Modélisation Objet d’un Simulateur Robot

Conception Objet

Enseignant : Michel Colette  
Polytech’ Nice-Sophia  
Si4 2016-2017

Auteurs

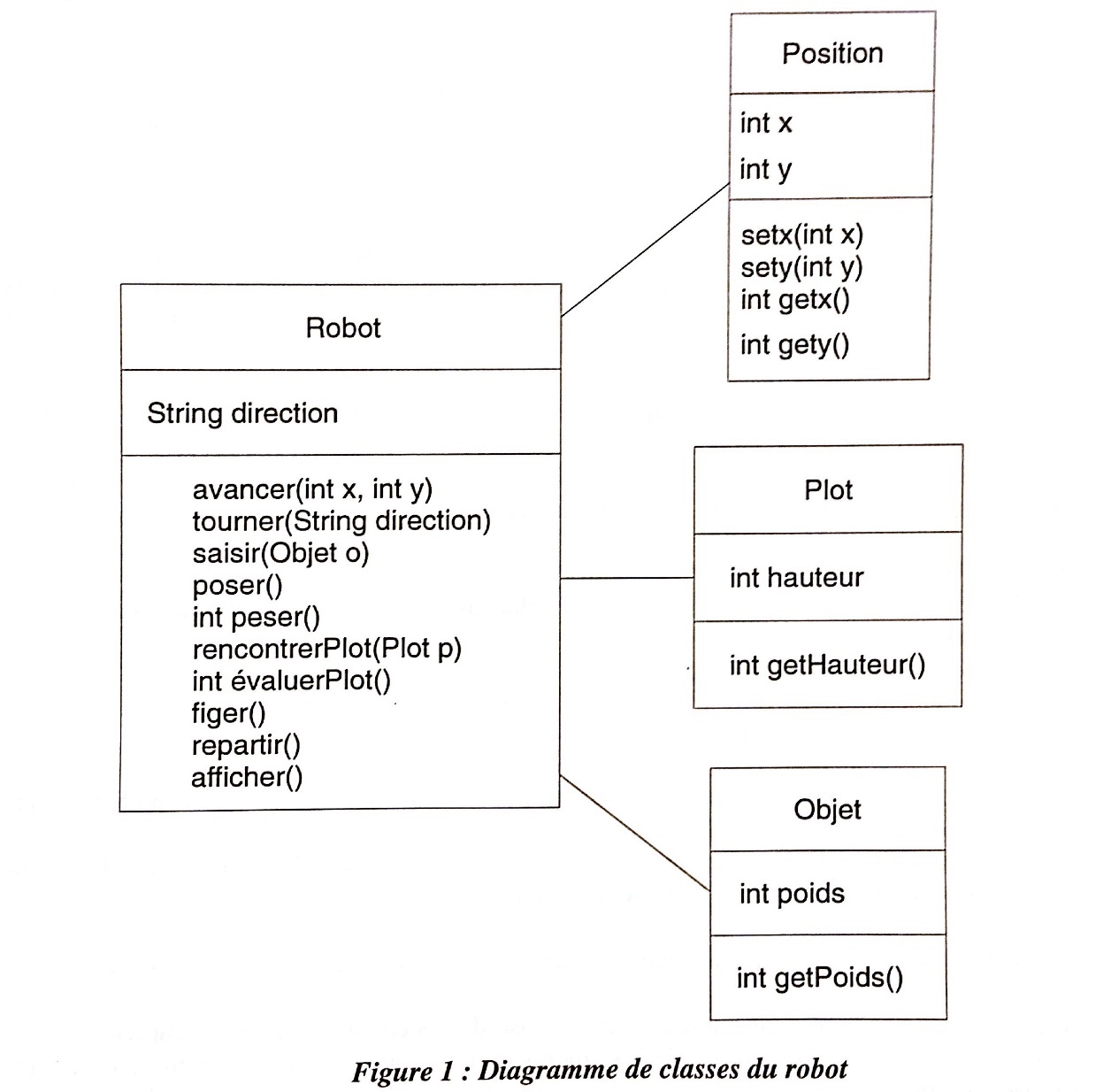
EL HAOUARI Zineb **&** GRIVON Justin

# Introduction et énoncé du problème

Nous souhaitons modéliser le comportement d’un robot disposant d’un certain nombre de fonctionnalités. Le Robot est capable de se déplacer sur un terrain muni une base de coordonnés (x,y). Des plots fixes sont posés sur ce terrain, et sont caractérisés par leur hauteur. Sur ces plots peuvent être posé des Objets d’un certain poids. Le robot peut interagir avec les pots et les objets à l’aide de commandes. Ces commandes peuvent être élémentaires ou composés d’autres commandes.

Pour modéliser ce problème, nous disposons d’un outil de programmation objet, ainsi qu’un ensemble de schémas de conception.

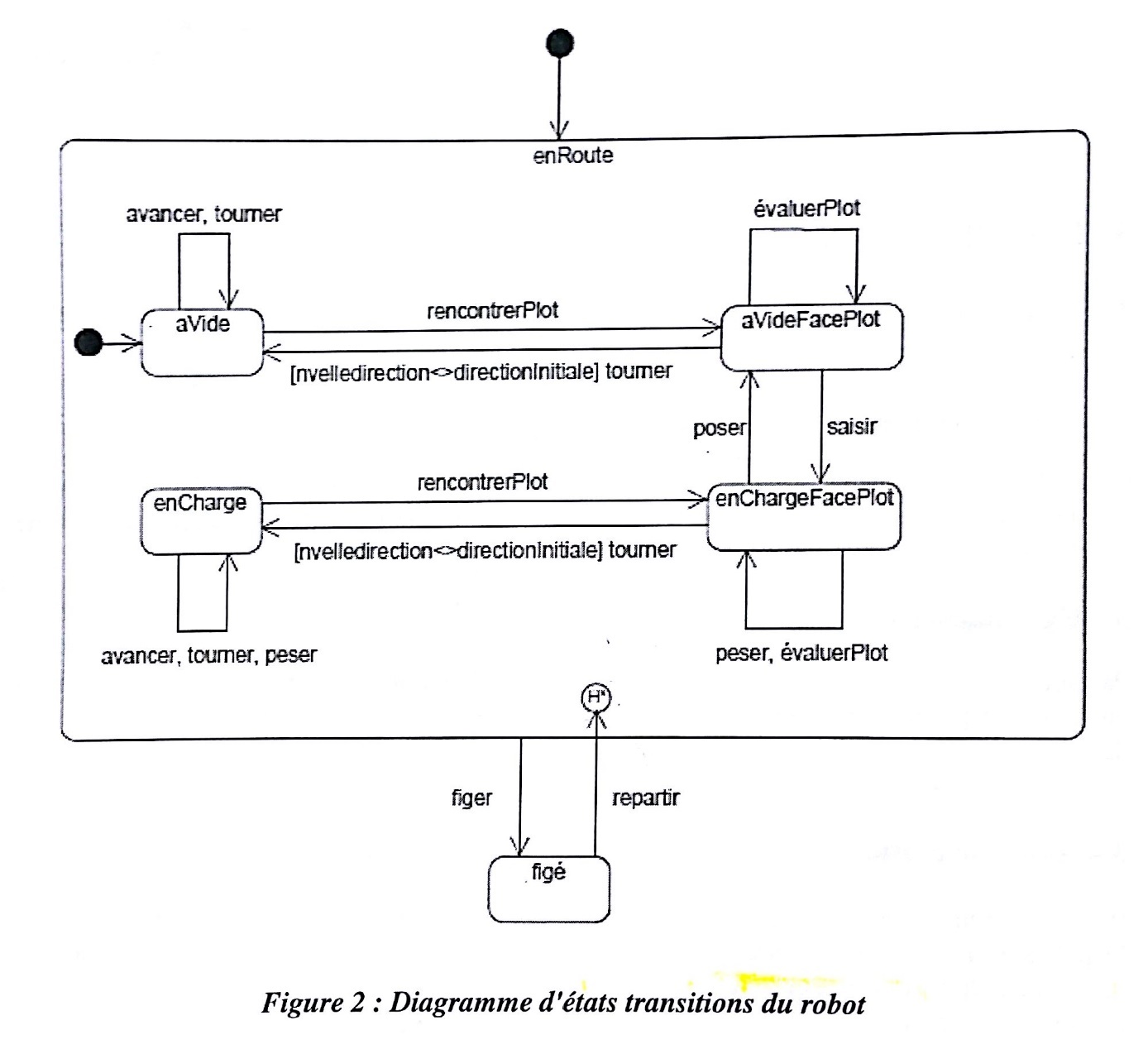
Nous partons d’une classe Robot, mise en évidence dans le diagramme de classe ci-dessous modélisant le comportement global du robot.



Le but de ce projet est d’implémenter le robot, en respectant les spécifications demandées, mais surtout, d’améliorer la structure du projet. Puisque ce projet est réalisé dans le cadre de la matière de Conception Objet, ces améliorations devraient permettre à l’implémentation finale de respecter au mieux les principes du paradigme objet. Le projet final doit être le plus ouvert à l’extension et le plus fermé à la modification.

# Le Schéma État

En plus d’être capable d’exécuter des actions le comportement du Robot doit respecter un ensemble de contraintes. À tout moment, le robot peut être caractérisé par son état selon s’il charge un objet, ou s’il est en déplacement. Certaines actions lui permettent de passer d’un état à un autre. Le comportement est décrit par le diagramme d’états transition ci-dessous.



*Afin de contrôler le comportement du robot et respecter le schéma transition, une solution naïve et rigide est de mettre en place une série de test qui vérifient l’état courant du robot et qui en fonction de cela permettent de lancer ou pas une méthode.*

*Mais cette solution n’est pas celle que nous avons employé, car elle ne respecte pas les principes que nous avons défini. En effet en plus d’être très couteuse en termes d’opérations, toute modification apportée au comportement du robot -ajouter un état, par exemple - impose de nombreuses modifications à apporter aux implémentations de toutes les méthodes de la classe Robot.*

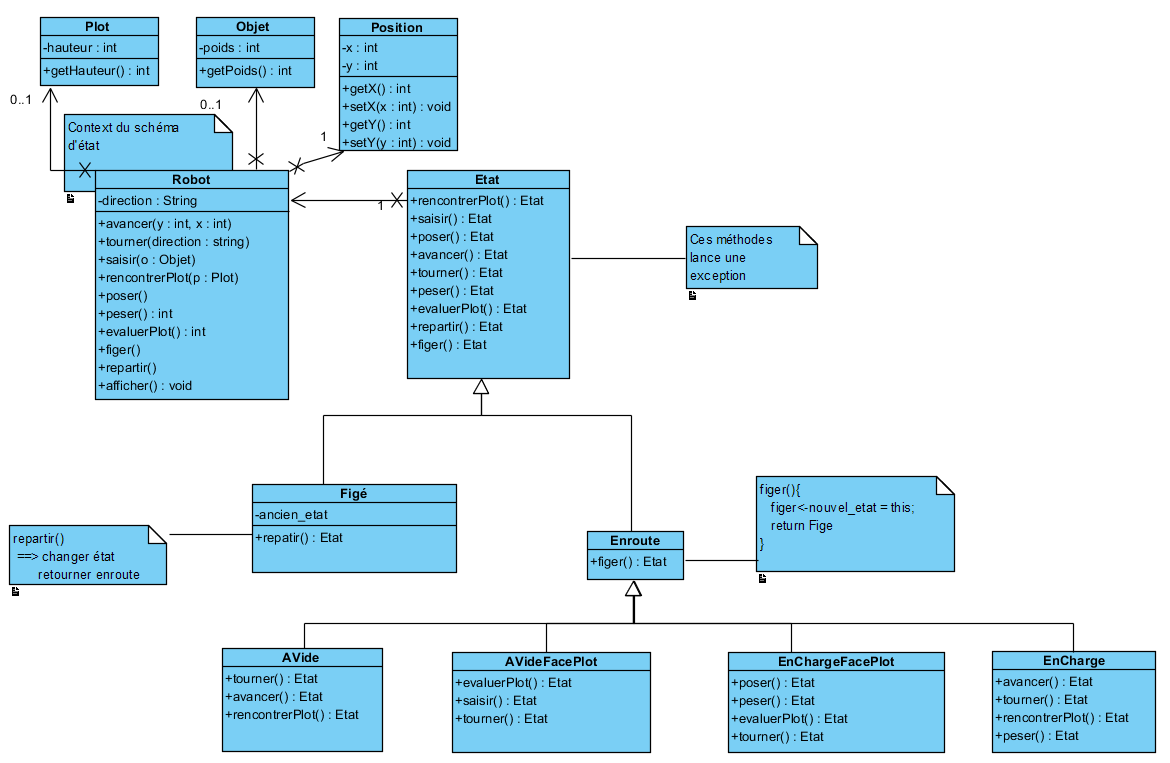
## Solution *évolutive*

La solution que nous avons adoptée est d’utiliser le schéma de conception « État », et ce pour plusieurs raisons. Tout d’abord, en étudiant le problème on comprend que le problème est comportemental. Le schéma d’état transition montre que le fonctionnement du robot est complexe. En effectuant des actions le robot est capable de changer d’état selon lequel il est permis ou pas d’effectuer une action donnée. Nous avions besoin de modéliser ça, à l’aide d’un outil de control.

L’utilisation de ce schéma permet de mettre en place ce control à l’extérieure de la classe Robot. Ceci permet d’alléger cette classe de point de vu du code et des responsabilités.

### Mise en place

*Afin de mettre en place ce schéma, on utilise le polymorphisme que nous permet le langage objet.* L’état du robot est donc une classe « Etat » qui généralise plusieurs classes qui sont les mêmes que ceux possible pour un robot. Elles correspondent aux états du schéma état transition. L’état d’un Robot peut changer de manière dynamique lors de l’exécution d’une action. Les méthodes de ces classes ne sont pas chargées du comportement des actions elles-mêmes, mais seulement du changement de l’état du robot.



1 Diagramme de du schéma état

Concrètement, les méthodes des classes état sont réduites, elles n’ont pas besoin de paramètres, et retourne toutes une instance d’un sous type de « Etat ». Certaines méthodes retourneront la classe elle-même : ceci correspond aux actions qui ne font pas changer d’état.

🢂 **AVideFacePlot**::**saisir**(){  
 **retourner** new **enCharge**;  
}

#### Que se passe-t-il lorsqu’une action n’est pas permise ?

Grâce au polymorphisme, le problème se résout très simplement. Les méthodes de la classe mère lance toutes des exceptions qui afficheront un message au client : « Action impossible ». Dans les états ou l’action est permise, il suffit alors de surcharger ces méthodes.

//ainsi que toutes les autres méthode de Etat

🢂 **Etat::saisir**(){   
 **throw** error\_action\_impossible ;  
 }

L’implémentation de saisir() dans la classe robot doit pouvoir catcher l’exception dans le cas ou elle est lancé, si aucune exception n’est lancée, on peut continuer la méthode et exécuter le code de l’action. Le pseudo-code ci-dessous l’explique clairement :

🢂**Robot::avancer**(**int** x, **int** y){  
 try{  
 **etat**.**avancer**() ; //retourne ce même etat   
 **this.x**<<**x** ;  
 **this.y**<<**y** ;  
 catch{  
 **afficher** "action impossible" ;  
 }  
 }

#### Et pour l’état figé ?

Le problème de l’état figé est qu’il faut être capable de mettre le robot en « pause » et de pouvoir retourner au moment de la reprise (l’état enRoute) au dernier état avant la pause. Pour cela nous avons tout simplement gardé en attribut l’ancien état. Au moment du passage à l’état « figé » on lui passe l’état courant. Ensuite pour retourner à l’état de départ, on retourne cet attribut dont le type dynamique correspond à l’état recherché.

🢂**enRoute::figer**(){  
 figer<-nouvel\_etat = this;  
 **return** new Fige ;   
 }

### Conséquences ?

La classe Robot reste responsable de la modification de ses données, et se charge par exemple de déplacer le robot sur le terrain. L’ajout de l’attribut de type générique « Etat » dans Robot est nécessaire, et chaque méthode de Robot doit utiliser une méthode du même nom cet état afin de permettre la vérification. Le Robot n’a pas accès aux noms des différents état, ce qui rend l’ajout d’un nouvel état particulièrement simple. En effet, un simple rajout d’une classe fille de « Etat » qui contiendra les actions permises dans cet état-là.

### 

# Pattern Singleton

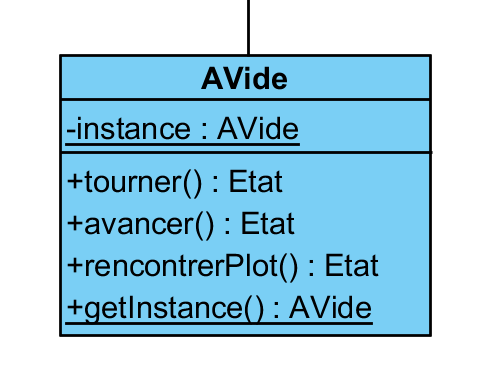
Après la mise en place du schéma « Etat » nous avons très vite été confronté à un problème. En effet, à chaque changement d’état on a besoin de créer une nouvelle instance Etat. Ainsi, pour un scénario ou le robot ferait un grand nombre d’actions, on créerait un grand nombre d’état, remplissant la mémoire inutilement. De plus, les classes Etat ne comportent pas d’attributs et ne contienne pas de données puisque toutes les informations du robot sont stockées dans la classe Robot.   
On souhaite donc pouvoir créer le minimum d’instance et pouvoir réutiliser les mêmes objets même après être passé à d’autre état.

## Solution proposée

Parmi les schémas de conception de création, on trouve le schéma Singleton. Ce schéma permet de créer qu’une seule instance sans passer par une variable globale.

### Mise en place

La mise en place de ce pattern demande tout d’abord de rendre le constructeur non public. En effet, l’accès aux nouvelles instances du singleton devient contrôlé. L’instance qui dans notre cas sera unique, est une variable statique, déclarée et instancié dans le .cpp de l’état. Dans notre projet, ceci revient à ajouter dans chaque classe d’état « static Etat instanceUnique = new Etat() ». Pour accéder à une nouvelle instance on passe par une méthode particulière, static qui retourne « instanceUnique ».



2 Schéma du Singleton pour l'état AVide

### Limite du polymorphisme

Rappelons que les classes singleton que nous souhaitons mettre en place sont des classe Etat. Elles sont toutes des classes filles de « Etat ». C’est grâce au polymorphisme que nous avons réussi à mettre en place le schéma état. D’une manière naïve, on peut penser à mettre l’« instanceUnique » dans la classe mère, puisque toutes les classes filles ont en besoin. Cependant, puisque le pattern singleton a besoin d’une instance static, on perd le type dynamique de cette variable. On est donc obligé de mettre dans chacune des classe filles l’instance et implémenter la méthode.

Dans chaque classe on a donc l’implémentation suivante :

**static** instanceUnique ;

…

🢂 **static** UneClasseEtat **getInstance()**{  
  **return** instanceUnique;  
 }

Il ne reste plus qu’à modifier les instanciations des état pour remplacer chaque « new Etat » par « Etat::getInstance ». Grâce à cela, nous avons pu économiser de la mémoire qui avant était utilisée inutilement.

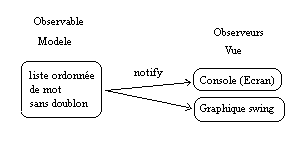
# Pattern Observateur

Après avoir implémenté la partie comportementale du robot qui s’occupe de la transition des états, on se rend compte que lorsqu’on appelle les actions sur le *Main*, on a besoin d’appeler la méthode afficher afin de pouvoir visualiser l’état et les informations du robot sur la vue – même si pour l’instant la vue est réduite à un affichage sur la sortie standard.

L’idéal serait de pouvoir à chaque méthode qui change l’état du robot, lancer un affichage de manière automatique. Par ailleurs, dans le cas où nous souhaitons remplacer l’affichage courant par une vue plus élaborée l’opération deviendra plus compliquée.

## Solution proposée

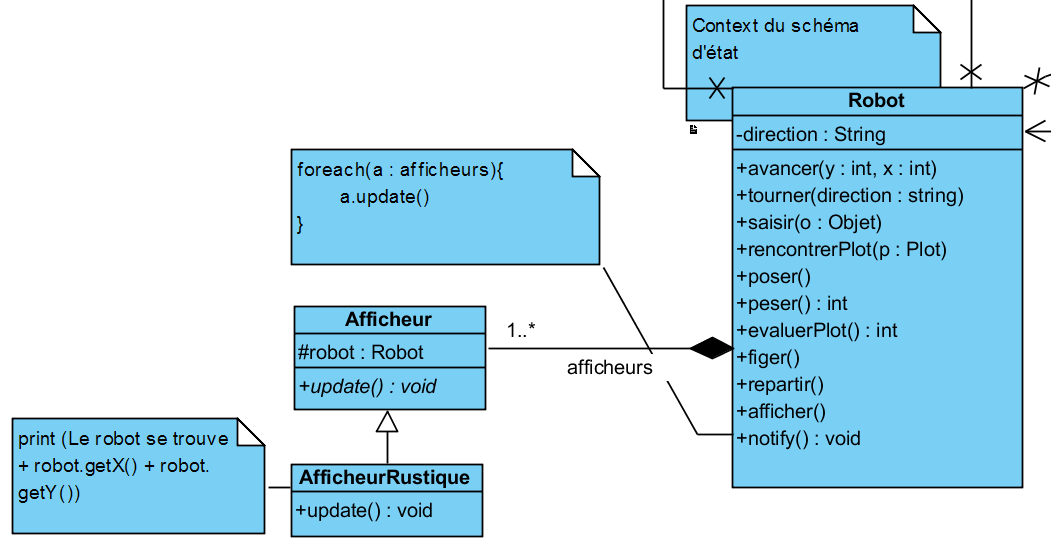
Pour cela on peut utiliser un « Observateur ». Le schéma « Observer-observable » est très utilisé dans le domaine d’IHM. Lorsque la vue est très élaborée, il n’est pas toujours possible de la recharger en entier à chaque changement de l’état. C’est pour cela qu’une vue observatrice est très souvent adaptée pour un affichage d’état. De plus, des programmes ont parfois besoin d’avoir plusieurs vues en même temps. C’est par exemple le cas pour des sites sur lequel on peut être utilisateur public, identifié ou administrateur, ou pour les applications sur plusieurs supports.



3 Exemple de cas d'utilisation d'affichage différents.

### Mise en place

Tout d’abord, même si pour l’instant nous n’avons besoin que d’un affichage, afin de permettre l’évolutivité de notre programme on utilise un Afficheur générique puis à l’aide du polymorphisme mettre en place un ou plusieurs afficheurs.



4 Diagramme de L'afficheur Observer

La classe Afficheur connait le Robot, afin de pouvoir lire les informations de celui-ci. Le robot sera « protected » afin de pouvoir y accéder depuis les classes filles : Afficheurs concrets. La méthode abstraite update()de Afficheur est implémenté dans la classe fille. Dans notre cas elle va se charger d’afficher les nouvelles positions du robot.

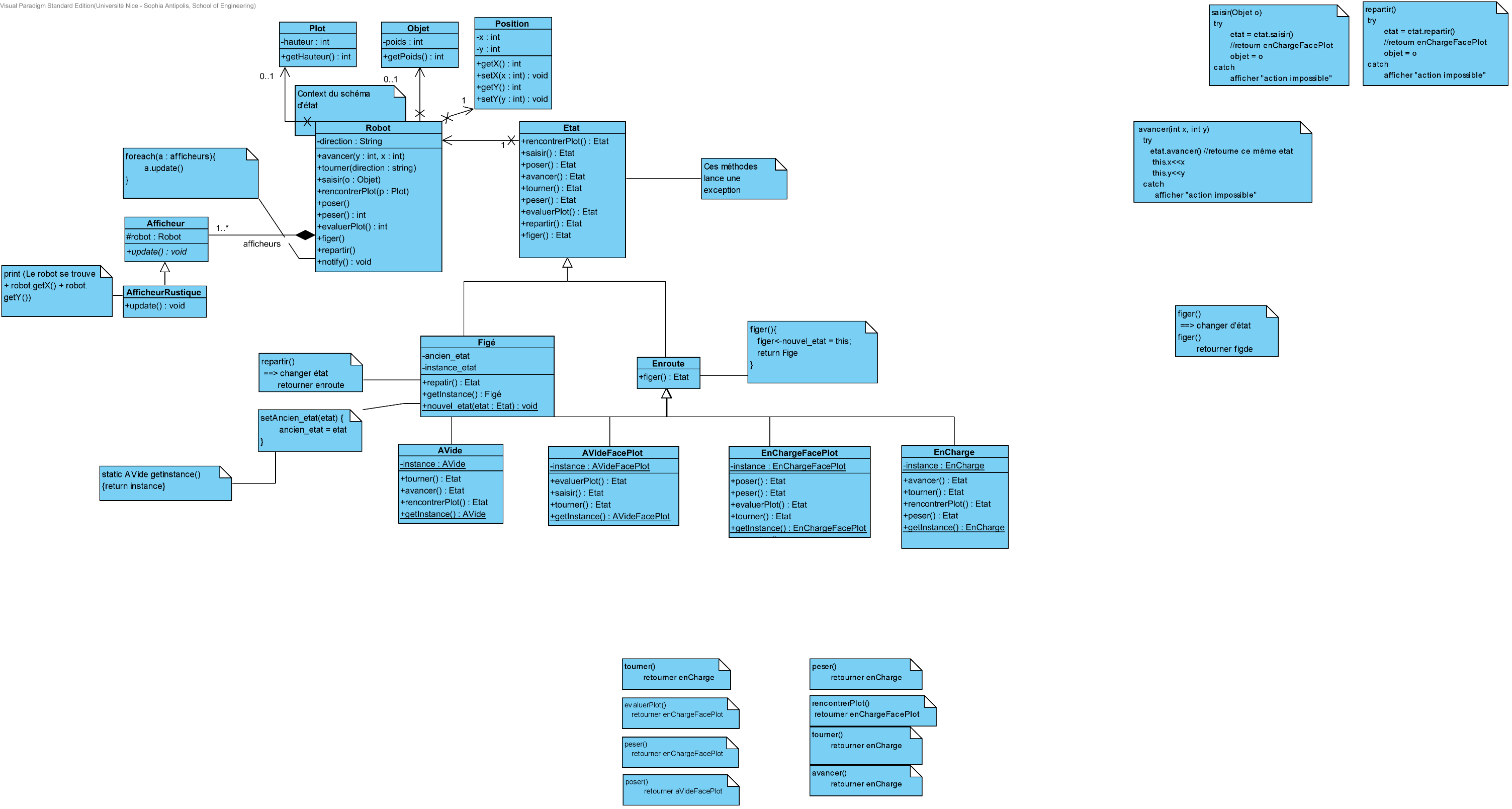
**AfficheurRustique::**

🢂 **update()**{  
 print(‘‘Le robot se trouve’’ + **robot**.getX() + **robot**.getY());  
 }

Par ailleurs, il faut que le Sujet, dans notre cas Robot, se compose de la liste de ses observateurs. Robot doit aussi implémenter la fonction notify(). Cette fonction effectue un parcours de toute la liste de observateurs de robot, et les « notifier » du fait qu’il y a eu un changement dans l’état du Robot. Dans les méthodes appropriées (ç.à.d. les actions qui changent les états du Robot). Enfin, voici ce que deviennent les méthodes de Robots :

🢂**Robot::notify()**{  
 foreach(a : **afficheurs**){  
 a.**update()**  
}

🢂**Robot::avancer**(**int** x, **int** y){  
 try{  
 **etat**.**avancer**() ; //retourne ce même etat   
 **this.x**<<**x** ;  
 **this.y**<<**y** ;  
 **notify();**  
 catch{ **afficher** "action impossible" ;}  
 }

Enfin, voici le diagramme global du robot, après avoir appliqué les modifications précédentes.

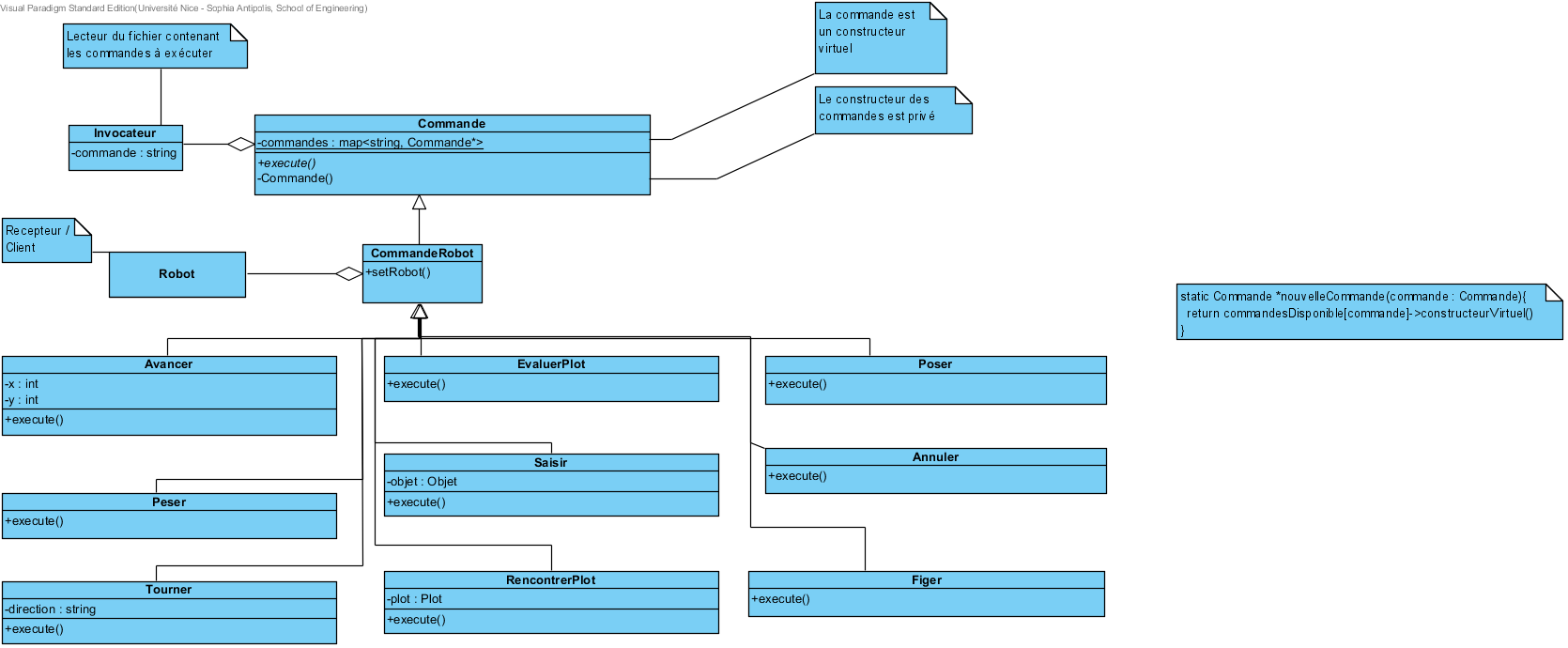
5 Diagramme de classe du robot (toutes les solutions intégrées)

# Pattern Commande

Afin de commander le robot nous avons pour l’instant utilisé une suite d’appels de méthodes sur le Main. Nous souhaitons maintenant mettre en place une interface pour écrire les commandes à exécuter sur un fichier par exemple. Pour cela nous avons mis en place le pattern Commande.

### Mise en place

Pour mettre en place le pattern commande, on fait hériter d’une classe mère Commande l’ensemble des classes commandes, elles représentent les différentes commandes dont dispose le robot. Voici le diagramme de classe correspondant à cet arborescence :



Notons que pour l’instant, les commandes ne traitent pas les paramètres.

L’invocateur lis le fichier contenant la liste de commandes, précédemment listées sur le co

### Conséquences

Interface avec l’utilisateur indépendante du programme, plusieurs interfaces de commandes possibles pour le même Robot logique.

Permet la mise en place de l’option dés-exécuter.

Nous souhaitons maintenant mettre en place une interface pour écrire les commandes à exécuter sur

# Pattern Constructeur virtuel

## Desexécuter

## Macro-Commande

[Fin Du Document]

