M1 Informatique - FMIN-107 - Ingénierie Logicielle
TD-TP
***Frameworks* : Architecture et Paramétrage**
Utilisation des *Design Pattern : "IOC", implantation avec "Abstract Fabric"*

1 Exercice 1

1.1 Un framework : *Compiler*

1.1.1 Utilisation

Soit le texte de programme suivant :

```
public static void main(String[] args) {
    try{
      System.out.println("-----------------------");
      Compiler c1 = new Compiler("Java");
      c1.compile(new ProgramText("..."));
      System.out.println("-----------------------");
      Compiler c2 = new Compiler ("C++");
      c2.compile(new ProgramText("..."));
      System.out.println("-----------------------");
      Compiler c3 = new Compiler("ADA");}
    catch(Exception e){System.out.println(e.getMessage());}}
```

Qui génére l'affichage suivant sur la sortie standard :

```
   -----------------------
   Compiling a: Java program.
   I am scanning a Java program text
   I am parsing a Java scaned text and I generate a Java AbstractSyntaxTree
   I am generating a JVM program text from a Java AbstractSyntaxTree
   Compilation finished
   -----------------------
   Compiling a: C++ program.
   I am scanning a C++ program text
   I am parsing a C++ scaned text a C++ AbstractSyntaxTree
   I am generating an assembler program text from a C++ AbstractSyntaxTree
   Compilation finished
   -----------------------
   Non supported Language : ADA, Extend the framework to support it */
```

1.1.2 Architecture

La classe `Compiler` est donc le point d'entrée d'un framework qui permet d'utiliser des compilateurs de différents langages, comme illustré dans le `main` précédent.

Le framework est à la base paramétré par composition de la façon suivante (extrait) :

```
public class Compiler{
protected Lexer lexer;
protected Parser parser;
protected Generator gen;
    ...
```

C'est un framework car il est extensible en créant de nouveaux types de *lexers*, de *Parsers* et de générateurs de code comme sous-classes des classes suivantes redéfinissant les méthodes abstraites suivantes :

```
abstract class Lexer {public abstract ScanedText scan(ProgramText t);};
abstract class Parser{public abstract AST parse(ScanedText t);}
abstract class Generator{public abstract File generate(AST a);}
```

De plus, si on étend le framework pour un nouveau langage (matérialisé par une sous-classe de `Lexer`, une de `Parser` et une de `Generator`, la méthode `compile` de la classe `Compiler` devra invoquer automatiquement leurs méthodes `scan`, `parse`, et `Generate`.

Le but de l'exercice est de mettre en place une architecture qui réalise cet objectif.

C'est une occasion de mettre en oeuvre explicitement les schémas d'"**injection de dépendance**" et d'**inversion de(of) contrôle** ("IOC") dont le cours a montré qu'ils sont centraux aux frameworks.

L'exercice pose eb plus le problème récurrent d'empêcher de combiner dans une même instance de `Compiler` un *parser* et un générateur de code qui soient incompatibles. Ce problème est suffisamment courant et général pour qu'un schéma de conception, "fabrique abstraite", lui soit consacré. Il est décrit dans la section suivante. Vous devez en premier lieu l'étudier.

1.2 Le Schéma "Fabrique Abstraite"

Le texte, le schéma et l'exemple qui suivent dans les sections 2.1 et 2.2, sont extraits de wikipedia : `http://fr.wikipedia.org/wiki/Fabrique_abstraite_(patron_de_conception)`.

1.2.1 Description

La fabrique abstraite est un patron de conception (design pattern) créationnel utilisé en génie logiciel (cf. figure 1). Une fabrique abstraite encapsule un groupe de fabriques ayant une thématique commune. Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique. Le client ne se préoccupe pas de savoir laquelle de ces fabriques a donné un objet concret, car il n'utilise que les interfaces génériques des objets produits. Ce patron de conception sépare les détails d'implémentation d'un ensemble d'objets de leur usage générique.

Un exemple de fabrique abstraite : la classe documentCreator fournit une interface permettant de créer différents produits (e.g. createLetter() et createResume()). Le système a, à sa disposition, des versions concrètes dérivées de la classe documentCreator, comme par exemple fancyDocumentCreator et modernDocumentCreator, qui possèdent chacune leur propre implémentation de createLetter() et createResume() pouvant créer des objets tels que fancyLetter ou modernResume. Chacun de ces produits dérive d'une classe abstraite simple comme Letter ou Resume, connues du client. Le code client obtient une instance de documentCreator qui correspond à sa demande, puis appelle ses méthodes de fabrication. Tous les objets sont créés par une implémentation de la classe commune documentCreator et ont donc la même thématique (ici, ils seront tous fancy ou modern). Le client a seulement besoin de savoir manipuler les classes abstraites Letter ou Resume, et non chaque version particulière obtenue de la fabrique concrète.

Une fabrique est un endroit du code où sont construits des objets. Le but de ce patron de conception est d'isoler la création des objets de leur utilisation. On peut ainsi ajouter de nouveaux objets dérivés sans modifier le code qui utilise l'objet de base.

Le code client n'a aucune connaissance du type concret, et ne nécessite donc aucun fichier header ou déclaration de classe requis par le type concret. Le code client n'interagit qu'avec la classe abstraite. Les objets concrets sont en effet créés par la fabrique, et le code client ne les manipule qu'avec leur interface abstraite.

L'ajout de nouveaux types concrets dans le code client se fait en spécifiant l'utilisation d'une fabrique différente, modification qui concerne typiquement une seule ligne de code (une nouvelle fabrique crée des objets de types concrets différents, mais renvoie une référence du même type abstrait, évitant ainsi de modifier le code client). Une fabrique est généralement un Singleton.

1.2.2 Exemple

```
public abstract class GUIFactory {
    public static GUIFactory getFactory() {
         int sys = readFromConfigFile("OS_TYPE");
         if (sys == 0) { return(new WinFactory()); }
         else { return(new OSXFactory());}}
    public abstract Button createButton();}

 class WinFactory extends GUIFactory {
     public Button createButton() {
         return(new WinButton());}}

 class OSXFactory extends GUIFactory {
     public Button createButton() {
         return(new OSXButton());}}

 public abstract class Button  {
     private String caption;
     public abstract void paint();
```

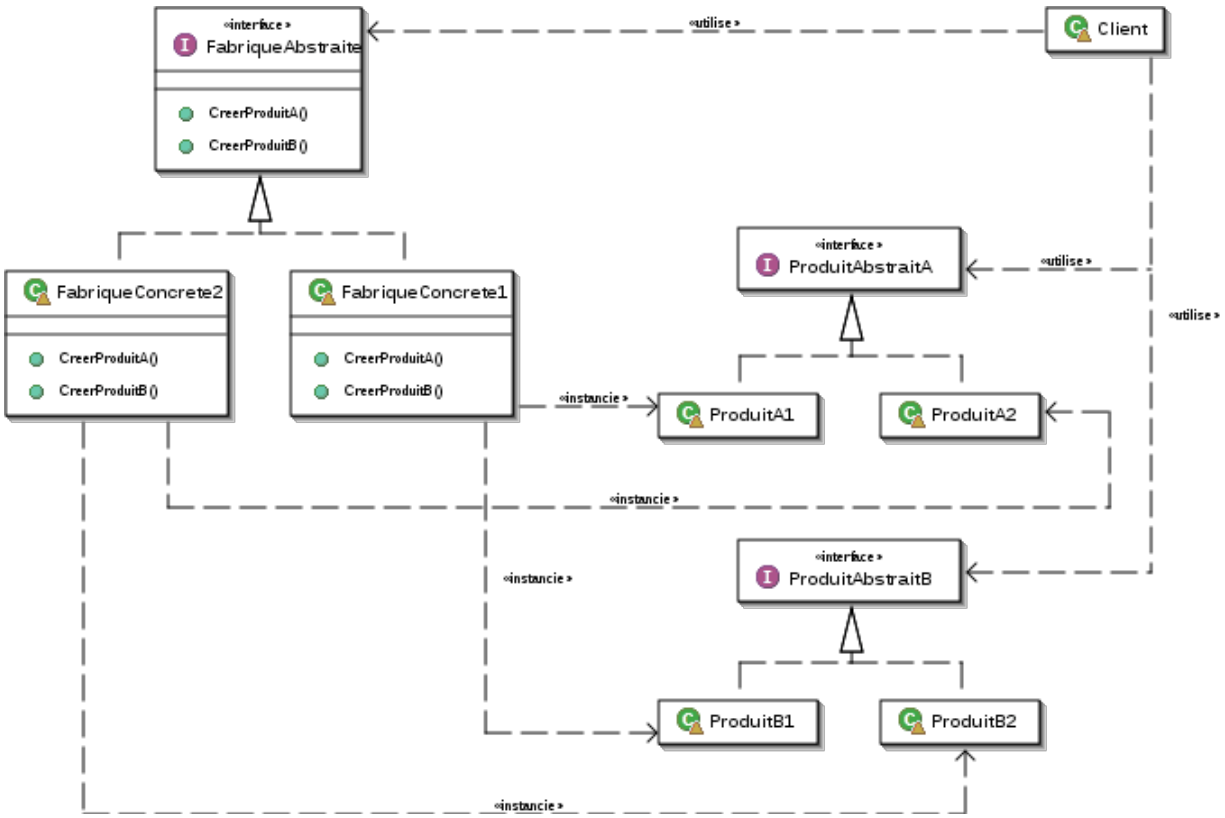

FIGURE 1 – Design Pattern "Fabrique Abstraite" d'après http ://fr.wikipedia.org/

```
    public String getCaption(){
        return caption;}
    public void setCaption(String caption){
        this.caption = caption;}}

class WinButton extends Button {
    public void paint() {
       System.out.println("I'm a WinButton: "+ getCaption());}}

class OSXButton extends Button {
    public void paint() {
       System.out.println("I'm a OSXButton: "+ getCaption());}}

public class Application {
    public static void main(String[] args) {
        GUIFactory aFactory = GUIFactory.getFactory();
        Button aButton = aFactory.createButton();
        aButton.setCaption("Play");
        aButton.paint();}
    //output is : "I'm a WinButton: Play" or "I'm a OSXButton: Play"
}
```

1.3 Application de *Abstract Factory*, *IOC*,*Dependency Injection*, au framework "Compiler"

En appliquant les schémas *Inversion de contrôle*, *Injection de dépendance* et *fabrique abstraite*, proposez une architecture et une implantation du framework réalisant le programme de la section 1. Bien sûr dans le code de la simulation que vous avez à réaliser, les méthodes `scan`, `parse` et `generate` ne font que des `println`

2 Exercice 2 : Un jeu de lettres avec AbstractFactory

On s'intéresse au développement d'un petit jeu de lettres. Dans ce jeu, on part d'une suite de mots prédéfinie, et on encode cette suite en associant à certaines lettres du mot un autre symbole (le même symbole pour chacune des occurences d'une même lettre). Cette suite encodée de mots est présentée au joueur, qui doit la décoder. Quand l'utilisateur propose une association lettre-symbole, selon que le jeu est assisté ou pas, on va lui indiquer ou pas si cette association est correcte. On présente en permanence au joueur la suite de mots en cours de décodage (dans son état de décodage courant), et les associations lettre/symbole que le joueur a proposées. Les lettres et leur version accentuée seront considérées dans une même classe d'équivalence, ainsi on travaille sur 26 lettres. On veut pouvoir choisir 3 niveaux pour le jeu :

- facile : les symboles utilisés pour l'encodage ne sont pas des lettres mais des symboles graphiques (étoiles, carrés, losanges, etc.). Lors de l'encodage on garde 30% de lettres non encodées. On assiste le jeu.
- moyen : les symboles utilisés pour l'encodage ne sont pas des lettres mais des symboles graphiques (étoiles, carrés, losanges, etc.). Lors de l'encodage on garde 20% de lettres non encodées. On n'assiste pas le jeu.
- difficile : les symboles utilisés pour l'encodage sont des lettres. Lors de l'encodage on garde 20% de lettres non encodées. On n'assiste pas le jeu.

Exemple. On encode : `LE PETIT CHAPERON ROUGE`. Il y a 20 occurrences de lettres, mais 13 lettres différentes.

- Niveau facile. On va encoder 70% des lettres, i.e. 9 lettres. On choisit par exemple d'encoder les 4 premières et les 5 dernières : L, E, P, T, R, O, N, U, G. On affiche donc par exemple :
  ✶✚ ■✚♦I♦ CHA■✚▲❖❍ ▲❖✈✐✚
- Niveau moyen et difficile. On va encoder 80% des lettres, i.e. 10 lettres. On choisit par exemple d'encoder les 5 premières et les 5 dernières : L, E, P, T, I, R, O, N, U, G. On affiche donc par exemple pour le niveau moyen :
  ✶✚ ■✚♦→♦ CHA■✚▲❖❍ ▲❖✈✐✚

On notera au passage que la difficulté résulte beaucoup dans la stratégie pour choisir les lettres non encodées, on pourra y réfléchir ...

Question 1 : Comment modéliser ce jeu et permettre d'en créer 3 variantes ?

Question 2 : Implémentez votre solution en TP. Pour simplifier les choses, utilisez comme alphabet de symboles des caractères ascii : chiffres, @, $, *, ...