# 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

* Expliquer en gros comment on en est arrivé aux choix de conception qu’on explique en-dessous, et pourquoi.
* Expliquer UML statique
* Expliquer Diagramme de séquence dynamique
* (ne pas oublier log et exception)
* Expliquer comment on aurait conçu notre historique si on avait eu le temps de le faire

## 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).