# Rapport de projet LO21 – UTC P12

Arthur VAN CEULEN

Qiuyi LI

Vous trouverez en annexe deux UML statiques et deux diagrammes de séquence dynamiques explicitant la partie calculatrice (interface, signaux utilisateur, onglets, historique, log et exception) pour deux d’entre eux, et la partie calcul (pile, constantes, expression, factory de nombres) pour les deux autres.

## Calculatrice

#### Choix de conception

Tout d’abord nous avons décidé de diviser le projet en deux principales parties :

* Interface avec système de log et d’exception
* Moteur de calcul et les expressions

Cette séparation nous paraît logique et nous permet aussi de pouvoir travailler à deux de façon indépendante.

#### A propos de l’UML statique de Calculatrice

La classe « Calculatrice » est implémentée en Singleton pour garantir qu’il y ait qu’une seule instance, et elle est composée de plusieurs « Onglet » qui héritent d’un QWidget. Elle utilise un seul QTabWidget pour contenir les onglets. Donc nous avons créé la méthode getTab() pour récupérer ce QTabWidget et le passer au nouvel onglet que l’on crée avec CTRL+N ou à l’onglet dupliqué grâce au CTRL+T. L’interface d’un « Onglet » est conçu à partir d’un formulaire UI de Qt, qui sera traduit en un fichier d’entête ui\_onglet.h pour qu’on puisse le réutiliser. Toutes les actions utilisateur seront traitées par la classe interne « evenement » de la classe « Onglet » pour respecter le principe d’encapsulation.

Pour la même raison que la classe « Calculatrice », nous avons implémenté la classe « LogSysteme » en Singleton aussi. Elle utilise sa classe interne « LogMessage » qui est une classe éphémère qui n'a pour seul but que de diffuser son message sur les flux voulu.

La classe « Exception » est conçue pour traiter les exceptions que l’utilisateur peut provoquer. Elle ne contient qu’un attribut qui est le message à envoyer à l’utilisateur ou au « LogSysteme » sous forme d’un QString. Dans la pratique, on ne l’utilise que lorsqu’on ne lançait pas d’exception avec throw, par exemple pour la perte de données quand on fait un cosinus d’entier.

La classe « FactoryException » et « OperationException » héritent tous de la classe « Exception » et elles représentent respectivement l’exception provoquée par une création de constante et celle produite par une opération. Il existe 2 catch par boucle try pour gérer leur déclenchement avec throw.

Nous avons le regret de ne pas avoir eu le temps suffisant pour implémenter la fonctionnalité d’historique (Annuler et Rétablir ) que l’on comptait réaliser avec le Mémento Pattern qui permet de restaurer un état précédent d'un objet sans violer le principe d'encapsulation.

#### A propos du diagramme de séquence de Calculatrice

Nous avons le regret de ne pas avoir eu le temps suffisant pour réaliser la fonction d’historique (Annuler et Rétablir) qu’on comptait implémenter avec le Mémento Pattern qui permet de restaurer un état précédent d'un objet sans violer le principe d'encapsulation.

Le scénario suppose que l’utilisateur ouvre l’instance de la « Calculatrice », qui appellera le constructeur de « Onglet » pour créer une instance d’onglet, configurer son UI, réaliser la connexion entre les signaux et les slot et enfin créer son moteur de calcul. Ensuite l’utilisateur appuie sur CTRL+T, ce qui va appeler la méthode dupOnglet() qui demande à la calculatrice de créer un nouvel onglet en dupliquant l’onglet actuel. Puis l’appui de CTRL+W de l’utilisateur provoquera l’appel de la méthode deleteOnglet() qui va supprimer l’onglet.

Ensuite l’utilisateur appuie sur le bouton « 1 » qui provoque un appel à la méthode traitement() de la classe interne « evenement » qui ajoutera le caractère « 1 » dans la QLineEdit. L’appui sur le bouton « Enter » appelle encore une fois la méthode traitement() qui va ajouter le contenu de la QLineEdit dans la pile de stockage et va appeler la méthode ajouterResoudre() pour ajouter ajouter « 1 » dans la pile de calcul.

Enfin l’utilisateur appuie encore une fois sur CTRL+W pour supprimer l’onglet restant.

## Namespace Calcul

#### A propos de l’UML statique de Calcul

L’interface entre la calculatrice et la résolution des calculs se situe au niveau de la classe MoteurCalcul, qui s’occupe de traiter directement la chaîne de caractère entré dans la QLineEdit, et on utilise ses accesseurs pour pouvoir afficher les éléments atomiques de la pile : les constantes, dérivant toutes de la classe abstraite Cst, (et donc agrégées dans la classe PileCst à travers une QStack) ont une méthode protégée setString() qui met directement leur string\_associé correctement en fonction de leur valeur et de leur type, l’attribut pouvant être accédé simplement avec l’accesseur getString().

Nous avons dans un premier temps souhaité concevoir la hiérarchie des constantes autour du design pattern composite, qui permettait de dire qu’une expression constante était en fait équivalente à la pile (qui aurait donc été le composite), et avec des objets opérateurs agrégeant les nombres, tous disposant d’une méthode évaluer(), mais cela nous a été fortement déconseillé. Néanmoins, nous pensons toujours que c’était la conception la plus judicieuse pour répondre à cette problématique.

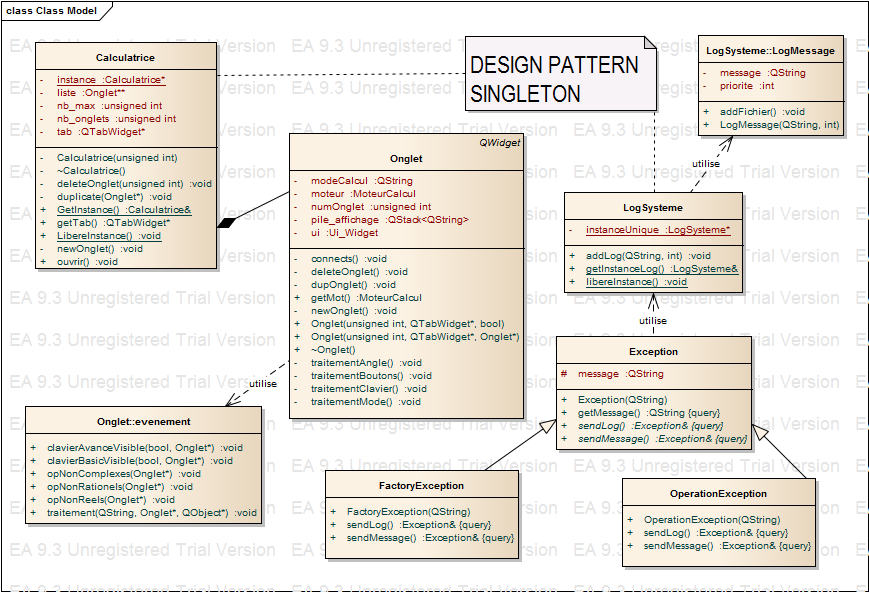
Nous n’avons donc finalement pas suivi de pattern pour cette hiérarchie assez classique, mais nous avons utilisé le design pattern factory method pour pouvoir créer tous ces objets de manière sécurisé à partir d’un string entré par l’utilisateur, et pouvant être n’importe quoi. Les informations d’une Expression sont simplement stockées dans son string\_associé, et les opérations sur celles-ci ne font que rajouter le string\_associé de l’autre opérande et l’opérateur à la fin de ce string\_associé. Complexe est une classe un peu plus compliquée puisque template : nous avons repris la classe complex de la STL. Ainsi, notre complexe agrège deux autres constantes (Reel, Rationnel ou Entier), et utilisent les opérations codées dans ces derniers pour implémenter ses propres opérations. Toutes les opérations sont déclarées virtuelle pures dans Cst, et renvoie un Cst&, ce qui nous a permis d’utiliser la covariance de type de retour (ainsi toute opération entre deux objets d’un même type T renvoie un T&). Nous avons choisi de renvoyer un objet alloué dynamiquement à chaque opération, et nous avons donc veillé dans la résolution des calculs à ce qu’il n’y ait pas de fuites mémoires.

#### A propos du diagramme de séquence de Calcul

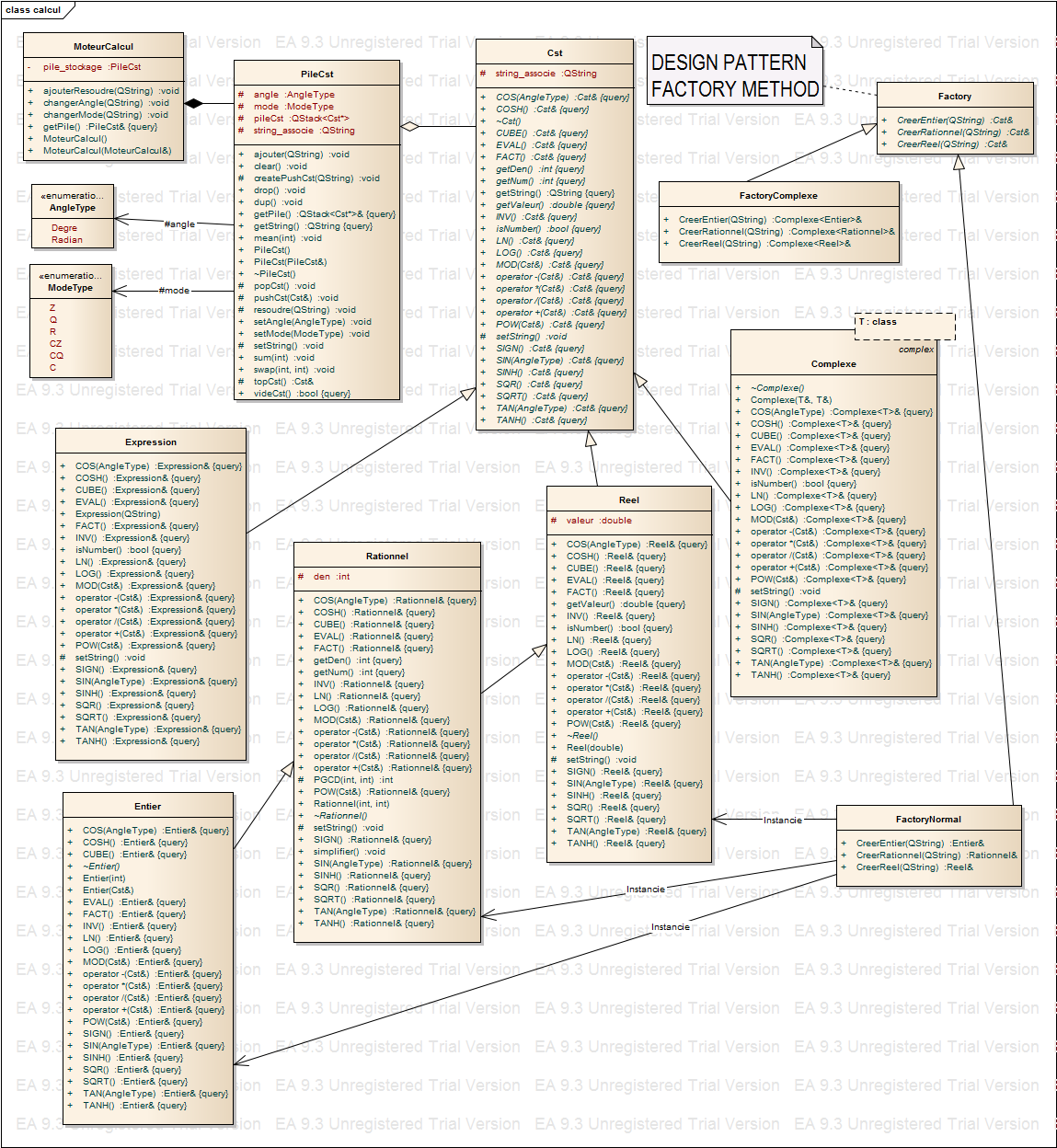
Le scénario veut que l’utilisateur ait entré « 3/4$5/6 7/8$1/2 + » soit une opération binaire en mode complexe rationnel. Les objets sont construits à partir du string passé à la pile, à travers l’interface de la factory. Lors d’une détection d’un opérateur dans ce string, elle en appelle à sa propre méthode résoudre, qui dépile et appelle les méthodes d’opérations des objets mis en jeu.

Ensuite, dans ce scénario classique, les accesseurs sont appelés pour que la partie calculatrice obtienne ses strings. Nous avons ensuite montrer la résolution de l’opérateur de pile clear(), puis des effets d’un changement de mode (vidage de la pile).

## Annexe I : UML Calculatrice



## Annexe II : UML Calcul



## C:\Users\William\Dropbox\lo21\projet\Sequence Calculatrice.pngAnnexe III : Diagramme de séquence Calculatrice

## Annexe IV : Diagramme de séquence calcul

