R3.04

## TP 5 Refactoring à l'aide de patrons de conception

**Principaux apports techniques:**
- compréhension d'un programme complexe par rétro-conception et rétro-documentation ;
- documentation d'une évolution sur des diagrammes UML ;
- Applications de patrons de conception

# Reprise de l'application Binôme

L'application **`Binome`** qui calcule les racines d'un polynôme du second degré semble a priori avoir une structure complexe ; les techniques de rétro-conception vues précédemment vont nous permettre d'en comprendre et d'en justifier l'architecture qui en fait a été choisie pour faciliter l'évolution.

### 1 - Pattern FactoryMethod

L'exemple présenté dans l'application Binome est une implantation du «*Pattern*» **`Factory Method`**.

Le patron de conception **`Factory Method`** est de la catégorie **`création`**. L'objectif de **`Factory Method`** est de localiser le code de choix d'instanciation de tel ou tel objet dans une méthode standard d'un objet particulier, la *fabrique ou créateur*. L'intérêt est que l'on réalise une double séparation :

- entre le code de détection de la situation et le code de réalisation des actions, d'une part,
- entre les codes de réalisation correspondant aux situations, de l'autre, ainsi le code interne de chaque classe fille devient très simple (sans test sur la situation) et facile à maintenir.

### A faire

- Ajouter une nouvelle classe à votre diagramme de classes que vous appellerez **Binome Factory**.
- Extraire (et supprimer) de la classe **`Binome`** la méthode **create** et l'insérez dans **BinomeFactory**, MAIS en tant que **méthode d'instance**.
- Modifier le code de la classe lanceur **Bin** pour ajouter la création d'un objet **`BinomeFactory`** et modifier les envois de message utilisant la méthode **`create`**.
- Faites tourner votre code pour vérifier que tout fonctionne comme avant.

### 2 - Pattern Singleton

On souhaite, dans l'application binome, déléguer tous les affichages à un **unique objet** instance d'une classe **`WriterFacade`**. Cette classe qui fournit des services d'affichage, dont une méthode **`println (String)`**, sous-traite l'affichage réel, dans cette première implantation, à l'objet `System.out`. Pour justifier l'utilisation de l'objet **`WriterFacade`** on pourrait imaginer que l'affichage se fasse dans un autre flux de sortie.

En fait, cette nouvelle classe est également une illustration d'un autre pattern fondamental, **Facade**.

Note : le pattern **`Facade`** permet de découpler un système A d'un autre système B à l'aide d'une classe intermédiaire qui masque la complexité de B au système A en ne lui proposant que les services pertinents. Cette classe **`Facade`** se borne à déléguer aux classes du système B les services

requis par le système A. Ce pattern diminue la complexité et surtout augmente considérablement l'évolutivité d'un programme.

## A faire

Montrer l'évolution du diagramme de classes de binome en appliquant le ***pattern Singleton***, c'est-à-dire :

- Ajouter la classe **WriterFacade** qui utilise le pattern Singleton dans votre diagramme de classes à l'extérieur du package binome (dans un autre package utils ; vous pouvez la mettre dans le même package que la classe Complex)),
- Implémenter en java la classe **WriterFacade**,
- Faire les modifications nécessaires au reste du code pour utiliser l'instance unique de **WriterFacade**,
- Compiler et vérifier que l'ensemble fonctionne.
- Retro-concevoir le diagramme final contenant toutes les modifications.

## A rendre

Déposer dans votre zone de rendu une archive contenant :

- le code entier de votre application Binome intégrant toutes les modifications
- l'image de votre diagramme de classes final intégrant toutes les modifications apportées dans ce TP et dans le précédent mis dans une fichier PDF avec le résultat de l'exécution de la classe Bin.