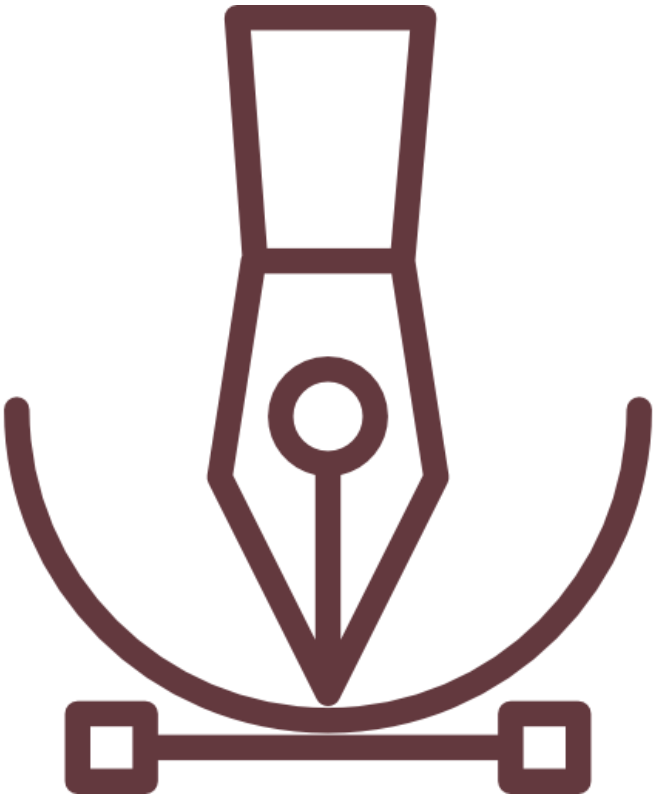


Conception Orientée Objet

Modélisation objet d'un simulateur de robot

Vincent
RAYBAUD

Rémy
KALOUSTIAN

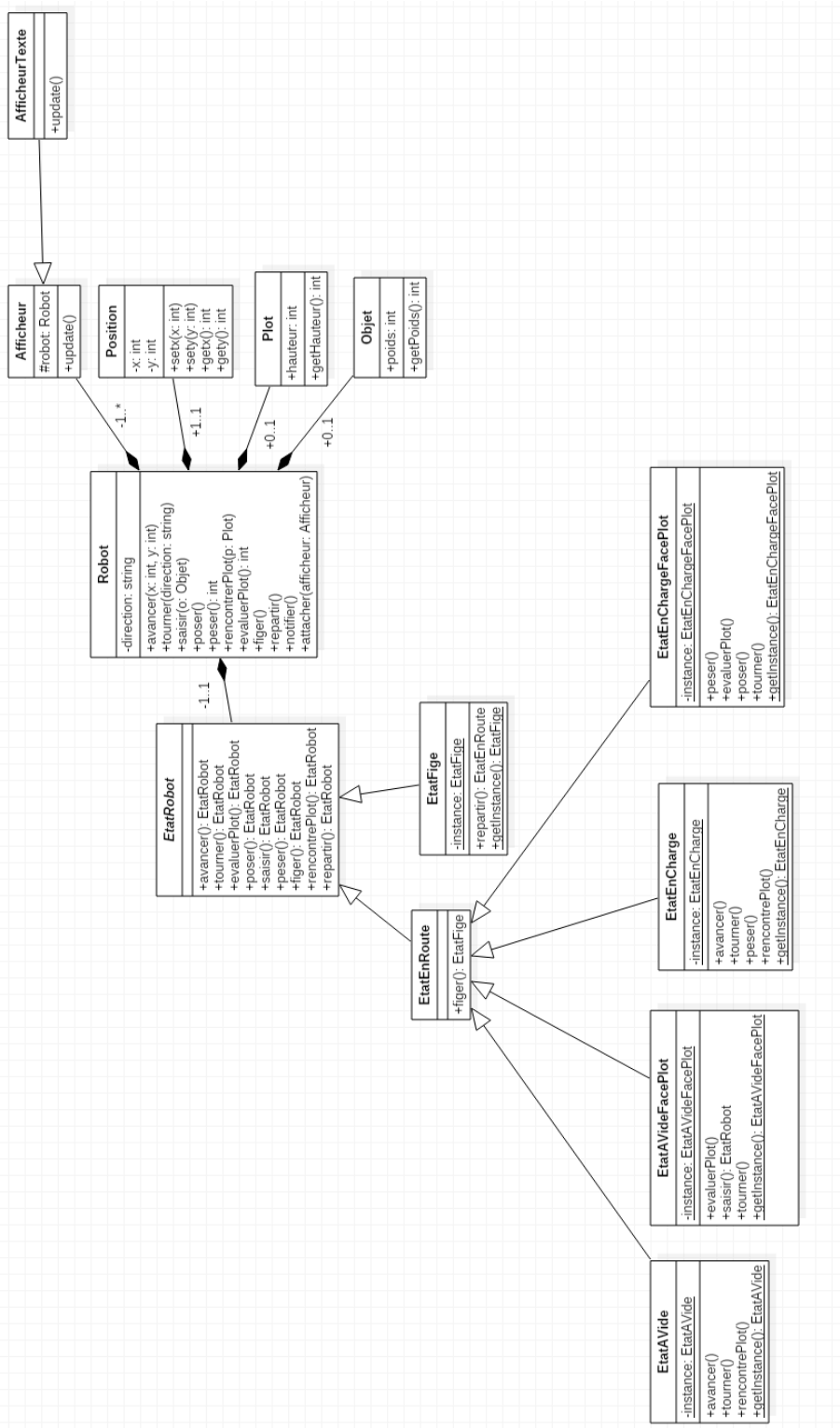
Polytech Nice-Sophia

SI4 – G3 - 2016

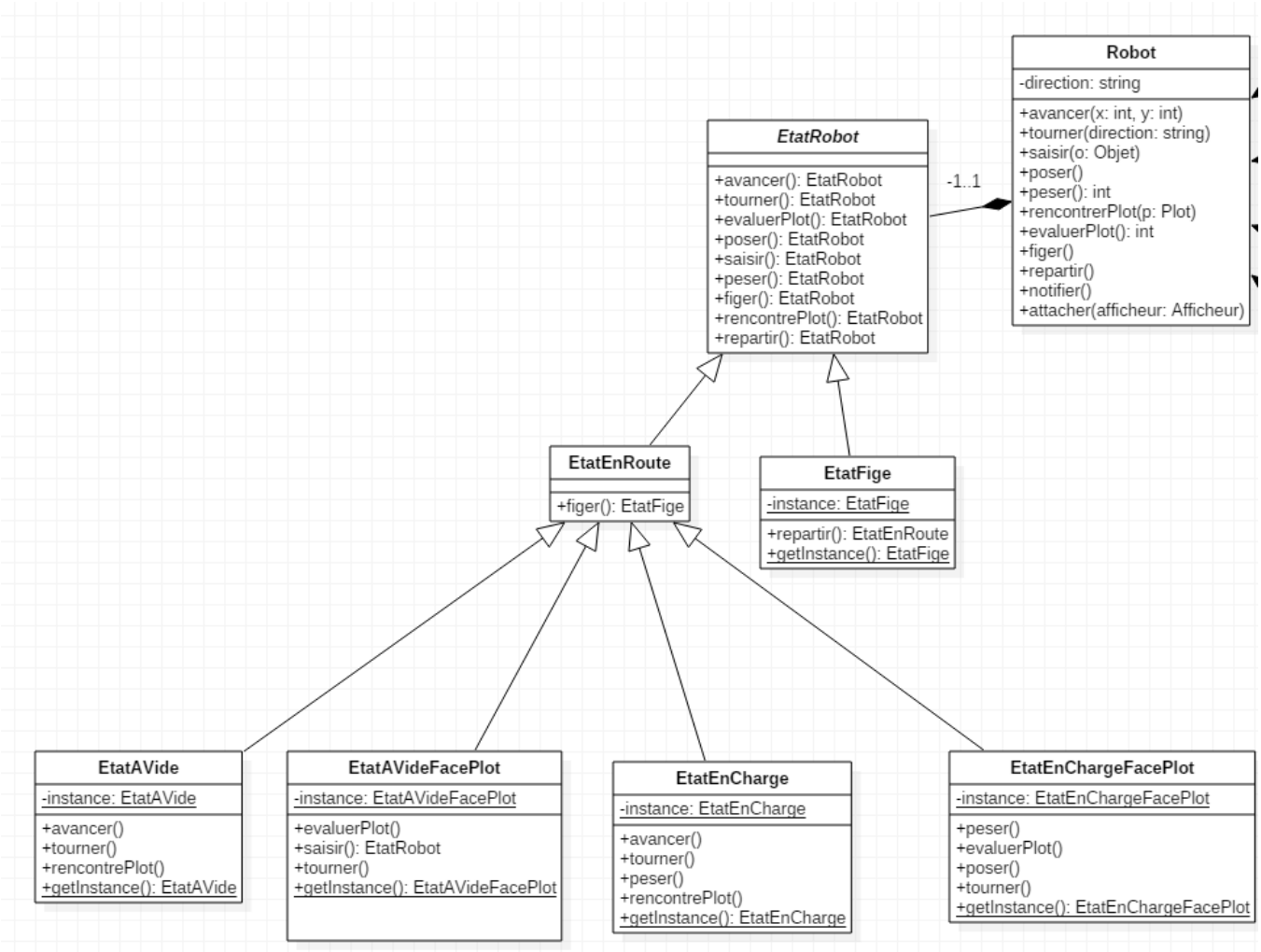
Introduction

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**



Pourquoi un État ?

Nous avions besoin d'un schéma qui pouvait 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 schéma 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.

Par ailleurs, 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 une 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 par ailleurs 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 serait 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 le passer directement 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 permettaient de 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 y a un changement d'état, on retourne le nouvel état, sinon, this (l'état lui-même), 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 EtatRobot, 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.

Pseudo code

➔ EtatRobot ::[toutes les méthodes]

```
EtatRobot methodeDansEtatRobot()
{
      throw UnauthorizedCallException ;
}
```

➔ EtatAVideFacePlot ::saisir()

```
EtatRobot saisir()
{
      return EtatEnCHargeFacePlot ::getInstance() ;
}
```

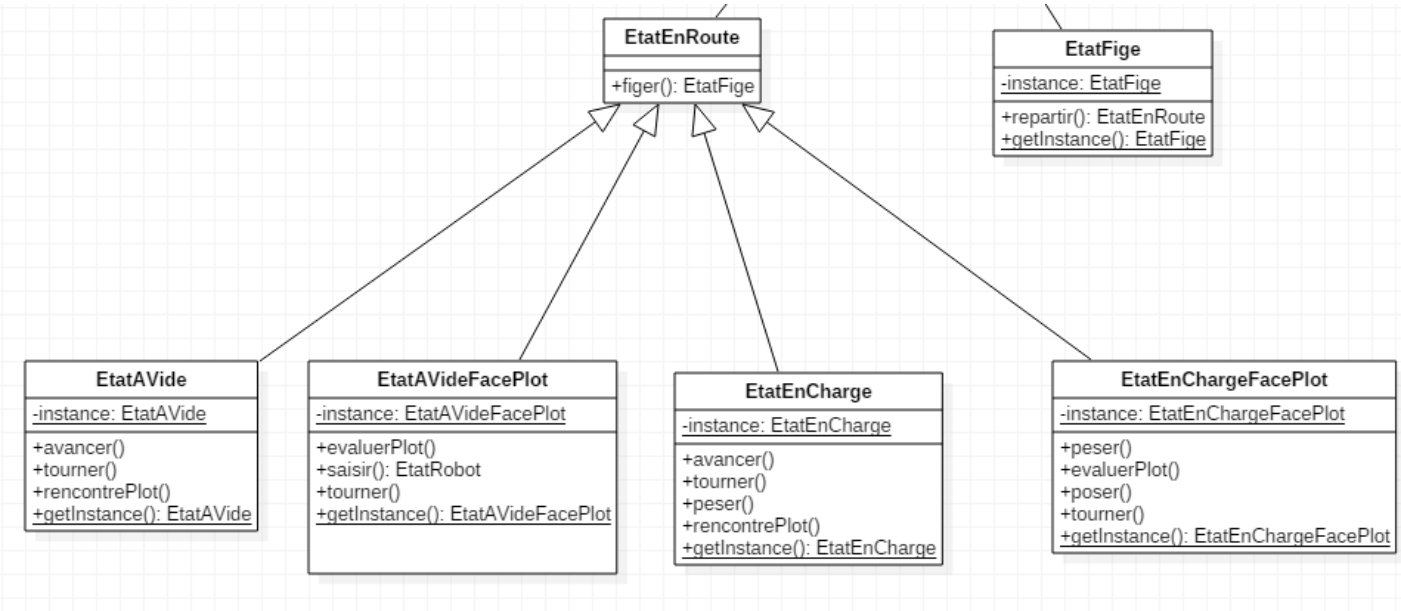
➔ EtatEnChargefacePlot ::figer() (qui est celle hérité de EtatEnRoute)

```
EtatRobot figer()
{
      return EtatFige ::getInstance(this) ;
}
```

➔ EtatFige ::repartir()

```
EtatRobot repartir()
{
      return ancienEtatRobot ;
}
```

1 L'état est (aussi) un **Singleton**



Pourquoi un Singleton ?

Nous nous sommes rendus compte qu'à chaque changement d'état, il y avait une création dynamique d'objet. Ainsi, si nous allions six fois dans l'état figé, il y aurait eu six objets EtatFige créés. La création de six pobjets n'est pas un problème en soi, le problème est que nous allons créer six objets totalement identiques, étant donné qu'il n'y a pas d'attributs dans ces états. De ce fait, il est préférable de ne créer qu'un seul objet par état. C'est pour cela que nous avons utilisé le schéma Singleton, qui nous permet de créer une seule instance d'un état, que nous pouvons réutiliser à volonté.

Quelles sont les conséquences ?

La conséquence principale est qu'il n'y a à présent qu'une seule instance par état, ce qui nous permet d'économiser de la mémoire, et surtout, d'améliorer la logique de notre programme, dans lequel nous n'avions aucune utilité de créer plusieurs instances d'un même état.

Le singleton est alors instancié dès la création de l'objet, et l'utilisateur n'aura même pas à s'en soucier.

Choix particuliers et problèmes

L'implémentation du schéma Singleton nous force à créer un singleton dans chaque classe fille d'état que l'on pourra instancier. Ainsi, l'ajout d'un nouvel état devra faire l'objet d'un ajout d'une variable static instance, et de la méthode getInstance(). Nous voyons donc là les limites du polymorphisme. Nous sommes obligés de créer les singletons dans les classes filles, si nous le faisions dans la classe mère, cela romprait le principe du singleton.

Par extension avec ce qui vient d'être dit, la création d'un singleton limite les possibilités d'héritage.

Nous pouvons aussi remarquer que l'implémentation d'un singleton nous fait nous éloigner légèrement du principe de responsabilité unique, puisque'ici, c'est à la classe elle-même de s'instancier, via un singleton.

Pseudo code

Nous montrons un exemple de la méthode getInstance() de la classe EtatAVide, les autres classes Etat (qui sont instanciables) suivent ce même principe.

➔ EtatAVide ::getInstance()

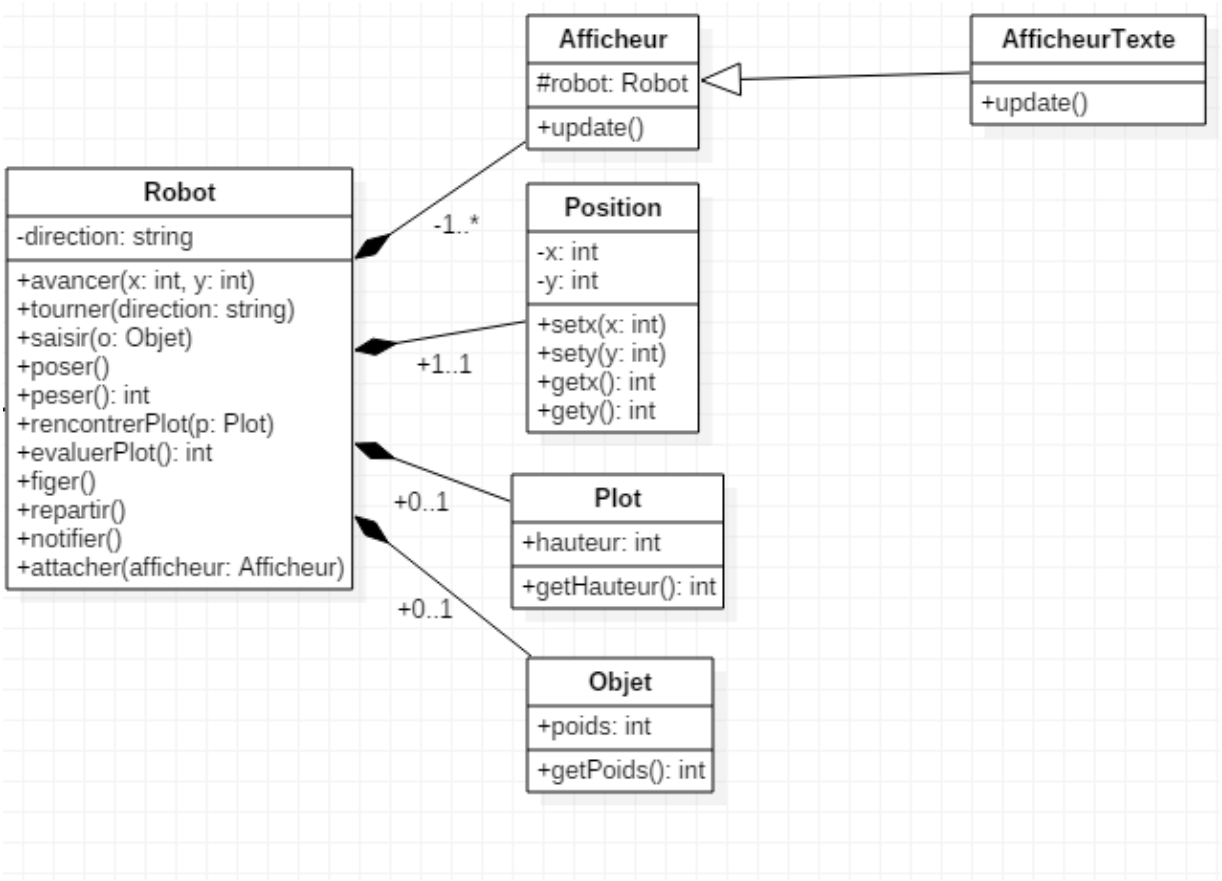
```
EtatAVide  getInstance()
{
      return EtatAVide ::getInstance() ;
}
```

Il y a un léger changement pour EtatFige qui doit conserver l'état précédent

➔ EtatFige ::getInstance()

```
EtatFige  getInstance(EtatEnRoute ancienEtat)
{
      instance.setEtatEnRoute(ancienEtat) ;
      return instance ;
}
```

L'afficheur est un **Observateur**



Pourquoi un Observateur ?

Il nous fallait un schéma qui puisse afficher des données, avec un afficheur par type d'affichage de données, tout en assurant que ces dernières sont correctes à tout instant.

Nous avons choisi le schéma Observateur tout d'abord pour la possibilité de représenter plusieurs données simultanément tout en isolant leur affichage (il peut y avoir plusieurs classes afficheur, mais chacune pour un type d'affichage de données précis).

Ensuite, le stockage des afficheurs dans le Robot (un vector d'Afficheur) nous permet d'enlever ou d'ajouter dynamiquement des Afficheurs. Par exemple, le passage dans un nouvel état peut requérir la création d'un nouvel afficheur, chargé d'afficher des données jusque-là absentes.

Enfin, la puissance de la fonction notifier() se révèle être fort pratique pour synchroniser les afficheurs avec le Robot. Utilisée avec l'héritage et le polymorphisme pour les afficheurs (via update()), notifier() permet d'appeler update() pour chaque Afficheur. Elle garantit ainsi la cohérence des données affichées dès qu'il y a un changement.

Quelles sont les conséquences ?

Nous pouvons maintenant modifier l'objet Robot sans avoir besoin de communiquer ce changement aux afficheurs explicitement. Il n'y a qu'à utiliser notifier() qui va mettre à jour les afficheurs. Ces derniers disposant du Robot, il n'y a pas besoin de leur transmettre les nouvelles données.

L'ajout d'un nouveau type d'Afficheur peut se faire simplement. En créant une classe héritant d'Afficheur, en implémentant update() pour cette classe, et en ajoutant une instance de cette classe dans la liste d'afficheurs du Robot.

Nous disposons d'un faible couplage entre le Robot et ses Afficheurs, le comportement de l'un ne dépend pas beaucoup du comportement de l'autre. Seule la mise à jour des données affichées dans les afficheurs crée une légère dépendance.

Choix particuliers et problèmes

Nous avons choisi de passer le Robot en paramètre à la construction de chaque Afficheur. Même si un Afficheur n'a pas forcément besoin de toutes les données d'un Robot, il est plus pratique de passer le Robot à la création et de sélectionner seulement les données utiles dans l'Afficheur, plutôt que de faire cette sélection au niveau du Robot et donc modifier les paramètres à la construction pour chaque Afficheur.

Le premier problème est justement que nous passons tout le Robot en paramètre, alors qu'un Afficheur n'a pas besoin de toutes ses informations. Toutefois, c'est un compromis nécessaire pour garantir la simplicité de création des afficheurs.

Un problème similaire à l'état est l'obligation de créer une classe par Afficheur. Là aussi, ce compromis présente plus d'avantages que d'inconvénients (notamment la séparation claire et extensible des affichages).

Un autre problème léger est que, de base, l'ordre des afficheurs n'est pas fixé. Ainsi, si certains afficheurs doivent être mis à jour avant d'autres, il faut rajouter une couche de traitement pour savoir lesquels notifier en premier. Par exemple, si on imagine une interface graphique basée sur cette application, et que le Robot trouve un Plot, il serait judicieux de mettre à jour les données affichées sur le Robot lui-même, et ensuite mettre à jour les données sur le Plot trouvé dans l'afficheur.

Pseudo code

➔ Robot ::notifier()

```
void notifier()
{
	for each (Afficheur a in ListeAfficheurs)
	{
		a.update() ;
	}
}
```

➔ AfficheurTexte ::update()

```
void update()
{
	cout<< '' Le Robot est en position ''<<
	_robot.getPosition().getx()<<'' : ''<< _robot.getPosition.gety() ;
}
```

NB : _robot est le Robot passé à la construction de l'afficheur.

La commande est une **Commande**

Pourquoi une commande ?

Nous avons remarqué que les commandes actionnables par el robot avaient la même structure. Chacune appelle une fonction du robot et chacune peut être annulée.

C'est donc logiquement que nous nous sommes tournés vers le schéma Commande, qui en plus d'être un choix logique de par son nom, est un choix logique de par sa structure.

Le schéma matérialise très bien les responsabilités d'une commande à savoir exécuter une action et proposer la possibilité de l'annuler. Nous avons trouvé cette manière de présenter les commandes du robot très pratique et l'avons alors intégrée dans notre projet.

Par ailleurs, la possibilité de stocker les actions faites dans une pile pour faciliter la désexécution nous a encouragés à choisir ce schéma.

Quelles sont les conséquences ?

La première conséquence qui facilitera grandement le travail à l'avenir est la facilité d'ajouter des nouvelles classes de commandes. En utilisant une fois de plus le polymorphisme et l'héritage, nous pouvons sans contraintes ajouter une classe de commande et bien implémenter excecute() et desexecute().

Ensuite, ce schéma permet une forte complétion des principes SOLID, en particulier la Single Responsibility et la Dependency Inversion.

Le premier principe évoqué se matérialise dans le fait qu'une commande soit représentée par une classe. On obtient alors des instances de commandes concrètes dont le seul but est d'exécuter et d'annuler. Nous limitons bien leurs responsabilités.

Le deuxième principe se retrouve dans la limitation du couplage entre les deux classes liées à la commande : L'invocateur et le récepteur (ici le robot). L'invocateur ne fait qu'appeler execute() sur une commande, il ne connait rien du robot, rien de la commandeet la commande ne le modifiera pas.

Le robot quant à lui ne connait pas du tout la commande, c'est même l'inverse (voir partie suivante). La commande est une interface à travers laquelle appeler implicitement (on n'appelle pas le robot depuis l'invocateur), les méthodes du robot.

Choix particuliers et problèmes

Le principal problème de conception que nous avons rencontré fut le lien entre le robot et la commande. Passer le robot en attribut ou en paramètre aurait marché. Néanmoins, le deuxième choix nous semblait moins logique.

En effet si nous passions le robot en paramètre de execute(), il aurait fallu le prendre en compte dans la classe abstraite Commande. Or, les commandes n'auront pas toutes obligatoirement besoin d'un robot. (voir fin de cette partie).

C'est pour cela que nous avons choisi de passer le robot à la création de la commande, qui le prendra en attribut. Cela évite de rompre avec le polymorphisme et nous permet de conserver une certaine généralité.

L'autre choix que nous avons fait, et qui nous paraissait logique et pertinent, est la création d'une classe abstraite intermédiaire entre la classe mère Commande et les commandes concrètes.

Nous nous sommes dit que dans l'éventualité d'un ajout de commandes qui ne travaillent pas sur le robot, nous pourrions créer une classe intermédiaire de commandes uniquement pour le robot. Les commandes concrètes pour le robot héritent alors de cette classe.

On peut toujours facilement ajouter une commande de robot, mais également ajouter un autre type de commande, seulement en étendant la classe mère abstraite Commande.

Le principal défaut de ce schéma et commun à certains vus jusque-là. Il implique en effet la création d'une classe par commande et vient ajouter alors un certain nombre de classes, faisant croître la hiérarchie du projet.

Cela n'est toutefois pas un énorme problème en soi.

Pseudo code

Nous avons choisi de montrer le pseudo code pour les commandes figer et avancer. Dans un cas, les attributs du robot ne sont pas modifiés, dans l'autre si.

```
CommandeFiger::executer()
{
	_robot.figer() ;
}

CommandeAvancer::executer()
{
	_lastx = _robot.getPosition().getx() ;
	_lasty = _robot.getPosition().gety() ;
	_robot.avancer(_x, _y);
}
```

Pour l'annulation, nous avons choisi un cas simple (repartir et figer, pour montrer leur complémentarité), et un cas qui utilise la sauvegarde de précédents attributs.

```
CommandeRepartir ::annuler()
{
	_robot.figer() ;
}

CommandeFiger ::annuler()
{
	_robot.repartir() ;
}

CommandeAvancer ::annuler()
{
	_robot.avancer(_lastx, _lasty) ;
}
```

Nous trouvions également pertinent de faire apparaître l'exécution de la commande Annuler.

```
CommandeAnnuler ::executer()
{
	if(pileActions.size() > 0)
{
	pilesActions.top().annuler();
	pilesActions.pop() ;
}
}
```

La commande utilise un **Constructeur virtuel**

Pourquoi un constructeur virtuel ?

La manière dont les commandes sont effectuées nous a poussé à utiliser un constructeur virtuel. En effet, l'utilisateur va saisir une commande et l'invocateur créera la commande correspondante. Sans schéma de conception, nous serions obligés de traiter la commande avec des if successifs. L'ajout d'une nouvelle classe de commande nous forcerait alors à ajouter un autre cas dans les if.

Cela est contraignant et peu orienté objet.

Le constructeur virtuel, grâce à sa map qui allie une commande à une entrée utilisateur (sous forme de string) et sa méthode constructeurVirtuel(), nous permet d'éviter de passer par des if et de créer à la volée une instance de la commande correspondante à ce que l'utilisateur vient d'entrer. C'est justement pour cela que nous l'avons choisi.

Quelles sont les conséquences ?

Les conséquences principales qui ressortent sont une grande simplification de la construction et de l'ajout.

Au niveau de la construction de commandes, il n'y a qu'à appeler la fonction statique nouvelleCommande(), qui grâce à l'association entre une string et une commande faite dans la map (commandesInscrites) renverra une instance correspondante à la commande tapée.

Pour ce qui est de l'ajout d'un nouveau type de commande, il suffit juste de rajouter une instance de cette commande lors de l'initialisation de la map, et l'associer à une commande. Cela revient presque à un ajout de cas dans des if successifs. La seule différence est le changement de responsabilité. Avec des if, le traitement ne s'effectuerait pas dans la commande. Or ici, l'ajout d'une nouvelle classe à l'instanciation est fait dans la commande. Nous limitons donc les responsabilités relatives aux commandes en dehors des classes commandes.

Ainsi, la structure de Commande est réutilisable, extensible, et les changements d'environnement autour influent finalement peu sur le schéma.

Choix particuliers et problèmes

Tout d'abord, nous devions trouver une solution pour remplir notre map de commande pour finaliser le pattern Constructeur Virtuel. Nous avons suivi le principe que tout stockage relatif à un objet doit se faire si possible au niveau de l'objet lui-même. Ainsi, chaque classe concrète de commande dispose d'une instance static créée dès le départ du programme, et qui grâce au constructeur de Commande, place l'instance dans la map avec le mot-clé correspondant. Cela nous permet alors de remplir notre map dès le début et en laissant ce traitement dans les commandes, plutôt que de faire une instanciation groupée (toutes les instanciations au même endroit dans le même fichier).

Nous avons ensuite choisi de créer une pile de commandes. Cette pile nous permet de stocker l'action effectuée, et nous pourrons revenir sur cette action dans le cas de l'annulation de la commande suivante. La pile permet un certain historique et permet un fonctionnement du mécanisme d'annulation assez intuitif :

1) On effectue une action 1, on la stocke dans la pile.
2) On effectue une action 2, on la stocke dans la pile.
3) On annule l'action 2, c'est la fonction annuler() de action 2 qui sera appelée (chaque commande a son propre annuler() en fonction des attributs qu'elle manipule).

Enfin, un choix qui était moins évident en étudiant le pattern, c'est celui de faire une commande pour l'annulation, autrement dit, une classe CommandeAnnuler. Nous avons en effet trouvé cela logique car l'annulation est une action comme une autre. La seule différence est que c'est cette action qui va appeler annuler() sur la commande au sommet de la pile, puis la retirer de la pile. Tout cela se fait dans sa fonction executer(). Outre la possibilité de bien isoler la responsabilité de modification de la pile en cas d'annulation, cette manière de procéder nous permet de limiter le code concernant le 'pop' de la pile à une seule classe. Nous gagnons donc en respect du Single Responsibility Principle, et nous économisons des lignes de code.