|  |  |
| --- | --- |
|  | UTComputer |
|  |  |
| P16 | Rapport de projet - LO21 |
|  | Rapport du projet UTComputer de LO21 en Printemps 2016  Par Orphée FAUCOZ et Titouan GALOPIN  *http://tgalopin.github.io/utcomputer/* |

# Introduction

Le projet UTComputer de LO21 (Printemps 2016) a eu pour objectif de développer une calculatrice programmable en C++ en utilisant les concepts de l’orienté objet découvert durant le semestre.

Dans un premier temps, avant même de commencer à réfléchir à notre architecture nous avons réalisé une liste succincte des fonctionnalités que nous souhaitions intégrer à notre calculatrice. Nous avons utilisé des schémas d’états pour représenter les différentes façons d’interagir avec notre logiciel.



Premiers graphes d’états qui nous ont permis de nous mettre d’accord  
sur le fonctionnement du logiciel

Nous avons aussi réalisé des dessins de ce que nous souhaitions mettre en place, afin de bien déterminer quels composants seront nécessaires pour développer l’application de manière maintenable.

# Présentation de l’architecture

Grâce à cette première analyse, nous avons pu aisément isoler quelles classes et quels design patterns seront nécessaires pour réaliser correctement le projet.

Notre projet est séparé en deux parties distinctes : la librairie (dossier « lib ») et la partie interface utilisateur (« ui »). Cette séparation permet deux choses essentielles :

* Nous sommes assurés que la librairie n’utilisera pas de composants liés à l’affichage (l’affichage doit utiliser la librairie, mais surtout pas l’inverse : cela créerait une dépendance circulaire et illogique)
* Nous avons pu réaliser des tests automatisés (cf. fin du rapport) sur la librairie seule, indépendamment de l’interface. Ces tests automatisés nous ont permis d’éviter de nombreux problèmes car à chaque modification d’une classe, nous étions certains que cette modification n’avait pas introduit de régression ou de bug.

De plus, cette séparation nous a permis de réduire le temps de compilation durant le développement : au début du projet, il ne nous était pas utile d’inclure la librairie graphique dans le projet (nous avons pu l’ajouter aisément quand nous en avions besoin seulement).

## La librairie

La première partie du projet, la plus importante et celle qui a réquisitionnée le plus nos connaissances en Orienté Objet, a été la librairie.

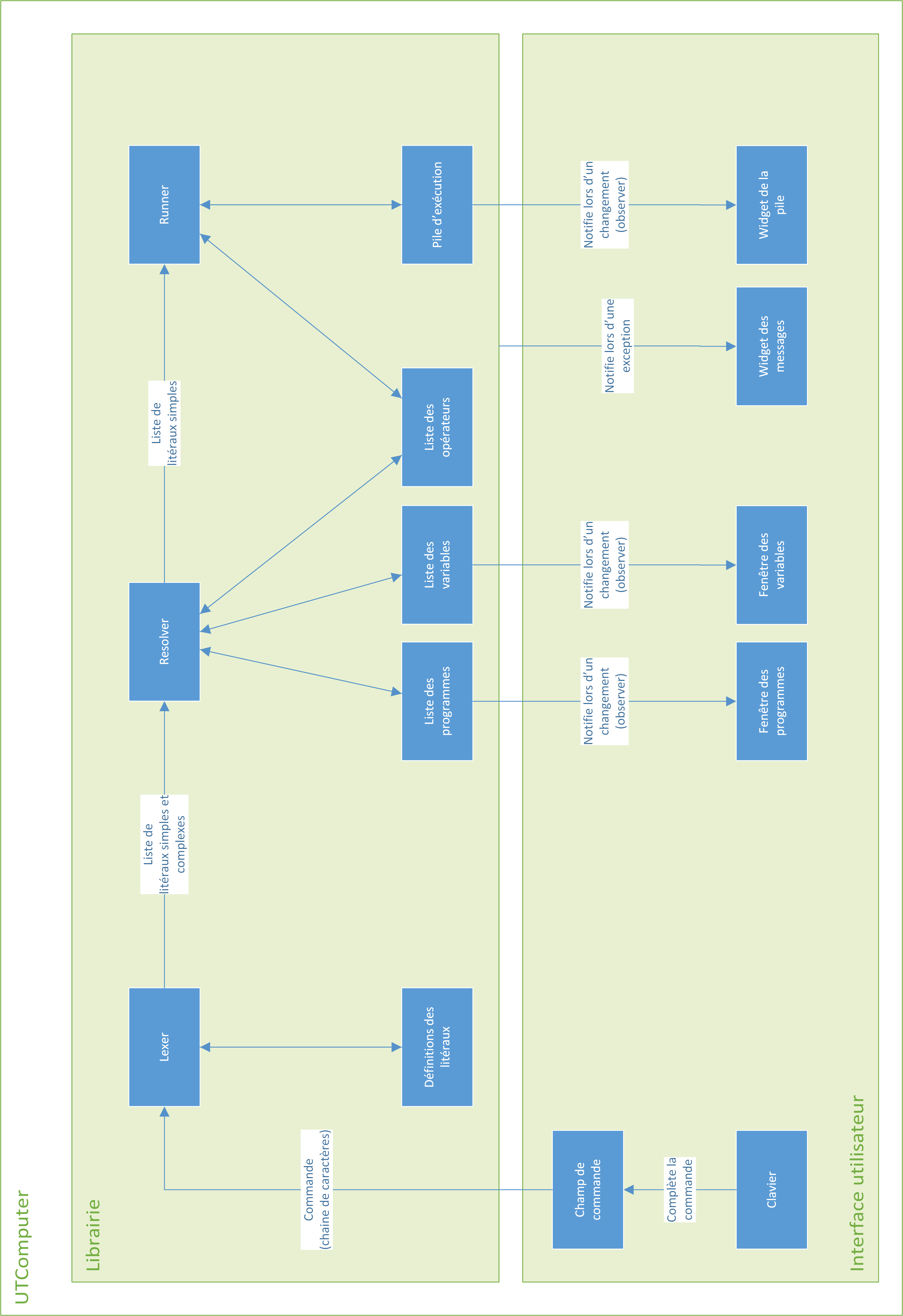
Le principe de base de la librairie est d’être une boite noire (principe d’encapsulation) à laquelle nous passons une chaine de caractères et qui l’exécute pour modifier une pile donnée. Cette boite noire sera ensuite simplement utilisée par l’interface graphique pour exécuter les commandes de l’utilisateur.

Pour réaliser cette boite noire, nous avons séparé l’analyse de la commande en trois parties selon le modèle des analyses lexicales qui constituent la plupart des langages de programmation (cf <https://en.wikipedia.org/wiki/Lexical_analysis>) :

1. Le **lexer** récupère la chaine de caractère et la transforme en une liste de « tokens », appelés dans le projet « littéraux ».
2. Le **resolver** récupère ensuite cette liste de littéraux et transforme tous les littéraux de noms (les « atomes ») en leur littéral équivalent variable/programme/opérateur. Il renvoi ensuite cette liste de littéraux « résolus ».
3. Le **runner** récupère cette liste résolue et l’exécute sur la pile en faisant appel aux opérateurs.

Cette organisation nous a permis de séparer le code en trois parties et en de multiples sous-parties encapsulées dans ces trois éléments. En séparant ainsi le code, chaque composant est indépendant ce qui apporte deux grands avantages :

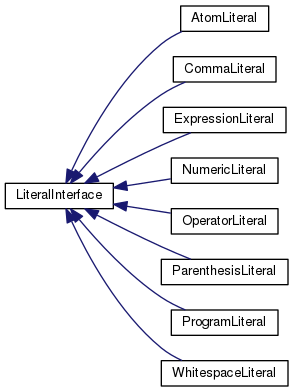
* Grâce à l’encapsulation et au principe de polymorphisme, il est possible pour n’importe qui de modifier le fonctionnement d’une partie de la librairie indépendamment du reste en remplaçant cette partie uniquement. Il est ainsi possible de développer son propre lexer sans influer sur le resolver et le runner.
* Comme chaque composant est indépendant, il devient possible de le tester indépendamment des autres composants. En utilisant un système de « mocks », implémentant de manière statique les interfaces des autres composants, seul le composant en lui-même est testé (par exemple, dans utcomputer/lib/tests/repository/map.cpp)



### Le lexer (http://tgalopin.github.io/utcomputer/classLexer.html)

### *Les littéraux (utcomputer/lib/literal)*

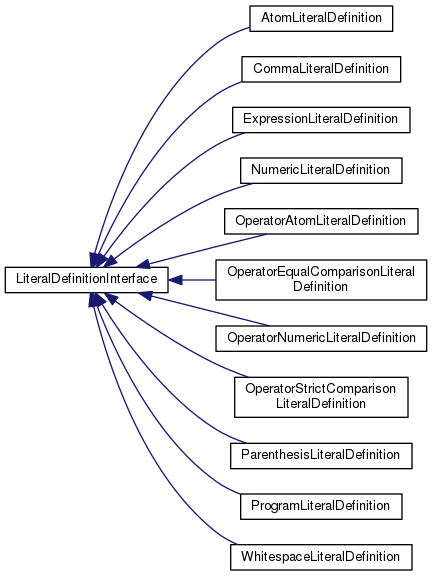
Les littéraux implémentent le pattern Strategy afin de pouvoir être utilisés indépendamment de leur type. Il est donc possible de créer de nouveaux types de littéraux en implémentant l’interface LiteralInterface.



Les littéraux implémentent le pattern Strategy

Notez qu’il n’y a qu’un seul littéral numérique. Cela permet de simplifier les calculs dans les opérateurs.

### *Les définitions de littéraux (utcomputer/lib/literal\_definition)*



Les définitions implémentent elles aussi le pattern Strategy

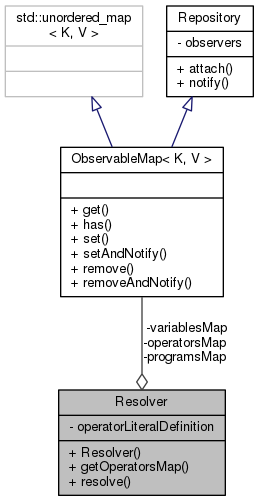
Même en implémentant le pattern Strategy, comme c’est le **lexer** qui créé les littéraux à partir de la chaine de caractères, si nous souhaitions créer un nouveau type de littéral, il aurait fallu modifier le **lexer**, ce qui n’est plus maintenable et va à l’encontre du principe de Strategy.

Pour résoudre ce problème, nous avons mis en place un système de définitions de littéraux basé à la fois sur le design pattern Visitor et le design pattern Factory.

En encapsulant une liste de définitions dans le **lexer**, au moment de la création des littéraux, nous parcourons toutes les définitions pour déterminer celle qui supporte la chaine actuelle (pattern Visitor) et nous utilisons cette définition pour créer le littéral associé à la chaine de caractère. Nous utilisons de plus les expressions régulières dans les définitions de littéraux pour permettre une grande granularité quant aux chaines supportées.

Ainsi, créer un nouveau littéral et donc étendre le **lexer** est beaucoup plus simple : il suffit de créer un objet littéral (implémentant LiteralInterface), créer une définition pour ce littéral (implémentant LiteralDefinitionInterface) et ajouter cette définition au **lexer**. N’importe qui peut aisément le faire depuis l’extérieur du code, sans connaitre la librairie dans ses moindres détails.

### Le resolver (http://tgalopin.github.io/utcomputer/classResolver.html)



A partir des littéraux créés avec le lexer, le **resolver** va itérer sur cette liste pour effectuer deux choses :

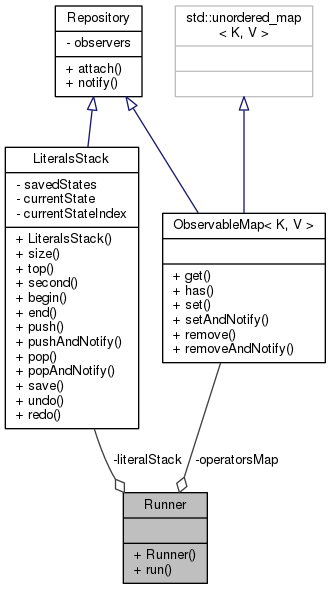
* Supprimer les espaces (ils doivent être ignorés dans la notation postfixée)
* Transformer les littéraux d’atomes en :
  + OperateurLiteral si l’atome existe dans la liste des opérateurs
  + Le littéral associé à la valeur d’une variable si l’atome a un nom de variable
  + La liste de littéraux associés à la valeur d’un programme si l’atome a un nom de programme

### *Les repositories (utcomputer/lib/repository)*

Le **resolver** a donc besoin de la liste des opérateurs, de la liste des variables et de la liste des programmes.

Chacune de ces listes implémente à la fois la hashmap standard de C++ (« unordered\_map » permettre un accès en O(1)) et l’interface RepositoryInterface. Cette interface met en place la partie « observable » du design pattern Observer, de telle sorte qu’il sera possible pour l’interface utilisateur d’être notifiée de toute changement des données du programme pour rafraîchir l’affichage.

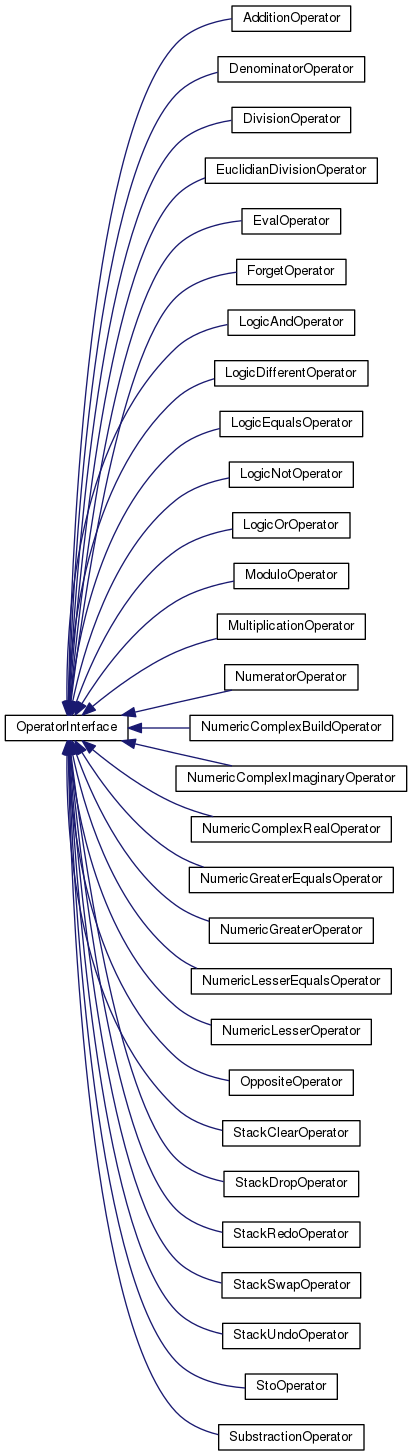
### Le runner (http://tgalopin.github.io/utcomputer/classRunner.html)



Le **runner** est le troisième composant de la librairie. C’est lui exécute concrètement l’action demandée sur la pile en utilisant les opérateurs. Il itère sur la liste de littéraux « résolue » et empile tous les littéraux sauf les opérateurs, qu’il exécute.

### *Les opérateurs (utcomputer/lib/operator)*

De la même manière que les définitions de littéraux, les opérateurs mettent en place le pattern Visitor et le pattern Strategy. Chaque opérateur a pour objectif de modifier la pile selon son action. Par exemple, l’opérateur AdditionOperator va dépiler deux opérandes, les additionner, et empiler le résultat. Chaque opérateur possède donc une méthode apply(LiteralsStack &stack) afin de modifier la pile.



Les opérateurs implémentent le pattern Strategy (http://tgalopin.github.io/utcomputer/classOperatorInterface.html)

## L’interface utilisateur

L’interface utilisateur utilise GTK et est composée de 7 widgets. Chacun de ces widgets implémente l’interface « RepositoryObserver » afin d’implémenter la partie « observer » du design pattern Observer.

Chaque widget met ensuite à jour l’affichage en fonction des données récupérées depuis le repository observé.

La création de la librairie en elle-même et son utilisation sont principalement défini dans ui/main.cpp.