Cahier des spécifications.

April 15, 2019

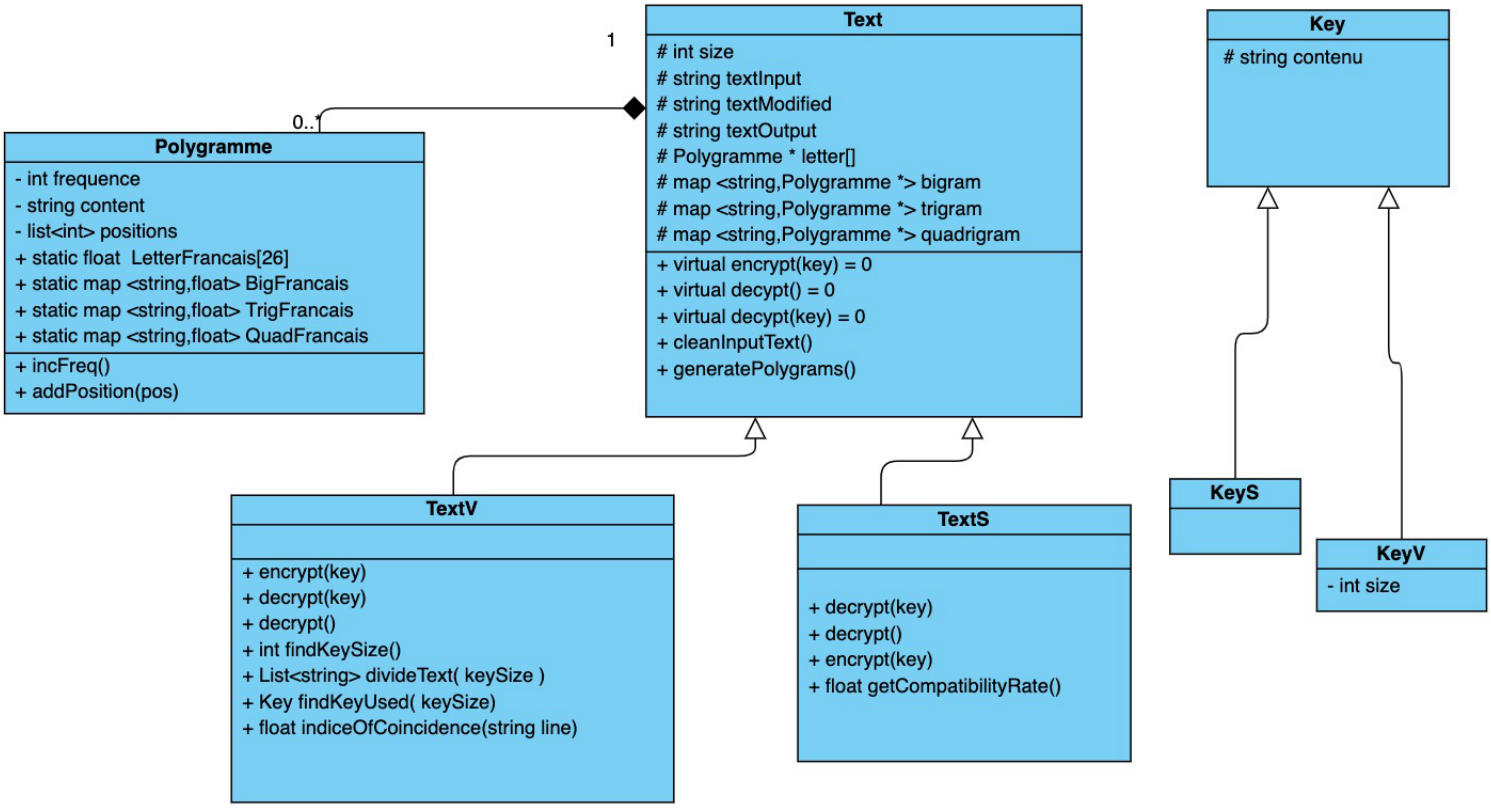
1 Introduction :

Le cahier des spécifications et le cahier des charges sont les deux rebords conceptuels d'un projet informatique qui une fois finalisés, laissent place à l'implémentation, et l'organigramme quant à lui sert de passerelle entre ces deux documents.
Après avoir achevé le cahier des charges de notre projet final d'étude en troisième année de licence intitulé outil automatique d'aide au décryptage et opter pour le C++ qui est un langage hybride offrant la possibilité d'utiliser l'aspect procédural afin de gérer tout ce qui touche aux routines mathématiques calculatoires et l'aspect orienté objet dans le but de modulariser le code, de l'aérer et d'en faciliter la modification, nous avons poursuivi le chemin de la conception en mettant au point le présent document qui décrira en détails tous les aspects techniques du projet, et ce, de façon modulaire.
Les deux cahiers ont été élaborés par l'ensemble du groupe, chaque membre est par conséquent, parfaitement capable de venir à bout de l'implémentation de n'importe quelle fonctionnalité quoique le travail à d'ores et déjà été départagé.

2 Glossaire des types :

- int : Entier sur 4 octets.
- float : Nombre en virgule flottante qui est représenté sur 4 octets.
- double : Flottant à double précision qui est représenté sur 8 octet .
- string : Chaîne de caractères .
- List $< T >$: Structure de données regroupant des éléments de même type T.
- map$< T1, T2 >$: Tableau associatif clé-valeur.
- void : Mot clé pour indiquer le type null .

3 Diagramme de classe :



Explication du diagramme de classe :

3.1 Classe Text :

La classe Text est la classe principale de notre projet, elle est abstraite, a 7 attributs et 5 méthodes dont 3 qui sont virtuelles pures. cette classe abstraite permet de définir les comportements des méthodes dont l'implémentation se fait dans les classes filles qui devront respecter le contrat défini par la classe mère, en effet, à ce stade on ne connaît pas encore le fonctionnement de ces méthodes tant que l'algorithme à utiliser n'a pas été choisi.

3.2 Classe TextV :

La classe TextV (pour texte Vignère) hérite de Text est la classe qui s'occupe de la manipulation du texte selon le chiffre de Vignère. Cette classe a 8 méthodes qui permettent de chiffrer ou de déchiffrer un texte selon l'algorithme de Vignère mais aussi d'effectuer une cryptanalyse sur les textes chiffrés par l'algorithme de Vignère. Trois des méthodes de cette classe ont un comportement polymorphique car comme cité plus haut TextV hérite de la classe abstraite Text mais à ce stade on connaît le fonctionnement des méthodes car les algorithmes à utiliser sont connus .

3.3 Classe TextS :

La classe TextS (pour texte substitution) hérite de Text est la classe qui s'occupe de la manipulation du texte selon l'algorithme de cryptage et décryptage par substitution. Cette classe a quatre méthodes dont deux qui vont être utilisées pour la cryptanalyse d'un texte chiffré par l'algorithme de chiffrement par substitution. Tout comme avec TextV, les méthodes issues de l'héritage de Text présentent un comportement polymorphique car on connaît désormais les algorithmes utilisés.

3.4 Héritage entre (TextV et Text) et entre (TextS et Text) :

Étant la classe principale de notre projet la classe Text ne peut, à elle seule, s'occuper de tous les sous modules de cryptage et de décryptage. Les différents algorithmes de chiffrement et de déchiffrement seront traités différemment bien qu'ils aient les mêmes attributs et partiellement les mêmes méthodes. Les attributs et les méthodes seront donc divisés en deux groupes, ceux en commun et ceux qui sont propres à chaque type de texte, c'est pourquoi nous avons pensé à l'héritage et faire hériter par conséquent les classe TextV et TextS de la classe Text.

3.5 Classe Polygramme :

La réalisation de notre logiciel nécessite la réalisation du sous-module décryptage sans clé, hors, celui-ci a besoin d'une classe essentielle qui est la classe polygramme cette dernière a six attributs dont trois qui sont static et deux méthodes. Elle a pour rôle de stocker tous les polygrammes présents dans le texte ainsi que leurs fréquences cette classe est un composé de la classe Text en effet notre texte est composé de polygrammes, la suppression de l'objet Text entraine la suppression de l'objet Polygramme.

3.6 Classe Key :

La classe Key manipule la donnée la plus sensible de notre projet qui est la clé, elle est le seul attribut de cette classe. Deux classes filles héritent de cette classe mère les classes KeyS et KeyV.

3.7 Classe KeyS :

La classe KeyS hérite de Key mais celle-ci dispose d'un constructeur lui permettant de générer une clé aléatoirement.

3.8 Classe KeyV :

KeyV est une classe qui hérite de Key, à la différence près qu'elle a un attribut supplémentaire qui est la longueur, car dans le chiffrement de Vignère la taille de cette dernière n'est pas fixe. Elle dispose aussi d'un constructeur lui permettant de générer une clé aléatoirement.

3.9 Encapsulation :

Vu qu'il existe des attributs délicats dans notre projet, l'accès à ces attributs ou bien leur modification n'importe comment risquerait fortement de compromettre les résultats finaux. Nous avons donc opté pour une encapsulation forte sauf dans les cas où les classes filles auront besoin d'accéder à ces attributs délicats, dans ces derniers cas, nous avons opté pour une encapsulation faible.

4 Elements en commun:

Vu que la majeure partie des attributs et des méthodes seront utilisés par la quasi-totalité des modules, nous allons commencer par citer tous les attributs et méthodes communs, nous listerons par la suite les méthodes propres à chaque module.

4.1 Attributs et structures de données:

4.1.1 Classe Text:

int size : Un entier contenant la longueur du texte.
string textInput :Une chaîne de caractères qui contient soit un texte clair soit un texte cryptée entrée par l'utilisateur via l'interface graphique ou bien à partir d'un fichier local.

String textOutput :Une chaîne de caractères contenant le Texte après traitement, c.à.d chiffré ou bien déchiffré .
String textModified : Une chaîne de caractères contenant le texte à crypter entré par l'utilisateur sans espaces ou caractères spéciaux.

Polygramme* letter[26] : Un tableau de pointeurs vers les polygrammes qui représentent les 26 lettres de l'alphabet.

map<string,Polygramme *> bigram : Un tableau associatif contenant tous les digrammes avec les informations liées à chaque digramme.
map<string,Polygramme *> trigram : Un tableau associatif contenant tous les trigrammes avec les informations liées à chaque trigramme.

map<string,Polygramme *> quadrigram : Un tableau associatif contenant tous les quadrigrammes avec les informations liées à chaque quadrigramme.

4.1.2 Classe Key:

String content : Contenu de la clé qui est une chaîne de caractères.

4.1.3 Classe KeyV:

int size : Attribut contenant la longueur de la clé de type Vigenère.

4.1.4 Classe Polygramme:

int frequence: Variable entière qui représente le nombre d'occurrences d'un polygramme dans le texte.

string content : Une chaîne de caractères contenant le polygramme en lui même.

list<int> positions : Une liste de type entier qui permet de stocker toutes les positions d'apparition du polygramme dans le texte.

static const float LettreFrancais[] : Un tableau qui contient la fréquence des 26 lettres de l'alphabet de la langue française.

Static const map<string,float> BigFrancais : Une map qui permet de stocker les digrammes les plus fréquents dans la langue française en associant à chacun sa fréquence.

static const map<string,float> TrigFrancais : Une map qui permet de stocker les Trigrammes les plus fréquents dans la langue française en associant à chacun sa fréquence.

static const map<string,float> QuadFrancais : Une map qui permet de stocker les quadrigrammes les plus fréquents dans la langue francaise en associant à chacun sa fréquence.

Les attributs précédents sont static car ils ne dépendent pas d'une instance de Polygramme mais de la classe Polygramme elle-même.

4.1.5 Structures de données utilisées:

float NomDuTableau[] : Dans la classe texte, nous avons opté pour un tableau de type float afin de pouvoir stocker les fréquences des lettres de la langue française qui sont connues (26 lettres alphabétique).

list< T > : Dans notre projet l'utilisation des listes au lieu de tableaux revient au manque d'informations préalable sur la taille de la structure.

map<string,Polygramme *> NomDeLaTable : Pour la sauvegarde et la manipulation des polygrammes, nous avons

opté pour des tableaux associatifs car on sera amené, lors du décryptage, à rechercher et à insérer les différents polygrammes et leurs fréquences et pour ce faire de manière optimal, l'utilisation de tableau associatif est la meilleure option.

4.2 Les constructeurs:

KeyS() : Constructeur par défaut de la clé qui assure la génération d'une clé aléatoire de 26 caractères.

KeyS(string contenu) : Constructeur de la clé à partir d'une chaîne de caractères passée en paramètre.

KeyV() : Constructeur par défaut de la clé qui assure la génération d'une clé aléatoire à partir d'un dictionnaire de mots afin d'avoir un mot qui a un sens.

KeyV(string contenu) : Constructeur de la clé à partir de la chaîne de caractères passée en paramètre.

TextS(string text) : Constructeur du texte à manipuler selon le principe de substitution qui stocke le text dans l'attribut textInput.

TextV(string text) : Constructeur du texte à manipuler avec la méthode de Vigenère qui stocke le text dans l'attribut textInput.

Polygramme(string contenu) : Constructeur de la Classe Polygramme qui prend en paramètre une chaîne de caractères qui représente le polygramme (de taille 1,2,3 ou 4 pour Lettre,Digramme,Trigramme ou Quadrigramme respectivement).

4.3 Les destructeurs:

TextS::∼TextS() : Destructeur par défaut de la classe TextS.

TextV::∼TextV() : Destructeur par défaut de la classe TextV.

KeyS::∼KeyS() : Destructeur de la classe KeyS.

KeyV::∼KeyV() : Destructeur de la classe KeyV.

4.4 Méthodes d'accès:

string Key::getContent() : Méthode qui retourne le contenu de la clé.

int KeyV::getSize() : Méthode qui retourne la longueur de la clé de type Clé Vigenère.

int Text::getSize() : Méthode qui retourne la longueur du Texte.

string Text::getTextInput(): : Méthode qui retourne le Texte entré par l'utilisateur.

string Text::getTextOutput() : Méthode qui retourne le texte après cryptage ou décryptage.

string Text::getTextModified() : Méthode qui retourne le texte après la suppression des espaces et des caractères spéciaux.

string Text::setTextOutput() : Méthode qui modifie l'attribut OutputText.

5 Cryptage

5.1 Bibliothèques utilisées:

<string> :Dans ce module nous manipulons des chaînes de caractéres et la bibliothéque string nous facilite cette manipulation.

5.2 Classe Text:

virtual void Text::cleanText();
Description: Méthode de la classe Text qui récupère le texte stocké dans l'attribut TextInput afin de supprimer les espaces, la ponctuation, les caractères spéciaux et qui transforme les lettres en majuscules, puis le stocke dans l'attribut TextModified, ceci nous facilitera le cryptage ou la cryptanalyse.

virtual void Text::encrypt(Key)=0;
Description: Méthode virtuelle qui permet le chiffrement de TextModified, son comportement dépend du type de cryptage choisi par l'utilisateur (Vigenère ou Substitution monoalphabétique) ,de ce fait cette méthode ne peut être définie dans la classe Text.
Paramétres:
Key: l'objet Key qui sert à crypter le texte.

5.3 Classe TextS:

void TextS::encrypt(Key);
Description:Redéfinition de la méthode virtuelle de la classe abstraite Text,qui permet la récuperation du contenu de l'attribut TextModified, et de le parcourir lettre par lettre afin de le crypter par substitution monoalphabétique en remplaçant chaque lettre par celle qui la désigne dans la clé passée en paramètre, le résultat sera stocké dans l'attribut TextOutput.
Paramétres:
Key: l'objet Key qui sert à crypter le texte.

5.4 Classe TextV:

void TextV::encrypt(Key);
Description: Redéfinition de la méthode virtuelle de la classe mère Text dans la classe fille TextV qui permet de récupérer le contenu de l'attribut TextModified, afin de le crypter par la méthode de Vigenère en utilisant la clé passée en paramètre, le resultat est stocké dans l'attribut textOutput.

6 Décryptage

6.1 Bibliothéques :

<list> : cette bibliothèque va nous permettre de manipuler les listes chaînées.
<cmath> : cette bibiothèque va nous permettre d'utiliser les formules mathématiques dont on a besoin dans le calcul du taux de compatibilité.
<string>: cette bibliothèque va nous permettre de manipuler les chaînes de caractères.

6.2 Classe Text:

virtual void Text::Decrypt(Key cle)=0;
virtual void Text::Decrypt()=0;
Description: Etant donne que le comportement de ces deux méthodes depend de l'algorithme à utiliser (Vigenère, Substitution), ce comportement ne peut être defini dans la classe abstraite Texte. Ces deux méthodes seront donc redefini dans les classes filles Text Vigenère ,TextSubstitution afin de fonctionner correctement selon le type de traitement (Vigenère, Substitution).
Text::generatePolygrams();
Description : avant de déchiffrer un texte sans la possession de la clé, cette méthode est appelée afin d'instancier et de structurer les polygrammes qui apparaissent dans le texte chiffré.

6.3 Classe TextVigenere:

void TextVigenere::Decrypt(Key cle);
Description: le rôle de cette méthode consiste à récuperer le texte contenu dans TextInput,afin de le décrypter en utilisant la clé qui lui est envoyée comme paramètre et de stocker le résultat dans Text Output.
Paramètres:
Key clé: La clé saisit par l'utilisateur qui est envoyée depuis l'interface graphique à cette méthode.

list< *string* > TextVigenere::divideTexte(int sizeOfKey);
Description: cette méthode à pour rôle de garantir l'une des étapes de la cryptanalyse de Vigenère, qui consiste à regrouper toutes les parties du texte qui ont été cryptées en utilisant le même caractère de la clé en une seule partie, et ce en se basant sur la longueur de la clé qui est passée comme paramètre.
Paramètres:
int sizeOfKey: Ce paramètre sert à déterminer le nombre de parties pour lesquel le texte doit être divisé.
(car si la clé est de longueur n on doit découper le texte en n parties de telle sorte à ce que chacune des parties est censée être cryptée en utilisant le même caractère).
Retour:
list< *string* > : une liste de chaînes de caractères ou chaque chaîne de la liste correspond à une partie du texte qui a été cryptée en utilisant le même caractère de la clé, l'utilisation d'une liste chainée revient au manque d'informations préalable sur le nombre de chaînes de caractères.

float TextVigenere::indiceOfCoincidence(string line);
Description: cette méthode calcule l'indice de coïncidence d'une chaîne de caractères .
l'indice de coïncidence permet de détecter si une chaîne de caractères a été chiffrée en utilisant un chiffrement monoalphabetique ou bien un chiffrement polyalphabetique.
si on calcule l'IC d'une chaîne de caractères chiffrée par un chiffrement mono alphabétique, on devrait trouver IC égal environ à 0.074 (en français). Si l'IC est beaucoup plus petit (p. ex-0.050), le chiffré est probablement polyalphabétique.
Paramétres:
string : une chaîne de caractères qui correspond à une partie du texte qui est censée être cryptée en utilisant le même caractère de la clé.
Retour:
float : un réel qui correspond à l'indice de coïncidence de la chaîne passée en paramètre.

int TextVigenere::findKeySize();
Description: cette fonction est chargée de trouver la longueur de la clé qui a servi lors du cryptage de TextInput et ce en analysant le placement des différents polygrammes, cette fonction fait appel à la fonction indiceofcoincidence() afin de déterminer la longueur la plus probable à partir d'une liste de longueurs possibles.

Retour:
int : l'entier qui représente la longueur de la clé utilisée pour crypter le texte.

Key TextVigenere::findKeyUsed(int keySize);

Description: le rôle de cette méthode consiste à trouver la clé qui a servi lors du cryptage et ce en se basant sur la longueur de celle-ci et en utilisant l'attaque statistique sur chaque partie cryptée en utilisant le même caractère de la clé.

Retour:
Key : l'objet Key qui représente la clé qui a servi lors du cryptage du texte et c'est cette même clé va nous permettre de décrypter le texte.

void TextVigenere::Decrypt();
Description: cette fonction utilise les fonctions expliquées précédemment afin de récupérer TextInput et de le décrypter, ensuite le stocker dans Text Output.

6.4 Classe TextSubstitution:

void TextSubstitution::Decrypt(Key cle);
Description: Permet de décrypter un texte déjà crypté par une substitution des lettres, cette fonction utilise le paramètre clé et l'attribut TextInput et stocke le résultat dans l'attribut TextOutput.
Paramètres:
Key clé: La clé saisit par l'utilisateur qui est envoyée depuis l'interface graphique à cette méthode.

void TextSubstitution::Decrypt();
Description: Afin d'avoir un texte clair par une cryptanalyse d'un texte chiffré, cette méthode utilise l'attaque statistique sur les différentes parties du texte tout en faisant appel à la fonction getcompatibilityrate() afin de trouver le texte décrypté le plus cohérent, et ensuite elle stocke le résultat dans attribut textOutput.

float TextSubstitution::getCompatibilityRate();
Description: Cette fonction calcule le taux de compatibilité entre TextOutput et la langue française, et c'est cette méthode qui va aider à détecter si le texte résultant est cohérent ou pas.

6.5 Classe Polygramme:

list< *int* > Polygramme::getPositions();
Description: cette fonction retourne les positions du polygramme dans le texte afin de calculer la distance qui les sépare dans le but d'extraire la longueur de la clé d'un texte chiffré.
Retour:
list<int> : une liste chainée contient les positions du même polygramme dans le texte, l'utilisation d'une liste chaînée revient au manque d'information préalable sur la taille de cette structure.

void Polygramme::incFreq();
Description: Lors du parcours du texte à chaque reconnaissance d'un polygramme existant, cette méthode est appelée pour incrémenter le nombre d'apparition de ce polygramme.

int Polygramme::getFrequency();
Description: Cette méthode retourne le nombre d'apparition du polygramme dans le texte.
Retour:
int : le nombre d'apparition du polygramme dans le texte.

void Polygramme::addPosition(int pos);
Description: Permet d'ajouter une position du polygramme à la liste des positions.
Paramètres:
int pos: la position à ajouter qui égale au nombre de caractères avant le polygramme en question.

7 Gestionnaire de fichiers :

Le gestionnaire de fichiers permettra à l'utilisateur de charger un texte à partir d'un fichier externe et d'exporter le résultat obtenu dans un autre fichier en respectant certaines régles de mise en page, ainsi qu'une structure définie. Ce module est en interaction avec le module Interface graphique, afin de faciliter les opérations chargement-sauvegarde à l'utilisateur.

7.1 Fonctions de sauvegarde des textes :

Les méthodes du gestionnaire de fichiers sont :

void exportTxt (char* text, char* chemin, char* nomFichier);
Description : Cette fonction va permettre à l'utilisateur de sauvegarder le résultat obtenu dans un fichier de type "TXT". Pour se faire, cette fonction prendra en entrée le texte à sauvegarder dans une variable de type chaîne de caractères, ainsi que le nom que l'utilisateur aura affecté à ce fichier en précisant le chemin du dossier où il souhaite le sauvegarder, ces deux informations doivent être également de type chaîne de caractères. À l'aide de la bibliothèque "fstream", nous allons pouvoir, dans cette fonction, ouvrir un fichier de type txt en mode écriture, écrire le résultat et le fermer ensuite. Si le nom du fichier existe déjà ou le chemin choisi n'existe pas, l'opération échouera, et de nouvelles valeurs seront demandées. Si l'opération se termine avec succès la valeur Vrai sera retournée par cette fonction comme variable booléenne, la valeur faux est retournée sinon.

void exportPdf (char* text, char* chemin, char* nomFichier);
Description : Cette fonction prend en entrée le résultat du cryptage ou décryptage, le nom à attribuer au fichier et le chemin de sauvegarde desiré dans des variables de type chaîne de caractères. À l'aide de la bibliothèque "fstream" ainsi que la bibliothèque "conio.h", nous allons pouvoir, dans cette fonction, ouvrir un fichier de type "MS WORD" en mode écriture, écrire le resultat en suivant une certaine structure et le fermer par la suite. Si le nom du fichier existe déjà, ou le chemin choisit n'existe pas, l'opération échouera, et de nouvelles valeurs seront demandées. Si l'operation se termine avec succès la valeur Vrai sera retournée par cette fonction comme variable booléenne, la valeur Faux sera retournée sinon.

void exportWord(char* text, char* chemin, char* nomFichier);
Description : Cette fonction sauvegardera un texte traité dans un fichier de type "PDF", et ceci se réalisera à l'aide de la bibliothèque "Haru" qui nous permettra d'écrire et sauvegarder dans un fichier pdf. Cette fonction aura besoin de trois chaînes caractères dont une pour attribuer un nom au fichier à sauvegarder, une deuxième pour préciser le chemin où l'utilisateur souhaite le sauvegarder, et une dernière pour designer le texte à exporter. Egalement, Si jamais le nom du fichier existe déjà, ou le chemin choisit n'existe pas, l'opération échouera, et de nouvelles valeurs seront demandées. Si l'opération se termine avec succès la valeur "Vrai" sera retournée par cette fonction comme variable booléenne, la valeur "faux" sera retournée sinon.

7.2 Fonctions de chargement des textes :

char* importTxt(char* chemin);

Description : Cette fonction va permettre à l'utilisateur de charger un texte à crypter ou bien préalablement crypter pour le décrypter depuis un fichier de type "TXT". Pour se faire, cette fonction prendra en entrée le chemin du fichier de type char* contenant le texte à charger. À l'aide de la bibliothèque "fstream", nous allons pouvoir ouvrir un fichier de type txt en mode lecture, lire le résultat, le stocker dans une variable de typer char* qui sera retournée par cette fonction et le fermer ensuite.

char* importWord(char* chemin);

Description : Cette fonction prend en entrée le chemin du fichier de type "MS WORD" de typer char* contenant le texte à crypter ou bien à décrypter. Grace à la bibliothèque "fstream" ainsi que la bibliothèque "conio.h", nous allons pouvoir ouvrir un fichier de type "MS WORD" en mode lecture, lire le résultat le stocker dans une variable de typer char* qui sera retournée par cette fonction et le fermer ensuite.

char* importPdf(char* chemin);

Description : Tout comme les deux fonctions précédentes, cette fonction chargera un texte à crypter ou bien préalablement crypté pour le décrypter mais cette fois ci depuis un fichier de type "PDF", et ceci se réalisera grâce à la bibliothèque "PODOFO" qui nous permettra de lire et d'importer un texte depuis un fichier "PDF". Cette fonction, aura besoin du chemin du fichier de type "PDF" de typer char* contenant le texte à charger.

8 Fenêtre graphique :

1 Définition :

L'interface graphique est le module qui permet à l'utilisateur de manipuler et d'intéragir avec le logiciel en choisissant d'abord soit de crypter ou de décrypter un texte en fournissant ou pas la clé de chiffrement, pour ensuite lancer l'un des processus cryptographiques que propose l'outil, et avoir enfin la possibilité de visualiser s'il le souhaite le déroulement des différents processus en temps réel. L'interface sera implémenté intégralement à l'aide du Framework multi-plateforme *Qt* qui offre une multitude d'objets et de mécanismes servant à produire des *GUI* agréables, intuitives et faciles d'accès. Pour se faire, l'environnement de développement QtCreator sera utilisé.

2 Les classes :

2.1 Définition :

Toutes les classes hériterons de la classe *QWidget* et contiendront la macro *Q_OBJECT* afin d'utiliser le mécanisme des signaux et des slots [voir 5].

2.2 Liste des classes (fenêtres):

1. **Manwindow** : Fêtre principale permettant à l'utilisateur de choisir de crypter ou de décrypter un texte.
2. **Encryptwindow** : Fenêtre permettant le chiffrement d'un texte.
3. **Decryptwindow** : Fenêtre permettant le déchiffrement d'un texte.
4. **Processwindow** : Fenêtre permettant de suivre en temps réel les différents processus.
5. **Resultwindow** : Fenêtre permettant d'afficher et de modifier un texte résultat.

3 Attributs :

3.1 Définition :

La totalités des attributs de ce module sont des objets de classe de la bibliothèque Qt, en effet une fenêtre graphique est un objet composé de différents autres objets; en d'autres termes, un *widget* principal est un conteneur des autres *widgets* qui compose ce tout visuel qu'est la fenêtre.

3.2 Les principaux objets utilisés :

- **QPushButton** : instancie comme son nom l'indique un bouton cliquable.
- **QTabWidge**t : instancie un nombre à définir de sous-conteneurs.
- **QGridLayout** : instancie une grille de layouts permettant de placer les différents objets sur la fentre.

- **QWidget** : instancie un conteneur.
- **QLineEdit** : instancie un *input* de type texte contentant une seule ligne.
- **QTextEdit** : instancie un *input* de type texte contentant plusieurs lignes.
- **QComboBox** : instancie une liste déroulante.
- **QChekBox** : instancie une case à cocher.
- **QProgressBar** : instancie une barre de progression.
- **QVBoxLayout / QHBoxLayout** : instancie un box d'alignement vertical ou horizontal des layouts.
- **QSlider** : instancie un curseur modifiable.

4 Les méthodes :

On utilisera en dehors des slots[voir 5] exclusivement les méthodes de classes qu'offre Qt et les constructeurs prendront comme arguments tous les objets placés à l'intérieur de la fenêtre, quant aux destructeurs ils ne seront pas implémentés car Qt gère les objets en mémoire et les *delete* automatiquement.

Exemples :

- **void setGeometry(int x, int y, int w, int h)**
- **int maximumWidth() const**
- **int minimumHeight() const**
- **void setFixedSize(int w, int h)**
- **void addLayout(QLayout * layout, int row, int column, int rowSpan, int columnSpan, Qt::Alignment alignment = 0)**
- **void addWidget(QWidget * widget, int row, int column, Qt::Alignment alignment = 0)**

5 Mécanisme des signaux et des slots:

5.1 Définition :

On utilisera tout au long de l'implémentation le mécanisme de *signaux* et de *slots* permettant de récupérer un signal émis suite au déclenchement d'un évènement et de l'envoyer avec un slot qui n'est rien d'autre qu'une méthode classique à la méthode de la classe **QObject *connect*** dont la signature est :

connect(const QObject *sender, const char *signal,
const char *method, Qt::ConnectionType type = Qt::AutoConnection) const

Remarque : *des signaux et des slots par défauts existent. On n'utilisera que les signaux par défauts, les slots eux seront implémentés pour combiner plusieurs actions au sein d'un même slot.*

Exemple :

1. Choisissons par exemple le signal par défaut "clique du boutton Decrypt" comme suit SIGNAL(clicked()).
2. On envoie ce signal à la méthode QObject::connect.
3. On choisit une méthode qui sera présentée comme slot, par exemple celle qui permettra de passer de la fenêtre decryptwindow à la fenêtre suivante processwindow tel que SLOT(cypher).

4. Dès que le bouton Decrypt sera cliqué la méthode connect prendra ses arguements et s'exécutera comme QObject::connect(Decrypt, SIGNAL(clicked()), this, SLOT(encypher())).

5.2 Les slots crées :

- **void mainwindow::cypher();** : slot permettant d'aller vers la fenêtre de chiffrement.
- **void mainwindow::encypher();** : : slot permettant d'aller vers la fenêtre de déchiffrement.
- **void encryptwindow::start_encrypt();** : slot permettant d'aller soit vers la fenêtre de déroulement soit vers la fenêtre résultat.
- **void encryptwindow::back_encrypt();** : slot permettant de revenir vers la fenêtre de d'accueil.
- **void decryptwindow::start_decrypt();** : slot permettant d'aller soit vers la fenêtre de déroulement soit vers la fenêtre résultat.
- **void decryptwindow::back_decrypt();** : slot permettant de revenir vers la fenêtre de d'accueil.
- **void processwindow::result();** : slot permettant d'aller vers la fenêtre de résultat.
- **void resultwindow::back_result();** : slot permettant de revenir vers la fenêtre de d'accueil.
- **void resultwindow::save();** : slot permettant d'enregistrer le texte dans un fichier externe.

6 Bibliothèques :

À fin d'utiliser tous les éléments cités plus haut, on devra inclure dans nos classes des headers qui portent les mêmes noms que les objets utilisés. Par exemple, afin d'utiliser des boutons dans la fenêtre, il faudra inclure au préalable <**QPushButton**>. Il faudra aussi inclure dans les classes les headers des fenêtres vers lesquelles elles peuvent basculer à l'aide d'un slot.

7 Utilisation et fonctionnement :

7.1 Fonctionnement :

1. L'utilisateur aura tout d'abbord en visu la fenêtre principal [ref] ou il pourra choisir de crypter ou de décrypter un texte en appyant sur l'un des deux boutons Encrypt / Decrypt.
2. Il basculera ensuite selon son choix vers l'une des deux fenêtre [ref] et [ref] et il pourra selon son choix crypter ou décrypter un texte :
   - La zone 1 désigne l'endroit dans lequel le texte pourra être tappé à l'aide du clavier.
   - Le boutton 2 permet de charger un texte à partir d'un fichier.
   - L'élement 3 est une liste déroulante avec laquelle l'utilisateur pourra choisir l'un des deux algorithmes (Vigenere, Substitution).
   - L'élément 4 est selon le choix d'utilisateur la zone d'insertion de la clée si aucune clée n'est introduite le processus se fera à l'aide d'une clée générée aléatoirement.
   - L'élément 5 est une case à coché permettant de choisir de visualiser ou pas les différents processus.
   - L'élements 6 est finalement le boutton qui selon le choix de l'utilisateur chiffrera ou déchiffrera le texte introduit.

3. Selon si la case à coché l'est ou pas l'utilisateur basculera vers la fenêtre du déroulement du processus qui affichera en temps réel les différents étapes, sinon il passera directement à la fenetre résultat qui affichera le texte après processus. L'utilisateur aura la main à ce moment là pour modifier s'il le souhaite son texte dans un fichiers externe et/ou de revenir à la fenêtre d'acceuil.

7.2 Utilisation :

L'utilisateur aura accès à une page d'aide et à une page réglages dans laquelles il pourra par exemple modifier les couleurs de la fenêtre ou encore enregistré une clée par défaut. Ces deux pages seront implémenté dans deux sous-conteneurs à l'intérieurs des fenêtres de chiffrement et de déchiffrement.

9 Conclusion

Pour chaque module et fonctionnalité de notre application, nous avons essayé de déterminer une structure optimale à l'implémentation. Nous avons commencé par élaborer une première version de notre diagramme de classe munie d'un nombre officieux de méthodes et d'attributs complétés et affinés au fur et à mesure jusqu'à obtention d'une version finale présentée dans ce document.
L'étape la plus ardue fut de décider quels types de structures de données nous allions utiliser. En effet, il fallait qu'elles soient les plus adéquates à nos besoins et qu'elles assurent parfaitement la cohérence des informations circulantes entre les différents modules. Ce choix est crucial et nous sommes convaincu qu'il impactera indéniablement la fluidité de l'implémentation.
Plusieurs débats passionnés ont vu le jour pendant cette étape, et ont finalement permis de mettre tout le monde d'accord.
À ce stade, nous avons pu identifier les besoins fonctionnels de notre outil dans le cahier des charges, et les nécessités techniques dans le cahier des spécifications, nous allons maintenant entamer l'étape suivante qui est celle de l'implémentation.