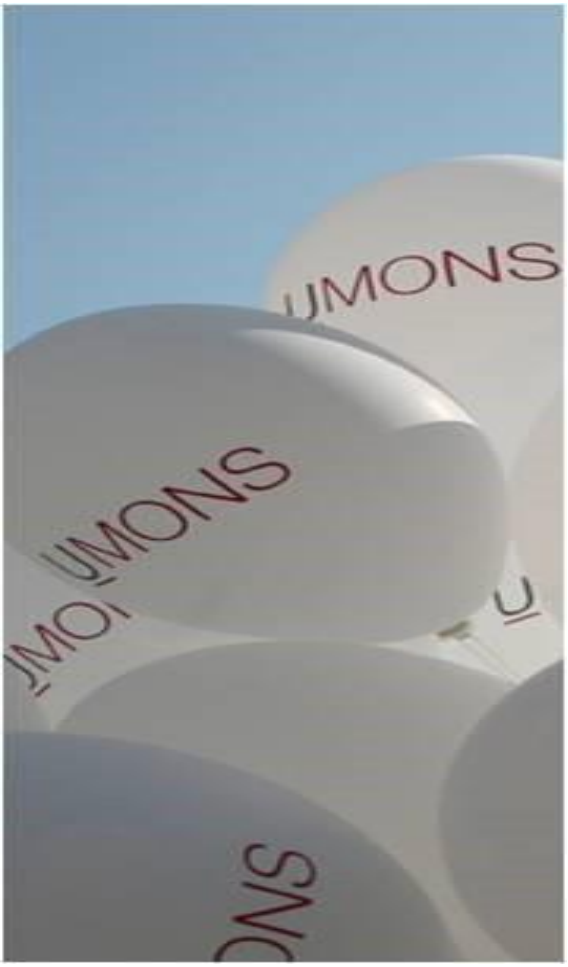
UMONS
Université de Mons

Faculté des Sciences



Développement dirigé par modèles

Séance 3 – Design pattern

Rapport
04 décembre 2021 à 11h
Valentin Marchand

Faculté des Sciences

Sous la direction de :
Prof. Stéphane DUPONT
Ass. Bastien VANDERPLAETSE

Année académique 2021-2022

1 Exercice 1

Question a : Donnez un diagramme de classes en UML 2 représentant cette application.

Diagramme situé dans le dossier exercice 1

Question b : Quel est le design pattern utilisé dans cette application ? Comment pouvez-vous renommer les classes et les méthodes pour que l'existence de ce design pattern transparaisse mieux ?

Le visitor : « ExplorerEntity » pourrais s'appeler « VisitorEntity » et « Explorable » pourrais s'appeler « Element »

Question c : On souhaite créer un deuxième explorateur, semblable à PrettyPrinter, dont la mission consisterait à censurer le groupe de discussion. La censure consiste à remplacer chaque occurrence d'un gros mot par trois étoiles : "***". Étendez le diagramme de classe afin d'ajouter cette fonctionnalité, puis proposez une modification du code source afin d'implémenter la modification. Vos modifications doivent se faire en respectant le design pattern présent. Vous pouvez modifier les classes existantes si vous en ressentez le besoin.

Modification disponible dans exercice 1

2 Exercice 2

Question 1 : Présentez en quelques mots les caractéristiques du design pattern Observer et expliquez dans quel contexte il est intéressant de l'utiliser. Aidez-vous d'un exemple si nécessaire.

On a différent observer qui sont inscrit à un contexte et lors d'un évènement comme une modification sur l'un des observer, le contexte va notifier chaque observer de se changement pour qu'il se met à jour.

Se pattern peut être utiliser pour une interface graphique où l'on a par exemple différents graphiques qui sont les observer, et lorsqu'on modifie l'un des graphiques les autres sont notifier afin de se mettre à jour.

Question 2 : Supposons que l'on souhaite donner une borne supérieure au nombre d'observateurs par sujet, et plus précisément limiter à 2 le nombre d'observateurs pour les objets de type MyCalendar. Modifiez le code source Java afin d'ajouter cette restriction.

Code modifié dans exercice 2

Question 3 : Supposons que l'on souhaite ajouter un nouvel observateur BirthdayView à notre calendrier dont le but serait d'afficher (uniquement) les anniversaires. Modifiez le code source Java afin d'ajouter cette fonctionnalité, en respectant le design pattern Observer.

Code modifié dans exercice 2

Question 4 : Modélisez le diagramme de classe correspondant au code source, tenant compte des ajouts faites en question (2) et (3).

Diagramme dans le dossier Exercice2

3 Exercice 3

Question 1 : Prédisez le résultat (succeed, failure ou error) pour les tests test1(), test2() et test3(). Pour chacun des résultats, expliquez pourquoi le résultat se produit.

Test1 : succeed, on regarde les évènements du CalendarView qui peut avoir n'importe quel évènement.

Test2 : succeed, l'erreur est gérée par l'exception « MinimalSizeException » et aucune vérification n'est effectué.

Test3 : succeed, comme le calendrier doit au moins avoir 1 observer la fonction detach renvoi bien l'exception « MinimalSizeException »

Question 2 : Suite à vos modifications du code source dans l'exercice précédente (partie 2), écrivez un test unitaire test4() afin de vérifier qu'une exception est bien lancée si plus de deux observateurs sont ajoutés au calendrier.

```
@Test
  public void test4(){
    cal.attach(new CalendarView("copy of Tom's calendar"));
    assertThrows(MaximalSizeException.class,()->{cal.attach(new Calendar-
View("copy of Tom's calendar'"));});
  }
```

Question 3 : Suite à vos modifications du code source, écrivez un test unitaire test5() qui modélise un scénario dans lequel un observateur BirthdayView est attaché à un calendrier et qui vérifie si tous les événements de type Birthday du calendrier sont bien présents dans la liste de l'observateur BirthdayView.

```
@Test
  public void test5(){
    ArrayList<Event> lst = b.getEvents();
    for(Event e:calList){
      if(e.getType() == EventType.Birthday){
        assertTrue(lst.contains(e));
      }
    }
  }
```

4 Exercice 4

Question 1 : Introduisez une interface Displayer disposant d'une méthode public String getDisplay(ClockTimer clock) qui retourne une chaîne de caractères correspondant à l'heure de l'horloge. Implémentez également la classe SimpleDisplayer, sous-classe de Displayer, qui n'affichera que les heures et les minutes.

Question 2 : Modifiez la classe ClockTimer pour qu'elle utilise un objet de type Displayer pour l'affichage plutôt que de décider elle-même du format.

Question 3 : Implémentez une classe abstraite DisplayDecorator qui implémente l'interface Displayer et créez les classes DateDisplayer et SecondsDisplayer qui permettront d'afficher la date et les secondes respectivement. Ces classes utiliseront le design pattern Decorator pour ce faire. Il doit être possible de n'afficher, en plus des heures et minutes, que les secondes, que la date ou les deux en même temps.