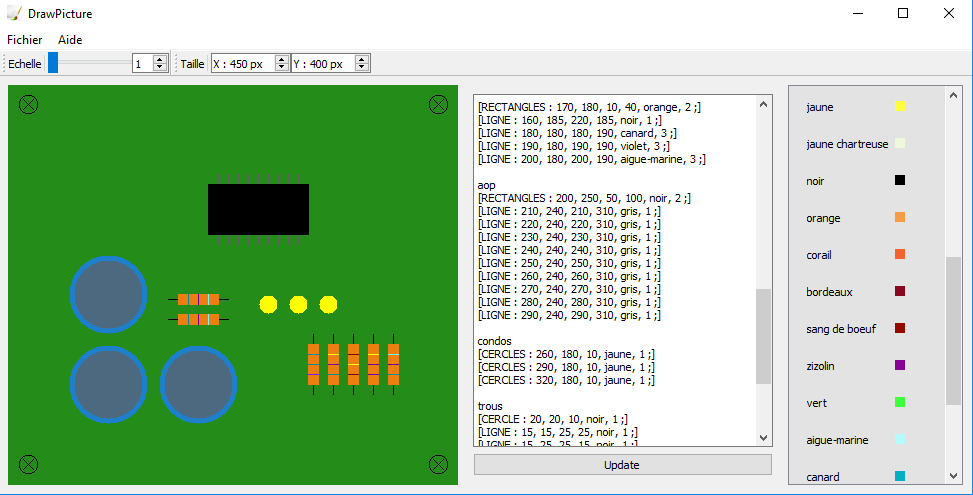
|  |
| --- |
|  |
| Rapport C++ |
| Dessin vectoriel |



|  |
| --- |
|  |



Electronique

Année 2

**Encadrant :**

Benoit Le Gal

**Etudiants :**

Florian Schaal

Margot Robin

SOMMAIRE

[**INTRODUCTION** 3](#_Toc481502142)

[**I.** **Figures** 4](#_Toc481502143)

[Architecture 4](#_Toc481502144)

[Classe « Figure » 4](#_Toc481502145)

[Algorithmes de remplissage de coordonnées 5](#_Toc481502146)

[Classe « Draw » 6](#_Toc481502147)

[**II. Dimension graphique** 8](#_Toc481502148)

[Présentation des fenêtres 8](#_Toc481502149)

[Choix des fichiers 8](#_Toc481502150)

[Fenêtre principale 9](#_Toc481502151)

[Barre d’outils 9](#_Toc481502152)

[**CONCLUSION** 11](#_Toc481502153)

# INTRODUCTION

Ce projet a pour objectif de coder un petit logiciel de dessin vectoriel. A partir d’une description géométrique, nous allons fabriquer un dessin. Les codes sources permettant de créer et dessiner pixel par pixel sont déjà fournis.

Le cahier des charges sera le suivant :

* Description vectoriel dans un fichier « .vec », une forme par ligne du type :

[FIGURE : x, y, (autres paramètres), COULEUR, (z), (alpha) ;]

* L’image sera enregistrée en « .bmp »
* Les paramètres « z » et « alpha » sont optionnels, servant respectivement à définir le plan sur lequel dessiner la figure et à définir la transparence de la figure
* Les différentes figures possibles sont :
* POINT
* LIGNE
* CARRE
* CARRES (plein)
* RECTANGLE
* RECTANGLES (plein)
* CERCLE
* CERCLES (plein)
* L’utilisateur ne doit pas savoir comment fonctionne le dessin des formes
* Les couleurs sont définies par l’utilisateur dans le fichier « color/color.col » sous la forme :

COULEUR rouge vert bleu

* L’utilisateur peut spécifier la taille de l’image ainsi qu’un facteur d’échelle afin de zoomer sur l’image sans qu’elle ne devienne floue
* L’exécutable prendra en paramètres les noms des fichiers vectoriel et bitmap, et pourra également prendre la taille et l’échelle de l’image à dessiner

# Figures

### Architecture

Dans un premier temps, nous allons définir les différentes classes à créer, afin de dessiner différentes figures, ainsi que les liens de relations entre elles. Ceci est fait grâce à la description UML donnée en annexe 1. Nous allons donc créer une classe mère, « Figure », qui contiendra tous les paramètres communs à toutes les figures (taille de l’image, coordonnées du premier point, couleur, plan et transparence). Nous allons maintenant voir à quoi correspond chaque méthode et argument de cette classe mère.

### Classe « Figure »

Cette classe sera la classe mère de toutes nos figures. Cependant, elle inclut une méthodes virtuelle, nommée fillCoord() qui, en la redéfinissant, permettra pour chaque figure de remplir un vecteur de coordonnées. Ainsi, cette classe est virtuelle et ne pourra donc pas être instanciée directement. Ses attributs et beaucoup de ses méthodes seront déclarées protected, afin que seules ses classes filles puissent les utiliser. Les constructeurs, destructeur et méthode pour dessiner seront quant à eux public, ce sont ces méthodes que l’utilisateur pourra appeler.

Afin de respecter le cahier des charges, prenant en compte deux arguments optionnels pour chaque figure, nous avons surchargé le constructeur de cette classe, dont les prototypes sont :

* Figure(int tailleX, int tailleY, int X, int Y, string couleur) ;
* Figure(int tailleX, int tailleY, int X, int Y, string couleur, int Z) ;
* Figure(int tailleX, int tailleY, int X, int Y, string couleur, int Z, int a) .

Les paramètres ‘tailleX’ et ‘tailleY’ correspondent aux dimensions de l’image, ’X’ et ‘Y’ aux coordonnées d’application de la figure, ‘couleur’ à la couleur que l’on souhaite donner à la figure, ‘Z’ au plan sur lequel la figure doit être dessinée et ‘a’ à la transparence de la figure.

Afin d’être sûr que la figure sera contenue dans l’image et ainsi éviter une erreur lors de la création de l’image, nous effectuons une vérification sur les coordonnées de la figure ; en effet, ils doivent obligatoirement se trouver dans l’image et non au-delà ou en-deçà, d’où la nécessité d’avoir les dimensions de l’image. Si la vérification échoue, nous changeons la valeur d’un booléen de contrôle, ‘writePossible’, à false.

Pour pouvoir stocker les coordonnées des futures formes, nous créons un vecteur d’un type défini par nos soins, comprenant deux entiers, ‘x’ et ‘y’. Il suffira, lors de l’appel de la classe fille, de remplir ce vecteur, grâce à la méthode push\_back(coordonnées).

Nous avons choisit, pour ce programme, de lire les couleurs disponibles dans un fichier texte. Ceci est réalisé grâce à la méthode readRGB(string fichier), où fichier correspond au fichier de couleurs, ici ‘./color/color.col’. Pour ce faire, il suffit d’ouvrir un flux en lecture seule grâce à un ‘ifstream’. Par la suite, nous lisons le fichier ligne par ligne. Les couleurs étant écrites de la forme : ‘couleur R G B’, nous pouvons isoler les composantes rouge, verte et bleue de la couleur. En effet, la dernière est contenue entre la fin de ligne et un espace, les deux autres entre deux espaces. Grâce aux méthodes de la classe ‘string’, rfind(string caract, size\_t pos=0) et substr(size\_t pos1, size\_t pos2), où ‘caract’ correspond à la chaîne de caractères à trouver, ‘pos’ la position à partir de laquelle chercher, ‘pos1’ la position du premier caractère à extraire et ‘pos2’ la taille de la chaîne à extraire. Ces fonctions permettent respectivement de chercher une chaîne de caractère dans une autre chaîne de caractères, à partir de la fin de chaîne, et d’extraire une chaîne de caractère à partir d’une autre chaîne de caractère. Ainsi, la couleur pourra être en plusieurs mots, séparés par des espaces. Nous stockons ces valeurs dans un vecteur d’un type que nous avons défini pour les couleurs, comprenant un nom sous forme de string, et les trois composantes sous forme d’entiers. Il suffit ensuite, grâce à la méthode setRGB(const string couleur), où ‘couleur’ définit la couleur à appliquer à la figure, de parcourir le vecteur créé. En cas d’échec de correspondance, ‘writePossible’ est mis à false.

Enfin, nous créons une méthode qui, une fois tous les paramètres acquis et calculés, permettra de modifier la CImage que nous dessinons : update(CImage\* img). Après avoir vérifié que l’image pouvait être dessinée, nous calculons les coordonnées spécifiques à la figure créée, puis nous faisons une boucle sur la taille du vecteur de coordonnées, acquise grâce à la méthode size(). Nous récupérons un pointeur sur le pixel à dessiner, acquis grâce à la méthode getPixel(int x, int y) de la classe CImage. Afin d’appliquer la transparence à la figure, nous accédons, grâce aux méthode Red(), Green() et Blue() les valeurs des composantes de ce pixel. Ensuite, nous appliquons la formule de transparence : , où ‘redA’ correspond à la nouvelle valeur de rouge du pixel, ‘redPixel’ son ancienne valeur et ‘redFigure’ la couleur de la figure opaque. Il suffit de faire le même calcul pour les composantes verte et bleue, puis, grâce au mutateur RGB(int red, int blue, int green) de la classe CImage, d’appliquer ces nouvelles valeurs au pixel pointé.

Pour finir, nous définissons un destructeur virtuel, qui sera redéfini, afin de désallouer l’espace mémoire nécessaire à la création d’une instance Figure.

### Algorithmes de remplissage de coordonnées

Il s’agit maintenant de définir les classes filles, celles qui seront instanciées par l’utilisateur. En effet, il faut redéfinir la méthode fillCoord(), permettant de remplir le vecteur de coordonnées. Pour chaque classe fille, nous définissons donc leurs propres constructeurs et destructeurs. Néanmoins, les constructeurs de ces classes appellent les constructeurs de la classe mère, afin d’éviter de redéfinir les initialisations pour chaque classe fille. Les classes CarreS, CercleS et RectangleS hériterons quant à eux respectivement des classes Carre, Cercle et Rectangle. En effet, il est inutile de redéfinir les conditions que doivent vérifier les différentes coordonnées afin que la figure soit dessinable.

L’algorithme de remplissage le plus simple et celui du point. En effet, il suffit d’ajouter au vecteur les coordonnées de ce point directement.

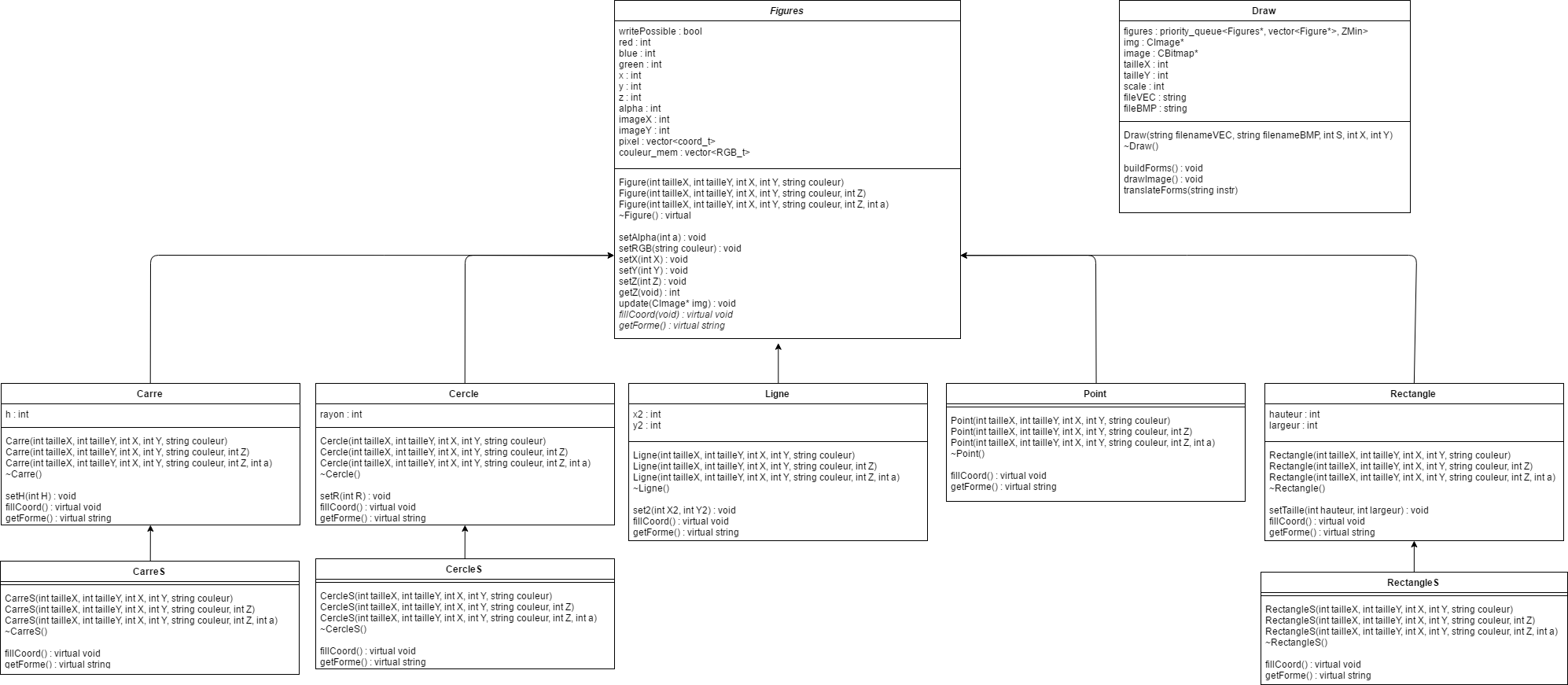
Ensuite, que ce soit pour le rectangle ou le carré, nous parcourons alternativement les côtés en x et en y, en remplissant le vecteur avec les balayements ainsi que leur coordonnées auxquelles s’ajoutent la taille de ces figures.

Afin de tracer une ligne dans toutes les directions possibles, nous utilisons l’algorithme de Bresenheim : nous balayons en x ou en y, suivant l’octant dans lequel se place la ligne, puis nous définissons si respectivement la valeur en y ou en x doit s’incrémenter.

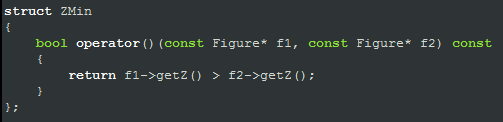
Nous utilisons le même principe pour dessiner un cercle, il suffit de tracer, par le même algorithme, les arcs de cercle dans tous les octants, puis de relier chaque octant. Ce faisant, nous évitons une duplication de mêmes coordonnées, ce qui serait gênant pour l’application de la transparence.

### Classe « Draw »

La dernière partie consiste à créer une classe qui lira le fichier vectoriel, le traduira, dessinera les figure souhaitées et enregistrera l’image au format bmp. Le diagramme suivant permet de connaître comment cette classe se comporte :



La solution retenue afin de dessiner l’image est de construire toutes les figures avant de les dessiner, une par une. Nous définissons donc une priority\_queue de ‘Figure\*’. En effet, nous pouvons, grâce à l’héritage, stocker un pointeur d’une classe fille dans un pointeur de sa classe mère. Il faut également redéfinir l’opération (), qui sera appelée afin d’ordonner les figures. Ceci ce fait de la façon suivante :



Ainsi, à chaque fois qu’un élément s’ajoutera dans le vecteur, il viendra se placer à se place en fonction de son ‘z’, son plan par rapport aux autres figures.

La première tâche à faire est de lire le fichier vectoriel. Ceci se fait de la même façon que précédemment. Il faut ensuite le traduire et construire les figures. Nous utilisons la fragmentation vue précédemment sur les string : en effet, le format pour décrire une figure est strict et nous pouvons donc identifier des séparateurs simples. Il suffit ensuite, en fonction du nombre d’arguments donnés à la figure, de choisir le bon constructeur surchargé.

Enfin, nous dessinons, tant que le vecteur de figures n’est pas vide, l’élément le plus haut. Puis nous devons détruire cette figure, puis l’enlever du vecteur. En itérant ainsi jusqu’à la fin, l’image est créée de la façon décrite par le fichier vectoriel.

Pour construire l’image ainsi que le bitmap, nous avons dû créer des pointeurs. Ainsi, dans le destructeur de cette classe, nous devons les détruire afin de ne pas remplir la mémoire à chaque appel à cette classe.

# Dimension graphique

### Présentation des fenêtres

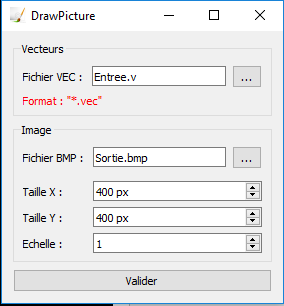
Nous allons utiliser, pour ce logiciel, la suite graphique portable Qt. Nous allons dans un premier temps construire une classe héritant d’une QApplication. Ainsi, cette classe personnalisée permettra d’afficher des fenêtres comme nous le souhaitons, à partir des méthodes propres à une QApplication.

Nous allons, afin de respecter le cahier des charges, créer deux fenêtres distinctes : l’une permettra de sélectionner les fichiers vectoriel et bitmap, ainsi que les paramètres de l’image, l’autre servira à afficher la description vectorielle, les couleurs disponibles, ainsi que l’image crée. Nous pourrons ainsi modifier l’image à la volée, afin de dessiner facilement.

Nous n’avons pas prévu de destructeur pour cette clase car toute la mémoire allouée à ce programme sera libérée lors de la fin de ce dernier.

### Choix des fichiers

La première fenêtre hérite d’un QWidget. Il sera, par conséquent, possible de l’instancier facilement dans l’application, grâce à la méthode show(). La fenêtre ainsi crée est montrée ci-après :

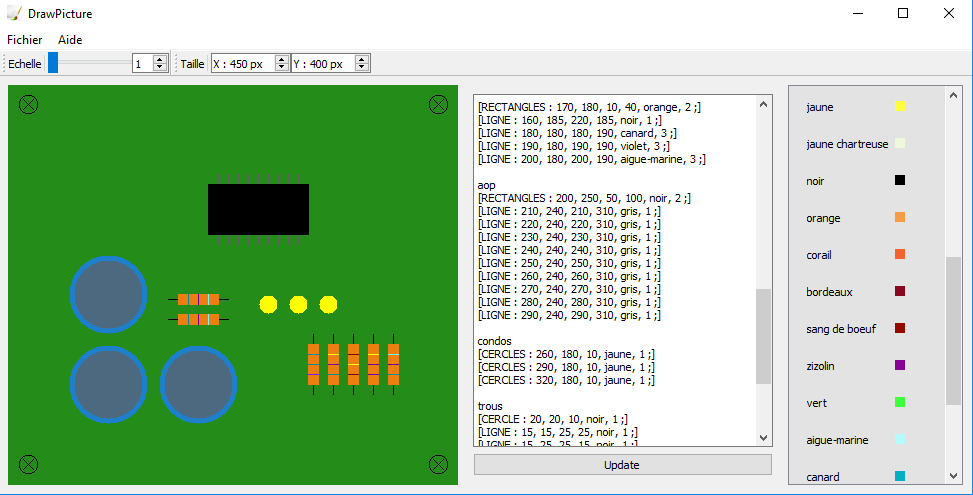


Nous pouvons apercevoir, dans un layout vertical, deux QGroupBox, qui héritent de QWidget également et un QPushButton afin de valider les paramètres entrés. Dans chaque QGroupBox, nous avons mis un layout du type QFormLayout, idéal pour entrer différents champs. Ce layout contient un autre QWidget contenant dans un layout horizontal un QLineEdit, ainsi qu’un QPushButton qui permettra, lors de l’émission d’un signal, d’ouvrir une fenêtre spécifique afin de sélectionner un fichier ‘\*.vec’ ou ‘\*.bmp’. Si l’extension n’est pas correcte, le slot permettant de valider les paramètres fera apparaître un message d’erreur, sous forme d’un QLabel. De plus, dans le second champ, nous avons inclus des QSpinBox, permettant d’entrer des entiers au programme.

Enfin, nous avons connecté le signal d’activation du bouton ‘Valider’ à un slot personnalisé, permettant soit d’afficher un message d’erreur, soit de charger les différents paramètres en mémoire et de cacher la fenêtre. Néanmoins, nous ne la détruisons pas. Elle prendra de la place en mémoire mais sera également plus rapide à afficher à nouveau dans le cas où nous décidons de charger d’autres fichiers.

### Fenêtre principale

La fenêtre principale sera composée comme montré ci-après :



Elle hérite d’un QMainWindow, permettant de gérer facilement plusieurs emplacements. Un menu est créé. Nous verrons par la suite son fonctionnement. De plus, nous avons défini deux barres d’outils, l’une permettant de changer l’échelle de l’image grâce à un QSlider et une QSpinBox, connectés entre eux, l’autre de changer à la volée la taille de l’image.

La zone principale est constituée d’un layout horizontal. Nous plaçons d’abord l’image créée dans un QLabel. Elle sera remise à une échelle de 1 afin que l’affichage ne dépasse pas l’écran. Ensuite, un QTextEdit permet d’éditer le fichier vectoriel facilement. Le fichier est entièrement chargé à partir du fichier texte avant d’être placé dans ce widget. Enfin, nous chargeons, lors de la construction de la fenêtre, les différentes couleurs disponibles, que nous disposons ensuite dans des QLabel. Afin de rendre l’utilisation plus facile, il est possible de sélectionner et copier les noms des couleurs. Enfin, nous avons placé un QPushButton qui sera connecté à un slot personnalisé. Ce slot permet d’enregistrer le fichier vectoriel, puis de dessiner l’image bitmap.

### Barre de menu

La barre de menu aura deux menus différents : l’un, nommé ‘Fichier’, permet d’afficher la fenêtre de sélection de fichier vu précédemment ainsi que de fermer le programme. Ces dernières actions peuvent également être réalisées grâce aux raccourcis clavier, respectivement ‘ctrl+n’ et ‘ctrl+q’.

Le second, nommé ‘Aide’, permet d’afficher une nouvelle fenêtre, contenant un QTextDocument. L’avantage de ce widget est qu’il peut lire des fichiers ‘\*.html’. Ainsi, nous pouvons mettre en forme l’aide de la manière que nous souhaitons.

# CONCLUSION

Lors de l’analyse de la mémoire allouée à ce programme en fonctionnement, nous nous apercevons que, même lors de plusieurs dessins à la suite, cette dernière reste constante. Nous en déduisons donc que l’allocution et la désallocution est faite correctement lors du fonctionnement normal du programme.

L’avantage du C++, par rapport au C, est de construire des boîtes noires, dont uniquement certaines méthodes sont appelables hors d’une classe. Ceci permet moins de flexibilité à l’utilisateur car il ne peut pas accéder directement aux attributs, mais ainsi ces derniers sont protégés. Par conséquence, l’architecture du programme est plus claire et lisible et les classes ainsi créées sont facilement portables. Un autre avantage indéniable du C++ est la définition, dans la librairie standard, de plusieurs classes, associées à plusieurs méthodes, permettant de manipuler des objets facilement, tel que les string. Néanmoins, le C++ étant de plus haut niveau que le C, nous ne pouvons pas maîtriser totalement la mémoire.

# C:\Users\Florian\Documents\Cours\ENSEIRB\2ème année\2ème semestre\C++\Projet\CPP(4) (1).pngAnnexe 1 : UML des figures