IUT de BAYONNE PAYS BASQUE - DÉPARTEMENT INFORMATIQUE
Semestre 6- 2024-2025

R6.A06.D05 Alt : Maintenance applicative
Feuille TD-TP n° 2

Refactoring avec patron de conception - Panier
éléments de correction

Refactoring des applications Jva / J2EE Jean-Philippe Rétaillé

Table des matières

1 Introduction

1.1 Présentation de l'application

L'application contient 3 classes :

- Classe Panier : dont l'attribut **contenu** est une liste contenant les produits achetés par un client (pas de lien à Client pour simplifier l'application)
- Classe GestionDeStock : dont la méthode traite (contenu) se charge de traiter le contenu acheté par le client (par exemple, déduire du stock les produits achetés du panier, …)
- Classe Compta : dont la méthode traite(contenu) qui se charge de traiter financièrement le contenu du panier. Par exemple : éditer la facture, …

Lorsque le client est satisfait du contenu de son panier, il déclenche la commande. La méthode declencherCommande() se charge de répercuter les actions correspondantes dans les stocks et la comptabilité.

```
package org.example;

import java.util.ArrayList;

public class Panier {
    private GestionDeStock stock;
    private Comptabilite compta;
    private ArrayList<String> contenu;

    public Panier (GestionDeStock pStock, Comptabilite pCompta) {
        this.stock = pStock;
        this.compta = pCompta;
        this.contenu = new ArrayList<>();
    }

    public void declencherCommande() {
        this.stock.traite(this.contenu);    // lien direct entre Panier et Stock
        this.compta.traite(this.contenu);
    }
}
```

```
package org.example;

import java.util.ArrayList;

public class GestionDeStock {
    private String gestionStock;

    public GestionDeStock(String pGestionStock) {
        this.gestionStock = pGestionStock;
    }
    public void traite(ArrayList<String> contenu) {
         contenu.add("contenu traite par " +  this.gestionStock);
    }
}
```

```
package org.example;

import java.util.ArrayList;

public class Comptabilite {
    private String comptabilite;

    public Comptabilite(String pComptabilite) {
        this.comptabilite = pComptabilite;
    }
    public void traite(ArrayList<String> contenu) {
        contenu.add("contenu traité par " + this.comptabilite);
    }
}
```

1.2 Problématiqur de l'application

La méthode **`traite()`** de la classe **`Panier`** est chargée de déclencher des actions dans les classes **`GestionDeStock`** et **`Comptabilité`**, alors que ces traitements ne la concernent pas vraiment.

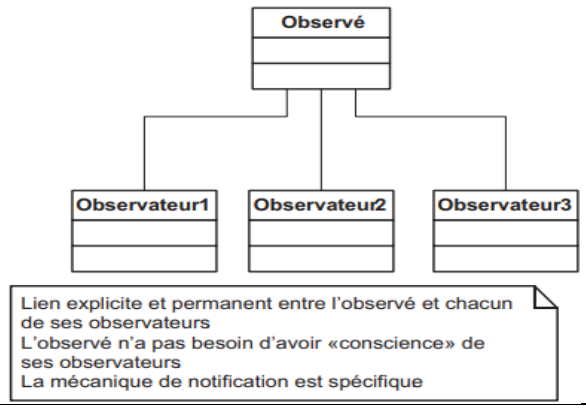
Le **`Panier`** entretient une relation unidirectionnelle vers **`Compta`** et **`GestionDesStocks`** (Observé vers Observateurs) qui est 'en dur', c'est à dire que la classe la classe **`Panier`** connaît explicitement ses observateurs, créant ainsi une dépendance au niveau du code (augmentation du couplage *afférent* = classes dépendant de cette classe, cf R5-Qualité de dév).

Fondamentalement, la classe **`Panier`** n'a pas besoin d'avoir ce lien direct. La validation d'une commande doit être vue par le panier comme un événement déclenchant plusieurs traitements indépendants, dont le détail n'a pas à être connu. L'idée est de créer une relation plus générique et dynamique, plus facile à maintenir.

L'utilisation du patron de conception Observation permettra à la classe observée (**`Panier)`**, de ne plus s'adresser directement aux classes observateurs qui dépendent d'elle lors du déclenchement d'une commande.

1.3 Patron Observation

1.3.1 Problématique traitée



Au cours de sa vie, un objet peut être amené à changer plusieurs fois d'état. Au niveau de l'application, ces changements d'état génèrent des traitements.

Par exemple, pour un traitement de texte, la modification du fichier ouvert doit activer la fonctionnalité d'enregistrement.

Si la classe de notre objet a la charge d'effectuer tous les traitements liés à ces changements d'état, nous pouvons arriver rapidement à une classe obèse, difficile à maintenir

1.3.2 Patron Observateur et mise en oeuvre en Java

Le design pattern observateur est simple à implémenter avec Java. L'API standard de J2SE fournit une interface (java.util.Observer) et une classe (java.util.Observable) offrant la base nécessaire

- **L'interface Observer** doit être implémentée par les classes des *observateurs*. Cette interface ne comprend qu'une seule méthode, **`update()`** (= **`réagir()`** dans le diagramme UML ci-dessous vu en R3.04). Cette méthode est appelée pour notifier l'observateur d'un changement au niveau du sujet d'observation.
- **La classe Observable** doit être utilisée par la classe *observée*. Cette classe fournit la mécanique d'inscription et de désinscription des observateurs (méthodes **`addObserver()`** et **`deleteObserver()`**) ainsi que la mécanique de notification (méthode **`notifyObservers()`**).

1.3.3 Utilisations possibles de la classe Observable

L'utilisation d'Observable par la classe observée peut se faire de deux manières.

- La première consiste à employer l'héritage, avec toutes les contraintes que cela impose (impossibilité d'hériter d'autres classes). C'est le principe vu en R3.04 et rappelé sur le diagramme ci-dessous :

- Tout `Observable` connaît les `Observateurs` intéressés par son changement d'état (cf. `mesObservateurs`).
- Lorsqu'un `Observable` change d'état il notifie tous ses observateurs (cf. méthode `notifierObservateurs`) afin que chacun d'eux puisse `réagir` à sa façon.
- Chaque `ObservateurConcret` connaît l'`ObservableConcret` qui l'intéresse.

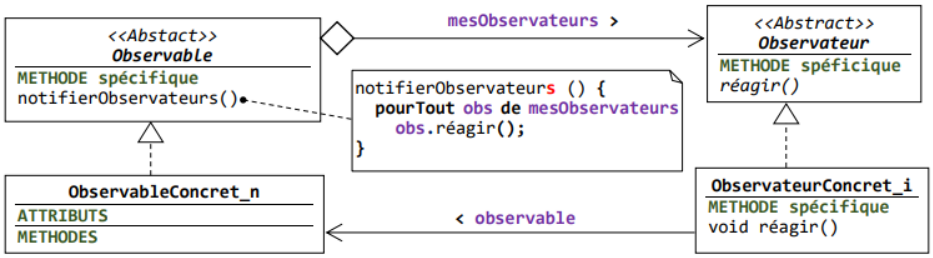
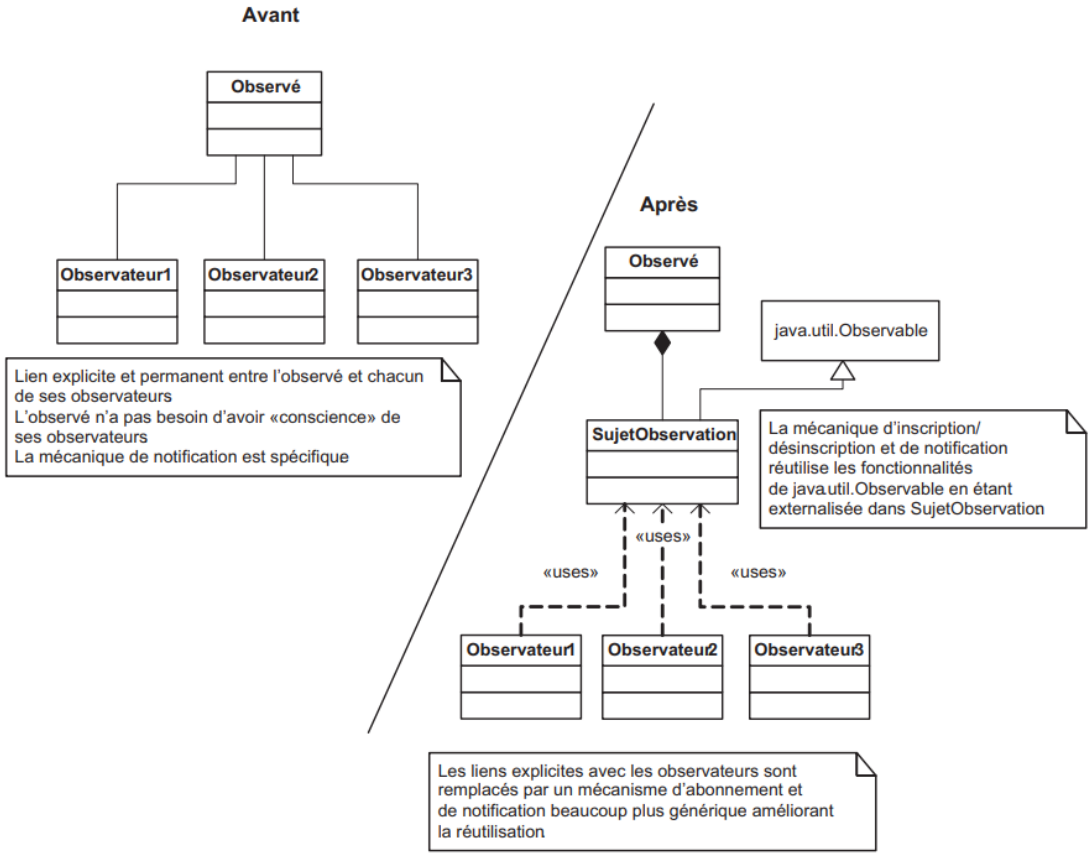


Diagramme.1 *- Représentation Schématique UML du patron Observateur*

– La seconde consiste à créer une **classe interne héritant d'Observable**. Cette dernière manière nous semble plus adaptée à un contexte de refactoring. Elle offre de surcroît davantage de flexibilité dans le cas où la classe observée comporte plusieurs sujets d'observation (il suffit de créer **une classe interne par sujet** d'observation)



2 Appliquer le patron Observation dans l'application existante

2.1 Identifier le Sujet d'Observation

- Le Sujet d'Observation de la classe **`Panier`** est le déclenchement d'une commande.
- Création d'une classe interne à **`Panier`**, nommée **`declenchementCommande`**, héritant de la classe Observable
  - Elle devra contenir une méthode **`notifyObservers()`**
- Modification d la classe Panier
  - Ajouter une instance de cette classe
  - Ajouter une méthode **`getSujetObservation()`** permettant d'accéder à cette instance
  - Modifier la méthode **`declencherCommande()`** afin qu'elle notifie ses observateurs via la méthode **`notifyObservers()`** de la classe interne **`declenchementCommande`**

2.2 Observateurs

- Ce seront les classes **`GestionDeStock`** et **`Compata`**
  - Elles implémenteront l'interface **`Observer`**
  - Ajouter une méthode **`update()`** dont le contenu sera de traiter le contenu du panier qui lui aura été passé par la méthode **`notyfyObserver()`** de la classe **`DeclenchementCommande`**

3 Avant de commencer : écrire les tests

3.1 Identifier les scénarios de test

1 seul scénario :
void devrait_activer_le_traitement_de_contenu_par_Comptabilite_et_GestionDesStocks()

3.2 Le test

```
package org.example;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import java.util.Observable;

import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.*;

class PanierTest {
    private GestionDeStock stock;
    private Comptabilite compta;
    private Panier panier;

    @BeforeEach
    void setUp() {
        stock = mock(GestionDeStock.class);     // création d'un mock
        compta = mock(Comptabilite.class);      // idem
        panier = new Panier(stock, compta);
    }

    @Test
    void testGetContenu() {
        assertThat(panier.getContenu()).isEqualTo("Contenu du Panier");
    }

    @Test
    void testDeclencherCommande() {
        Panier.DeclenchementCommande sujet = panier.new DeclenchementCommande();

        sujet.addObserver(compta);
        sujet.addObserver(stock);
        sujet.setChanged();
        sujet.notifyObservers(panier.getContenu());

        verify(compta, times(1)).update(any(Observable.class),
               eq("Contenu du Panier"));
        verify(stock, times(1)).update(any(Observable.class),
              eq("Contenu du Panier"));
    }
}
```

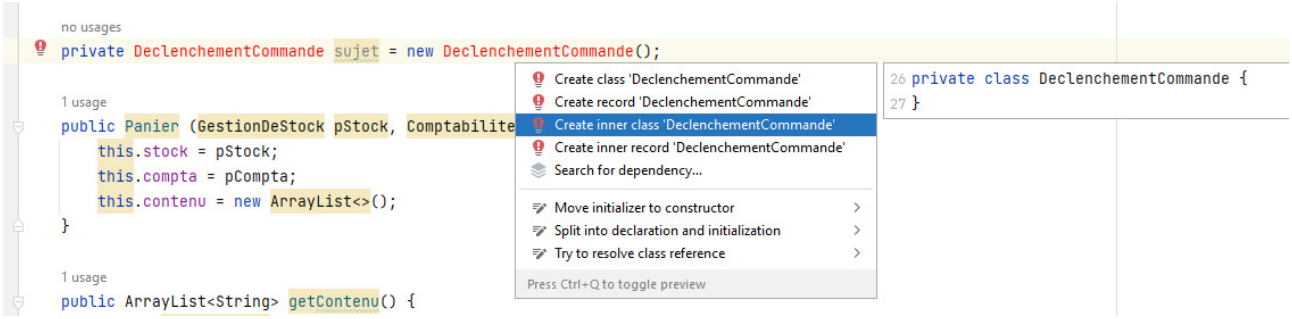
3.3 Imports nécessaires sur la classe de tests

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.catchThrowable;
import static org.assertj.core.api.AssertionsForClassTypes.assertThat;
```

4 Refactoring - étape 1 : Création de la classe interne DeclenchementCommande

4.1 Dans la classe Panier

- Création de l'attribut privé, puis curseur sur le nom de la classe et **Alt + Entrée**



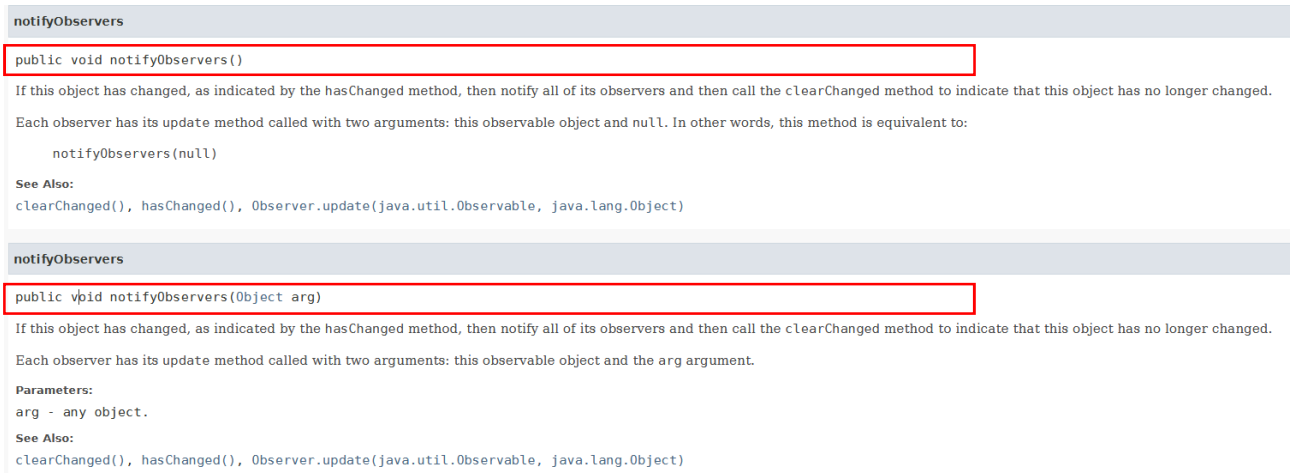
- La classe interne est créée dans la classe Panier :

```
public class DeclenchementCommande {
}
```

4.2 Compléter la classe DeclenchementCommande

- Elle hérite de la classe Java Observable
- Ajouter les 2 méthodes **`notifyObservers`**, 1 avec paramètre, l'autre sans paramètre. Plus tard, nous utilierons celle avec paramètre, puisque l'on notifiera les Observateurs **`Compta`** et **`GestionDeStock`** en leur passant en paramètre le contenu du panier

  https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html



4.3 Dans Panier : déclencher la notification

- On modifie la méthode **`declencherCommande()`** pour qu'elle appelle la méthode **`notigyObservers()`** de la classe **`DeclenchementCommande`**

```
public void declencherCommande() {

    sujetObservation.notifyObservers(this.contenu);
    this.stock.traite(this.contenu);
    this.compta.traite(this.contenu);
}
```

- Les 2 autres instructions disparaîtront plus tard, lorsque les classes **`GestionDesStocks`** et **`Comptabilite`** auront été modifiées elles aussi.

4.4 Résultat pour les 2 classes

```
package org.example;

import java.util.ArrayList;
import java.util.Observable;

public class Panier {
    private GestionDeStock stock;
    private Comptabilite compta;
    private String contenu;

    private DeclenchementCommande sujet = new DeclenchementCommande();

    public Panier (GestionDeStock pStock, Comptabilite pCompta) {
        this.stock = pStock;
        this.compta = pCompta;
        this.contenu = new String("Contenu du panier") ;
    }

    public String getContenu() {
        return this.contenu;
    };

    public void declencherCommande() {
        this.sujet.notifyObservers(getContenu()) ;

        this.stock.traite(this.contenu);
        this.compta.traite(this.contenu);
    }

public class DeclenchementCommande extends Observable {
    public void notifyObservers() {
        super.setChanged(); // indique que le sujet d'observation a changé
d'état
        super.notifyObservers();
    }

    public void notifyObservers(Object pObject) {
        super.setChanged(); // indique que le sujet d'observation a changé
d'état
        super.notifyObservers(pObject);
    }
}

}
```

5 Mise en conformité des Observateurs

Il s'agit des classes Comptabilité et GestionDeStocks

Elles implémentent l'interface Observer, il n'y a qu'une méthode à implémenter : méthode update()

5.1 La doc Java

java.util

Interface Observer

`public interface Observer`

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:

`JDK1.0`

See Also:

`Observable`

Method Summary

All Methods	Instance Methods	Abstract Methods

Modifier and Type	Method and Description
`void`	`update(Observable o, Object arg)` This method is called whenever the observed object is changed.

Method Detail

update

```
void update(Observable o,
            Object arg)
```

This method is called whenever the observed object is changed. An application calls an `Observable` object's `notifyObservers` method to have all the object's observers notified of the change.

Parameters:

`o` - the observable object.

`arg` - an argument passed to the `notifyObservers` method.

5.2 La modification des classes Comptabilité et GestionDesStocks = ajout de la méthode update()

```
package org.example;

import java.util.ArrayList;
import java.util.Observable;
import java.util.Observer;

public class GestionDeStock implements Observer {
    private String gestionStock;

    public GestionDeStock(String pGestionStock) {
        this.gestionStock = pGestionStock;
    }
    public void traite(String contenu) {
         contenu.add("contenu traite par " +  this.gestionStock);
    }
    public void update(Observable pSujet, Object pObject) {
        traite((String) pObject);
    }
}

/*  Généralisation
Si cette classe est Observateur de plusieurs sujets d'Observation, sa méthode
sera la suivante :
    public void update(Observable pSujet, Object pObject) {
        if (pSujet instanceof Panier.DeclenchementCommande)
        {traite ((ArrayList)pObject);}
    }

    La méthode est traitéeprévue pour traiter différents sujets d'observation.
    L'idntification des sujets se fait en testant sa classe.
 */
```