REANEY Olivia *GI02*
DAMBRINE Florian *GI02*

Projet LO21

Calculatrice à notation polonaise inversée

I) Choix d'implémentation

Notre système de correspondance entre les différences classes héritant de Constante est un peu particulier. En effet, à la lecture du sujet, nous avions tout d'abord compris qu'un Rationnel était composé de deux entiers (son numérateur et son dénominateur) mais en tant qu'objets de classe Entier, car si nous devions réencapsuler une classe pour les Entier, pourquoi ne devrions-nous pas les considérer comme les plus petites unités de base? Un Rationnel utilise donc deux Entier pour son numérateur et son dénominateur.

De la même façon, nous estimons qu'un nombre complexe peut avoir différent type de nombre en tant que partie réelle et partie imaginaire. Celui-ci peut-être un entier, un réel ou un rationnel, car ceux-ci se correspondent (un entier est un réel sans partie décimale, un rationnel est un entier dont le dénominateur est 1, etc.). Comme nous avons créé une classe Nombre abstraite afin d'encapsuler les objets ayant des méthodes en commun, un complexe a donc pour partie réelle un pointeur vers un Nombre, et un pointeur vers un autre Nombre pour partie imaginaire.

Les relations entre les différents types de constante nous a permis de faciliter les opérations entre constantes. Nous avons réalisé des méthodes pour chaque classe afin de convertir le type d'un Nombre en un autre. Comme la plupart des opérations binaires simples sont symétriques, il nous a fallu développer une spécialisation d'opération pour un type, puis, dans un autre type, de transformer celui-ci et d'appeler la fonction concernant l'opération correspondante pour l'argument en lui passant an paramètre l'objet transformé.

Par exemple : un Entier additionné à un Réel transforme l'Entier en Réel et retourne un Réel. Et un Réel additionné à un Entier appellera la fonction addition de l'Entier en passant le Réel en paramètre. Cela permet de traiter les cas les plus complexes d'abord pour ensuite faciliter les autres cas. De cette façon, lorsqu'on a un complexe additionné à n'importe quel Nombre, celui-ci sera transformé en un complexe, et on n'aura plus qu'à additionner les deux complexes ensemble.

Nous avons décidé que pour garder le plus de précision possible, l'objet créé à partir de l'objet le plus précis. Un entier multiplié par un réel renverra un réel, tandis qu'un réel multiplié par un rationnel renverra un rationnel. Seule la division fait une exception, en vérifiant le type de constante sélectionnée par l'utilisateur. Dans le cas de la division, il nous a également fallu vérifier si la valeur d'une Constante n'était pas égale à 0, et dans le cas d'un Rationnel, si les deux Entier ne sont pas égaux, auquel cas on pourrait simplifier ce Rationnel par un Entier valant 1. Pour cela, il nous a fallu implémenter des opérateurs d'égalité, soit entre deux Nombre, soit entre un Nombre et un int passé en paramètre.

Nous avons eu également recours à de nombreux dynamic_cast tout au long de nos méthodes, même si ceux-ci sont coûteux, car pour tester tous les cas possibles pour les opérations, il nous faut vérifier les types des Constante. Cela nous permet un contrôle total sur la précision des résultats, la gestion des erreurs, et la possibilité de simplifier certaines méthodes en évitant les redondances de codes (comme par exemple le cas décrit auparavant sur les opérations binaires).

II) Les desgin pattern abordés dans le projet :

1) Template Method :

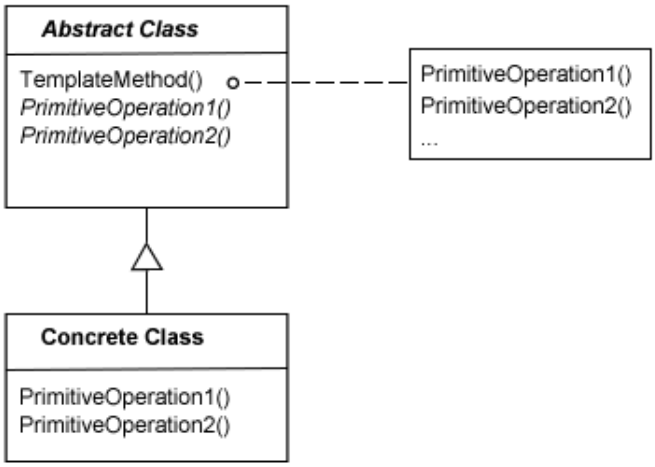


Illustration 1: Schéma UML du design pattern "Template/Method"

Nous avons utilisé le design pattern Template/Method dans la classe **Constante**. Ce patron de conception utilise des méthode concrètes (operator+(...) / operator-(...), operator*(...), operator/(...)) qui appelle des méthodes virtuelles pures (addition(...), soustraction(...), multiplication(...), division(...)).

Ces méthodes sont définies dans les classes dérivées afin d'implémenter correctement ces opérations arithmétiques sur des objets des classes **Complexe**, **Rationnel**, **Reel** et **Entier**.

Le template méthode est également utilisé pour des fonctions d'affichage comme afficher(...).

2) Singleton :

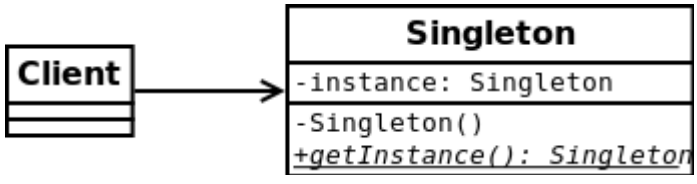


Illustration 2: Schéma UML du design pattern "Singleton"

Ce patron nous assure l'instanciation d'une unique instance de classe.

Nous l'avons utilisé dans les classes **Calculatrice**, **Gardien**, **Option, Fabrique**.

3) Memento :

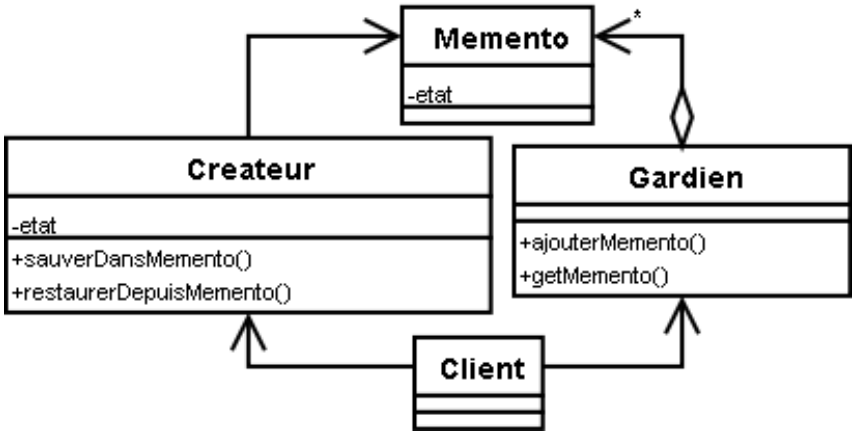


Illustration 3: Schéma UML du design pattern "Memento"

Pour réaliser l'opération undo/redo, nous avons implémenté le design pattern Memento.

La classe pile encapsule une classe memento. Cette classe permet de réaliser une sauvegarde de la pile à un instant donné.

La classe Cliente, ici la classe MainWindow, réalise des sauvegardes de la pile lorsque celle ci est modifiée.

Le Gardien mémorise une liste de Memento qu'il peut restituer quand on lui demande.

Ainsi quand l'utilisateur demande une opération de undo, le Gardien restitue le Memento à l'index i et décrémente son index.

Quand celui-ci demande une opération de redo, le Gardien restitue le Memento d'index i et incrémente son index.

La classe Pile n'a plus qu'a restaurer sa pile depuis un mémento donné.

4) Iterator :

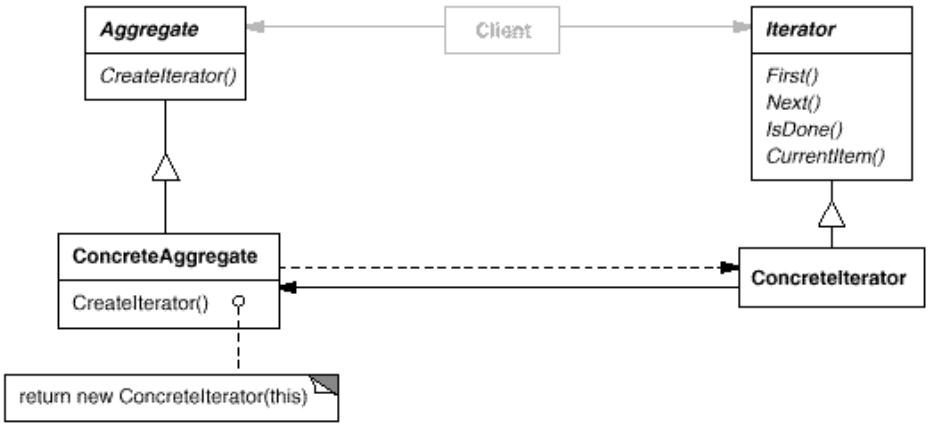


Illustration 4: Schéma UML du design pattern "Iterator"

Nous avons utilisé le design pattern Iterator de la librairie Qt. En effet le parcours de pile se fait grâce à un iterateur de la classe Qstack.

L'utilisation de l'iterateur nous permet de parcourir aisaiment la pile contenant des pointeurs Expression et d'en afficher le contenu.

Le second avantage est d'utiliser l'iterateur à l'envers (fonctionnement en reverse iterateur) pour afficher le sommet de la pile vers le haut de la zone d'affichage.

5) Factory :

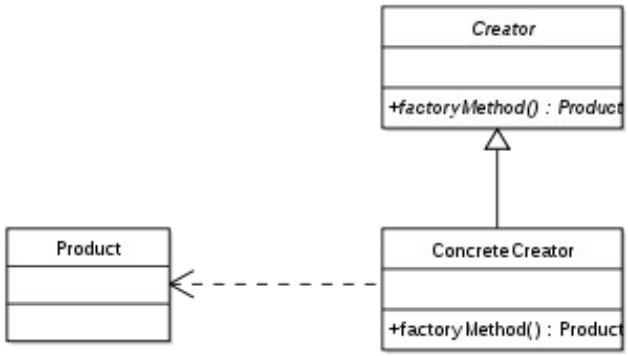


Illustration 5: Schéma UML du design pattern "Factory"

L'utilisation de ce patron nous à permis de gérer correctement la création de constante avant de l'empiler.

Contrairement au schéma proposé ci-dessus, nous avons directement implémenté la

Fabrique concréte.

Cette fabrique est en charge du traitement de la chaine de caractère saisie par l'utilisateur.

Elle s'occupe de réorganiser cette chaine en vérifiant la présence d'expressions, d'opérateurs ou de constantes.

Celle-ci crée alors le bon type d'objet pour finalement faire appel à la classe **Pile** afin d'empiler l'objet.

6) Composite :

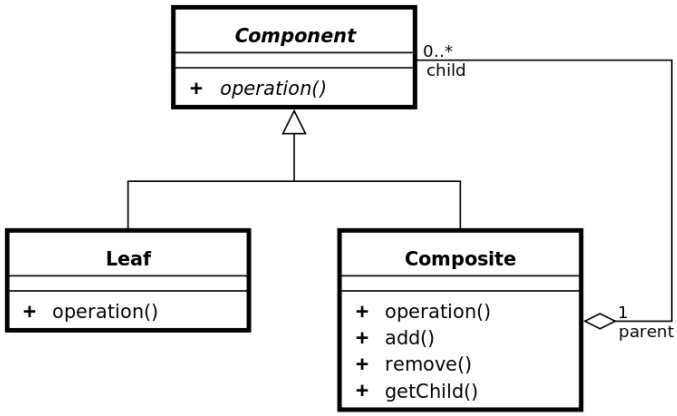


Illustration 6: Schéma UML du design pattern "Composite"

La gestion d'expression nous à ammené à chercher une solution de conception adaptée.

Nous nous sommes donc penché sur le design pattern Composite. Celui ci permet de d'agréger au sein d'un objet des composants. En effet une expression est un objet complexe composé de constantes et d'opérateurs.

Nous n'avons pas implémenté ce pattern en envisagant de traiter les expressions en tant que chaine de caratères.

Ainsi nous stockons les expressions dans une objet de type **Exp**. Quand l'utilisateur demande son évaluation, on réinjecte cette chaine dans la fabrique qui s'occupe de réaliser les opérations demandés.

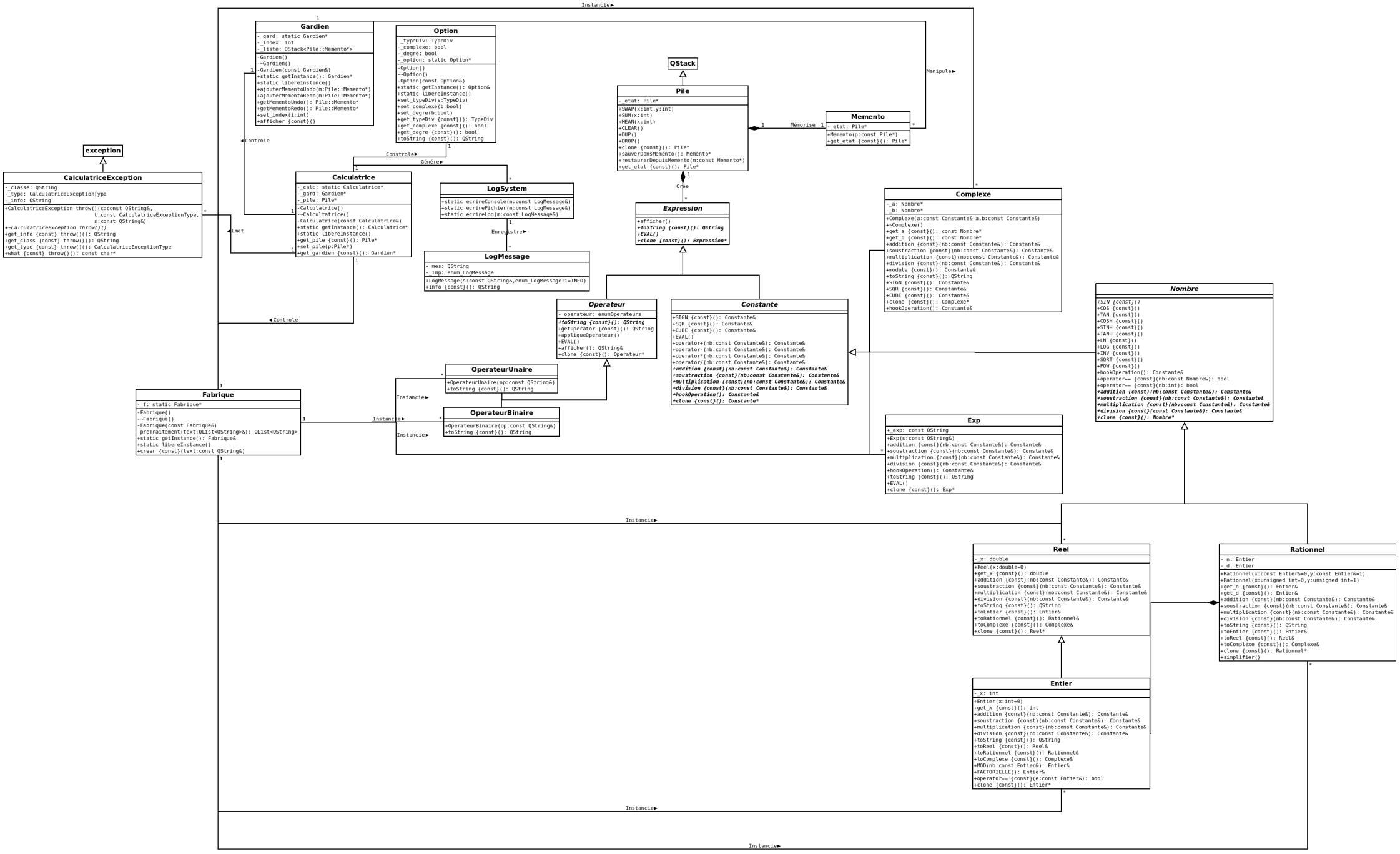
De plus, la gestion de la concaténation était bien plus simple en gérant des Qstrings plutôt que des Composants.

III) Bilan

Ce projet, même si très scolaire dans son sujet, nous a permis d'approfondir les concepts de design pattern, vus en cours. Ils nous ont été très utiles pour la conception des relations entre les classes et le stockage des données. Nous avons pu observer également que nous avons mis en place, inconsciemment, un modèle MVC : la classe MainWindow s'occupe de l'interface, donc de la vue, la classe Calculatrice contient les éléments nécessaires au fonctionnement de celle-ci, et correspondrait donc à un contrôleur, et c'est par ces éléments qu'on va pouvoir effectuer des calculs et stocker diverses données, le côté modèle.

Nous avons également réalisé qu'avec certains concepts vus dans la dernière partie du cours, une grande partie du projet aurait été grandement facilité, et cela nous a permis d'appréhender la notion de patron de conception, et au fur des TDs nous avons réalisé comment nous aurions pu améliorer la calculatrice avec ces idées. Le fait d'ajouter une véritable interface graphique à l'aide de Qt nous permets d'appréhender le fait que toute l'obscure programmation a une incidence réelle sur le fonctionnement d'un programme, on peut finalement voir le résultat final de nos algorithmes, et son fonctionnement, chose qu'on aperçoit rarement dans les enseignements de programmation, où les affichages restent souvent seulement textuels. On perd toute une partie de la complexité également en ne voyant pas l'interface textuelle, en se privant de découvrir la notion de signaux et de slots dans un programme. Cela dit, pour la calculatrice, ceux-ci étaient relativement aisés, car nous n'avons pas eu affaire à des signaux asynchrones.

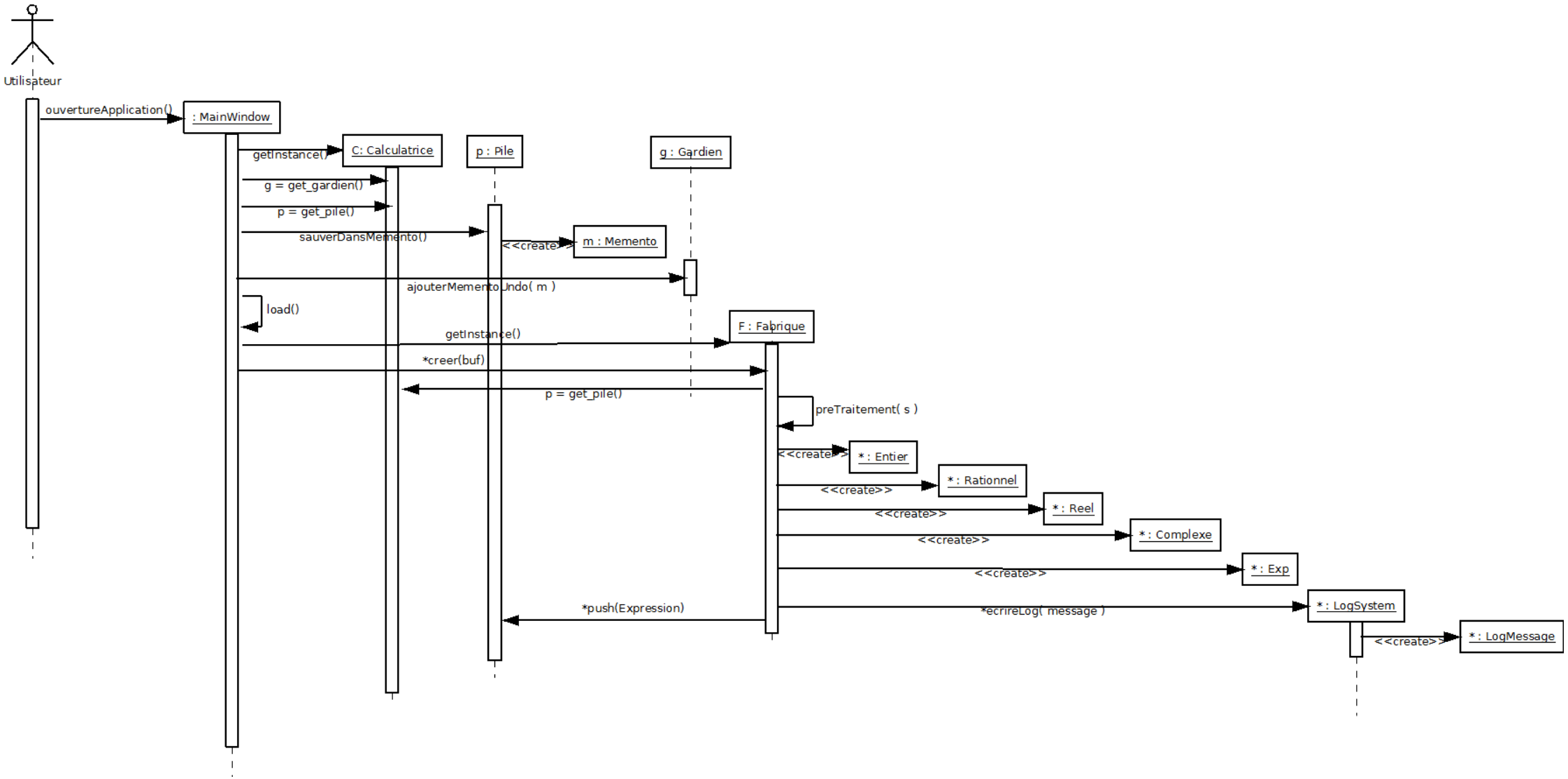
Projet LO21 – Diagramme de classe UML



DAMBRINE Florian – REANEY Olivia

Projet LO21 – Diagramme de séquence 2 UML

Ouverture de l'application



Projet LO21 – Diagramme de séquence 1 UML

Saisie d'un entier addition d'un entier et d'un réel (on suppose que le réel est déjà dans la pile)