Projet de c++

/

Alan ASENSIO

Bohua LIU

Hao SUN

Mengxin WU

Table des matières

Chapter I. UML du projet....3
Chapter II. Patron de conception....4
A. Singleton....4
1. Rôle....4
2. La raison de choix....4
3. Détails des algorithmes....5
B. MVC....5
1. Vue – Observateur....6
2. Modèle – Observé....11
3. Contrôleur....16

Chapter I. UML du projet

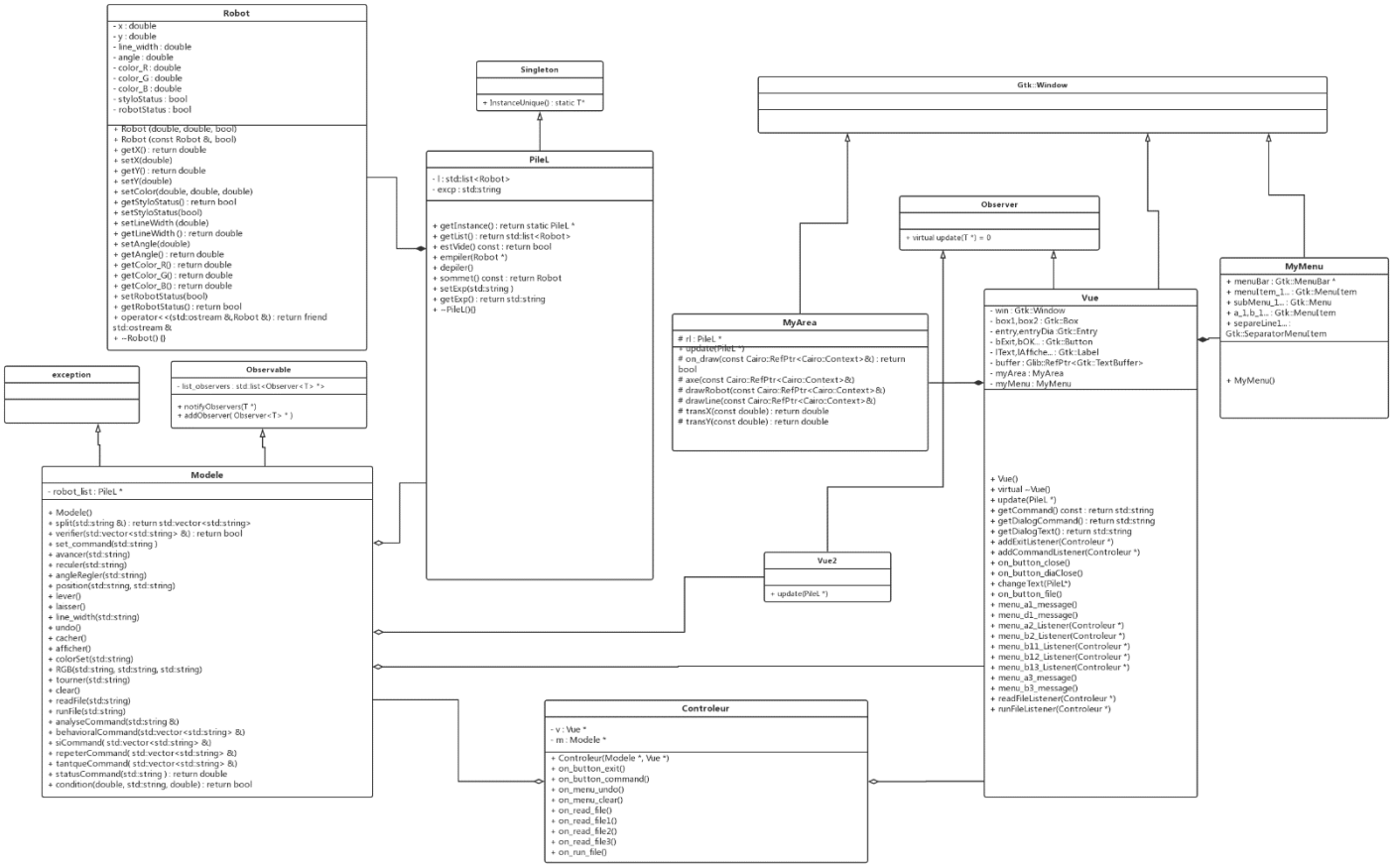

Figure 1 schéma de UML

Voici le schéma UML de notre projet. Nous créons un *Robot* et le stockons dans la *PileL*. Ce robot peut effectuer des opérations de dessin dans la zone géographique d'une interface graphique. Les méthodes de déplacement du robot sont écrites dans le modèle. Et par la fonction *update,* les données sont transférées directement à la classe *myArea*. Et ensuite nous pouvons mettre à jour l'interface graphique.

Cette interface graphique contient également des boutons, des étiquettes, des menus, des zones de saisie et plus encore afin de faciliter son utilisation.

Une introduction détaillée sera donnée au chapitre suivante.

Chapter II. Patron de conception

A. Singleton

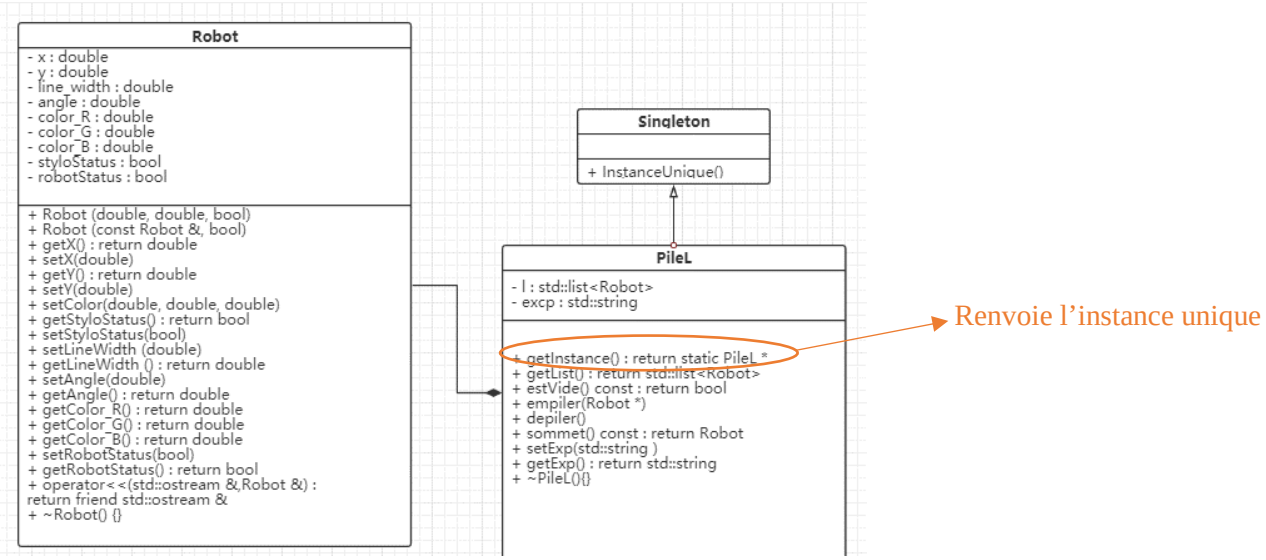

Figure 2 UML de Singleton

1. Rôle

Nous avons choisi le patron de conception *Singleton*. Le but de ce patron est de restreindre l'instanciation d'une classe à un seul objet. Nous avons utilisé ce patron de conception sur la classe *PileL*. Cela signifie que la classe *PileL* n'a qu'une seule instance et fournit un point d'accès global pour y accéder.

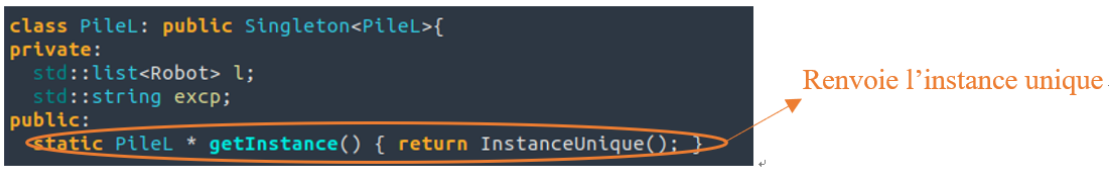

Figure 3 Classe PileL

2. La raison de choix

Dans la classe Area, nous avons besoin de tous les états du robot en vue de le tracer ainsi que sa trajectoire. Donc nous définissons une instance de *PileL*. A l'aide du patron *Singleton*, nous pouvons obtenir cette instance unique qui conserve tous les états du robot jusqu'à maintenant.

```
MyArea::MyArea(){
  rl = PileL::getInstance();
}
```

Figure 1 Constructeur de la classe Area

3. Détails des algorithmes

i. La classe Singleton

- static T *InstanceUnique()
  Cette méthode permet de créer une instance unique de n'importe quel type.

ii. La classe Robot

Cette classe contient les divers états, les accesseurs et mutateurs de ces états. Les coordonnées (x, y), la couleur (color_R, color_G, color_B), l'état de stylo (0 : levé, 1 : baissé), l'angle, la largeur de la ligne et l'état du robot(cacher ou afficher).

iii. La classe PileL

- static PileL *getInstance()
  Cette méthode permet d'obtenir une instance unique de type PileL.
- bool estVide() const
  Dans cette méthode, on renvoie 1 si la taille de la liste du robot est vide, sinon 0.
- void empiler(Robot *r)
  Cette méthode permet d'empiler un pointeur de type Robot à la liste du robot. Ce pointeur représente l'état courant du robot.
- void depiler()
  Cette méthode permet de dépiler le denier élément de la liste du robot.
- Robot sommet () const
  Cette méthode renvoie le dernier élément de la liste du robot.
- void setExp (std::string e)
  Cette méthode renvoie une exception.

B. MVC

Nous avons choisi aussi le patron de conception *Observateur*. Ce patron est bien souvent utilisé avec le modèle MVC. Ici le modèle est le composant observé et les vues sont les composants observateurs. Chaque modification des données sera notifiée aux vues qui pourront alors la visualiser.

1. Vue – Observateur

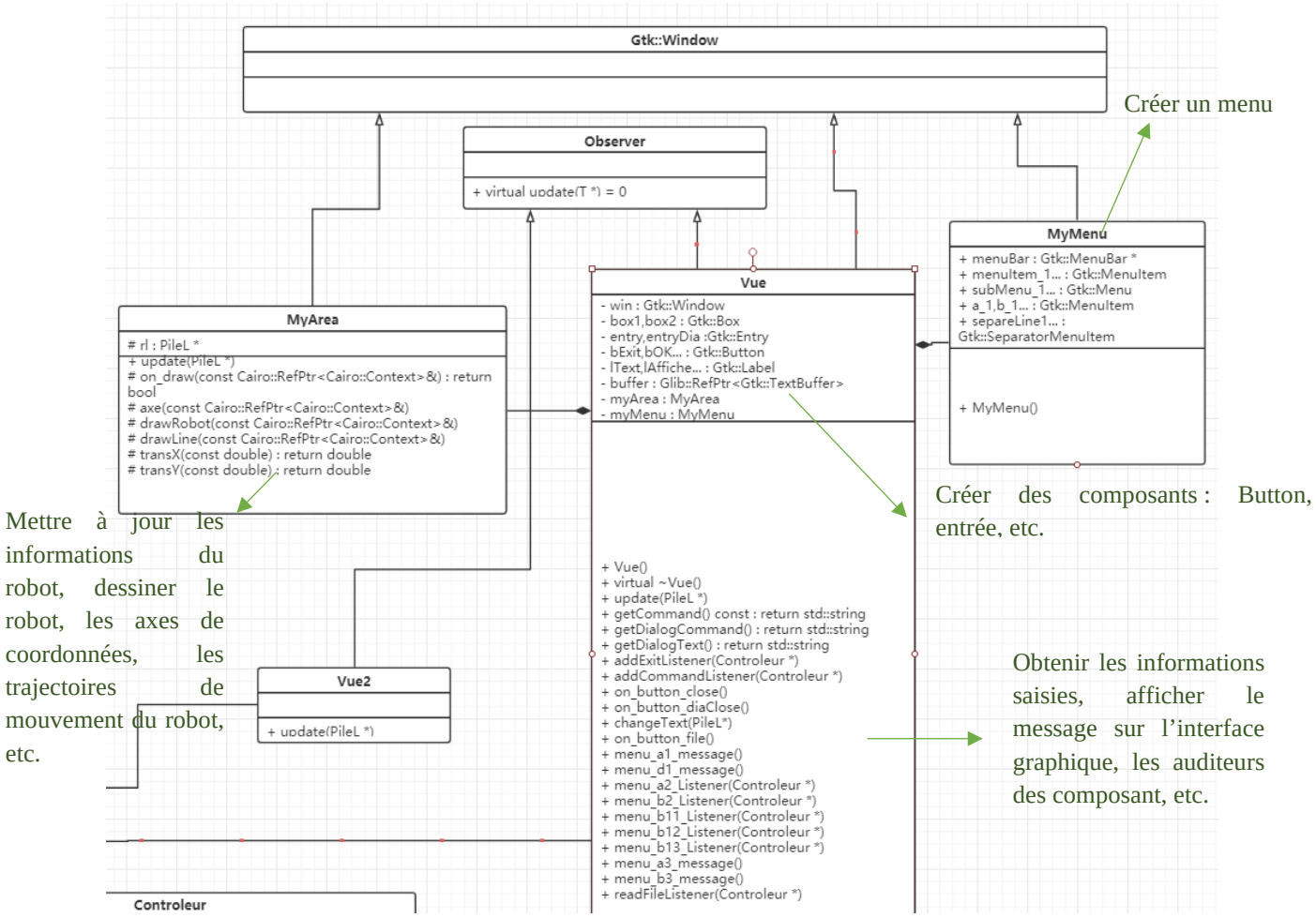

Figure 4 UML d'Observateur

i. Rôle

Cette partie consiste à visualiser les données et les résultats calculés par le modèle et à traiter les événements transmis par le contrôleur. D'abord voici trois exemples de notre programme :

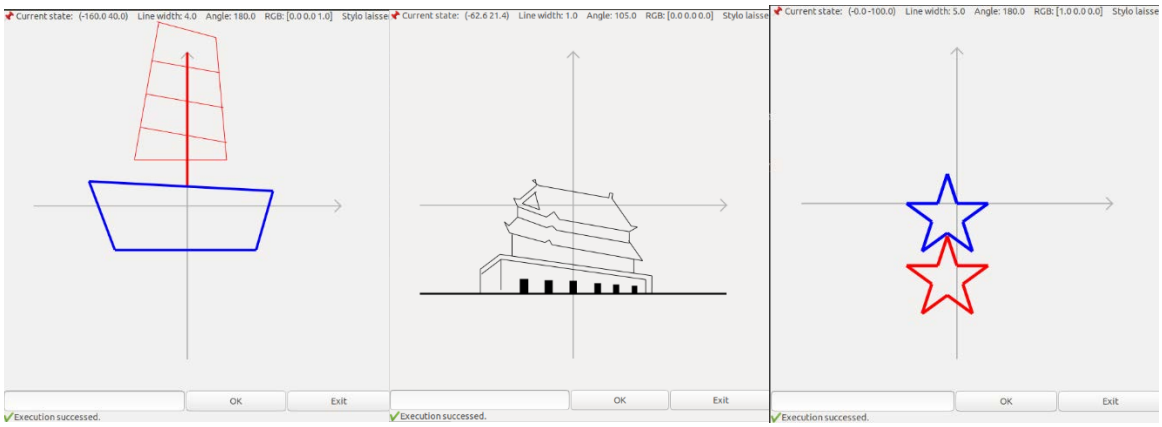

Figure 5 trois exemples

Nous avons deux vues différentes d'un même modèle :

- Une interface graphique

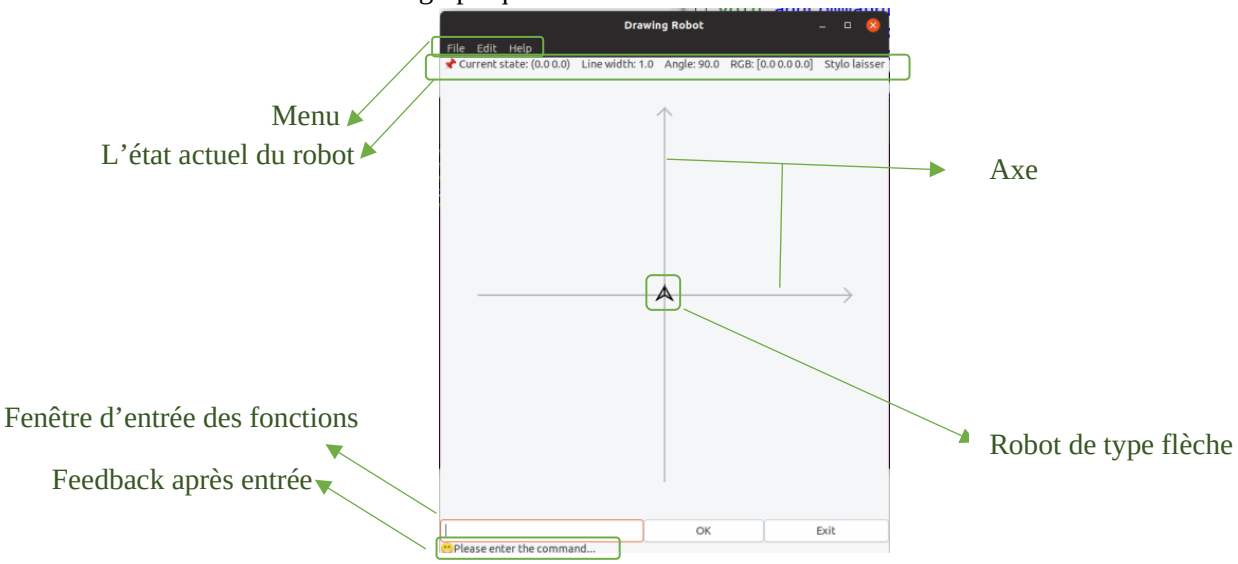

Figure 6. Interface graphique

- La sortie du terminal.

```
lbohh@ubuntu:~/C++/PROJET$ ./main
Current_state: (0 0) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laisser
Current_state: (2.02035e-09 45) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laiss
er

Current_state: (0 0) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laisser
Current_state: (2.02035e-09 45) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laiss
er
Current_state: (2.02035e-09 45) Line width: 1 Angle: 45 RGB: [0 0 0] Stylo laiss
er

Current_state: (0 0) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laisser
Current_state: (2.02035e-09 45) Line width: 1 Angle: 90 RGB: [0 0 0] Stylo laiss
er
Current_state: (2.02035e-09 45) Line width: 1 Angle: 45 RGB: [0 0 0] Stylo laiss
er
Current_state: (2.02035e-09 45) Line width: 1 Angle: 45 RGB: [0 0 1] Stylo laiss
er
```

Figure 7 la sortie du terminal

ii. La raison des choix

Nous fabriquons un robot en forme de flèche pour clarifier sa direction, afin de permettre à l'utilisateur d'obtenir les informations de coordonnées du robot, nous ajoutons l'axe de coordonnées et fixons la position d'origine au centre de la zone pour que celui-ci soit bien visible dès le premier coup d'œil.

Nous ajoutons une ligne d'étiquette au-dessus de la zone de dessin, qui affiche toutes les informations du robot actuel, telles que la couleur, l'angle, l'état du stylo, etc. Ceci permet à l'utilisateur d'obtenir des informations plus rapidement sur l'état du robot, mais apporte également beaucoup d'aide pendant la phase de test du programme.

Nous ajoutons une ligne d'étiquettes au-dessous de la zone de dessin, qui sert de feedback pour les commandes, elle indique à l'utilisateur si le format de la commande est correct, comment faut -t- il entrer la commande, si le mouvement du robot dépasse les limites, etc.

Nous utilisons le patron <<Observateur>> avec le modèle MVC, grâce à ça, nous pouvons créer plusieurs observateurs pour le même modèle. Dans ce projet, l'information modifiée sera mise à jour par l'interface graphique et par le terminal, le terminal étant le deuxième observateur.

iii. Détails des algorithmes

Nous installons des composants comme button, entry comme lors du TD12, puis nous ajoutons un menu, une zone de dessin, des axes.

1) Zone de dessin MyArea

- Double MyArea::**transX**(const double x) et double MyArea::**transY**(const double y)
  C'est pour convertir les pixels de la zone de dessin en coordonnées.
- void MyArea::**update** (PileL *info)
  Après exécution, nous obtenons une nouvelle chaîne de robot qui est créé par modèle. Ensuite le robot est transferé par la fonction *update* à la classe *myArea*. Nous mettons à jour la PileL dans *myArea*. Et quand nous exécutons la fontion *queue_draw*, nous mettons à jour la zone de dessin.
- void MyArea::drawRobot(const Cairo::RefPtr<Cairo::Context>& cr) et
  void MyArea::axe(const Cairo::RefPtr<Cairo::Context>& cr)
  Afin de dessiner le robot et les axes, nous utilisons principalement deux fonctions :
  - ✓ move_to(double x, double y) : Déplacer vers le point (x, y).
  - ✓ line_to(double x,double y) : Tracez une ligne à partir de la coordonnée actuelle jusqu'à (x, y)
- void MyArea::drawLine(const Cairo::RefPtr<Cairo::Context>& cr)
  Afin de tracer la trajectoire du robot, nous examinons les variables du robot, en fonction de l'état du stylo, nous déterminons s'il faut tracer la ligne(line_to) ou se déplacer(move_to).
- bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override
  Cette méthode est la méthode dans la bibliothèque de **Gtk**::**DrawingArea** pour tracer un région. Chaque fois nous créons le *DrawingArea* ou nous le mettons à jour, nous appelons cette méthode. Donc nous écrivons toutes les fonctions de tracer dans cette méthode y compris la fonction de *drawRobot*, *axe* et *drawLine*.

2) Barre de menu MyMenu

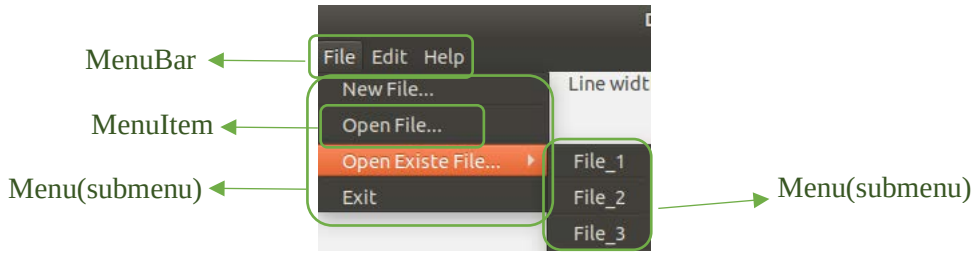

Figure 8 la forme de menu.Menu

Nous ajoutons d'abord les **menuItem**(File,Edit,Help) à la **MenuBar**, puis nous ajoutons un **subMenu** à chaque **menuItem** et ajoutons les nouveaux **menuItem** au **subMenu**. La fonction utilisée pour ajouter les MenuItem est : append (Gtk :: MenuItem menuItem).

- Exit

Une fois que nous cliquons la fenêtre pop-up *File->Exit*, la méthode addExitListener est appelée. Puis le programme se ferme.

- DiaText

Une fois que nous cliquons sur la fenêtre pop-up *File->NewFile*, la méthode *menu_a1_message* est appelée. Une zone d'édition de texte apparaît pour entrer plusieurs lignes de commande.

- `void Vue::menu_a1_message()`

Dans cette méthode, nous avons utilisé la classe **Gtk::TextView** dans la bibliothèque de gtkmm pour créer un Widget d'édition de texte multiligne. Et après nous utilisons cette fonction pour l'ajouter :

box_->pack_start(view);

Puis nous avons utilisé la méthode *view.get_buffer()* pour afficher le texte tapé dans la zone d'édition de texte *view*.

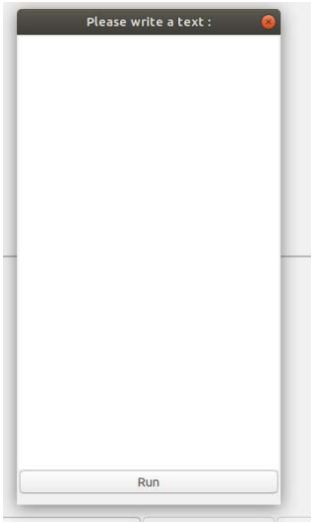

Figure 9 Zone d'édition de texte multiligne

- `std::string Vue::getDialogText()`

Cette méthode est pour récupérer le texte de la zone d'édition, c'est à dire les commande multiligne inscrit dans le document à l'aide de méthode *buffer->get_text*.

- DiaFichier

Une fois que nous cliquons sur la fenêtre pop-up *File->OpenFile*, une zone de dialogue apparaitra pour écrire le nom du fichier que nous voulons ouvrir. Donc pour réaliser cette fonction, il suffit que créer une fenêtre de dialogue avec un *entry* de simple ligne. Et après nous utilisons cette fonction pour ajouter l'*entry* :

box_->pack_start(entryDia);

Il faut bien faire attention à mettre *entryDia* comme variable globale pour réserver les donnée en dehors de la fonction void Vue::**menu_d1_message**(). Grâce à ça, nous pouvons ajouter la fonction std::string Vue::getDialogCommand() pour directement prendre les messages dans l'*entry*. Le résultat apparait comme ci-dessous :

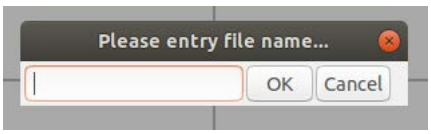

Figure 10 OpenFile

- Open Existe File

Dans la submenu du Open Existe File, nous avons trois fichiers qui ont déjà été écrits, il suffit de les sélectionner pour dessiner les graphiques associés.

- Edit

Dans la submenu Edit, on peut exécuter undo et effacer l'écran.

- Help

Une fois que nous cliquons la fenêtre pop-up *Help->About the command*, la méthode *menue_a3_message* est appelée. Ensuite, un guide d'utilisation des commandes apparaîtra dans l'interface graphique.

Une fois que nous cliquons la fenêtre pop-up *Help->About the programmable robot*, la méthode *menue_b3_message* est appelée. Ensuite, un guide d'utilisation pour les robots programmables apparaîtra dans l'interface graphique :

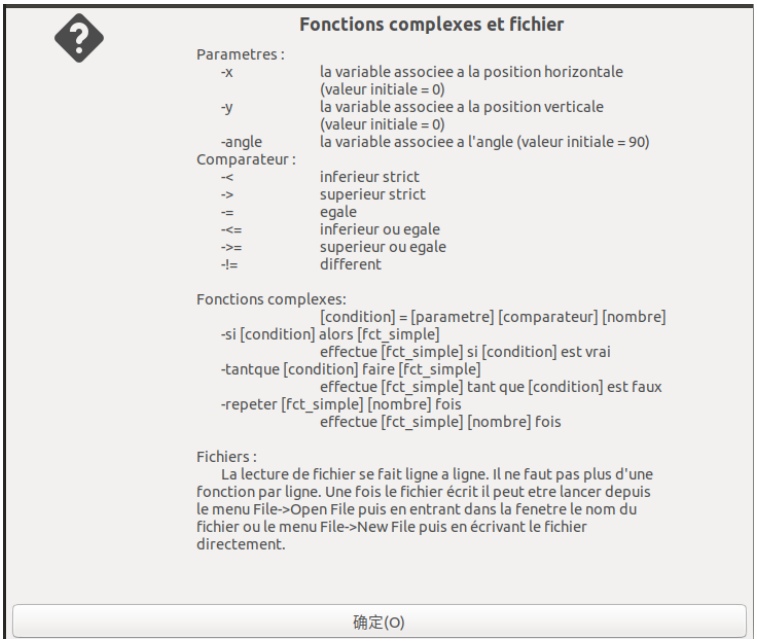

Figure 11 Help de fonctions complexes et fichier

Figure 12 Help de fonctions somples

2. Modèle – Observé

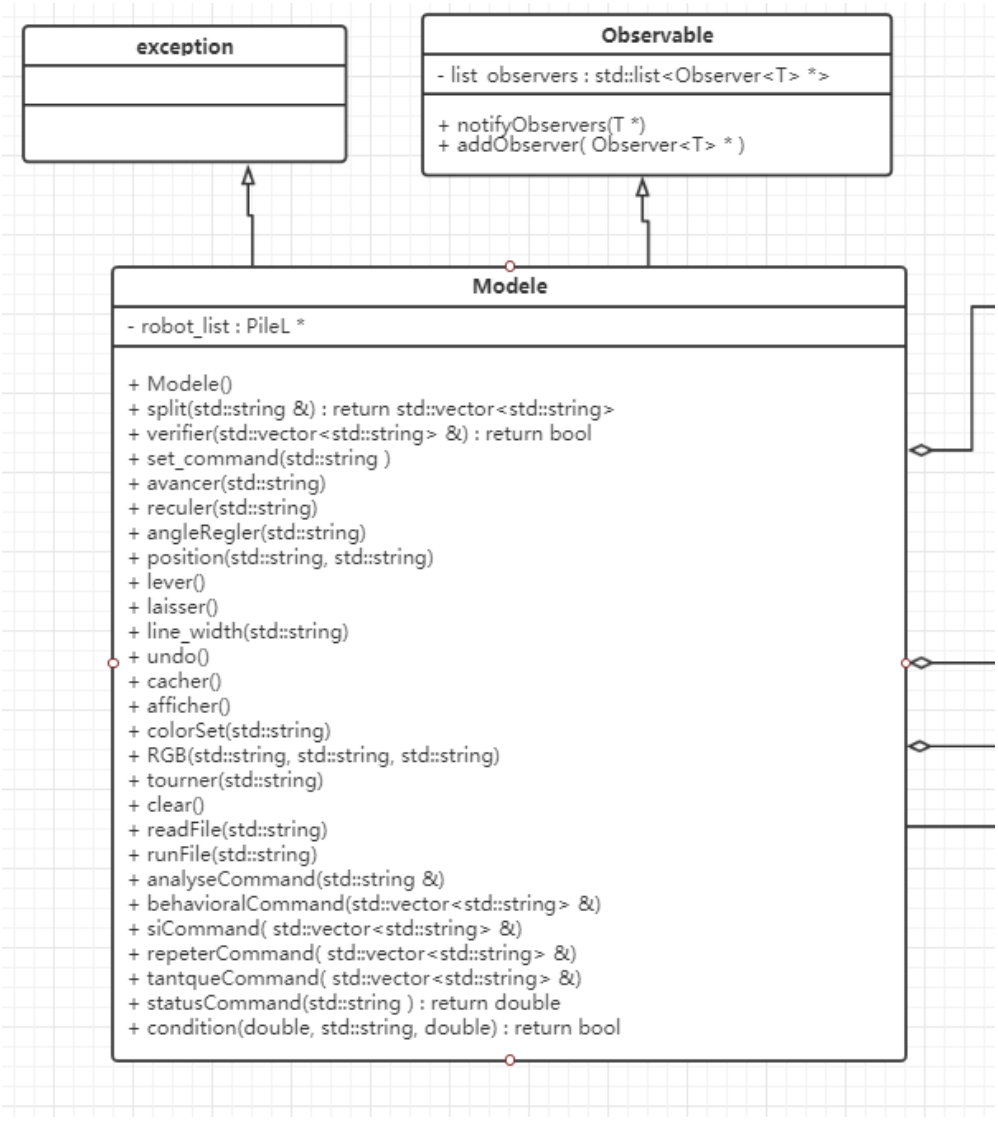

Figure 13 UML de Modèle

i. Rôle

Nous avons créé une classe de « Modèle ». Elle représente le noyau de l'application. Elle contient les analyses de commande, le traitement du stylo et fournit les résultats sous forme de présentation. Elle a une fonction *update* permettant de mettre à jour les données dans la liste de robot à tracer sur l'interface de graphique et a une fonction *addObserver* pour ajouter les observateurs dans la chaine.

ii. La raison des choix

Nous avons fait ce choix pour séparer les codes de traitement et les codes de création de l'interface graphique. Ceci rend les codes plus clair et lisible. Et aussi, nous pouvons initier et utiliser la chaine de robot directement dans la classe *Modèle*, nous n'avons pas besoin de transférer les pointeurs.

iii. Détails des algorithmes

1) void Modele::set_command (std::string command) :

Cette méthode est la plus importante fonction de la classe. D'abord, le contrôleur l'appelle et il transfert la chaine de caractères avec cette méthode. Ensuite elle fait une pré-analyse et transfert à la méthode de

analyseCommand pour une analyse approfondie. Enfin les données sont traitées et nous mettons à jour les observateurs par

notifyObservers(robot_list);

2) void Modele::analyseCommand(std::string &cmd) :

D'abord, nous convertissons toutes les lettres de cette chaîne en minuscules :

std::transform(cmd.begin(),cmd.end(),cmd.begin(), [] (unsigned char c) {return std::tolower(c);});

Et ensuite, nous avons séparé la chaîne de caractères et l'avons enregistré dans un vecteur en utilisant la fonction ci-dessous :

std::vector<std::string> commandVector = split(cmd);

Après nous utilisons cette méthode pour diviser la commande pour savoir s'il s'agit d'une commande de statut (si … alors, tantque…faire… etc.) ou une commande comportementale (avancer, color, position, etc.) en fonction du premier mot de commande. Et ensuite, nous transférons la chaine de caractères à la fonction correspondante.

3) void Modele::behavioralCommand(std::vector<std::string> &command) :

Nous utilisons cette méthode pour tester la commande comportementale. Si le premier mot est le mot qui nous intéresse, nous transformons le 1er, 2èmeet 3ème mot (s'il existe) en la fonction correspondante.

4) void Modele::avancer(std::string s) :

Pour cette méthode, d'abord, nous avons créé une instance dynamique qui est égale au sommet de la pile (pour prendre toutes les propriétés d'état courant) en utilisant la fonction ci-dessous :

Robot * robot = new Robot(robot_list->sommet(),true);

Et ensuite, nous prenons la position courante et faisons avancer le robot selon l'angle courant comme ceci :

```
robot->setX(x+dst*cos(angle));
robot->setY(y+dst*sin(angle));
```

Et après nous empilons ces instructions dans la pile et libérons l'espace dynamique :

```
robot_list->empiler(robot);
delete robot;
```

5) void Modele::reculer(std::string s):

Comme la méthode avancer sauf que la fonction de calcul change :

```
robot->setX(x-dst*cos(angle));
robot->setY(y-dst*sin(angle));
```

6) void Modele::angleRegler(std::string s) :

Nous changeons la fonction de traitement à :

robot->setAngle(des);

7) void Modele::position(std::string x, std::string y) :

Comme la méthode avancer sauf que la fonction rentre les valeurs directement :

```
robot->setX(x_des);
```

```
robot->setY(y_des);
```

8) void Modele::lever() :

Nous mettons la variable booléenne *StyloStatus* de *Robot* sur faux :

```
robot->setStyloStatus(false);
```

9) void Modele::laisser() :

Nous mettons la variable booléenne *StyloStatus* de *Robot* sur vrai :

```
robot->setStyloStatus(true);
```

10) void Modele::line_width(std::string s) :

Nous changeons la largeur de la ligne directement par la méthode de *Robot* :

```
robot->setLineWidth(des);
```

11) void Modele::undo() :

Si la taille de chaîne de robot est supérieure à 1, c'est-à-dire s'il n'est pas dans l'état initial, nous pouvons faire l'opération de undo (depiler).

```
if(robot_list->getList().size()>1)
      robot_list->depiler();
```

12) void Modele::cacher() :

Nous mettons la variable booléenne *RobotStatus* de *Robot* sur faux :

```
robot->setRobotStatus(false);
```

13) void Modele::afficher() :

Nous mettons la variable booléenne *RobotStatus* de *Robot* sur vrai :

```
robot->setRobotStatus(true);
```

14) void Modele::colorSet(std::string s) :

Nous identifierons le deuxième mot, si c'est notre couleur établie, nous allons définir la couleur du robot en fonction des données que nous avons établies.

```
 if (s=="black") {
    robot->setColor(0,0,0);
      } else if (s=="white") { … …
```

15) void Modele::RGB(std::string r, std::string g, std::string b) :

Nous définissons la couleur en fonction des trois valeurs RGB entrées par l'utilisateur en utilisant la fonction ci-dessous :

```
robot->setColor(r_,g_,b_);
```

16) void Modele::tourner(std::string a) :

Nous utilisons cette fonction pour faire pivoter d'un certain angle. L'angle va augmenter dans le sens des aiguilles d'une montre.

```
robot->setAngle(ang*180/PI+des);
```

17) void Modele::clear() :

Nous dépilons toutes les actions sauf la première.

18) double Modele::statusCommand(std::string command) :

Si le premier mot de commande est un mot de commande dites complexes comme si … alors, tantque…faire… etc. Nous appelons la méthode correspondante. Dans cette commande, il existe une fonction de comparaison comme x>2, nous utilisons cette méthode pour analyser le premier mot qui est la variable à comparer. Nous lisons la valeur de la variable correspondant dans l'état courant. Par exemple :

```
if(command =="angle")
  return robot_list->sommet().getAngle()*180/PI;
else if(command == "x") … …
```

Nous pouvons comparer trois variables : angle, x et y.

19) bool Modele::condition(double c1, std::string condition, double nom) :

La fonction de condition est celle qui se charge d'évaluer si la condition est correcte ou non. Le premier paramètre est celui que nous avons obtenu par la méthode *statusCommand*. Et il faut vérifier la condition à partir du deuxième paramètre, voici les codes ci-dessous :

```
if(condition==">")
  return c1>nom;
else if(condition=="<") … …
```

20) void Modele::repeterCommand(std::vector<std::string> &command) :

Nous savons que la commande est de la forme **repeter… n fois**, donc nous voulons chercher le mot fois d'abord. Nous utilisons la fonction *find* pour le trouver facilement. Nous mettons la commande après le mot **repeter** dans un vecteur *cmd*. Ensuite, nous pouvons dire que la chaîne avant de lui est le nombre de fois. Du coup, nous écrivons une boucle *while* pour le réaliser :

```
for(int i=0;i<l;i++)
  behavioralCommand(cmd);
```

Nous aussi utilisons la fonction de *behavioralCommand* pour effectuer à chaque fois d'appel de command.

21) void Modele::tantqueCommand(std::vector<std::string> &command) :

Nous savons que la commande est de la forme **tantque… faire…**, donc nous voulons chercher le mot **faire** d'abord. Nous utilisons la fonction *find* pour le trouver facilement. Nous mettons la commande après le mot **faire** dans un vecteur *cmd*.

Et après, on vérifie si la condition est vraie ou non en utilisant la fonction de *condition* (que nous avons introduit paragraphe *19)*. Voici les codes ci-dessous :

```
while(condition(statusCommand(command[1]),command[2], atof(command[3].c_str())))
{
  if(abs(robot_list->sommet().getX()) <= WIDTH/2
           && abs(robot_list->sommet().getY()) <= HEIGHT/2
           && n<10000) {
      behavioralCommand(cmd);
      n++;
  } else {
      break;
  }
}
```

Pour limiter le nombre maximum d'exécution, nous avons ajouté une variable *n* et quand la boucle s'exécute 1000 fois, elle s'interrompt.

22) void Modele::siCommand(std::vector<std::string> & command) :

Nous savons que la commande est de la forme **si… alors…**, donc nous voulons chercher le mot **alors**. Nous utilisons la fonction *find* pour le trouver facilement. Nous mettons la commande après le mot **alors** dans un vecteur *cmd*.

Et après, on vérifie si la condition est vraie ou non en utilisant la fonction de *condition* (que nous avons introduit paragraphe *19)*. Comparer avec la dernière méthode, il suffit que changer le mot clé *while* par *if*, voici les codes ci-dessous :

```
if(condition(statusCommand(command[1]), command[2], atof(command[3].c_str())))
    behavioralCommand(cmd);
```

Pour limiter le nombre maximum d'exécutions, nous avons ajouté une variable *n* et au bout de 1000 exécutions, on interrompt la boucle afin d'empêcher une boucle infinie.

23) void Modele::readFile(std::string in) :

Pour lire les fichiers, nous avons écrit cette méthode. Elle permet d'ouvrir un fichier et lit les chaînes de caractères ligne par ligne. Pour chaque chaîne, nous appelons la fonction *set_command* donc nous pouvons réaliser la fonction de lire le fichier.

24) void Modele::runFile(std::string text) :

Nous avons réalisé la fonction d'exécuter un nouveau fichier que nous créons en utilisant cette fonction. Nous utilisons la fonction ci-dessous :

```
for (std::string::iterator it = text.begin() ; it != text.end() ; it++)
      oos.put(*it);
```

Et après nous réappelons la fonction *readFile* pour l'exécuter.

iv. Traitement des Exceptions

Pour éviter les situations anormales, nous avons fait quelques traitements des exceptions. Nous avons créé 10 classes héritant de la classe std::exception qui représente les différentes exceptions comme ci-dessous. Puis nous avons associé les exceptions aux instructions correspondantes.

- La classe CrossBorder : Le robot a traversé la frontière de la fenêtre
- La classe MissCommand : Commande d'entrée incomplète
- La classe CommandError : Erreur de commande d'entrée
- La classe FormatError1 : Erreur de format de commande ''si ... alors ...''
- La classe FormatError2 : Erreur de format de commande ''repeter ... n fois''
- La classe FormatError3 : Erreur de format de commande ''tanque... faire ...''
- La classe FormatError4 : Erreur de format de commande de l'état
- La classe ColorError : Erreur de commande de la couleur
- La classe ConditionError : Erreur de l'opérateur conditionnel
- La classe FichierError : Erreur du nom de fichier

3. Contrôleur

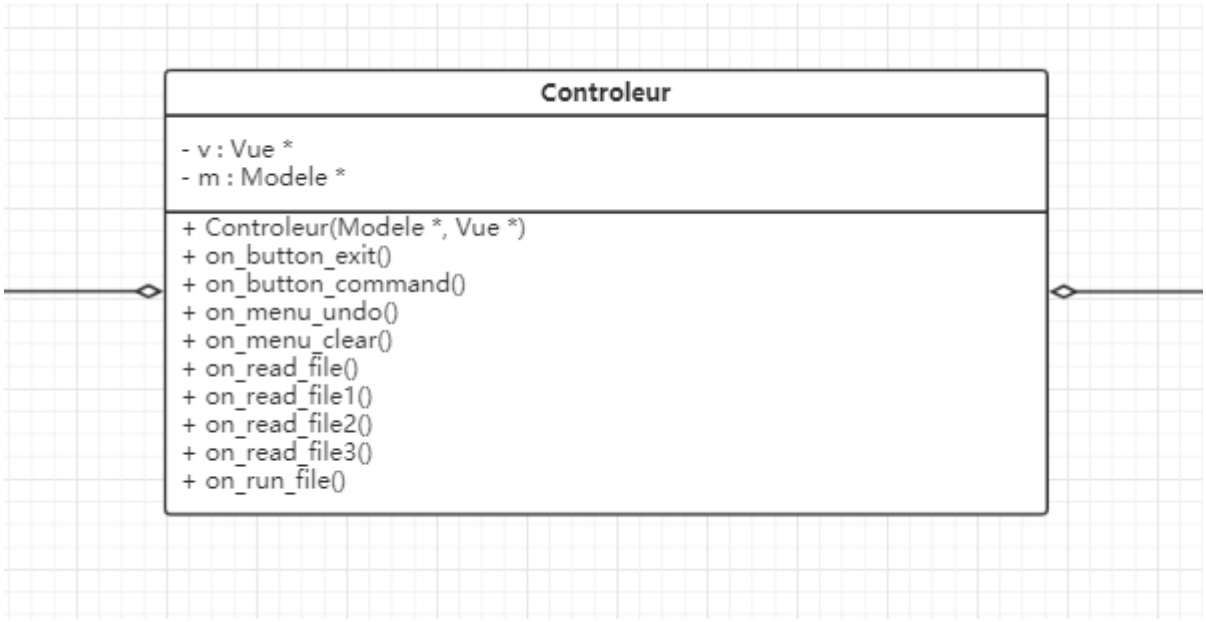

Figure 14 UML de contrôleur

i. Rôle

Nous avons créé une classe Controleur. Elle assure la connexion entre les 2 parties précédentes (vue et modèle). Elle enregistre les gestionnaires d'événements pour les buttons *exit, go, undo, clear, New File…, Open File…, run.* Elle associe deux membres privés de type Modele et Vue. Donc elle a accès aux instances de ces deux classes.

ii. La raison des choix

Nous avons choisi le contrôleur parce que nous voulons connecter les deux parties en utilisant une classe. Grâce à ça, nous pouvons utiliser le modèle et l'observateur de même temps. C'est pour augmenter la liaison entre deux parties.

iii. Détails des algorithmes

1) Controleur(Modele *mm, Vue *vv) : m(mm), v(vv)

Dans cette méthode, on appelle tous les *listener* écrit dans la classe Vue en passant par le pointeur de Controleur.

2) void on_button_exit()

Cette méthode est pour gérer l'événement de quitter le programme. On appelle la méthode *on_button_close* par l'instance de Vue.

3) void on_button_command()

Cette méthode est pour gérer l'événement d'exécuter une commande entrée par l'utilisateur. On appelle la méthode *getCommande* par l'instance de Vue pour obtenir la commande. Puis on appelle la méthode *set_commande* par l'instance de Modele pour mettre la commande.

4) void on_menu_undo()

Cette méthode est pour gérer l'événement d'annuler l'opération. On appelle la méthode *set_commande("undo")* par l'instance de Modele.

5) void on_menu_clear()

Cette méthode est pour gérer l'événement de nettoyer d'interface graphique. On appelle la méthode *set_commande("clear")* par l'instance de Modele.

6) void on_read_file()

Cette méthode est pour gérer l'événement de lire un fichier. On appelle la méthode *getDialogCommand()* par l'instance de Vue pour obtenir le nom du fichier. Puis on appelle la méthode *readFile* par l'instance de Modele pour lire ce fichier.

7) void on_run_file()

Cette méthode est pour gérer l'événement d'exécuter un fichier. On appelle la méthode *getDialogText()* par l'instance de Vue pour obtenir le contenu du fichier. Puis on appelle la méthode *runFile* par l'instance de Modele pour exécuter toutes les commandes dans ce fichier.