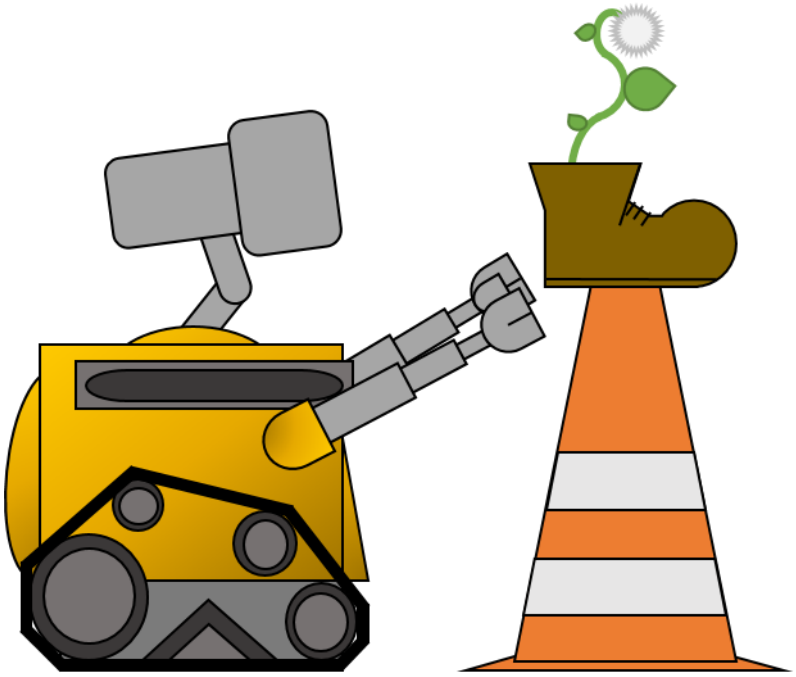
POLYTECH
NICE-SOPHIA

Rapport de

Conception Orienté Objet



AIELLO Axel
PEPIN Nicolas

SI4 25.11.16
G2

Sommaire

Introduction

Dans le cadre du cours de « Conception orientée objet », nous avons comme objectif de modéliser un robot avec pour support les notions vues en cours.
Nous devons réaliser un schéma de conception avec des diagrammes de classes et des explications sur nos choix de conception. Un pseudocode sera fourni sur les fonctions demandées. Notre conception doit répondre au diagramme d'état suivant :

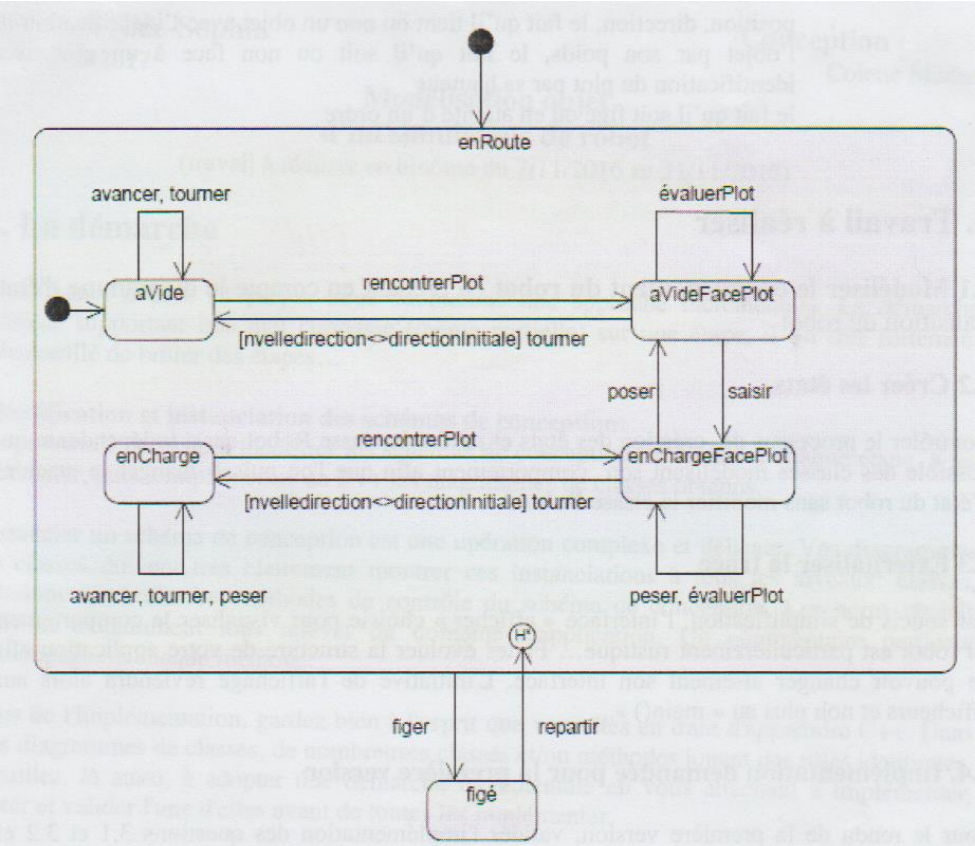


Figure 2 : Diagramme d'états transitions du robot

Pour cela nous allons vous présenter dans un premier temps une version simple contenant de l'héritage pour les différents états du robot, une seconde version avec une optimisation sur les instances des états et l'utilisation du patron de conception « Singleton » et enfin une dernière partie où on intègre le schéma Observateur pour les afficheurs de toutes les informations du robot.

1. Modélisation du comportement du robot

a. Diagramme de classes

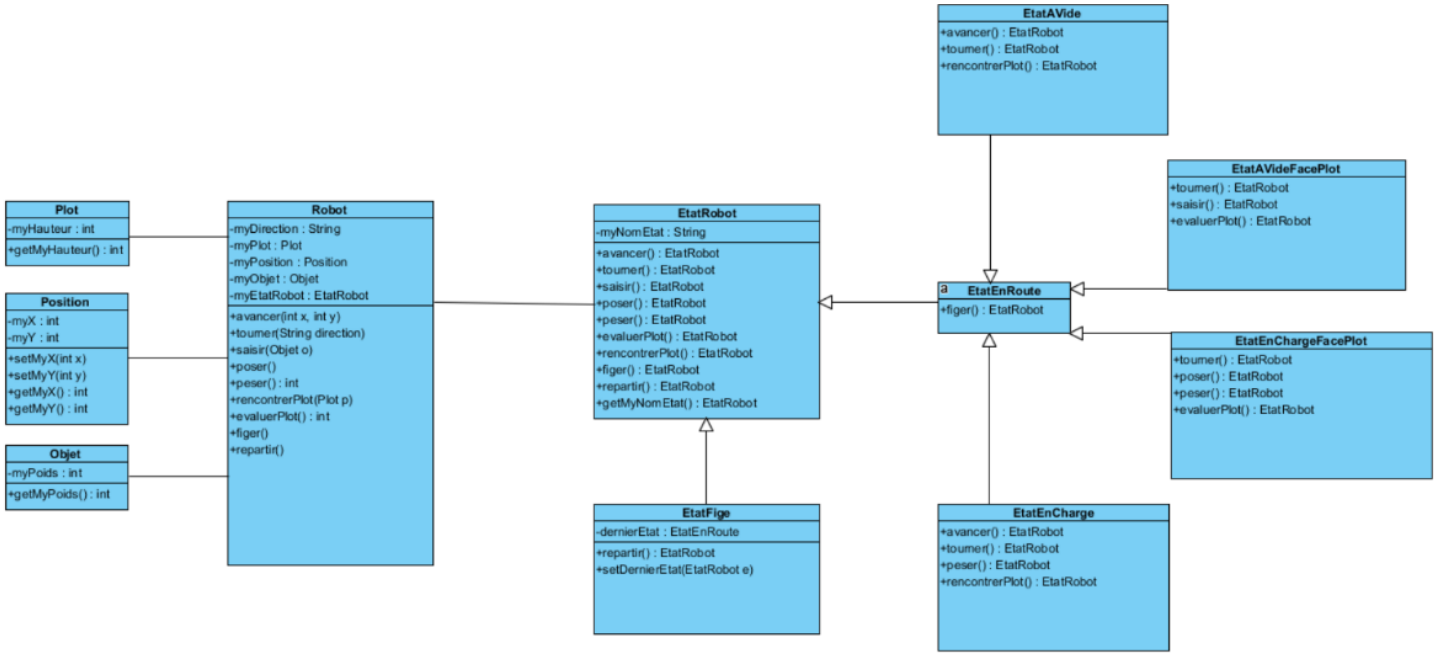


Figure 1 : diagramme de classes pour l'étape 1

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 1, avec la fonction saisir() et repartir().

A. Saisir

Robot	saisir(Objet o) {
	si (il n'y a pas d'exception) {
	myEtatRobot = myEtatRobot.saisir();
	} sinon si (il y a une exception) {
	on signale l'exception ;
	fin ;
	}
	myObjet = o ;
	fin ;
	}

EtatRobot	EtatRobot saisir() {
	on signale une exception ;
	}

EtatAVideFacePlot	EtatRobot saisir() {
	on return un nouveau objet EtatEnChargeFacePlot ;
	}

B. Repartir

Robot	repartir() {
	si (il n'y a pas d'exception) {
	myEtatRobot = myEtatRobot.repartir();
	} sinon si (il y a une exception) {
	on signale l'exception ;
	fin ;
	}
	fin ;
	}

EtatRobot	EtatRobot repartir() {
	on signale une exception ;
	}

EtatFige	EtatRobot repartir() {
	on return le dernier Etat ;
	}

c. Explication

Le diagramme de classe pour l'étape 1 représente principalement la modélisation des différents états possibles du robot au cours du programme.
Dans un premier temps le robot pouvant être « figé » ou « en route », nous retrouvons les deux classes qui représentent les deux états « principaux ».
Cette implémentation des différents états principaux nous permet de faciliter le changement entre ces derniers.

Puis nous retrouvons les classes filles de EtatEnRoute qui correspondent aux différents états « secondaires » dans lesquels le robot peut être lorsque ce dernier n'est pas figé.

Cette modélisation offre la possibilité que le robot ait une réaction différente avec des actions propres à l'état dans lequel il se trouve, tout en étant dépendant aux actions qui seront réalisées sur le robot.

Notre implémentation permet de réaliser une action dans la classe Robot ; pour cela, Robot appelle EtatRobot qui va voir, en fonction de l'état dans lequel se trouve le robot, si l'action demandée peut être réalisée ou si l'état actuel ne le permet pas. Si l'action ne peut pas être réalisée dans cet état, une exception est levée en informant que le robot ne peut pas réaliser cette action ; en revanche si l'action est possible, elle est réalisée et l'état du robot change en fonction de l'action exécutée.

Nos actions sont réalisées dans Robot afin de ne pas avoir à déplacer une instance complète du robot à chaque demande d'action, même si cela nous aurait permis de modifier l'état du robot directement dans la fonction correspondant à l'état actuel. Nous avons donc opté pour le changement de l'état du robot dans la classe Robot.

2. Création des Etats par Singleton

a. Diagramme de classes

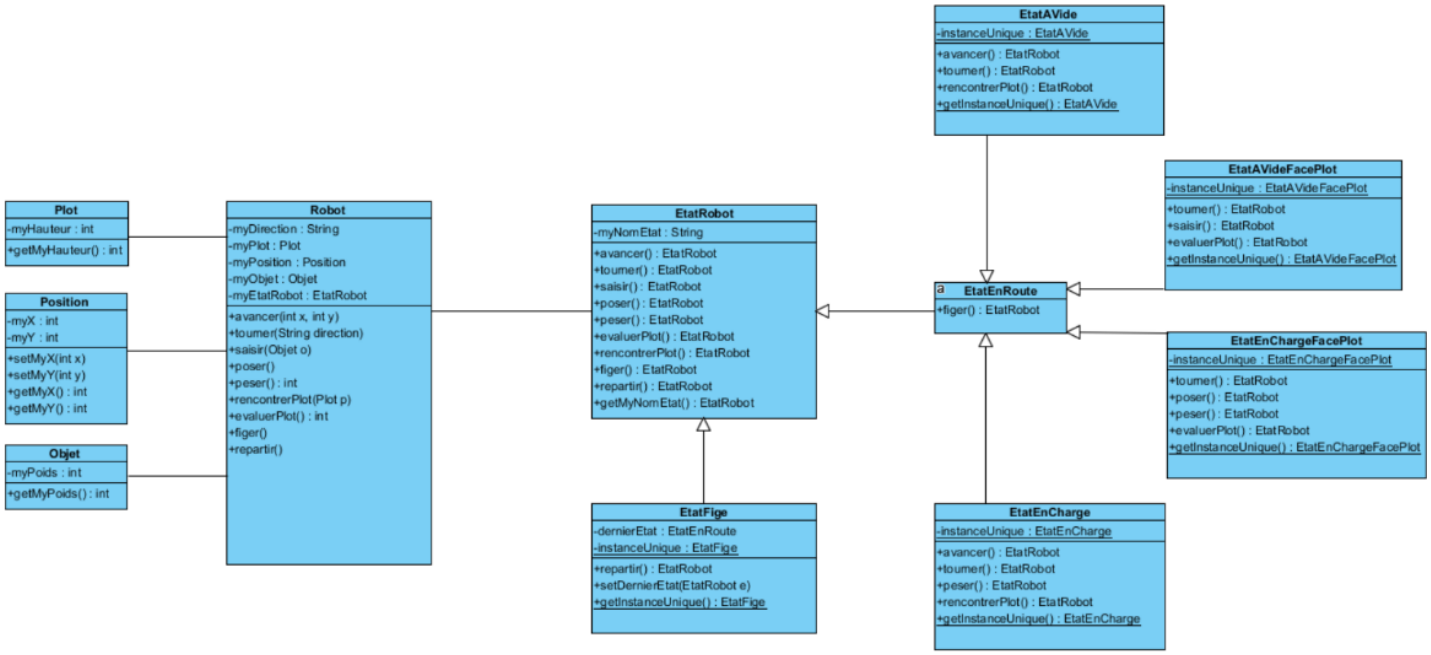


Figure 2 : diagramme de classes pour l'étape 2

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 2, avec la fonction saisir().

A. Saisir

Robot	saisir(Objet o) {
	si (il n'y a pas d'exception) {
	myEtatRobot = myEtatRobot.saisir();
	} sinon si (il y a une exception) {
	on signale l'exception ;
	fin ;
	}
	myObjet = o ;
	fin ;
	}

EtatRobot	EtatRobot saisir() {
	on signale une exception ;
	}

EtatAVideFacePlot	**Création de l'instance de la class EtatAVideFacePlot ;**
	
	EtatRobot saisir() {
	on return l'instance de la class EtatEnChargeFacePlot ;
	}

	EtatAVideFacePlot getInstance() { **on return l'instance de la class EtatAVideFacePlot ;** **}**

c. Explication

Le diagramme présenté ci-dessus présente les mêmes fonctionnalités que le diagramme précédent ; cependant le diagramme a subi une modification avec l'ajout du patern Singleton qui permet d'éviter la création, à chaque changement d'état, d'un nouvel objet de l'état. Cela permet d'avoir une seule instance par état lorsque ce dernier est créé.
Cette optimisation du système est minimale à notre échelle mais, si le robot réalise beaucoup d'actions, elle deviendra indispensable.

Afin d'ajouter cette optimisation, nous avons adapté la construction des états avec l'ajout d'un attribut « instanceUnique » qui permet de garder l'unique instance de l'état de manière statique. Il ne nous reste plus qu'à avoir accès à cette instance via la fonction statique instance(), qui effectuera l'initialisation de l'instance et retournera l'état créé lors de son premier appel, sinon elle retournera l'instance qui avait été créée précédemment.

3. Externalisation de la trace

a. Diagramme de classes

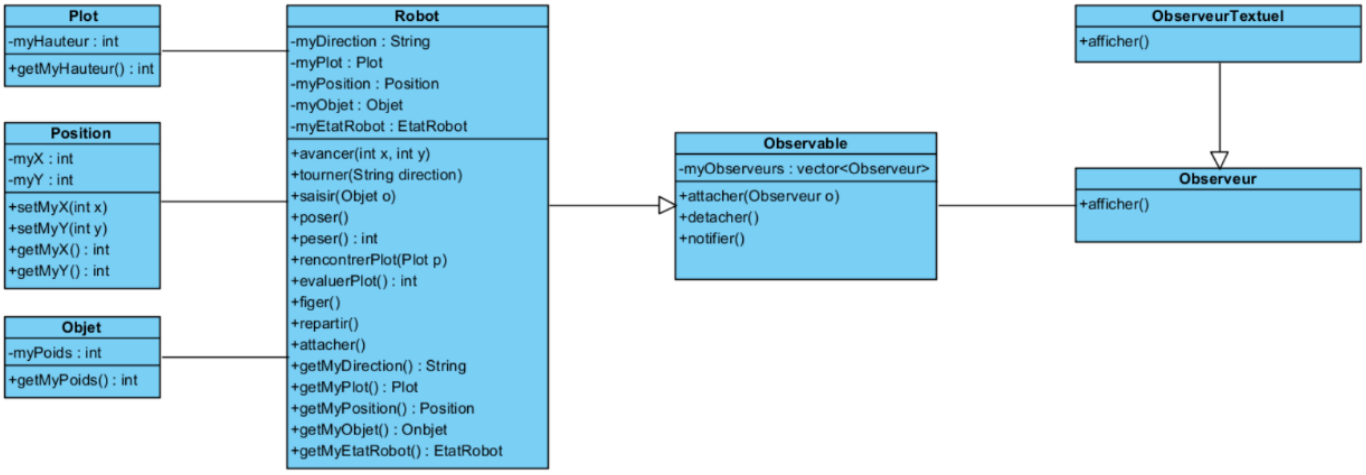


Figure 3 : diagramme de classes pour l'étape 3

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 3, avec la fonction saisir().

A. Saisir

Robot	saisir(Objet o) {
	si (il n'y a pas d'exception) {
	myEtatRobot = myEtatRobot.saisir();
	} sinon si (il y a une exception) {
	on signale l'exception ;
	fin ;
	}
	myObjet = o ;
	notifier() ;
	fin ;
	}
	
	notifier() {
	pour i parcourant la liste des myObserveurs {
	myObserveurs(i).afficher();
	}
	}

EtatRobot	EtatRobot saisir() {
	on signale une exception ;
	}

EtatAVideFacePlot	EtatRobot saisir() {
	on return l'instance de la class EtatEnChargeFacePlot ;
	}

Observable	**notifier() {** **}**

Observeur	afficher() {
	Afficheur de base, n'affiche rien ;
	}

Observeur Textuel	afficher() {
	Afficheur Textuel, affiche sur la sortie standard;
	}

c. Explication

Ce diagramme de classe présente les afficheurs possibles qui donneraient toutes les informations du robot.
Afin d'externaliser l'affichage du robot pour qu'il n'ait pas un seul afficheur possible, on a implémenté une classe Observable, dont la classe Robot hérite et qui contient la liste des différents afficheurs et une classe Observateur qui aura une méthode afficher() en Virtual afin qu'elle puisse appeler la fonction afficher() de l'afficheur correspondant au type d'affichage que l'on souhaite.
Enfin la classe Observable contient deux fonctions qui sont attacher() et detacher() et qui permettent d'ajouter ou d'enlever des Observeurs.

4. Implémentation pour la 1ère

Voir les fichiers situés dans le dossier "src"

5. Commander le Robot

a. Diagramme de classes

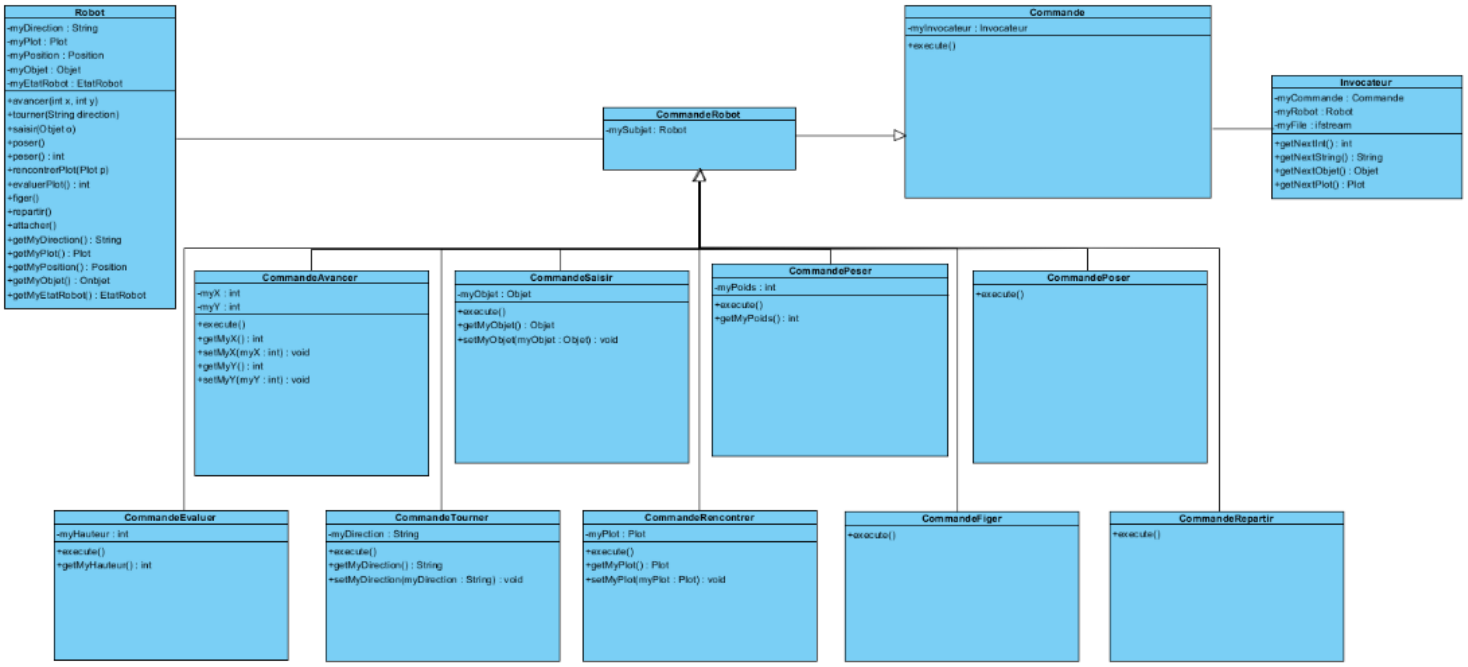


Figure 4 : diagramme de classes pour l'étape 5

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 5, avec la fonction saisir().

A. Saisir

Invocateur	Main(){
	on crée le robot ;
	pour chaque ligne du fichier {
	on crée la commande voulu avec le robot du dessus ;
	on exécute la commande ;
	}
	}
	
	Objet getNextObjet() {
	récupère dans le fichier le prochain int ;
	créé un plot et le renvoie ;
	}

Commande	Abstraite execute(){ }

CommandeSaisir	execute() {
	récupère myObjet par getNextObjet() de myInvocateur
	appel de la fonction saisir() sur mySujet avec l'objet myObjet;
	}

c. Explication

Le diagramme présenté ci-dessus expose les différentes commandes que le robot peut réaliser. L'implémentation du Schéma Commande permet de réaliser une interface de communication évolutive avec le Robot. Nous avons réalisé un parseur qui sert à l'extraction des commandes du fichier de l'utilisateur et à la génération des commandes correspondantes. Ce parseur est lié à la classe Commande qui possède les correspondances entre les strings utilisés pour appeler une commande, ainsi que la classe commande qui correspond.

Une sous classe CommandeRobot regroupera la liste des commandes qui s'appliqueront directement au Robot, ces commandes hériteront de la classe CommandeRobot. Grâce à cela, il serait possible de réaliser par exemple une CommandeVoiture qui pourrait avoir ses propres commandes mais en utilisant quand même Commande.

Enfin, chaque commande possède sa propre fonction execute(), qui corresponds a l'action à effectuer sur le robot en fonction de cette dernière.

Cependant cette version n'est pas optimisée, le point suivant sera plus poussé.

6. Créer les commandes

a. Diagramme de classes

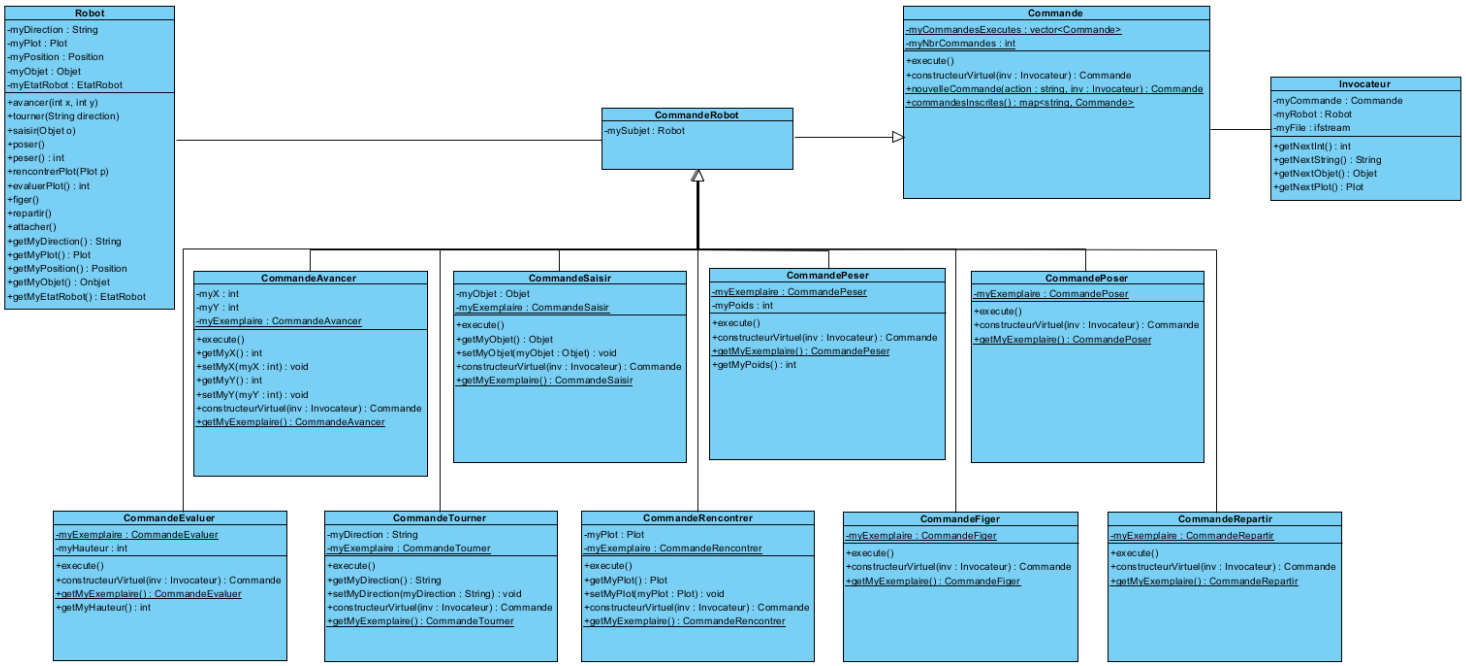


Figure 5 : diagramme de classes pour l'étape 6

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 6, avec la fonction saisir().

A. Saisir

Invocateur	Main(){
	on crée le robot ;
	on attribue à myCommande un objet Commande
	pour chaque ligne du fichier {
	myCommande = myCommande.nouvelleCommande(nom, this) ;
	on exécute la commande ;
	}
	}
	
	Objet getNextObjet() {
	récupère dans le fichier le prochain int ;
	créé un plot et le renvoie ;
	}

Commande	Abstraite execute(){
	}
	
	Commande nouvelleCommmande(string action, Invocateur myInvocateur){
	renvoie commandesInscrites()[action]->constructeurVirtuel(myInvocateur) ;
	}

	map<string, Commande> commandesInscrites(){ **static map<string, Commande> cmdInscrites = new map<string, Commande> ;** **renvoie cmdInscrites ;** **}**

CommandeSaisir	execute() {
	appel de la fonction saisir() sur mySujet avec l'objet myObjet;
	}
	
	Commande constructeurVirtuel(Invocateur myInvocateur){
	renvoie new CommandeSaisir(myInvocateur) ;
	}
	
	CommandeSaisir(Invocateur myInvocateur) {
	récupère myObjet par getNextObjet() de myInvocateur
	}

c. Explication

Nous avons dû intégrer le schéma Constructeur Virtuel, qui permet l'instanciation de nouvelles instances de la classe de la commande appelée au fur et à mesure de leurs appels lors de la durée de vie du Robot.

Nous avons mis en static afin qu'il n'y ait qu'une seule instance des états qui soit faite et non pas plusieurs ce qui permet de ne pas trop consommer en terme de mémoire.

7. Défaire

a. Diagramme de classes

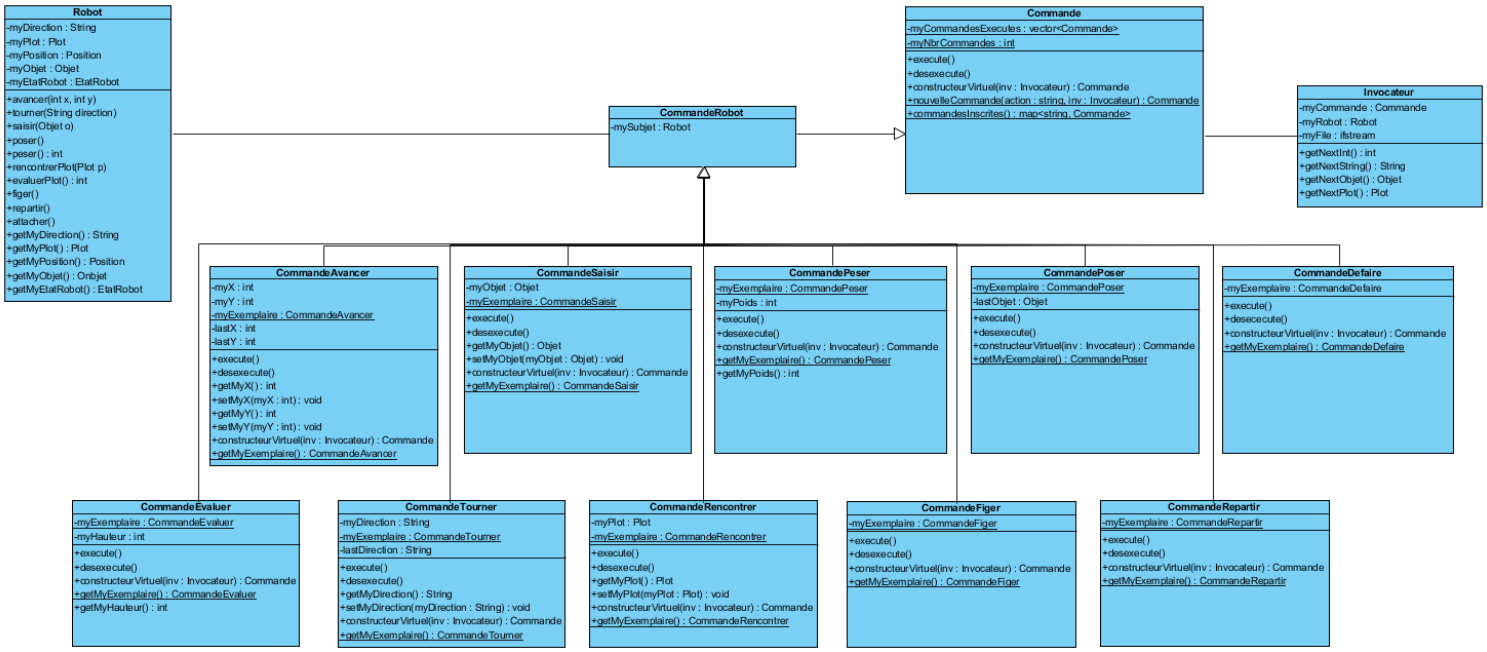


Figure 6 : diagramme de classes pour l'étape 7

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 7, avec la fonction saisir().

A. Saisir

Invocateur	Main(){
	on crée le robot ;
	on attribue à myCommande un objet Commande
	pour chaque ligne du fichier {
	myCommande = myCommande.nouvelleCommande(nom, this) ;
	on exécute la commande ;
	}
	}
	
	Objet getNextObjet() {
	récupère dans le fichier le prochain int ;
	créé un plot et le renvoie ;
	}

Commande	Abstraite execute(){} **Abstraite desexecute(){}**

	Commande nouvelleCommmande(string action, Invocateur myInvocateur){
	renvoie commandesInscrites()[action]->constructeurVirtuel(myInvocateur) ;
	}
	
	map<string, Commande> commandesInscrites(){
	static map<string, Commande> cmdInscrites = new map<string, Commande> ;
	renvoie cmdInscrites ;
	}

CommandeSaisir	execute() {
	appel de la fonction saisir() sur l'objet mySujet;
	}
	
	desexecute () {
	décrémentation de myNbrCommandes ;
	on enlève la dernière commande (saisir) de myCommandesExecutes ;
	appel de la fonction poser() sur l'objet mySujet ;
	}
	
	Commande constructeurVirtuel(Invocateur myInvocateur){
	Commande cmd = new CommandeSaisir(myInvocateur) ;
	si (cmd est reversible) {
	ajout de la commande cmd dans myCommandesExecutes ;
	incrémentation de myNbrCommandes ;
	}
	renvoie cmd ;
	}
	
	CommandeSaisir(Invocateur myInvocateur) {
	récupère myObjet par getNextObjet() de myInvocateur
	}

B. Défaire

CommandeDefaire	execute() {
	on récupère la dernière commande (n° myNbrCommandes) de myCommandesExecutes ;
	on appel la fonction desexecute() sur cette commande ;
	}
	
	desexecute () {
	lance une exception ;
	}
	
	Commande constructeurVirtuel(Invocateur myInvocateur){
	renvoie new CommandeDefaire(myInvocateur) ;
	}
	
	CommandeDefaire(Invocateur myInvocateur) {
	}

c. Explication

La classe Commande possède une pile qui correspond aux commandes que l'on peut annuler si on le souhaite avec la commande DEFAIRE.

Nous avons pris l'initiative que la pile serait composée de commandes et serait dans la classe Commande afin de ne pas avoir à chaque fois une nouvelle instance du robot dans la pile mais simplement des commandes pour réaliser le retour de ces dernières. Pour faire correspondre le fait que l'on puisse avoir différentes commandes dans la pile qui puissent être annulées (myComandeExecutes).
Enfin, chaque commande possède son propre desexecute() qui correspond à l'annulation de la commande effectuée (le desexecute d'un avancer(x,y) est par exemple un avancer(lastX, lastY)).

Lorsque la commande « DEFAIRE » est appelée, cette dernière appelle à son tour la fonction desexecute() de la commande qui se trouve en sommet de pile, après avoir annulé la commande, on supprime la commande du sommet de pile.
Nous avons choisi de n'ajouter que les commandes réversibles à la pile afin de ne pas avoir de « DEFAIRE » qui supprimerait juste la commande.

8. Macro-commandes

a. Diagramme de classes

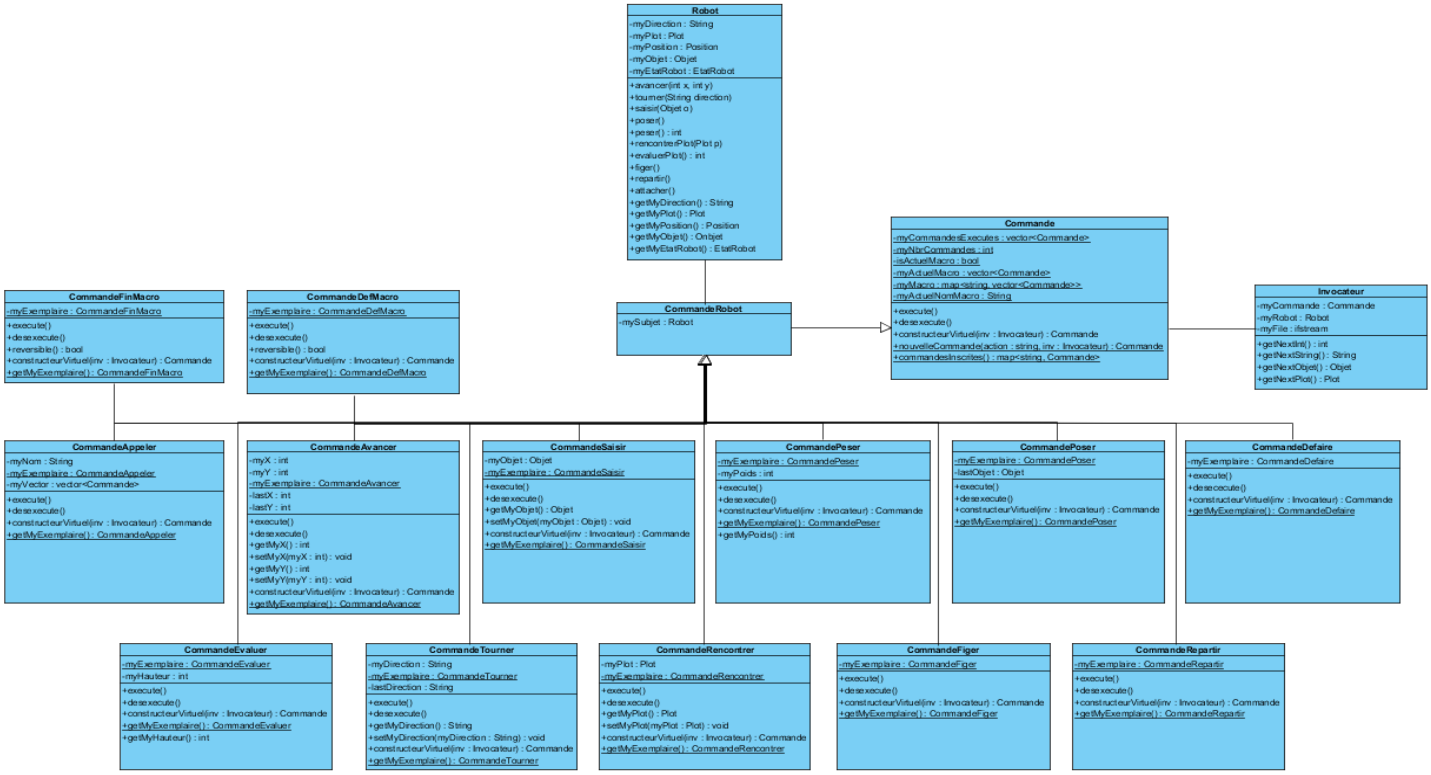


Figure 7 : diagramme de classes pour l'étape 8

b. Pseudocode

Voici des exemples de pseudocodes pour l'étape 8, avec la fonction saisir().

A. Macro

Invocateur	Main(){
	  on crée le robot ;
	  on attribue à myCommande un objet Commande
	  pour chaque ligne du fichier {
	  myCommande = myCommande.nouvelleCommande(nom, this) ;
	  }
	}
	
	Objet getNextObjet() {
	  récupère dans le fichier le prochain int ;
	  créé un plot et le renvoie ;
	}

Commande	Abstraite execute(){} Abstraite desexecute(){}

	Commande nouvelleCommmande(string action, Invocateur myInvocateur){
	renvoie = commandesInscrites()[action]->constructeurVirtuel(myInvocateur) ;
	}
	
	map<string, Commande> commandesInscrites(){
	static map<string, Commande> cmdInscrites = new map<string, Commande> ;
	renvoie cmdInscrites ;
	}

CommandeDefMacro	**execute() {** **isActuelMacro devient vrai ;** **récupère myActuelNomMacro par getNextStringt() de myInvocateur** **initialisation du vecteur myActuelMacro ;** **}** **desexecute () {** **lance une exception** **}**

CommandeFinMacro	**execute() {**
	met dans myMacro, myActuelMacro avec comme nom myActuelNomMacro
	efface le vecteur myActuelMacro ;
	}
	
	desexecute () {
	lance une exception
	}
	
	CommandeFinMacro(Invocateur myInvocateur) {
	isActuelMacro devient false ;
	}

CommandeAppeler	**execute() {** **on récupère myNom par getNextStringt() de myInvocateur ;** **on récupère myVector par myMacro avec myNom ;** **pour chaque Commande cmd de myVecotr {** **on execute cmd ;** **}** **}** **desexecute () {** **pour chaque Commande cmd de myVecotr parcourut dans le sens inverse {** **on desexecute cmd ;** **}** **}**

CommandeSaisir	execute() {
	si (isActuelMacro est vrai) {
	appel de la fonction saisir() sur l'objet mySujet;
	}
	}
	
	desexecute () {
	décrémentation de myNbrCommandes ;
	on enlève la dernière commande (saisir) de myCommandesExecutes ;
	appel de la fonction poser() sur l'objet mySujet ;
	}

	Commande constructeurVirtuel(Invocateur myInvocateur){
	Commande cmd = new CommandeSaisir(myInvocateur) ;
	si (isActuelMacro est faux**) {**
	ajout de la commande cmd dans myCommandesExecutes ;
	incrémentation de myNbrCommandes ;
	} sinon {
	on met la commande dans myActuelMacro
	}
	
	}
	
	CommandeSaisir(Invocateur myInvocateur) {
	récupère myObjet par getNextObjet() de myInvocateur
	}

c. Explication

Cette dernière partie qui est Macro-Commande, permet la réalisation d'une suite d'exécutions de commandes ; afin de réaliser cela, il a fallu ajouter à CommandeRobot 3 commandes supplémentaires, dont 2 qui interagissent. La première est CommandeDefMacro : elle permet la création d'une macro et l'enregistrement de la suite de commandes à effectuer ; cette macro se terminera lors de la lecture du FINMACRO (la commande CommandeFinMacro). Ces deux commandes fonctionnent ensemble afin de réaliser la macro que l'on souhaite.

La troisième commande est APPELER (la commande CommandeAppelerMacro) ; cette dernière permet l'exécution de la macro que l'on a enregistrée en fonction de son nom et donc elle permet d'exécuter la suite de commande que l'on a enregistrée grâce aux commandes précédentes.

Dans la classe Commande, on enregistre la liste des macros commandes dans la map correspondante (myMacro), cela permettra d'exécuter la bonne macro lors de l'appel en fonction du nom.

Précisions

Il faut préciser que nous avons dû faire des choix par manque d'information, ainsi :

- Lorsqu'on fait AVANCER 1 1, on fait x = 1 et y = 1 et non x = x + 1 et y = y + 1
- On ne peut pas désexécuter les commandes DEFMACRO, FINMACRO et APPELER.

Conclusion

Afin de réaliser ce robot nous nous sommes reposés sur les éléments du cours avec comme objectif de fournir la conception la plus optimale possible avec de l'héritage au niveau des états par exemple, mais aussi de l'optimisation avec l'utilisation du pattern Singleton pour éviter l'instanciation répétitive d'un état, ou encore pour la portabilité, si l'on souhaite ajouter un afficheur ou un état supplémentaire pour le robot, le travail à fournir sera plus simple et plus rapide. Enfin la partie des commandes est importante dans sa structure via le schéma Commande mais aussi le schéma Constructeur Virtuel , qui permet l'instanciation de nouvelles instances de la classe de la commande appelée au fur et à mesure de leurs appels lors de la durée de vie du Robot.