGRAMME DE CLASSE



PSEUDO CODE DE LA METHODE NOTIFIER DE LA CLASSE OBSERVABLE

Observable contient une liste d'Observateur associés.

```
void notifier() {
        pour tous les observateurs faire :
                Observateur -> afficher() ;
}
```

PSEUDO CODE DE LA METHODE AFFICHER DE LA CLASSE OBSERVATEURCONCRET

Un ObservateurConcret a en paramètre un pointeur sur l'Observable auquel il est associé.

```
void afficher() {
        Observable -> afficher ;
}
```

Dans notre cas l'observable est le robot.

PSEUDO CODE DE LA METHODE AFFICHER DE LA CLASSE ROBOT

```
void afficher() {
        cout << robot ;
}
```

Nous avons surchargé l'opérateur << afin que celui affiche pour un robot ces différentes caractéristiques (etat, hauteur du plot, poids de l'objet, etc ...).