Arnaud GARNIER
Lisa JOANNO

POLYTECH NICE-SOPHIA

SI4 - 2015/2016 - G3

Analyse & Modélisation orientée objet

Modélisation objet d'un simulateur de robot

SOMMAIRE

1. Introduction
2. Étape 1 : Modélisation du comportement du robot
3. Étape 2 : Création des états
4. Étape 3 : Externalisation de la trace
5. Conclusion

1. Introduction

Le projet de modélisation du robot fait intervenir trois différents schémas de conception : schéma état, schéma singleton et schéma observateur. Nous allons pour chaque étape nous focaliser sur une partie de la modélisation du robot, ainsi que le schéma de conception qu'elle fait intervenir.

2. Étape 1 : Modélisation du comportement du robot

Dans cette étape, on met en oeuvre le **schéma état**.

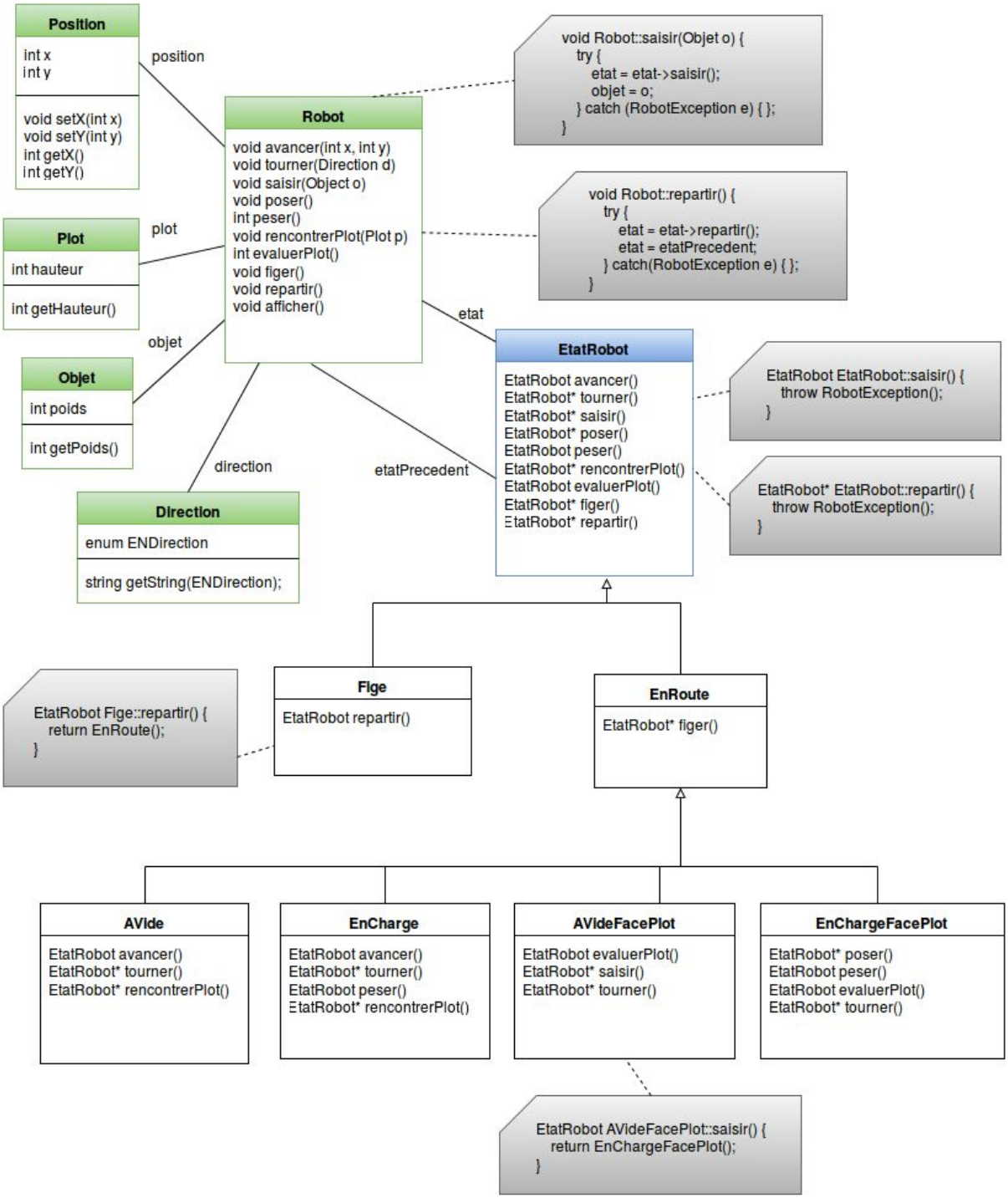

Figure 1 : Diagramme de classe du comportement du robot

Le pseudo-code de chaque méthode se ressemblant, on ne fait apparaître ici que celui des méthodes saisir() et repartir().

On utilise le schéma état car le comportement du robot est modélisé par des états (en route, figé, à vide…). Le diagramme d'états transitions du robot qui nous est

fourni fait apparaître tous ces états, avec les changements d'état en fonction des méthodes appelées sur le robot. La classe robot, en plus de ses caractéristiques (telles que sa position, l'objet qu'il tient, le plot se trouvant devant lui si il existe), possède une variable d'instance EtatRobot. C'est l'état courant du robot. Cette classe va ensuite être dérivée en tous les états qui apparaissent sur le diagramme d'états transitions. EtatRobot étant "l'interface" de tous ces états, elle contient toutes les méthodes du robot. Cependant, pour qu'une méthode ne puisse pas être appelée dans le mauvais état, toutes ces méthodes lèvent une exception (RobotException). Dans les sous classes d'EtatRobot (où une méthode peut être appelée et avoir un effet), on implémente les méthodes utiles et elle renvoie l'état dans lequel se trouve le robot après exécution (par exemple, l'appel à rencontrerPlot() dans l'état EnChargeFacePlot est impossible, mais dans l'état AVide, il est possible et le robot se trouve ensuite dans l'état AVideFacePlot). Ainsi, dans Robot, pour chaque méthode, on essaie d'appeler la méthode sur l'état, et si cela fonctionne, on applique les effets de la méthodes. Si l'état ne le permet pas, l'exception est attrapée et les effets de la méthode ne sont pas appliqués.
Pour ajouter un état au robot, il suffit de dériver une nouvelle fois la classe EtatRobot.
Durant l'implémentation de ce schéma, nous avons rencontré des problèmes multiples. Le premier était les inclusions multiples, résolus grâce aux directives au préprocesseur (#ifndef, #define…). Le second était le choix des pointeurs sur EtatRobot. Nous avions, en premier lieu, tenté de se passer de pointeurs, pour finalement déclarer un EtatRobot* au lieu d'un EtatRobot dans Robot. Nous avons donc dû opérer aux changements adéquats dans le renvoi des méthodes.

3. Étape 2 : Création des états

Dans cette étape, on mettra en oeuvre le **schéma singleton**.

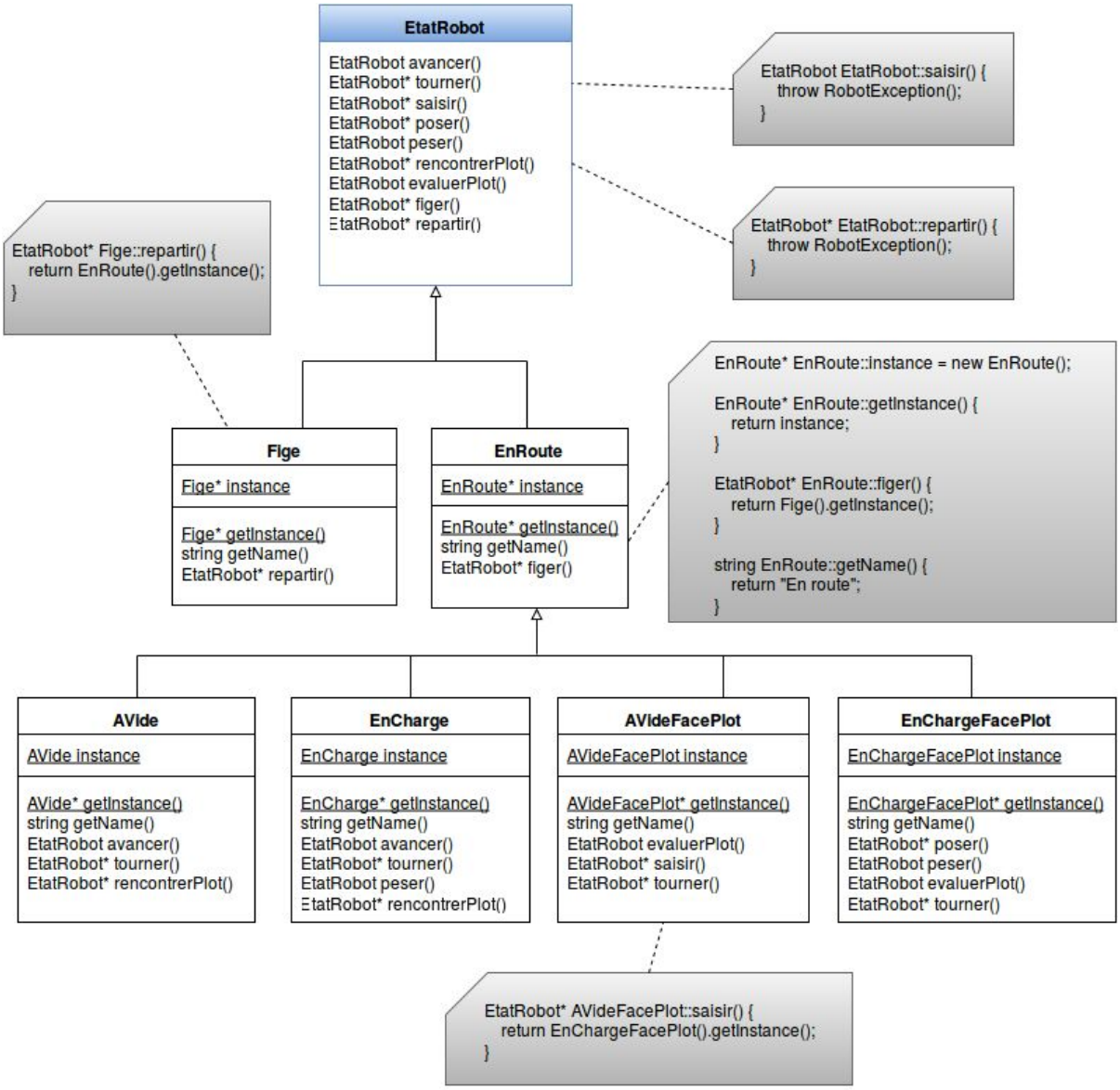

Figure 2 : Diagramme de classe des états du robot

On utilise le schéma singleton car les états n'ont pas besoin d'être instanciés plusieurs fois et on veut contrôler l'accès à une instance unique. On crée dans chaque classe une instance statique de la classe. Par la suite, quand on veut changer d'état, on appelle l'instance de cette classe grâce à getInstance(). Ceci permet de ne pas créer un nouvel état à chaque changement d'état. On s'assure ainsi qu'il n'existe qu'une et une seule instance dans l'espace et dans le temps durant toute la simulation du robot.

Durant l'implémentation de ce schéma, nous avons rencontré des problèmes de reconnaissance de la variable statique instance.

4. Étape 3 : Externalisation de la trace

Dans cette étape, on mettra en oeuvre le **schéma observateur**.

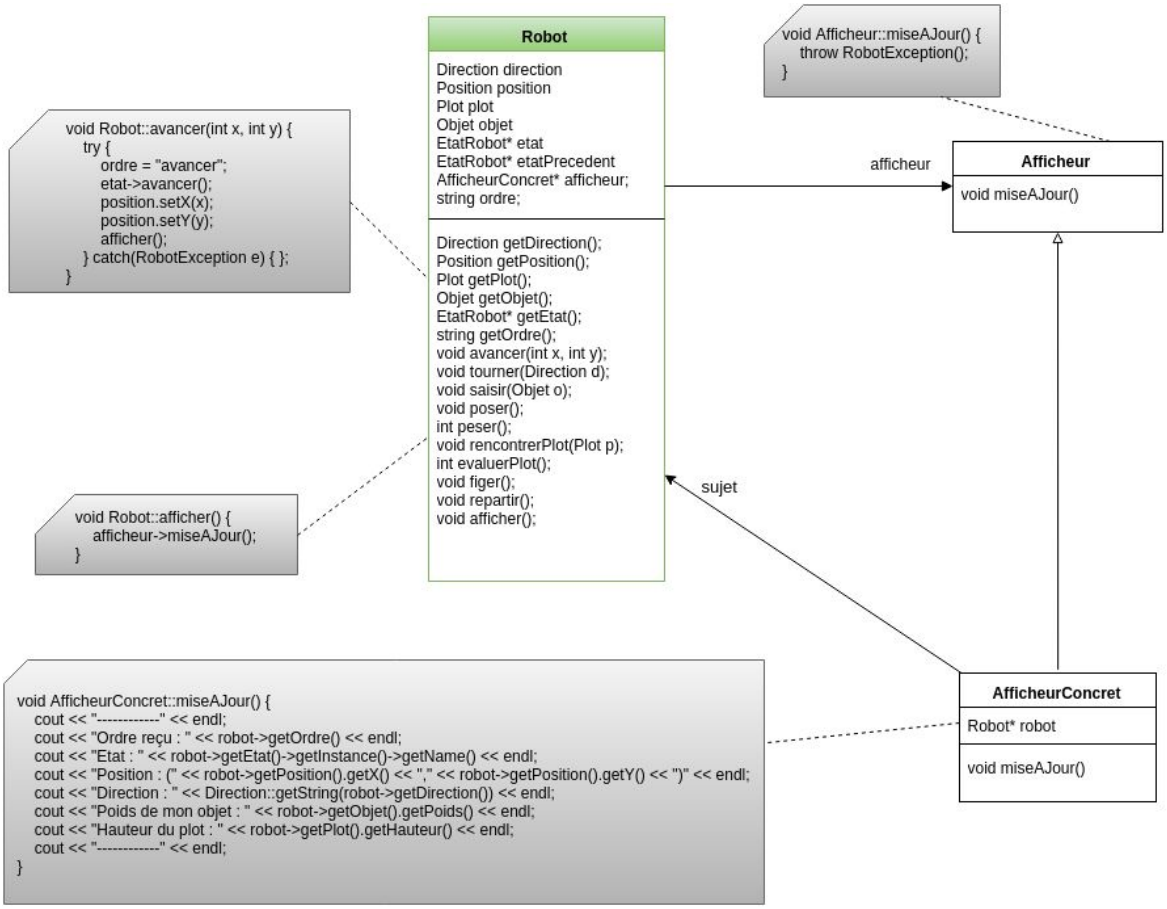

Figure 3 : Diagramme de classe de l'externalisation de la trace du robot

On utilise le schéma observateur parce que l'on souhaite, à chaque changement d'état, que l'affichage se fasse automatiquement. On crée une interface Afficheur et une classe AfficheurConcret qui en hérite. Ici, on a qu'un seul afficheur, mais si on en voulait un second (pour une interface graphique, par exemple), il suffirait de créer AfficheurInterfaceGraphique et de la faire hériter d'Afficheur.

Le fonctionnement est le suivant : dès que le robot changera d'état, la fonction afficher() de Robot sera appelée. Celle-ci notifie les afficheurs en les mettant à jour. Si on avait eu plusieurs afficheurs, afficher() aurait notifié tous les afficheurs à l'aide d'une boucle for. La classe AfficheurConcret n'a plus qu'à afficher (ici sur la sortie standard) les différentes caractéristiques du robot : l'ordre donné, son état, sa direction, sa position, s'il tient ou non un objet avec l'identification par son poids, s'il est ou non face à un plot avec l'identification du plot par sa hauteur.
Durant l'implémentation de ce schéma, nous avons également rencontré des problèmes d'inclusions multiples, et surtout d'inclusions mutuelles. En effet, Robot inclut AfficheurConcret, qui inclut Robot. Ce problème a été résolu en pseudo-déclarant la classe `class Robot;` dans AfficheurConcret, et `class AfficheurConcret;` dans Robot. Nous avons également dû choisir entre le fait de placer Robot en paramètre de la méthode miseAJour, ou bien placer Robot en variable d'instance de AfficheurConcret. Nous avons finalement choisi cette dernière solution.

5. Conclusion

Nous avons vu que chaque schéma de conception joue un rôle très précis dans la modélisation du comportement du robot. Ces trois schémas s'adaptent très bien à l'exemple car :

- ➔ le robot passe d'un état à un autre
- ➔ ces états sont des singletons
- ➔ chaque changement d'état entraîne une mise à jour de l'affichage.