R3.04 Conception cours n°6

I.Borne

- Etude de patrons de conception
  - Classement par intention
    - Intention de construction : AbstractFactory
    - Intention d'opération : TemplateMethod
    - Intention de responsabilités : Observer
- Patron d'architecture
  - Model-View-Controller

Les patrons de conception du GoF

purpose / scope	Creational	Structural	Behavioral
class	Factory Method	Adapter (class)	Interpreter Template Method
object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State, Strategy, Visitor

Classification des patterns selon leur intention

Intention	Patterns
Interfaces	Adapter, Facade, Composite, Bridge
Responsability	Singleton, Observer, Mediator, Proxy, Chain of Responsability, Flyweight
Construction	Builder, Factory Method, Abstract Factory, Prototype, memento
Operations	Template Method, State, Strategy, Command, Interpreter
Extensions	Decorator, Iterator, Visitor

Le point de vue par intention

- Interface
- Responsabilité : distribuer les responsabilités dans chaque classe
- Construction : faciliter la construction des objets
- Opération: spécification d'un service
- Extension

1- Intention de construction

- Leur rôle consiste à faciliter la construction d'objets.
- Ce sont des conceptions qui laissent un client construire un nouvel objet par un moyen autre que celui qui consiste à appeler un constructeur de classe.
- Par exemple :
  - Factory Method : différer la décision de quelle classe instancier
  - Abstract Factory : construire un famille d'objets qui partagent une caractéristique.

Motivation (Abstract Factory)

Motivation

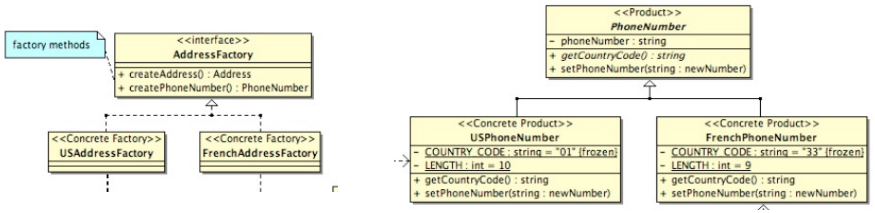
Répertoire de numéros de téléphone qui génère automatiquement le numéro selon la région et le pays.

– +33 2 97 62 64 46

Evolution pour d'autres numéros d'autres pays

- par exemple USA : +1 202 738 94 63

Modélisation des numéros de téléphone



Une hiérarchie de produits à laquelle on associe
une hiérarchie de créateurs des produits qui définissent une factory method :

```
public PhoneNumber createPhoneNumber(){
        return new FrenchPhoneNumber();
}
```


Abstract Factory (creationnal)

création - construction

Intention

Fournir une interface pour créer des familles d'objets sans avoir à spécifier leurs classes concrètes. L'interface délègue les appels de création à une ou plusieurs classes concrètes afin créer des objets spécifiques.

S'utilise quand

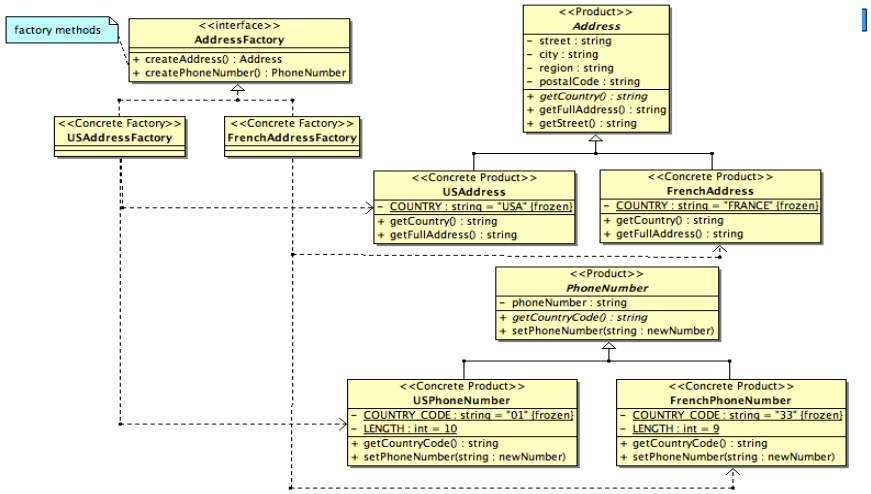
La création des objets doit être indépendante du système qui va les utiliser.

Les familles d'objets doivent être utilisées ensemble.

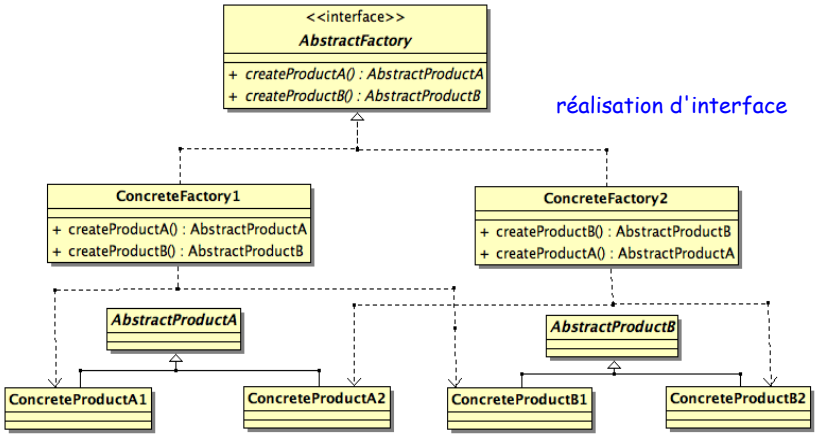
Les bibliothèques doivent être publiées sans exposer les détails d'implémentation.

Les classes concrètes sont découplées des clients.

Exemple `AddressFactory`



AbstractFactory (avec interface)



Bénéfices

- AbstractFactory aide à accroître la flexibilité générale d'une application, lors de la conception et à l'exécution.
- Lors de la conception on ne peut pas prédire toutes les utilisations futures d'une application : framework générique.
- A l'exécution l'application peut facilement intégrer de nouvelles caractéristiques et ressources.
- Pour tester on peut prévoir des classes : TestConcreteFactory, TestConcreteProduct.
- Il est important de bien définir l'interface générique des produits abstraits.

Patterns utilisés avec AbstractFactory

- Factory Method : utilisé pour implémenter la *factory* abstraite.
- Singleton : souvent utilisé dans la *factory concrete*

2- Intention d'opérarion

- Une opération est une spécification d'un service qui peut être requis d'une instance d'une classe.
- Une méthode est une implémentation d'une opération.
- Un algorithme est une procédure (une séquence d'instructions) qui prend des valeurs en entrée et produit des valeurs en sortie.

- Le polymorphisme permet de distribuer une opération parmi plusieurs classes.

Objectifs de Template Method

Définir le squelette d'un algorithme à l'aide d'opérations *abstraites* dont le comportement concret se trouvera dans les sous-classes, qui implémenteront ces opérations

Cette technique, très répandue dans les classes abstraites, permet de :

- fixer clairement des comportements standards qui devraient être partagés par toutes les sous-classes, même lorsque le détail des sous-opérations diffère ;
- factoriser du code qui serait redondant s'il se trouvait répété dans chaque sous-classe.

Particularités de Template Method

La technique du patron de méthode a ceci de particulier que c'est la méthode de la classe parent qui appelle des opérations n'existant que dans les sous-classes.

C'est une pratique courante dans les classes abstraites, alors que d'habitude dans une hiérarchie de classes concrètes c'est le contraire : ce sont plutôt les méthodes des sous-classes qui appellent les méthodes de la super-classe comme morceau de leur propre comportement.

Le pattern Template Method (comportement)

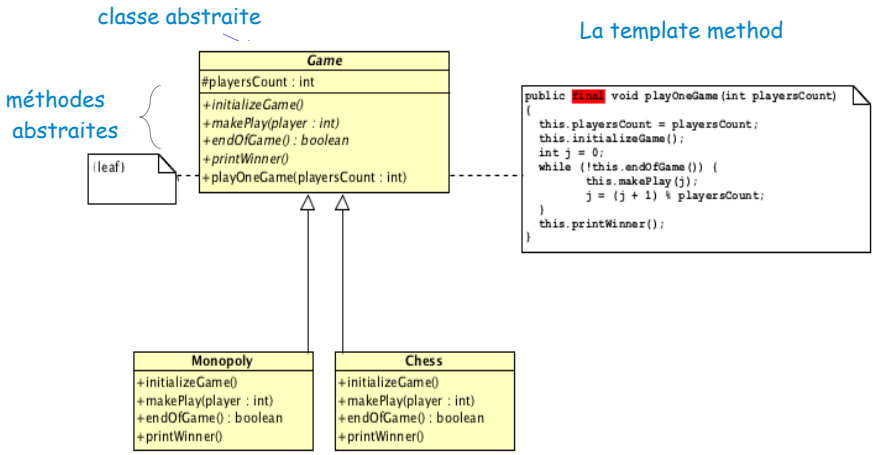
Intention

Fournir une méthode qui permet aux sous-classes de masquer des parties de la méthode sans la ré-écrire.

Applicability (usage)

- Fournir un squelette pour une méthode, permettant aux sous-classes de redéfinir des parties spécifiques de la méthode.
- Centraliser les morceaux d'une méthode qui sont définis dans tous les sous-types d'une classe, mais qui ont toujours une petite différence dans chaque sous-classe.
- Contrôler quelles sont les opérations que les sous-classes doivent masquer.

Exemple de Template Method



La classe Game

```
public abstract class Game {
    protected int playersCount;
    public abstract void initializeGame();
    public abstract void makePlay(int player);
    public abstract boolean endOfGame();
    public abstract void printWinner();

    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        this.initializeGame();
        int j = 0;
        while (!this.endOfGame()) {
            this.makePlay(j);
            j = (j + 1) % playersCount;
        }
        this.printWinner();
    }
}
```

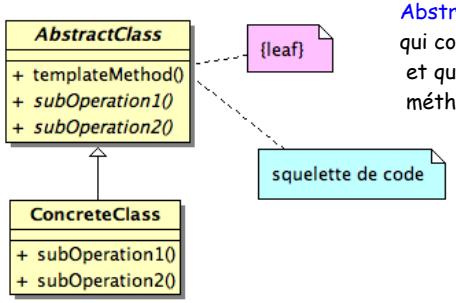

La sous-classe Monopoly

```
public class Monopoly extends Game {

    public void initializeGame() {
        // Initialize money
    }
    public void makePlay(int player) {
        // Process one turn of player
    }
    public boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
       return true;
    }
    public void printWinner() {
        // Display who won
    }
}
```


La sous-classe Chess

```
public class Chess extends Game {
    public void initializeGame() {
        // Put the pieces on the board
    }
    public void makePlay(int player) {
        // Process a turn for the player
    }
    public boolean endOfGame() {
        // Return true if in Checkmate or
        // Stalemate has been reached
      return true;
    }
    public void printWinner() {
        // Display the winning player
    }
}
```

Template method (solution)



AbstractClass : classe abstraite qui contient la template method et qui définit une ou plusieurs méthodes abstraites

ConcreteClass : implémente les méthodes abstraites

Avantages et inconvénients de *Template Method*

Principal bénéfice : promouvoir la réutilisation de code, essentielle pour les frameworks

La *template* méthode appelle trop de méthodes abstraites
Il est préférable de limiter le nombre de méthodes abstraites

Variantes :

- La méthode n'appelle que des méthodes concrètes
- *AbstractClass* fournit une implémentation par défaut des méthodes appelées par la méthode *Template*
- *AbstractClass* n'est pas forcément abstraite.

2- Intention de responsabilité

Les objets possèdent certaines responsabilités :

- un objet travaille sur ses propres données (encapsulation)
- le développement basé sur les objets distribue les responsabilités de telle sorte que chaque objet fait son propre travail.

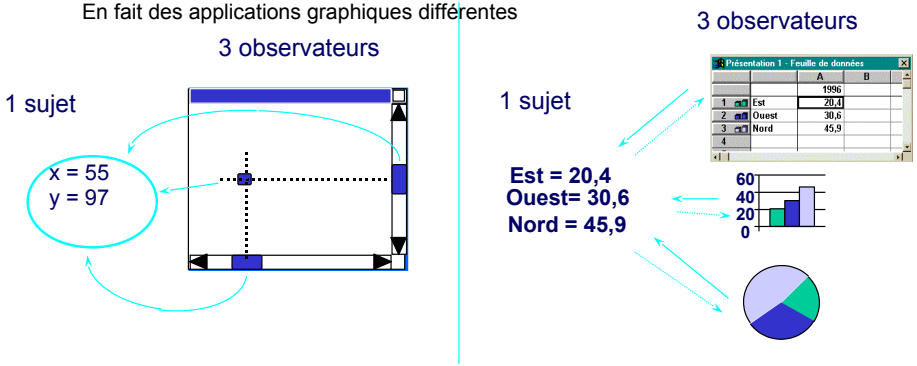
Certains patterns déplacent la responsabilité vers un intermédiaire ou un objet particulier.

Le pattern Observer (behavior)

- **Intention** :
  - définir une dépendance, de multiplicité un à plusieurs, entre des objets de sorte que si un objet change d'état alors tous ses dépendants sont automatiquement avertis.
- **Alias** : Dependents, Publisher-Subscriber

Observer (motivation)

Le système fournit des données pour différents clients.
En fait des applications graphiques différentes



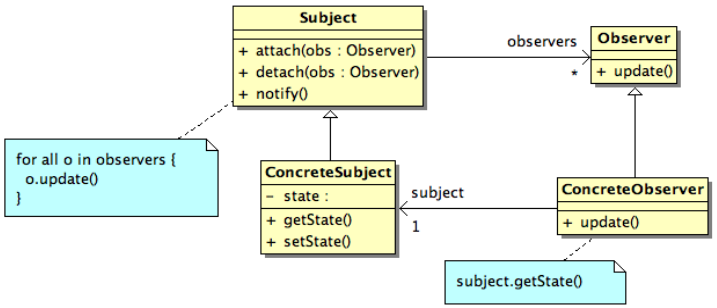
Quand utiliser Observer ?

- Quand une abstraction a deux aspects, l'un dépendant de l'autre.
- Quand un changement pour un objet requiert des changements sur d'autres, et que l'on ne sait pas combien d'objets ont besoin d'être modifiés.
- Quand un objet doit pouvoir avertir les autres sans faire d'hypothèses sur qui ils sont.
- On ne veut pas que ces objets soient étroitement couplés.

Structure pour Observer

Subject : Connaît les observateurs et fournit une interface pour lier et délier les observateurs (attach, detach)

Observer : définit une interface de mise à jour pour les observateurs (update).



Participants

Subject

- Connaît ses *observers*. Un nombre quelconque d'objets *Observer* peut observer un sujet.
- Fournit une interface pour lier et délier les objets de type *Observer*

Observer

- Définit une interface de mise à jour pour les objets qui doivent être avertis des changements du subjet.

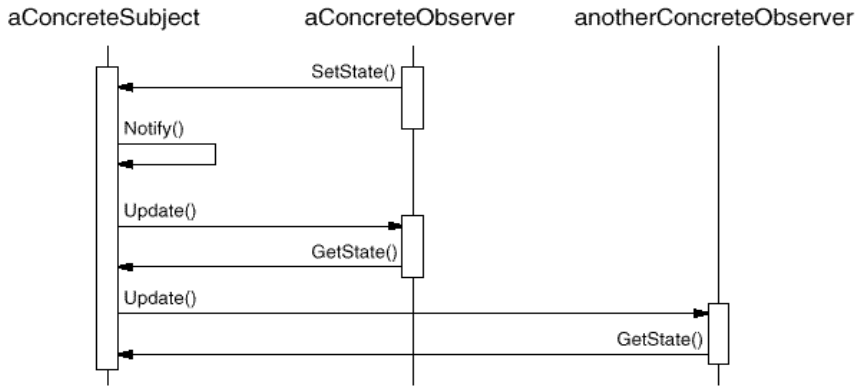
ConcreteSubject

- Envoie une notification à ses observateurs quand son état change, ils recoivent le message update()..

ConcreteObserver

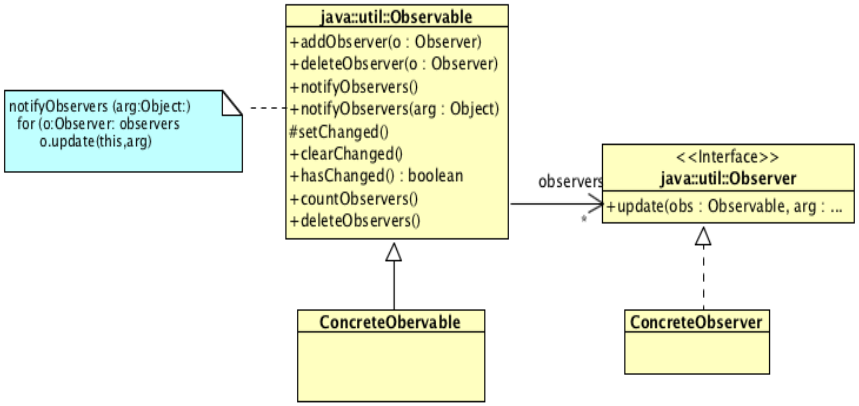
- Maintient une référence sur un objet *ConcreteSubject*.
- Converve l'état qui doit rester cohérent avec celui du sujet.
- implémente l'interface de mise à jour de Observer pour garder son état cohérent avec celui du sujet.

Collaboration entre un sujet et deux observateurs



L'envoi du message SetState() va déclencher le notify() qui envoie le message update() à chaque observateur.

Le pattern Observer en Java (package util)

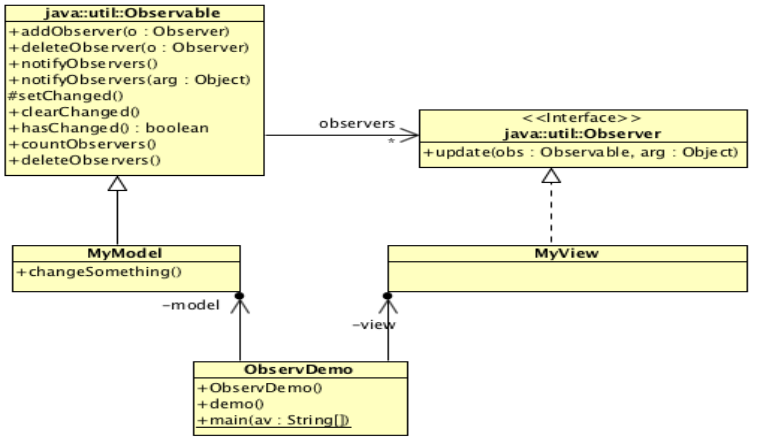


Explications

- Java est apparu après la définition des patrons de conception, c'est pourquoi certains patrons sont intégrés à Java.
- Dans le package java.util on trouve la classe Observable et l'interface Observer
- La classe Observable fournit le code pour attacher, détacher les observateurs et pour les notifier. Donc la classe dont les objets sont à observer doit être une sous-classe de Observable.
- L'interface Observer donne la spécification de la méthode update(). Les classes des objets observateurs doivent implémenter cette interface.

Remarque : ces classes sont indiquées comme « deprecated » depuis Java 9

Exemple utilisant le patron Observer



Code de l'exemple

```
import java.util.Observable;
/** The Observable normally maintains the data */
public class MyModel extends Observable {
    public void changeSomething() {
      // Notify observers of change
      this.setChanged();
      this.notifyObservers();
    }
  }
-------------------------------------------------------------
import java.util.Observer;
  /** The Observer normally maintains a view on the data */
public class MyView implements Observer {
 /** For now, we just print the fact that we got notified. */

    public void update(Observable obs, Object x) {
      System.out.println("update(" + obs + "," + x + ");");
    }
  }
```


La classe lanceur de l'exemple

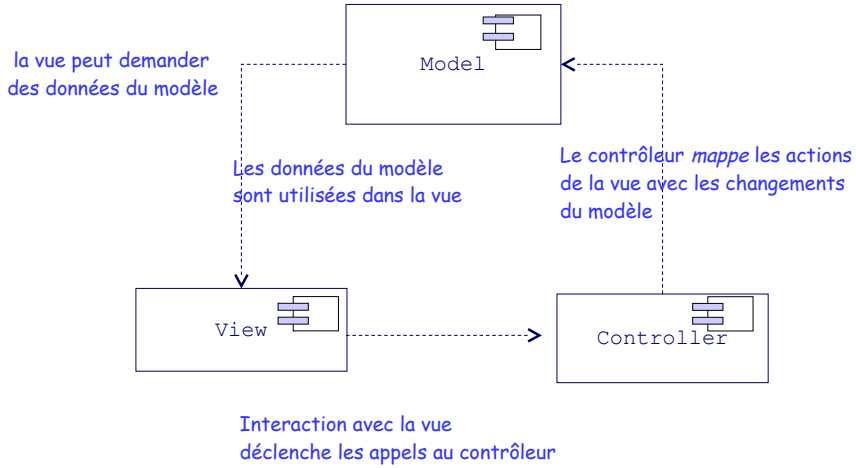
```
import java.util.Observable;

public class ObservDemo{
  private MyView view;
  private MyModel model;

  public ObservDemo() {
    view = new MyView();
    model = new MyModel();
    model.addObserver(view);
  }
  public void demo() {
    model.changeSomething();
  }
  public static void main(String[] av) {
    ObservDemo me = new ObservDemo();
    me.demo();
  }
}
```


Model-View Controller (architecture)

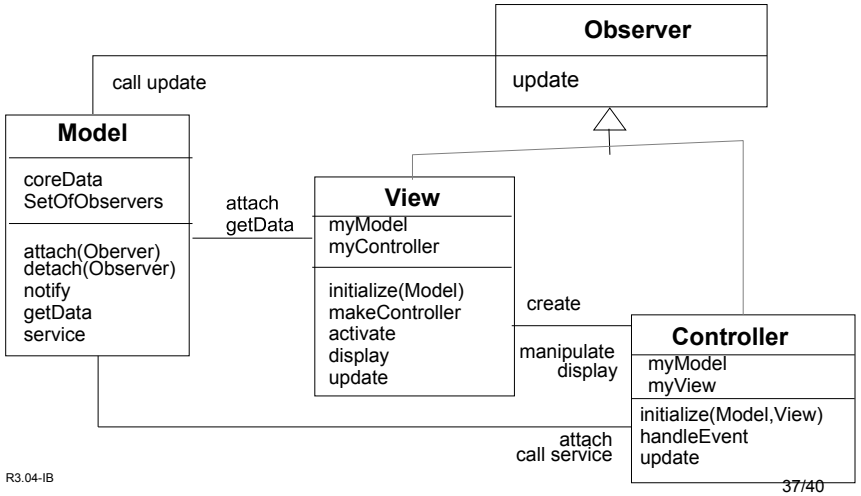
- Diviser un système en trois parties logiques : un modèle , une vue et un contrôleur,
- et faciliter le changement et l'adaptation de chaque partie.

- MVC et un patron d'architecture qui est utile lorsqu'un composant ou un sous-système possède certaines des caractéristiques suivantes :
  - Différentes vues du composant
  - Différents types possibles de comportement (sources multiples d'invocation)
  - Comportement ou représentation change
  - Adapter, réutiliser un composant dans des circonstances variées avec un taux de codage minimum

Structure générique de MVC (composants)

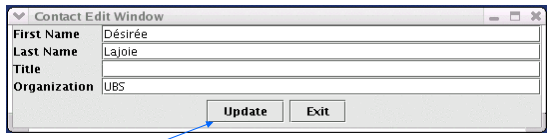


MVC : Structure (classes)



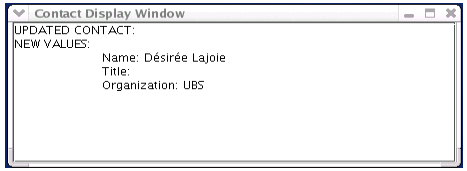
Exemple d'un modèle avec deux vues

la 2ème vue pour mettre à jour le contact



L'ActionListener est le contrôleur, il met à jour l'objet ContactModel

la 1ère vue pour afficher l'information sur le modèle mise à jour



Interface java.util.Observer

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.

classe java.util.Observable

Methods

Modifier and Type	Method and Description
`void`	`addObserver(Observer o)` Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
`protected void`	`clearChanged()` Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the `hasChanged` method will now return `false`.
`int`	`countObservers()` Returns the number of observers of this `Observable` object.
`void`	`deleteObserver(Observer o)` Deletes an observer from the set of observers of this object.
`void`	`deleteObservers()` Clears the observer list so that this object no longer has any observers.
`boolean`	`hasChanged()` Tests if this object has changed.
`void`	`notifyObservers()` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`void`	`notifyObservers(Object arg)` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`protected void`	`setChanged()` Marks this `Observable` object as having been changed; the `hasChanged` method will now return `true`.