

Exercice : Les Design Patterns

# Enoncé

L’objectif de cet exercice est d’implémenter 4 scénarios différents reposant sur trois Design Patterns. La vérification se fait par le biais de tests unitaires. Le but est de faire passer tous les tests en vert.

Vous disposez pour cela d’une solution comprenant 2 projets :

* ITI.DesignPattern, contenant 4 dossiers correspondant aux différents scénarios (le code écrit ne permet rien de plus que de pouvoir compiler sans erreur).
* ITI.DesignPattern.Tests, contient les tests unitaires.

Pour faire tourner les tests unitaires, exécuter les tests via l’onglet *Test* de Visual Studio.

Pour les autres tests, à vous de faire en sorte qu’ils passent en vert. Pour cela, vous avez le droit de faire ce que bon vous semble dans le projet ITI.DesignPattern (modifier l’implémentation des méthodes, des propriétés ou des constructeurs, ajouter des membres *internal* ou *private*, créer des nouveaux types, etc.).

En revanche, vous ne devez pas modifier le projet ITI.DesignPattern.Tests.

Les tests unitaires sont là pour spécifier la façon détaillée les fonctionnalités attendues.

Voici un petit laïus de ce qu’est un Design Pattern : Les Design Patterns sont des concepts servant à résoudre des problèmes particuliers.

Le pattern du Singleton, par exemple, répond à la question suivante : Comment mettre à disposition un objet unique en tout point de l’application en respectant la conception objet et sans recourir à un objet statique, l’inconvénient des objets statique étant qu’il n’expose que des méthodes static là où un Singleton peut implémenter une interface, être passé en paramètre et utiliser comme un objet normal.

Cependant voici une courte description de ce qui est attendu :

* AbstractFactory (Creational Pattern)
  + Disons que vous avez une usine automobile
    - Cette usine construit des voitures pour différentes marques
      * Les voitures ont donc les mêmes propriétés de base (4 roues, un moteur, un chassis, etc.) mais sont différentes par plusieurs aspects comme la marque, le modèle (une Clio n’est pas une Mégane ou une Polo…), etc.
      * Toutes les marques de voitures peuvent implémenter la fabrique de voiture
      * Cela permettra notamment si vous demandé une Polo de ne pas vous retrouver avec une Twingo… (ce n’est pas le même gabarit, ni la même classe)
  + Pour synthétiser vous avez donc une factory qui possède des factory spécifiques : ce concept permet de générer une multitude d’objets différents à partir de la même classe selon les paramètres qui lui sont fournis
* Observer (Behavioral Pattern)
  + On peut donner comme exemple une compagnie qui écrit des newsletters chaque semaine sur le domaine IT
    - Vous vous inscrivez donc via le site de la compagnie pour recevoir ces newsletters
      * Vous êtes donc sur leur mailing liste d’envoi de news
      * Lorsque la compagnie va créer une newsletter :
        + Vous allez recevoir une notification de ladite newsletter
      * Mais ! Vous n’êtes pas le seul à être abonné aux newsletters de la compagnie, votre amie Michu aussi est abonné
      * Et encore mieux admettons que votre père aussi
      * Tout le monde reçoit donc les notifications lorsqu’une newsletter sort !
    - Et en tant que grand passionné de cuisine, vous vous abonnés également à une autre newsletter, vous en recevrez donc 2 en simultanée
    - Bien entendu, vous pouvez vous désabonner si votre compagnie de cuisine ou d’IT vous a déçu (ce sont des choses qui arrive…) de l’une des deux ou des deux et soit continuer à recevoir uniquement une des deux newsletters, soit aucune.
* Decorator (Structural Pattern)
  + Par exemple, prenons deux étudiants qui sont en retard pour aller en cours :
    - Il peut partir en cours à pied et arriver en retard ou, pour augmenter ces chances d’arriver à l’heure, il met ses rollers avant de sortir
    - On peut donc obtenir deux situations, la première où l’étudiant part à pied et la seconde ou il met ses rollers et va plus vite
  + Comme vous avez pu le comprendre avec cet exemple, le but du Décorateur est de modifier le comportement de l’objet
* AbstractFactoryWithObserverWithDecorator
  + Cette implementation reprend les trois Design Pattern expliqué plus haut.
  + Normalement, toutes les clefs sont entre vos mains et vous devriez être capable de vous en sortir.
  + Petit conseil supplémentaire : ne vous décourager surtout pas, si c’est difficile pour vous, n’hésitez pas à demander de l’aide aux professeurs ou à chercher sur Internet.
  + Bon courage 😊

Il y a 22 tests à passer au vert :

Une image contenant texte

Description générée automatiquement

Au boulot !

Fait par : 🐱‍👤🐉👹