TP1 – Factory

Programmation par composants

OBJECTIF : Le but de ce TP est de manipuler la notion de Fabrique d'objets (*Factory*) en commençant par un exemple très simple, puis en implémentant une application plus complexe, basée sur des fabriques génériques. Enfin, l'outil *Ant* pourra être abordé.

Cet énoncé, ainsi que toutes les ressources à télécharger mentionnées ci-dessous, sont disponibles à l'adresse http://www-info.iut2.upmf-grenoble.fr/intranet/enseignements/composants/2010-2011/index_files/Page374.htm.

Partie 1 – Client / Serveur

Sans factory

Pour une première approche des fabriques, nous allons créer une application avec un client et un serveur où le client fait appel à une fabrique pour créer le serveur.

- Créer un nouveau projet TP1.
- Modifiez les propriétés du projet
  - Répertoire source : src
  - Répertoire de sortie des fichiers .class : cls
  - Ceci vous permet une meilleure organisation et évite d'avoir un mélange de fichiers sources et binaires.
- Créer un package spécifique pour cette partie.
- Créer l'interface **`IServeur`**. Cocher la case « Generate comments » pour générer certains commentaires dans le fichier source. Après validation, l'interface est créée et son code est affiché. Remarquez les commentaires et suivez les consignes pour les personnaliser avant de poursuivre le TP (« Window > Preferences > Java > Code Style > Code Templates »). Changez au moins les commentaires liés aux fichiers (files) et aux types (types → classes et interfaces). Faire le reste si vous avez le temps.
- Supprimer et recréer l'interface **`IServeur`**. Vérifier que vos modifications (commentaires automatiques) ont été bien prises en compte.
- Déclarer les deux méthodes publiques :
  - **`encode(…)`** qui reçoit en argument un message (chaîne) et qui retourne le code morse correspondant (chaîne).
  - **`decode(…)`** qui réalise l'opération inverse.
- Créer la classe Java (**`Serveur`**) qui implémente la fonctionnalité d'encodage/décodage. Cette classe implémente l'interface **`IServeur`**.
- Pour gagner du temps, téléchargez la classe **`Serveur`** fournie à partir du site web (vous pouvez écraser la classe que vous avez créée avec la classe téléchargée au lieu de faire des copier/coller).

- Créer maintenant un **`Client`** qui permet de tester le serveur.
- Créer la classe **`Client`** dans le même package que le serveur. Cette classe doit comporter une méthode principale (**`main`**) qui permet de lancer l'application. Pour générer cette méthode automatiquement, il suffit de cocher la case correspondante dans le dialogue de création de la classe. Ci-dessous un exemple d'implémentation du client :

```
private IServeur serveur = null;

public Client() {
	serveur = new Serveur();
}

public void appelerServeur() {

	System.out.println("\t*** Appel du Serveur ***");

	String message = new String("MON PREMIER TP");
	String codeResultat = serveur.encode(message);

	System.out.println("Message initial : " + message);
	System.out.println("Code résultat : " + codeResultat);

	String messageResultat = serveur.decode(codeResultat);
	System.out.println("Message résultat : " + messageResultat);
}

public static void main(String[] args) {

	Client client1 = new Client();
	client1.appelerServeur();
}
```

- Tester l'application (en l'exécutant).

Avec Factory

Nous allons maintenant ajouter le concept de *fabrique* (factory) à notre application Client / Serveur. Une fabrique est un objet particulier auquel sera déléguée la responsabilité d'instanciation (création) de certains objets de l'application. Au lieu qu'un client crée lui-même les objets qu'il utilise, il s'adresse plutôt à la fabrique, ce qui permet d'augmenter son indépendance (un pas vers les composants).

Remarquez par exemple que notre client *Morse* crée lui-même l'instance du serveur qu'il doit utiliser. Vous allez alors créer une fabrique spécifique au serveur, que le client va interroger pour demander l'instance dont il a besoin. La Figure 1 montre le principe de fabrique.

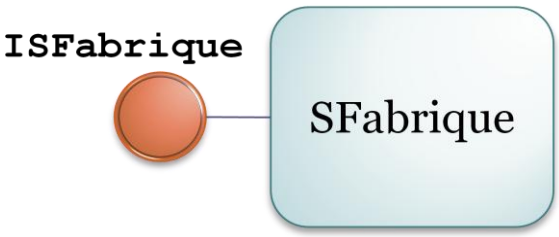

Figure 1 : La classe `SFabrique` implémente l'interface `ISFabrique`.

- D'abord, commencez par créer l'interface de la fabrique (**`ISFabrique`**). Cette interface comporte une seule méthode (**`creerServeur()`**) qui retourne une instance du serveur (de type **`IServeur`** !). Elle ne reçoit rien en arguments.
- Créer la classe d'implémentation de la fabrique (**`SFabrique`**) telle que vous l'avez vue en cours.
- Modifier le client pour utiliser la fabrique. Faire des tests.

```
// Dans le constructeur
public Client(ISFabrique fabrique) {
      serveur = fabrique.creer();
}

// Dans le main
ISFabrique fabrique = new SFabrique();
Client client1 = new Client(fabrique);
```

- Quel problème peut poser cette implémentation ?
- Si vous créez deux clients, combien obtenez-vous de fabriques ?
- Quelle solution pouvez-vous proposer ?
- Implémentez et testez cette solution.

Partie 2 – Application d'affichage de formes

Vous devez maintenant implémenter une application d'affichage de formes. La classe principale est la classe **`Appli`**. Elle propose à l'utilisateur le choix entre différentes formes à afficher (triangles, carrés, ronds, etc.). La forme choisie est alors affichée trois fois « en escalier » : la première forme est tout à gauche, la deuxième est sous la première et décalée vers la droite, de même pour la troisième. La Figure 2 montre un exemple d'affichage produit.

```
****** Application de dessin de formes ******
*** Menu :
1 : Dessiner des triangles
2 : Dessiner des carrés
3 : Dessiner des ronds
4 : Dessiner la forme précédente
5 : Quitter
*** Faites votre choix :
1
  *
 ***
*****
        *
       ***
      *****
              *
             ***
            *****
```

Figure 2 : affichage produit par l'application.

La classe **`Appli`** instancie un client de dessin (classe **`ClientDessin`**) à qui elle demande de dessiner les formes, sans savoir à l'avance quel type de forme ce sera. **`ClientDessin`** fait appel à une fabrique générique pour créer les bonnes formes, puis il leur demande de s'afficher avec le bon décalage. Le diagramme de classes correspondant est fourni Figure 3. La classe **`ClientDessin`** est totalement indépendante des formes qu'elle dessine. Elle ne connait que l'interface **`Forme`** et l'interface **`FormeFactory`**. Lorsque l'utilisateur choisit une forme, **`Appli`** donne à **`ClientDessin`** la bonne fabrique grâce à la méthode **`setFactory(...)`**. A tout moment, **`ClientDessin`** a donc une instance de **`FormeFactory`** de la bonne classe, lui fournissant la forme choisie par l'utilisateur. On peut donc ajouter des formes à volonté, sans changer une ligne de **`ClientDessin`**. De plus, si l'on souhaite transformer cette application en interface graphique, il suffirait de changer la méthode **`afficher()`** des différentes formes, et de modifier **`Appli`**.

Remarque : dans la Figure 3, les flèches ⟶ représentent des liens de dépendance, alors que les flèches ⟶▷ représentent des liens d'héritage ou implémentation d'interface.

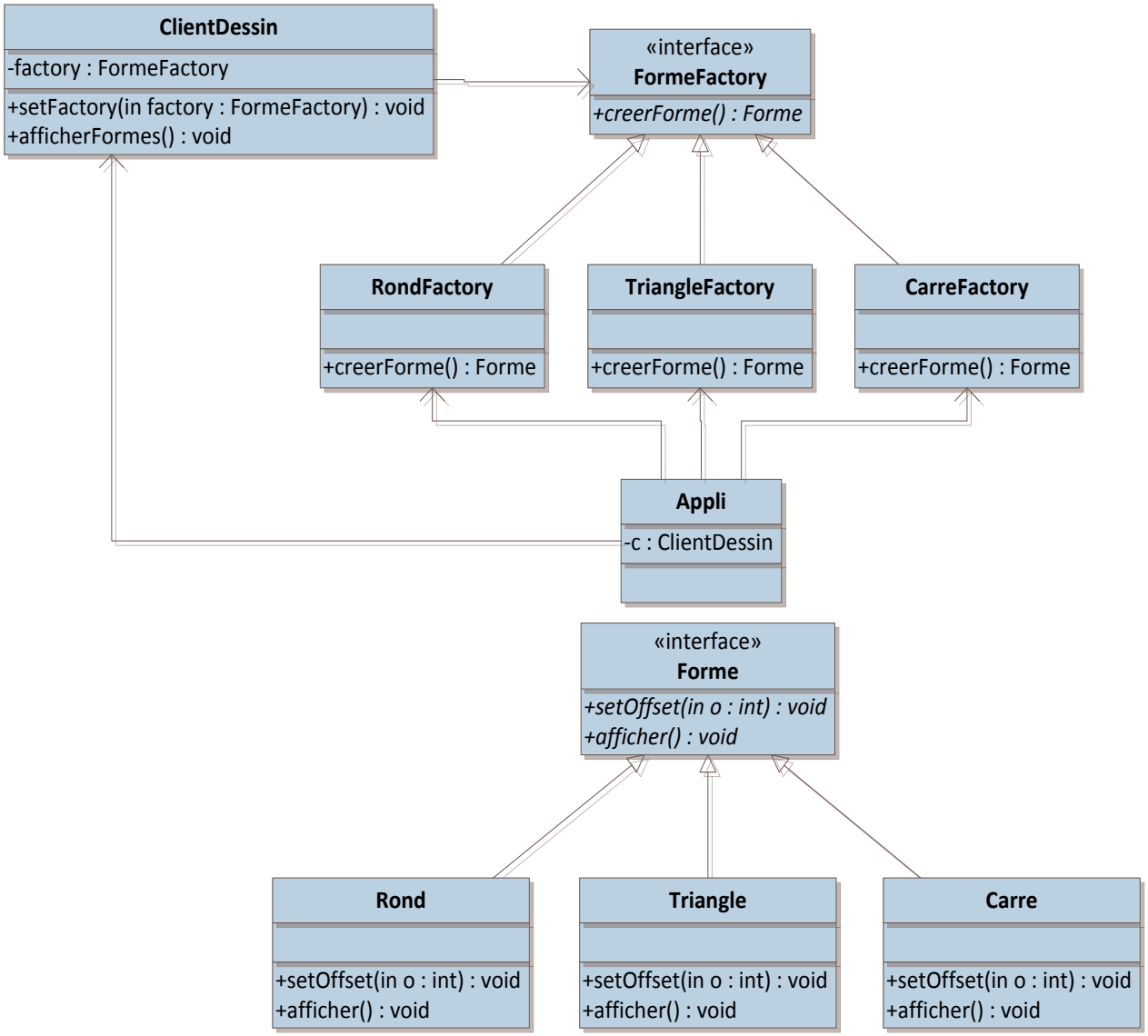

Figure 3 : Diagramme de classes de l'application d'affichage de formes.

Partie 3 – Prise en main de Ant

L'objectif est de créer une tâche **`Ant`** qui permet de packager l'application (**`Client.class`**, **`Serveur.class`**, **`IServeur.class`**, **…**) afin de pouvoir la déployer et la lancer indépendamment d'Eclipse.

- Créer (dans la racine du projet : « new > Folder ») un répertoire « build ».
- Créer le fichier de configuration **`build.xml`** dans le répertoire build. Ce fichier sera interprété par Ant. Remarquez qu'Eclipse intègre un éditeur dédié à Ant et qui permet de créer aisément les fichiers de configuration.

- Lorsque vous commencez l'édition, dès que vous tapez '<', Eclipse vous propose un squelette d'un fichier que vous pouvez utiliser. Vous pouvez également choisir de saisir vous-même entièrement votre fichier. Le script peut ressembler à ceci :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ======================================================================
     26 févr. 09 17:14:57

     TP1
     Mon premier TP

     ====================================================================== -
->
<project name="tp1" default="archiver" basedir="..">
      <description>
            Mon premier TP
      </description>

      <!-- ================================
          target: archiver
        ================================ -->
      <target name="archiver">
            <jar destfile="tp1.jar" basedir="bin" includes="dessin/*">
                  <manifest>
                        <attribute name="Main-Class" value="dessin.Appli" />
                  </manifest>
            </jar>
      </target>

</project>
```

- Lorsque vous terminez l'édition de votre fichier, vous pouvez lancer l'exécution avec Ant (en cliquant avec le bouton droit sur le fichier build.xml et en choisissant « Run > Ant Build ». Ceci n'est possible que si votre **`build.xml`** ne comporte pas d'erreurs.
- Testez le package généré (fichier .JAR) en lançant directement votre application depuis la ligne de commande.