

Interfaces et Patron observer



Abdelmoumène Toudeft, maitre d'enseignement

abdelmoumene.toudeft@etsmtl.ca

1.	Interfaces.....	2
1.1.	Définition d'une interface	2
1.2.	Implémentation d'une interface	3
1.3.	Propriétés des interfaces.....	4
2.	Patron Observer	6

1. Interfaces

Une interface est comme une classe abstraite qui ne contient que des méthodes abstraites et des constantes (les attributs d'une interface sont automatiquement constants).

1.1. Définition d'une interface

Une interface est définie à l'aide du mot-clé **interface** au lieu de **class** :

```
public interface MonInterface {  
    void methode1(int x, double y);  
    double methode2(String s);  
}
```

Une interface permet de définir (ou plutôt d'imposer) la liste des méthodes que des classes doivent fournir pour interagir avec.

Par exemple, il existe différents types de messages : message texte, message audio, message vidéo, ... On peut imposer que tous les types de messages fournissent une méthode **lire()** pour lire le message :

```
public interface Message {  
    void lire();  
}
```

Autre exemple, on peut définir une interface **Pile** pour imposer les méthodes qu'une pile doit fournir, indépendamment de la façon dont elle est implémentée :

```
public interface Pile {  
    boolean empiler(Object x);  
    Object depiler() throws Exception;  
    boolean estVide();  
    Object peek() throws Exception;  
    int taille();  
    void vider();  
}
```

On peut faire la même chose pour les files (queues) et les listes.

1.2. Implémentation d'une interface

Une classe peut **implémenter une interface**. Elle est alors **obligée** de définir toutes les méthodes de l'interface (sinon la classe doit être abstraite) :

```
public class MaClasse implements MonInterface {
    //attributs et méthodes...
    //Définition des 2 méthodes de l'interface :
    void methode1(int x, double y) {
        //...
    }
    double methode2(String s) {
        //...
    }
}
```

Par exemple, l'interface **Message** va être implémentée par toutes les classes de messages :

```
public class MessageTexte implements Message {
    //attributs et méthodes...
    //Définition de la méthode de l'interface :
    void lire() {
        //lire le message texte
    }
}
public class MessageAudio implements Message {
    //attributs et méthodes...
    //Définition de la méthode de l'interface :
    void lire() {
        //lire le message audio
    }
}
public class MessageVideo implements Message {
    //attributs et méthodes...
    //Définition de la méthode de l'interface :
    void lire() {
        //lire le message vidéo
    }
}
```

De même, les classes **PileStatique**, **PileChaineSimple** et **PileChaineDouble** vont implémenter l'interface **Pile** et fournir les mêmes méthodes imposées par l'interface mais implémentées différemment dans chacune des classes :

```
public class PileStatique implements Pile {
    //...implémentation statique avec tableau
}
```

```
public class PileChaineSimple implements Pile {
    //...implémentation avec chainage simple
}

public class PileChaineDouble implements Pile {
    //...implémentation avec chainage double
}
```

En *Java*, une classe ne peut dériver que d'une classe mère (héritage simple) mais **peut implémenter plusieurs interfaces** :

```
public class MaClasse2 extends MaClasse1 implements
    MonInterface1, MonInterface2, MonInterface3 {
    //attributs et méthodes...
    //Définition de toutes les méthodes de toutes les interfaces
    //...
}
```

1.3. Propriétés des interfaces

✓ On peut faire de l'héritage (même multiple) d'interfaces :

```
public interface MonInterface1 {
    void f();
}

public interface MonInterface2 {
    void g();
}

public interface MonInterface3 extends MonInterface1 {
    void h();
}
```

MonInterface3 contient les méthodes **h()** et **f()** (héritée de **MonInterface1**).

```
public interface MonInterface4 extends MonInterface1,
    MonInterface2 {
    void r();
}
```

MonInterface4 contient les méthodes **r()**, **f()** (héritée de **MonInterface1**) et **g()** (héritée de **MonInterface2**).

✓ On peut déclarer des variables de type interface :

```
Message message;
File pile;
```

Mais, on ne peut pas instancier une interface :

```
message = new Message(); //NON  
pile = new Pile(); //NON
```

Ces variables doivent référencer des objets de classes concrètes :

```
message = new MessageTexte(); //OK  
pile = new PileChaineSimple(); //OK
```

- ✓ On peut tester si un objet est « instance » d'une interface. Cela veut dire que l'objet est instance d'une classe qui implémente l'interface :

```
PileChaineDouble pile2 = new PileChaineDouble();  
if (pile2 instanceof Pile) { //teste si la classe de pile2  
                           //implémente l'interface Pile.  
}
```

2. Patron Observer

Il arrive souvent qu'un objet **A** veuille être informé qu'un objet **B** a changé pour qu'il réagisse à ce changement.

Par exemple, un personnage d'un jeu est un objet en mémoire dont la position et l'énergie change lorsqu'il fait des actions (bouge, court, ...).

Un objet **vue1** s'occupe d'afficher le personnage sur une scène à l'écran. Cet objet a besoin de redessiner le personnage à sa nouvelle position chaque fois qu'il change de place (bouge, court, ...).

Un autre objet **vue2** s'occupe d'afficher la quantité d'énergie qui reste au personnage. Cet objet a besoin de mettre à jour la quantité affichée chaque fois que le personnage agit (bouge, court, ...).

Comment demander à **vue1** et **vue2** de se rafraichir lorsque le personnage change ?

Première solution : couplage fort

Cette solution est **inacceptable**. Le personnage doit connaître **vue1** et **vue2** et détenir des références vers eux afin de les informer des changements.

Le couplage fort établit des liens forts entre les objets. Il devient alors difficile de faire évoluer un objet sans influencer les autres. Il est aussi difficile de tester les objets de manière isolée les uns des autres.

Deuxième solution : couplage faible

Le personnage ne doit pas connaître **vue1** et **vue2** et doit pouvoir évoluer sans se soucier de quel(s) objet(s) est/sont intéressé(s) par ses changements.

C'est ce que permet de réaliser le **patron de conception Observer**. Ce patron définit 2 types d'objets :

- **Les observables** : objets dont le changement d'état peut intéresser d'autres objets (les observateurs);
- **Les observateurs** : objets qui observent les objets observables et réagissent à leurs changements d'états.

Pour concrétiser ce patron, on définit une interface qui représente les observateurs et une classe (abstraite) qui représente les observables :

```
public interface Observateur {
    //Permet à l'observateur de se mettre à jour lorsqu'il est notifié par
    //l'objet Observable qu'il observe :
    public void seMettreAJour(Observable observable);
}

public abstract class Observable {
    //Liste des objets qui observent cet Observable :
    private ArrayList<Observateur> observateurs = new ArrayList<>();

    //Ajoute un observateur à la liste, s'il n'y est pas déjà :
    public boolean ajouterObservateur(Observateur observateur) {

        boolean ajoutEffectue;

        if (this.observateurs.contains(observateur))
            ajoutEffectue = false;
        else {
            this.observateurs.add(observateur);
            ajoutEffectue = true;
        }
        return ajoutEffectue;
    }

    //Demande à tous les observateurs de se mettre à jour :
    public void notifierObservateurs() {

        ListIterator<Observateur> itérateur = this.observateurs.listIterator();
        while (itérateur.hasNext())
            itérateur.next().seMettreAJour(this);
    }
}
```

Les classes des observateurs (objets *vue1* et *vue2*, par exemple) doivent implémenter l'interface **Observateur** et définir leur réaction au changement de l'observable dans la méthode **seMettreAJour()** :

```
public class ClasseVue1 implements Observateur {
    //...attributs et méthodes

    public void seMettreAJour(Observable observable);

    if (observable instanceof Personnage) {
        Personnage p = (Personnage)observable;
        //chercher la position de p - p.getPosition() - et afficher
        //le personnage p à cette position.
    }
}

public class ClasseVue2 implements Observateur {
    //...attributs et méthodes
```

```
public void seMettreAJour(Observable observable);

    if (observable instanceof Personnage) {
        Personnage p = (Personnage)observable;
        //chercher l'énergie de p - p.getEnergie() - et l'afficher.
    }
}
```

La classe du personnage étend la classe **Observable** (pour rendre les personnages observables) :

```
public class Personnage extends Observable {
    //...attributs et méthodes

    public void bouge() {
        //changer la position et l'énergie...
        //Informer les observateurs :
        this.notifierObservateurs();
    }
    public void court() {
        //changer la position et l'énergie...
        //Informer les observateurs :
        this.notifierObservateurs();
    }
}
```

On peut maintenant connecter les observateurs à l'observable :

```
Personnage personnage = new Personnage();
ClasseVue1 vue1 = new ClasseVue1();
ClasseVue1 vue2 = new ClasseVue2();

personnage.ajouterObservateur(vue1);
personnage.ajouterObservateur(vue2);
```

À partir de là, le personnage peut évoluer à sa guise. À chacun de ses changements, les observateurs **vue1** et **vue2** sont alertés et réagissent en conséquence, chacun à sa manière.

Signalons, pour terminer, que le **modèle de délégation**, utilisé dans la gestion des événements en programmation graphique avec **Swing**, est un autre exemple d'implémentation du patron **Observer**.