1 TP 1 : prise en main de QT

1.1 Première utilisation

On va compiler et exécuter le programme écrit pour QT en ligne de commande pour commencer. Comme vous vous rappelez du cours, vous savez que QT fournit un descripteur de projet (les `.pro`) qui contient (en texte modifiable) les besoins et contraintes de votre application, les noms des fichiers sources, les dépendances...

Un fichier `.pro` est fourni dans l'archive : `premierEssai.pro`, ouvrez-le et identifiez où la source `premierEssai.cpp` apparaît. Pour compiler : `qmake` puis `make` Cela crée normalement un binaire « `premierEssai` », lancé avec « `./premierEssai` » en ligne de commande. Si vous modifiez le source `.cpp` il suffit de refaire un "`make`" pour lancer la compilation. Si vous modifiez l'usage des ressources QT (avec des signaux, des widgets...) il faut relancer `qmake` et `make`.

ce premier exemple ne fait pas grand-chose mais introduit déjà :

— il faut des `include` spécifiques à QT

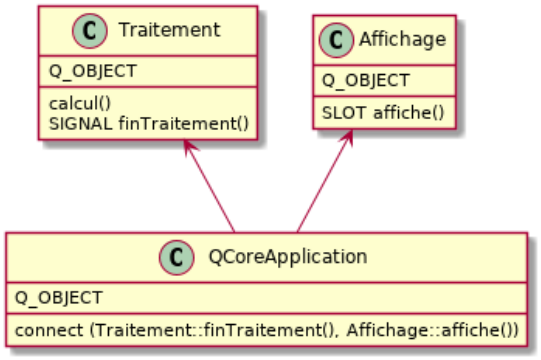
— QT a ses propres types qui sont différents de ceux de C++

Une fois que ce premier exemple est compris, on passe à la suite.

1.2 QT application, signaux, `Q_OBJECT`

On va maintenant construire une application un peu plus complexe, utilisant les `signaux` et `slots`. Pour cela, on utilise 3 objets, une instance d'une application QT (`QCoreApplication`), une classe `Traitement` que l'on va définir et qui contiendra une fonction `calcul()`, une classe `Affichage` qui contiendra une fonction `affiche()`.

On veut les relations suivantes de nos classes :



Les fichiers sources fournis contiennent ces éléments. Si vous ouvrez le fichier `main.cpp`, vous verrez :

— l'instanciation d'une `QCoreApplication`

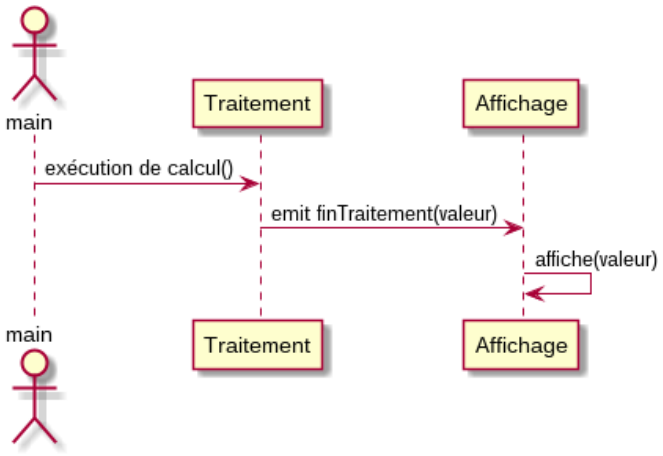
— l'instanciation d'un objet `Traitement`

— l'instanciation d'un objet `Affichage`

— et la boucle de l'application (« `app.exec` »)

On va ensuite connecter la fin de la fonction `Traitement::calcul()` avec le lancement de la fonction `Affichage::affiche()`. On utilise `connect()` dans le `main.cpp` :

```
QObject::connect(&trait, SIGNAL(finTraitement(QString)), &aff, SLOT(affiche(
    QString)));
```



Le diagramme de séquence de ces actions est le suivant :

Il n'y a dans l'exemple de base que le passage d'une information (un `QString` qui vaut "10") depuis la fin de `Traitement::calcul()` vers la fonction `Affichage::affiche()`. En fait aucun calcul n'est effectué, on a juste une preuve que cela fonctionne (un « POC » [1]).

On aurait pu faire comme cela sans signaux (en séquentiel dans un programme) :

```
        std::string resCalcul = trait.calcul();
        aff.affichage(resCalcul);
```

Comme les 2 commandes sont séquentielles, on a le même ordre d'opérations (c'est **synchrone**). Par contre, si on voulait plusieurs processus de traitement ou d'affichage, on devrait utiliser une version **asynchrone** de ces opérations, c'est à celui qui calcule d'avertir l'affichage (ou l'application) qu'il a terminé.

Il faut maintenant :

1. modifier la fonction `Traitement::calcul(unsigned v)` pour qu'elle calcule $v = v+1$. Passer cette valeur via le signal à `Affichage::affiche(QString)` (cette fonction n'est pas modifiée).

   Note : pour créer un `QString` à partir d'un nombre :

   ```
   QString s;
   s.setNum (v);
   ```

2. modifier les déclarations de classes pour refléter ces changements (fichiers .h)

On a donc découplé le calcul et l'affichage dans l'application.

1.3 Exercice final : plus petit, plus grand

Grâce à ce qui a été fait dans cette séance, construire un jeu de « plus petit, plus grand ». Évidemment, on utilisera un maximum de signaux/slots pour faire communiquer des parties de l'application.

Notre application contiendra 2 acteurs : un joueur (`J`) et un maître du jeu (`MdJ`). On utilisera une classe `Affichage` et une `QCoreApplication`.

Le rôle du maître du jeu est de tirer un nombre secret au hasard, de le conserver, d'évaluer les propositions du joueur et de lui renvoyer si le nombre proposé est plus petit, plus grand ou égal au nombre secret.

Les activités de traitement possibles sont visibles sur l'image 1.1.

On va ajouter des messages (signaux/slots) entre les acteurs et les interfaces (`Affichage` et `QCoreApplication`) :

1. `https://fr.wikipedia.org/wiki/Preuve_de_concept`

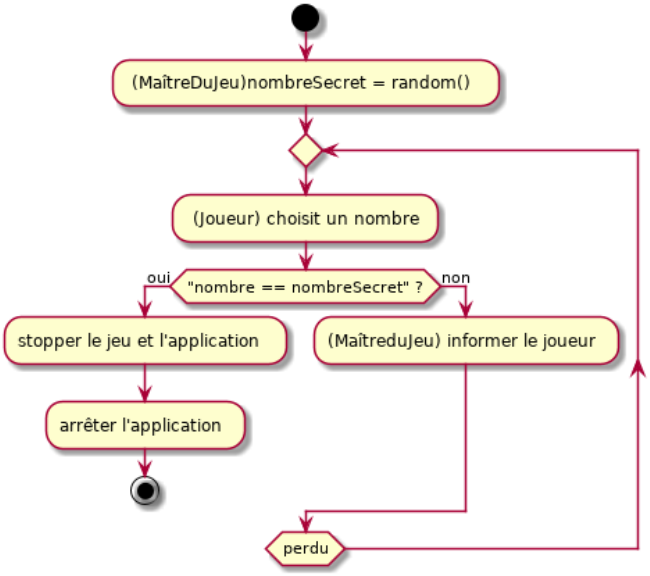
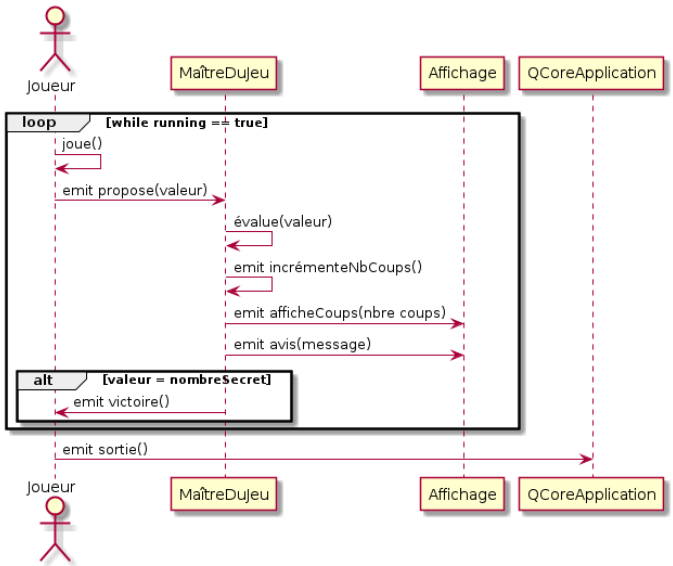


FIGURE 1.1 – activités de traitements

1. `J` ⟶ `MdJ` , émission d'une proposition de valeur. Le `MdJ` compare cette valeur au nombre secret, selon ce traitement :
   — `MdJ` ⟶ `Affichage`, l'avis suite à l'évaluation (« plus petit », « plus grand », « égal »),
   — si le nombre proposé est **égal** au nombre secret, `MdJ` émet en plus un message de victoire vers `J`,
   — `MdJ` ⟶ lui-même, pour incrémenter le nombre de coups et afficher ce nombre : `MdJ` ⟶ `Affichage`,
2. `J` ⟶ `QCoreApplication`, lorsque le jeu est gagné, pour quitter l'application.

Les séquences à implémenter sont :



On stocke :
— chez `MdJ`, le nombre secret et le nombre de coups (des `unsigned`)
— chez `J`, un booléen « `running` » qui donne l'état du jeu (`true`, en cours, `false` arrêté)

À vous de jouer ! (ça veut dire à vous de tout implémenter)

Les en-têtes des classes « Maître du Jeu » et « Joueur »< sont données ici. Le maître du Jeu :

```
class MaitreJeu: public QObject
{
        Q_OBJECT

private:
        unsigned secret;
        unsigned nbcoups;

public:
        //constructeur
        explicit MaitreJeu(QObject *parent = nullptr);

public slots:
        //reçoit et évalue la proposition du joueur
        void evalue(unsigned);

        //ce slot peut être "private" car de MdJ à lui-même
        void incrementationCoups();

signals:
        void avis(QString);
        void incrementeCoups();
        void afficheCoups(QString);
        void victoire();

};
```

Dans l'`Affichage`, pour convertir un `std::string` en `QString` :

```
void Affichage::affiche(QString texte)
{
        //comme d'habitude, il faut récupérer un type C++ compatible
        std::cout << texte.toStdString() << std::endl;
}
```

Et pour le Joueur :

```
class Joueur : public QObject
{
        Q_OBJECT

private:
        //le jeu est il en cours ?
        bool running;

public:
        explicit Joueur(QObject *parent = nullptr); //"explicit" est ajouté par
            QTCreator à la création d'une classe héritant de QObject
        void joue();

signals:
        void propose(unsigned);
        void sortie();

public slots:
        //reçoit le message du MdJ et demande la sortie du jeu
        void gagne();
};
```