Kraczek Sébastien
Benhaddou Inès
Le Trung Hieu
Applincourt Joris

Compte rendu de projet de conception orienté objet

Robot

SOMMAIRE

PRÉSENTATION DU PROJET

Le projet qui nous a été proposé consiste en la création d'un robot pouvant se déplacer dans un environnement 2D. Notre robot doit être capable non seulement de se déplacer (avancer, reculer, tourner) mais aussi aussi de détecter des obstacles et de laisser une trace derrière lui lorsqu'il se déplace et l'effacer.
La seconde partie de se projet consiste à rendre le robot programmable. En effet l'utilisateur doit pouvoir charger ses commandes dans le robot, soit en les chargeant à partir d'un fichier, soit en les rentrant directement à partir de l'interface graphique.

La structure de notre projet est représentée dans le diagramme de classe mis en annexe.

PRÉSENTATION DU CODE

La classe Point

Elle prend en paramètre deux doubles. Cette classe crée des points de coordonnée (x,y). Elle permet aussi de convertir les données soit en cartésien soit en polaire et d'obtenir la distance entre deux points.
Nous avons effectué la surcharge des opérateurs +,= et / pour faciliter les calculs que nous devions effectuer par la suite. De plus, afin de pouvoir afficher les résultats, nous avons fait la surcharge de l'opérateur créneau créneau.

La classe Hitbox :

Cette classe permet de créer la hit box des obstacles et du robot pour savoir si ceux-ci rentrent en collision.
Nous avons défini les hit box de telle sorte qu'elles soient rectangulaires. Ainsi, elles sont créées à partir de quatre points. Elles sont définies par le point central, la hauteur, la largeur et la direction.

La méthode bool collision(Hitbox h) permet de savoir si deux hit box rentrent en collision. Afin de simplifier le calcul de collision, on transforme nos rectangle OBB (Oriented Bounded Box) en un rectangle AABB (Axis Aligned Bounded Box) l'incluant.

La classe Item :

C'est une interface pour pouvoir facilement rajouter d'autres types d'objet. L'intérêt de l'interface est qu'elle va jouer le rôle de découpleur, c'est à dire que le reste du code ne voit que l'interface Item qui va resté inchangé quelque soit la réelle implémentation des objets derrière.

Pour tous les items nous avons des getter pour obtenir : la largeur, la hauteur, la direction, la hitbox, les coordonnées (point central de l'objet) et le mode de dessin.

Nous avons aussi des setter afin de modifier : la largeur, la hauteur et le mode de dessin.

Nous incluons aussi les méthodes virtuelles pures :

virtual void avancer(double dist)=0;

virtual void reculer(double dist)=0;

virtual void tourner(double angle)=0;

virtual void poser(double x_r, double y_a)=0;

Nous implémenterons ces méthodes dans la classe Robot et la classe Obstacle.

De plus, nous incluons la surcharge de l'opérateur < < afin de pouvoir afficher les items.

La classe Robot :

Cette classe utilise le patron de conception singleton car nous allons l'implémenter qu'une seule fois comme nous n'avons besoin que d'un robot. Ce patron nous assure de tenir cette spécification.

Cette classe dérive de la classe Item car le robot est un item particulier.

Le robot possède une largeur, une hauteur, un point central, une direction, une hitbox ainsi qu'un mode de dessin.

Par défaut le robot est en (0,0) avec une direction 0 et un mode de dessin « pinceau levé », c'est à dire qu'il ne laisse pas de trace.

Dans la classe Robot nous implémentons toute les méthodes de l'interface Item :

- La méthode avancer : void avancer(double dist)

  Le but de cette méthode est de faire avancer le robot. Cette méthode permet de créer un point delta en coordonnée polaire, ce point correspond à une certaine distance et direction dans laquelle on veut le déplacer. On convertit ce point delta en coordonnée cartésienne pour en faire un vecteur sur lequel on va déplacer les coordonnées du robot et enfin nous mettons à jour la hitbox.

- La méthode reculer : void reculer(double dist)

  Cette méthode a pour but de faire reculer le robot. Elle est similaire à la méthode avancer sauf qu'ici on ajoute 180° à la direction afin de reculer. Puis nous mettons à jour la hitbox.

- La méthode tourner : void tourner(double angle)

  Cette méthode permet de faire tourner le robot en ajoutant l'angle en degré. Nous mettons ensuite à jour la hitbox.

- La méthode poser : void poser(double x_r, double y_a)

  Cette méthode permet de poser le robot à un certain point dans le plan juste en précisant les coordonnées de ce point.

Nous utilisons à la fois des coordonnées cartésiennes et polaires car il est plus simple de se déplacer en utilisant les données polaires (direction et distance) mais nous utilisons un plan cartésien. C'est pourquoi nous effectuons des conversions entre les deux repères.

<u>La classe Obstacle :</u>

Cette classe dérive de la classe Item car les obstacles sont des items particuliers. Cette classe est similaire à la classe Robot sauf qu'elle possède des fonctionnalités en moins. En effet contrairement à la classe Robot nous ne pouvons pas modifier la taille, la largeur, le mode de dessin, ni faire bouger l'obstacle.

<u>La classe Fabrique à Item :</u>

Cette classe utilise le patron de construction fabrique. Afin de gérer sa création nous utilisons le patron de conception singleton.

L'intérêt d'utiliser une fabrique est que cela permet de cacher comment sont réellement créés nos objets. Ceci permet d'homogénéiser notre code.

Afin de créer un item, il suffit de choisir le type d'item (en string) ainsi que d'ajouter la hauteur et la largeur (tous deux de type double). En paramètre optionnel, nous pouvons choisir les coordonnées ainsi que la direction de l'item. Les paramètres optionnels sont mis à zéro par défaut.

MVC (Modèle Vue Contrôleur) :

Ceci permet de séparer le modèle et la vue afin que le dispositif soit bien dissocié.

La classe Observateur :

On utilise la patron observateur, la classe est templatisé. Ainsi, on peut mettre n'importe quel type pour associer un modèle à une vue (où le même type est observé).

Cette classe possède juste une méthode virtuelle virtual void update(T info) =0 permettant de mettre les observateurs à jour.

La classe Observable :

Cette classe est templatisé dans le même but qu'observateur.

Cette classe possède une liste d'observateurs et deux méthodes :

- void notifierObservateurs(T info)

  Cette méthode permet de notifier les observateurs

- void ajouterObservateur(Observateur<T> * observateur)

  Cette méthode permet d'ajouter des observateurs à la liste d'observateurs.

La classe Modele :

Cette classe dérive de la classe Observable (donc le modèle est observable).

Le modèle définit le type de données à observer (le template) qui est ici un vecteur d'info.

Cette classe est une interface. Ainsi, cela nous permet de rajouter d'autre modèle facilement si on le désire. Modèle joue le rôle de découpleur, c'est à dire que le reste du programme ne voit que le modèle qui va rester inchangé quelle que soit la réelle implémentation du modèle.

Cette classe contient les méthodes virtuelles suivantes :

virtual void avancerRobot(double dist)=0

virtual void reculerRobot(double dist)=0

virtual void tournerRobot(double angle)=0

virtual void poserRobot(double x_r, double y_a)=0

virtual void setSizeRobot(double h, double w)=0

virtual void setDessinRobot(DRAWINGTYPE dessin)=0

virtual void creerObstacle(double h, double w, double x_r, double y_a, double dir)=0

virtual void setArea(double height, double width)=0

Ces méthodes seront implémentées dans la classe ModeleEnv2D.

La classe Info :

Info est une classe qui possède deux points, une taille, une direction et un mode de dessin.

Nous partons du principe que toutes les formes simples que nous utilisons (lignes et rectangles) peuvent être représentées par des lignes.

Les paramètres direction et type de dessin sont optionnels. Par défaut, la direction est à zéro et le mode de dessin est par défaut ‘autre’.

Le mode de dessin est un type énuméré comprenant :

- lever : c’est à dire que le « pinceau est levé » ainsi nous ne dessinons pas
- dessin : ce mode nous permet de dessiner la trace
- gomme : ce mode nous permet d’effacer la trace
- autre : mode par défaut

La classe ModeleEnv2D :

Pour implémenter le modèle définit dans la classe Modèle, nous utilisons la classe ModeleEnv2D.

ModeleEnv2D utilise une area de type Hitbox (zone de travail du modèle), une fabrique de type FabriqueItem, un robot qui est un pointeur sur Item, obstacles qui est un vecteur de pointeur sur Item et dessin de type vecteur d’info qui mémorise tous les tracés que l’on a fait avec le robot.

Pour construire le modèle il faut une hauteur et une largeur pour :

- la zone dans le lequel le robot va se déplacer, par défaut 100 x 100
- le robot, par défaut 30 x 30

À la fin de la construction, il y a une méthode de notification void notification(). Cette méthode ne prend aucun paramètre et va créer un vecteur d’info qui va être un conteneur de toutes les informations du modèle. Dans ce vecteur d’info, on a au début le vecteur contenant les infos de dessin (tracé). Puis, on y place à la suite les infos associées au vecteur d’obstacle. Pour cela, on va

se déplacer dans le vecteur d'obstacles pour créer pour chaque obstacle, l'info associée à celui ci. Nous faisons de même pour le robot et mettons son Info à la suite.

Finalement nous notifions les observateurs avec le vecteur d'info.

→ La méthode pour créer l' Info associée à l'objet est getItemInfo. Cette méthode est définie deux fois, on peut soit l'appeler avec :

- un pointeur sur item afin de récupérer l'info des objets

  Info getItemInfo(Item * i)

- une hitbox et un type de dessin afin de récupérer l'info de dessin

  Info getItemInfo(Hitbox h,DRAWINGTYPE d)

Pour l'implémentation de avancerRobot et reculerRobot, on utilise un méthode centrale qui est :

void bougerRobot(double dist)

→ La méthode bougerRobot vérifie si le déplacement est valide, c'est à dire si on est bien dans la zone de déplacement et si on ne rencontre pas d'obstacle avec la méthode :

bool deplacementValide(Item * item, double dist) const

La méthode deplacementValide utilise les méthodes :

- bool inArea(Hitbox h) afin de vérifier si on est bien dans la zone de déplacement.
- bool poserValide(Hitbox h) afin de vérifier si on ne rencontre pas d'obstacle.

Si le déplacement n'est pas valide, on se déplace pixel par pixel jusqu'à rencontrer un obstacle.

Une fois que le déplacement du robot est fini, une notification est envoyée aux observateurs.

→ La méthode tournerRobot tourne 1 degré par 1 degré jusqu'à ce qu'on tourne jusque là où nous l'avons demandé ou jusqu'à ce qu'on rencontre un obstacle. Puis, une notification est envoyée.

→ La méthode poserRobot vérifie que le déplacement ait bien lieu dans la zone définie et que poserValide soit vérifiée. Puis la méthode pose le robot à la coordonnée demandée et envoie une notification. Dans le cas où poserValide n'est pas vérifié, un message d'erreur est envoyé.

La méthode setSizeRobot permet de modifier la taille du robot. Si poserValide est juste même après modification de la taille, on envoie une notification. Si poserValide est faux, on envoie un message d'erreur et on ne modifie pas la taille.

La méthode setArea permet de créer la zone de travail. On crée une hitbox pour la zone centrée en (0,0), de taille donnée en paramètre et alignée avec l'axe des abscisses. Une fois la hitbox créée, on crée quatre obstacles correspondant aux quatre 'murs' délimitant la zone. Ainsi, si le robot essaye de sortir de la zone, il y aura collision.

La classe Vue :

Cette classe va observer le vecteur d'info. C'est une interface qui dérive d'observateur. Ainsi, si on le désire, on pourra ajouter facilement plus de vue différentes (interface).

Cette classe inclut les méthodes virtuelles suivantes :

- virtual void on_button_close()=0 permet de fermer la fenêtre
- virtual void on_button_help()=0 permet d'ouvrir la fenêtre d'aide
- virtual void addHelpListener(Controleur *c)=0 entend l'interaction sur le bouton Help et le dit au controleur
- virtual void addExitListener(Controleur *c)=0 entend l'interaction sur le bouton Exit et le dit au controleur
- virtual void addGetCommandeListener(Controleur *c)=0 entend l'interaction sur le bouton Run et le dit au contrôleur
- virtual void addGetfileListener(Controleur *c)=0 permet d'ouvrir le bouton Ouvrir le script et le dit au contrôleur
- virtual void setArea(double height, double width)=0 permet de définir la taille de la zone de travail pour vue (Drawing Area).

Partie Graphique

La classe VueGraphique :

La classe VueGraphique dérive de la classe Vue. Elle implémente tout les boutons de la classe Vue ainsi que la méthode 'update' d'observateur.

Ses variables membres sont :

- des bouttons Gtk::Button bExit, Gtk::Button bHelp, Gtk::Button bGetCommande, Gtk::Button bGetFile, Gtk::Entry EntryCode

- une zone de dessin Canvas ActiveZone, Canvas est une classe agrégée à VueGraphique
- des box de rangement Gtk::Box ButtonBox, EntryBox, MainBox

La classe Canvas :

Cette classe dérive de Gtk::DrawingArea. Sa variable membre est le vecteur d'info.

Le constructeur utilise une largeur et une hauteur pour définir la taille de la Drawing Area. Ces paramètres sont optionnels (par défaut 100x100)

Il possède la méthode void setInfo(std::vector<Info> info) pour modifier le vecteur d'info de la classe que l'on va ensuite afficher. On exécute ensuite la commande queue_draw permettant de réactualiser la zone. Queue_draw va enlever tous les dessins sur la drawing area et va appeler la méthode bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr) qui est une méthode virtuelle pure de drawing area. On l'override et on met les dessins à l'intérieur. Pour cela, nous regardons la taille allouée au widget (la hauteur et la longueur) grâce à get_allocation. Ensuite, nous allons translater les coordonnées de la drawing area au centre c'est à dire à la coordonnées (hauteur/2; longueur/2) pour que l'origine (0,0) ne soit plus en haut à gauche mais au centre comme le modèle observé.

Une fois le repère translaté, on se déplace dans le vecteur d'info grâce à des itérateurs. À l'intérieur, nous allons utiliser get_dessin afin de connaître le mode de dessin et ensuite nous effectuons un switch case sur dessin.

Si on est en mode :

- dessin alors la trace est orange.
- gomme alors on trace de la même couleur que le fond par défaut.
- default alors on trace en vert. C'est dans ce cas qu'on peut tracer un obstacle.

Une fois sorti du switch nous allons vérifier que le mode de dessin est différent du mode 'lever' (le mode où on ne dessine pas). Nous allons ensuite modifier la largeur de la ligne. Nous nous déplaçons ensuite au point ouest de ce que nous allons dessiner avec move_to puis nous traçons une ligne jusqu'au point est avec la méthode line_to. Nous affichons ensuite la ligne avec la méthode stroke.

Sur le vecteur d'info, on s'arrête un élément avant la fin car nous savons que le dernier élément correspond au robot.

Le robot va être traité différemment des autres éléments. On trace bien la ligne comme expliqué précédemment (cette fois ci en rouge). Cependant, on rajoute une forme géométrique pour savoir la direction dans laquelle le robot regarde. Nous avons modéliser cela par un arc de cercle rouge placé dans le rectangle représentant le robot. Nous traçons cet arc de cercle en se plaçant au centre du robot et en traçant un arc de cercle de 180° autour de la direction du robot. On affiche ensuite cet arc de cercle avec stroke.

Interpréteur de code :

La classe Token_list :

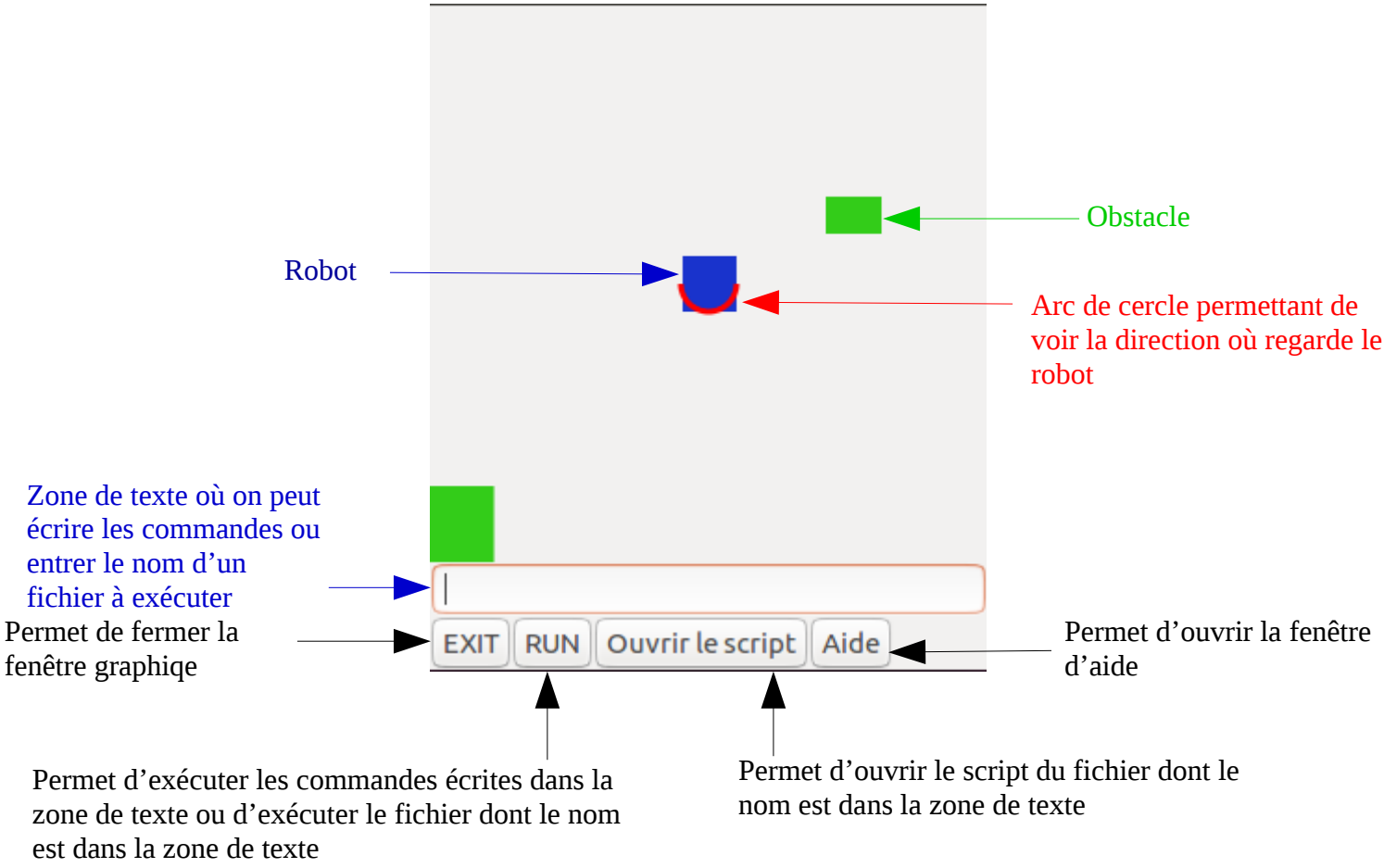
Cette classe permet de faire l’exécution séquentielle du code.

La classe Token :

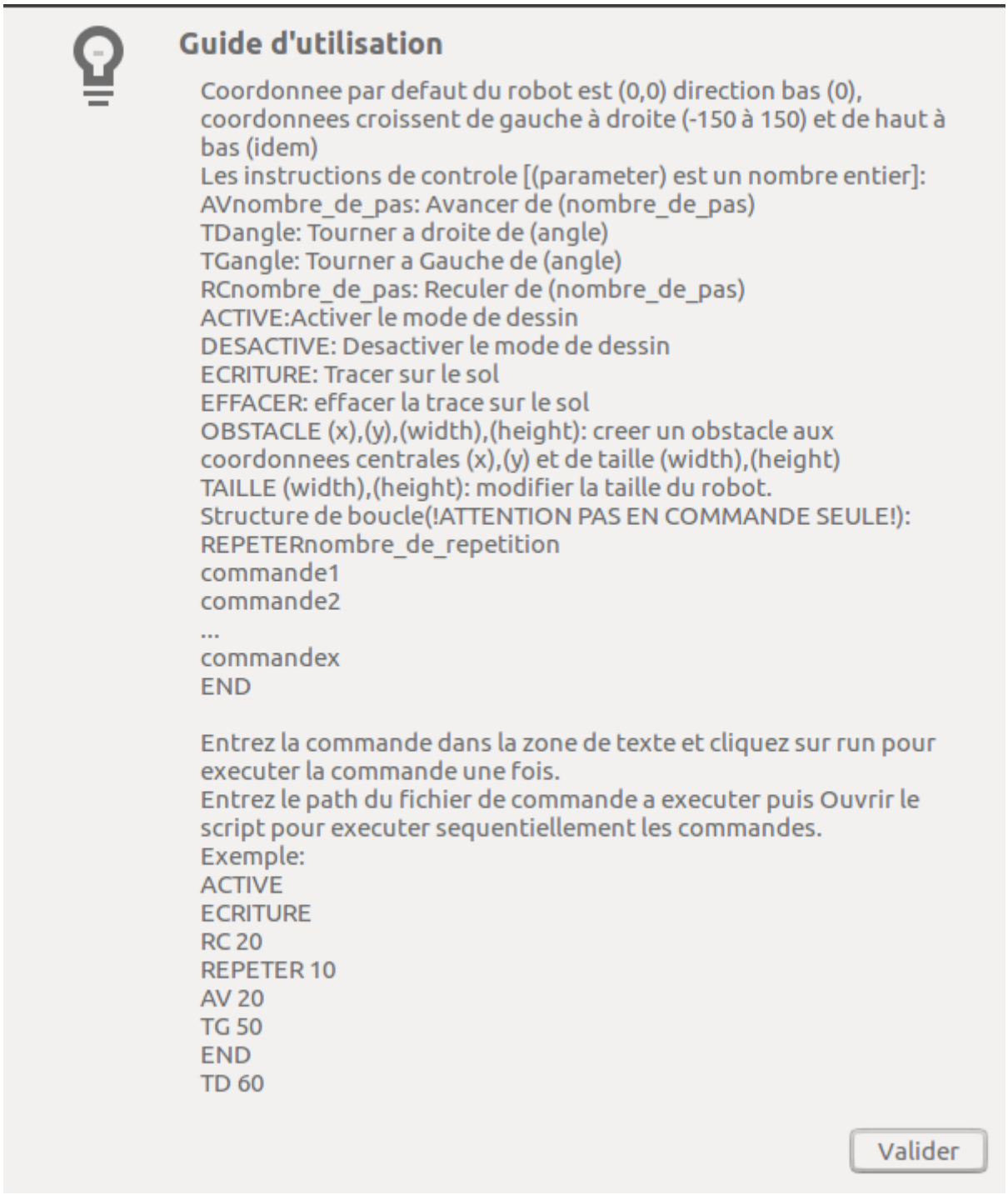
Cette classe qui contient l’expression de la commande et les paramètres associés.

Ils interagissent avec le modèle à travers le contrôleur.

A l'issu de notre projet nous obtenons le robot suivant :



Fenêtre d’aide :



De plus nous obtenons bien les résultats voulu lorsque :

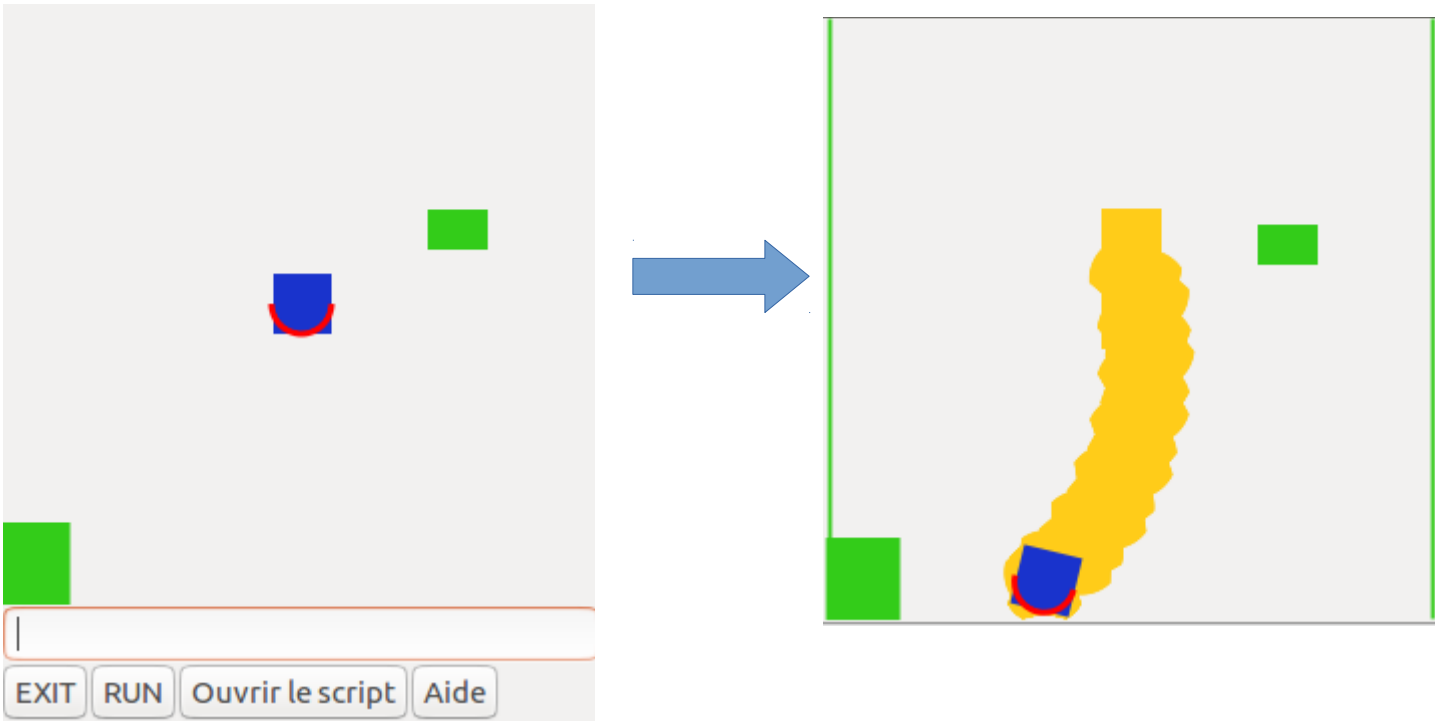
- nous rentrons des commandes directement dans l'interface graphique :
  ACTIVE
  ECRITURE
  AV20
  TG50
  AV50

Afin d'activer la trace, d'avancer de 20 pas, de tourner à gauche de 50° et d'avancer de 50 pas.
Nous obtenons bien le résultat escompté.

- nous exécutons un fichier testfile

Script du fichier testfile :

ACTIVE
ECRITURE
RC 20
RC 20
REPETER 10
AV 20
TG 50
AV 10
TD 60
END
REPETER 2
AV 1
END

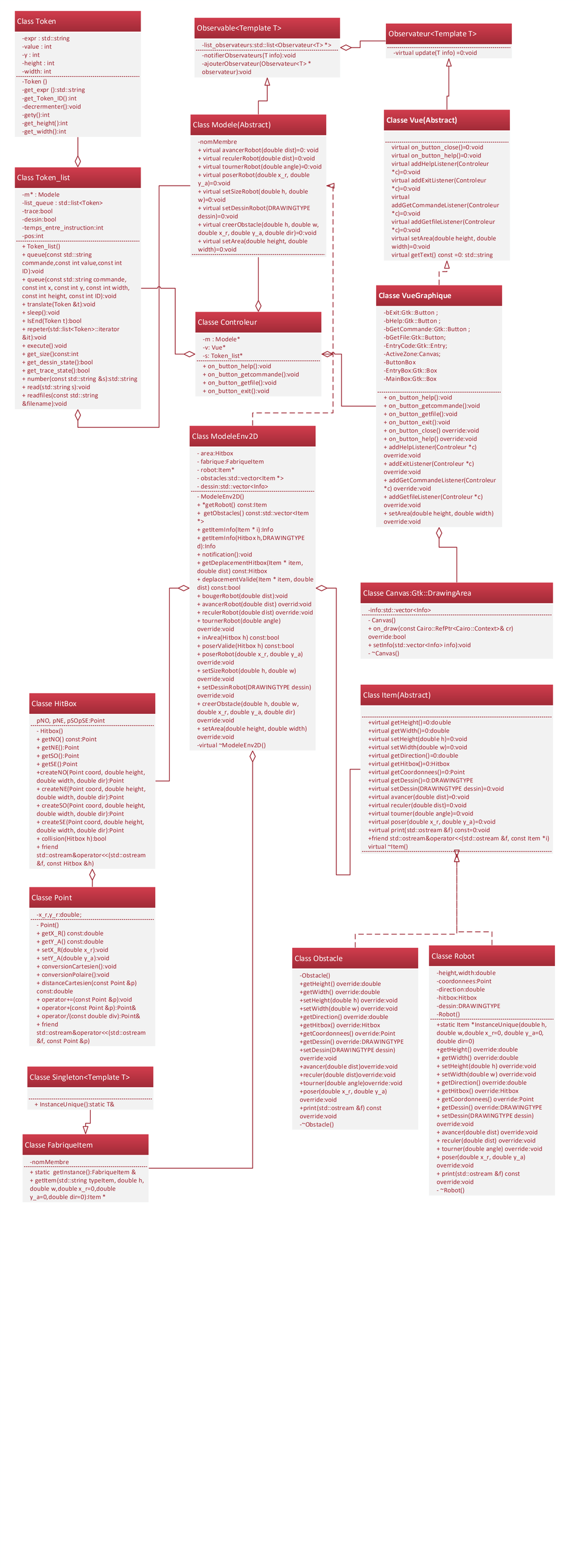


Ainsi nous obtenons bien les résultats voulu.

Nous avons représenté ici quelque cas de figure, mais nous vous laissons le soin de tester toute les possibilités de notre Tortue Robot.

Conclusion :

Nous avons choisi de faire une méthode plus mathématique pour le modèle. Nous avions pensé à deux méthodes : une méthode matricielle et une méthode plus mathématique. Le principe étant qu'avec une méthode matricielle les calculs sont moins important mais il faut allouer une certaine zone mémoire au départ qui ne sera pas facilement adaptable si on veut changer notre Vue. Alors que la méthode plus mathématique que nous avons choisi permet de s'adapter à notre Vue et est facilement adaptable pour toute nouvelle vu plus ou moins précise sachant que dans notre projet nous avons pu nous permettre de perdre en précision afin de nous adapter à notre vue. Au aurait aussi pu choisir une vue précise que celle que nous avons choisi d'utiliser. De plus au niveau mémoire la méthode mathématique permet de consommer moins car on ne sauvegarde que les objets dont on a besoin et non pas toute la zone (les informations de chaque pixel de la zone).


Class Token
-expr : std::string
-value : int
-y : int
-height : int
-width: int
-Token ()
-get_expr ():std::string
-get_Token_ID():int
-decrermenter():void
-gety():int
-get_height():int
-get_width():int
Observable<Template T>
-list_observateurs:std::list<Observateur<T> *>
-notifierObservateurs(T info):void
-ajouterObservateur(Observateur<T> * observateur):void
Observateur<Template T>
-virtual update(T info) =0:void
Class Modele(Abstract)
-nomMembre
+ virtual avancerRobot(double dist)=0: void
+ virtual reculerRobot(double dist)=0:void
+ virtual tournerRobot(double angle)=0:void
+ virtual poserRobot(double x_r, double y_a)=0:void
+ virtual setSizeRobot(double h, double w)=0:void
+ virtual setDessinRobot(DRAWINGTYPE dessin)=0:void
+ virtual creerObstacle(double h, double w, double x_r, double y_a, double dir)=0:void
+ virtual setArea(double height, double width)=0:void
Classe Vue(Abstract)
virtual on_button_close()=0:void
virtual on_button_help()=0:void
virtual addHelpListener(Controleur *c)=0:void
virtual addExitListener(Controleur *c)=0:void
virtual addGetCommandeListener(Controleur *c)=0:void
virtual addGetfileListener(Controleur *c)=0:void
virtual setArea(double height, double width)=0:void
virtual getText() const =0: std::string
Class Token_list
-m* : Modele
-list_queue : std::list<Token>
-trace:bool
-dessin:bool
-temps_entre_instruction:int
-pos:int
+ Token_list()
+ queue(const std::string commande,const int value,const int ID):void
+ queue(const std::string commande, const int x, const int y, const int width, const int height, const int ID):void
+ translate(Token &t):void
+ sleep():void
+ IsEnd(Token t):bool
+ repeter(std::list<Token>::iterator &it):void
+ execute():void
+ get_size():const:int
+ get_dessin_state():bool
+ get_trace_state():bool
+ number(const std::string &s):std::string
+ read(std::string s):void
+ readfiles(const std::string &filename):void
Classe Controleur
-m : Modele*
-v: Vue*
-s: Token_list*
+ on_button_help():void
+ on_button_getcommande():void
+ on_button_getfile():void
+ on_button_exit():void
Classe VueGraphique
-bExit:Gtk::Button ;
-bHelp:Gtk::Button ;
-bGetCommande:Gtk::Button ;
-bGetFile:Gtk::Button;
-EntryCode:Gtk::Entry;
-ActiveZone:Canvas;
-ButtonBox
-EntryBox:Gtk::Box
-MainBox:Gtk::Box
+ on_button_help():void
+ on_button_getcommande():void
+ on_button_getfile():void
+ on_button_exit():void
+ on_button_close() override:void
+ on_button_help() override:void
+ addHelpListener(Controleur *c) override:void
+ addExitListener(Controleur *c) override:void
+ addGetCommandeListener(Controleur *c) override:void
+ addGetfileListener(Controleur *c) override:void
+ setArea(double height, double width) override:void
Class ModeleEnv2D
- area:Hitbox
- fabrique:FabriqueItem
- robot:Item*
- obstacles:std::vector<Item *>
- dessin:std::vector<Info>
- ModeleEnv2D()
+ *getRobot() const:Item
+ getObstacles() const:std::vector<Item *>
+ getItemInfo(Item * i):Info
+ getItemInfo(Hitbox h,DRAWINGTYPE d):Info
+ notification():void
+ getDeplacementHitbox(Item * item, double dist) const:Hitbox
+ deplacementValide(Item * item, double dist) const:bool
+ bougerRobot(double dist):void
+ avancerRobot(double dist) overrid:void
+ reculerRobot(double dist) override:void
+ tournerRobot(double angle) override:void
+ inArea(Hitbox h) const:bool
+ poserValide(Hitbox h) const:bool
+ poserRobot(double x_r, double y_a) override:void
+ setSizeRobot(double h, double w) override:void
+ setDessinRobot(DRAWINGTYPE dessin) override:void
+ creerObstacle(double h, double w, double x_r, double y_a, double dir) override:void
+ setArea(double height, double width) override:void
-virtual ~ModeleEnv2D()
Classe Canvas:Gtk::DrawingArea
-info:std::vector<Info>
- Canvas()
+ on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override:bool
+ setInfo(std::vector<Info> info):void
- ~Canvas()
Classe HitBox
pNO, pNE, pSOpSE:Point
- Hitbox()
+ getNO() const:Point
+ getNE():Point
+ getSO():Point
+ getSE():Point
+createNO(Point coord, double height, double width, double dir):Point
+ createNE(Point coord, double height, double width, double dir):Point
+ createSO(Point coord, double height, double width, double dir):Point
+ createSE(Point coord, double height, double width, double dir):Point
+ collision(Hitbox h):bool
+ friend std::ostream&operator<<(std::ostream &f, const Hitbox &h)
Class Item(Abstract)
+virtual getHeight()=0:double
+virtual getWidth()=0:double
+virtual setHeight(double h)=0:void
+virtual setWidth(double w)=0:void
+virtual getDirection()=0:double
+virtual getHitbox()=0:Hitbox
+virtual getCoordonnees()=0:Point
+virtual getDessin()=0:DRAWINGTYPE
+virtual setDessin(DRAWINGTYPE dessin)=0:void
+virtual avancer(double dist)=0:void
+virtual reculer(double dist)=0:void
+virtual tourner(double angle)=0:void
+virtual poser(double x_r, double y_a)=0:void
+virtual print(std::ostream &f) const=0:void
+friend std::ostream&operator<<(std::ostream &f, const Item *i)
virtual ~Item()
Classe Point
-x_r,y_r:double;
- Point()
+ getX_R() const:double
+ getY_A() const:double
+ setX_R(double x_r):void
+ setY_A(double y_a):void
+ conversionCartesien():void
+ conversionPolaire():void
+ distanceCartesien(const Point &p) const:double
+ operator+=(const Point &p):void
+ operator+(const Point &p):Point&
+ operator/(const double div):Point&
+ friend std::ostream&operator<<(std::ostream &f, const Point &p)
Class Obstacle
-Obstacle()
+getHeight() override:double
+getWidth() override:double
+setHeight(double h) override:void
+setWidth(double w) override:void
+getDirection() override:double
+getHitbox() override:Hitbox
+getCoordonnees() override:Point
+getDessin() override:DRAWINGTYPE
+setDessin(DRAWINGTYPE dessin) override:void
+avancer(double dist)override:void
+reculer(double dist)override:void
+tourner(double angle)override:void
+poser(double x_r, double y_a) override:void
+print(std::ostream &f) const override:void
-~Obstacle()
Classe Robot
-height,width:double
-coordonnees:Point
-direction:double
-hitbox:Hitbox
-dessin:DRAWINGTYPE
-Robot()
+static Item *InstanceUnique(double h, double w,double x_r=0, double y_a=0, double dir=0)
+getHeight() override:double
+ getWidth() override:double
+ setHeight(double h) override:void
+ setWidth(double w) override:void
+ getDirection() override:double
+ getHitbox() override:Hitbox
+ getCoordonnees() override:Point
+ getDessin() override:DRAWINGTYPE
+ setDessin(DRAWINGTYPE dessin) override:void
+ avancer(double dist) override:void
+ reculer(double dist) override:void
+ tourner(double angle) override:void
+ poser(double x_r, double y_a) override:void
+ print(std::ostream &f) const override:void
- ~Robot()
Classe Singleton<Template T>
+ InstanceUnique():static T&
Classe FabriqueItem
-nomMembre
+ static getInstance():FabriqueItem &
+ getItem(std::string typeItem, double h, double w,double x_r=0,double y_a=0,double dir=0):Item *