Cours - Patron de conception - #4

Observateur

Guillaume Santini

25 janvier 2024

Plan

1 Motivations

2 Mauvaise conception
- Cas d'étude

3 Une bonne conception
- Principes SOLID
- Modélisation
- Implémentation
- Principe d'inversion des dépendances non respecté

4 Le patron de conception Observateur
- Structure du patron
- Implémentation du patron
- Principes de conception

5 Credits

Motivations

Objectifs : définir une relation entre objets de type un-à-plusieurs

- lorsque un objet change d'état, tous ceux qui en dépendent sont notifiés et mis à jour automatiquement.
- La liste des objets notifiés est déterminée (et peut être modifiée) à l'exécution du programme.

Motivations

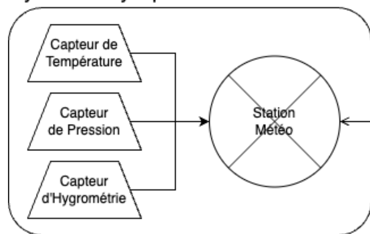
Objectifs : définir une relation entre objets de type un-à-plusieurs

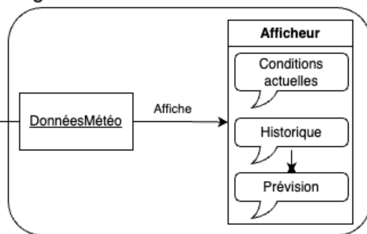
- lorsque un objet change d'état, tous ceux qui en dépendent sont notifiés et mis à jour automatiquement.
- La liste des objets notifiés est déterminée (et peut être modifiée) à l'exécution du programme.

Principe SOLID

- Inversion des dépendances,
- Open/Closed,
- Ségrégation des interfaces.

Cas d'étude

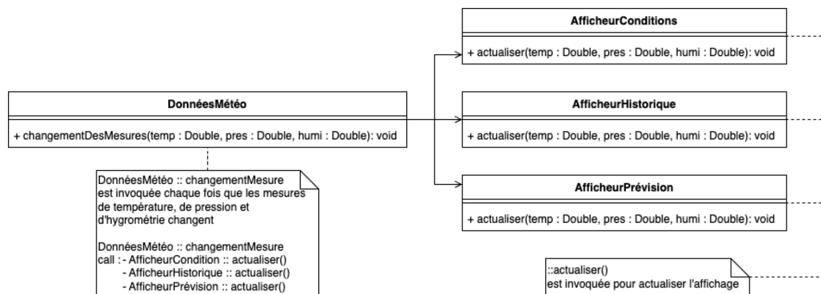




Modélisation d'une station d'affichage d'informations météorologiques

- Une station météo réalise des mesures physiques,
- L'objet DonneesMeteo communique avec la station pour obtenir la température, la pression et l'humidité qu'il communique à l'afficheur,
- Il y a trois options d'affichage : conditions actuelles, statistiques et prévisions.

Composer des classes en fonction des besoins



Composer des classes en fonction des besoins

```
public class DonneesMeteo {

  ...
  private AfficheurConditions aff_cond;
  private AfficheurHistorique aff_hist;
  private AfficheurPrevision aff_prev;

  public void changementMesures(Double temp, Double pres, Double humi){
    ...
    this.aff_cond.actualiser(temp, pres, humi);
    this.aff_hist.actualiser(temp, pres, humi);
    this.aff_prev.actualiser(temp, pres, humi);
  }
}
```

Principe SOLID Open/Closed

- Dès qu'on ajoute ou retire un afficheur, il faut modifier le code de la classe `DonnéesMétéo`.

Principe SOLID Inversion des dépendances

- Couplage fort : La classe `DonnéesMétéo` dépend des 3 classes *concrètes* d'afficheurs.

Mise en œuvre des principes SOLID

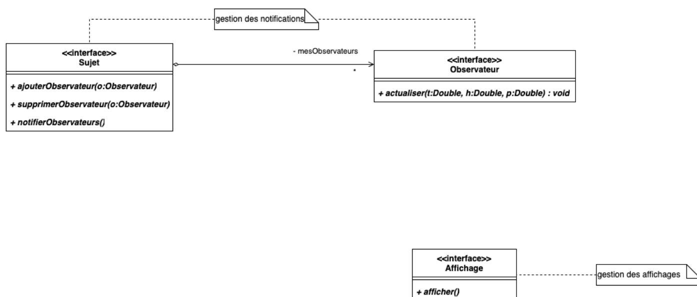
Inversion des dépendances

- Maintient d'un faible couplage entre les classes, la station météo n'a pas a dépendre des implémentations des composants chargés de l'affichage ou des composants devant être notifiés (pour mise à jour des affichages) lorsque les mesures changent.

Open/Closed

- On doit pouvoir ajouter des modalités d'affichages et des agents à notifier des mises à jour des mesures sans avoir à modifier les classes déjà développées (et testées).

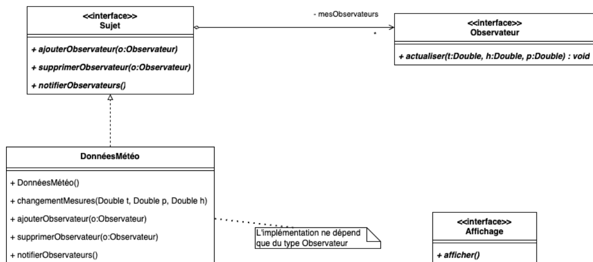
Modélisation #1



Principe SOLID Ségrégation des interfaces

- Séparation des interfaces de déclaration des contrats de notification et d'actualisation (`Sujet` et `Observateur`) et d'affichage (`Affichage`).

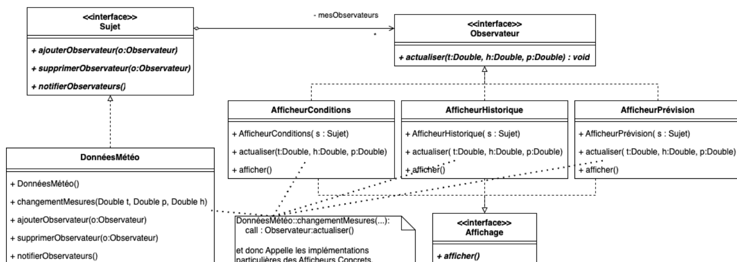
Modélisation #2



Principe SOLID Inversion des dépendances

- La classe `DonneesMeteo` ne dépend que d'un contrat abstrait et non pas d'implémentations (multiples d'affichage ou d'actualisation).

Modélisation #3



Principe SOLID Open/Closed

- Les classes implémentent et dépendent d'interfaces,
- de nouvelles classes implémentant les mêmes interfaces peuvent être développées.

Implémentation #1 - les interfaces

```
public interface Sujet {
  public void ajouterObservateur( Observateur o);
  public void supprimerObservateur( Observateur o);
  public void notifierObservateurs();
}
```

```
public interface Observateur {
  public void actualiser(Double t, Double h, Double p);
}
```

```
public interface Affichage {
  public void afficher();
}
```

Principe SOLID Ségrégation des interfaces

- Séparation des interfaces de déclaration des contrats de notification et d'actualisation (`Sujet` et `Observateur`) et d'affichage (`Affichage`).

Implémentation #2 - la classe DonneesMeteo

```
public class DonneesMeteo implements Sujet {

  private Double temp, press, humi ;

  private ArrayList<Observateur> observateurs ;

  public DonneesMeteo(){ this.observateurs = new ArrayList<Observateur>();}

  public void ajouterObservateur( Observateur o){ this.observateurs.add(o); }

  public void supprimerObservateur( Observateur o){ this.observateurs.remove(o); }

  public void notifierObservateurs(){
    for (Observateur o : this.observateurs) {
      o.actualiser(this.temp, this.humi, this.pres);
    }
  }

  public void changementMesures(Double t, Double p, Double h){
    this.temp = t; this.press = p; this.humi = h;
    this.notifierObservateurs();
  }
}
```

Principe SOLID Inversion des dépendances

- La classe DonneesMeteo ne dépend que d'un contrat abstrait et non pas d'implémentations (multiples d'affichage ou d'actualisation).

Implémentation #2 - la classe DonneesMeteo

```
public class DonneesMeteo implements Sujet {

  private Double temp, press, humi ;

  private ArrayList<Observateur> observateurs ;

  public StationMeteo(){ this.observateurs = new ArrayList<Observateur>();}

  public void ajouterObservateur( Observateur o){ this.observateurs.add(o); }

  public void supprimerObservateur( Observateur o){ this.observateurs.remove(o); }

  public void notifierObservateurs(){
    for (Observateur o : this.observateurs) {
      o.actualiser(this.temp, this.humi, this.pres);
    }
  }

  public void changementMesures(Double t, Double p, Double h){
    this.temp = t; this.p = press; this.humi = h;
    this.notifierObservateurs();
  }
}
```

Principe de fonctionnement

- À chaque fois que l'état de la classe `DonneesMeteo` change d'état (modification de `temp`, `press` et `humi`) les observateurs sont notifiés.

Implémentation #3 - les Observateurs concrets

```
public class AffichageConditions implements Observateur, Affichage {
  private Double temp, humi, pres ;

  public AffichageConditions(Sujet s) {
    s.ajouterObservateur(this) ;
  }
  public void actualiser(Double t, Double h, Double p) {
    this.temp = t ; this.humi = h ; this.pres = p ;
    this.afficher() ;
  }
  public void afficher() {
    System.out.println("temperature = " + this.temp) ;
    System.out.println("humidite = " + this.humi) ;
    System.out.println("pression = " + this.pres) ;
  }
}
```

Principe de fonctionnement

- Lors de leurs instanciations les `Observateurs` s'enregistrent auprès de `DonneesMeteo`,
- L'actualisation(`actualiser()`) de l'`Obervateur` provoque le rafraichissement de l'affichage (`afficher()`).

Implémentation #3 - les Observateurs concrets

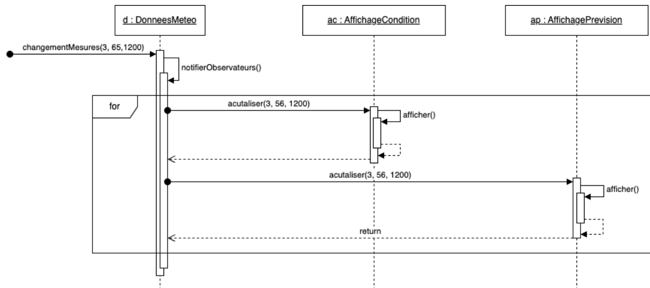
```
public enum Condition {BEAU, VARIABLE, MAUVAIS, TEMPETE}

public class AffichagePrevision implements Observateur, Affichage {

  private Condition prevision ;

  public AffichagePrevision( Sujet s) {
    s.ajouterObservateur(this);
  }
  public void actualiser(Double t, Double h, Double p) {
    if ( p > 1020 ) this.prevision = BEAU ;
    else if ( p < 1020 p >= 1010 ) this.prevision = VARIABLE ;
    else if ( p < 1010 p >= 1000 ) this.prevision = MAUVAIS ;
    else if ( p < 1000 ) this.prevision = TEMPETE ;
    this.afficher();
  }
  public void afficher( ) {
    System.out.println(" prevision = ", this.prevision); }
}
```

Principe de fonctionnement

- Plusieurs implémentations concrètes d'Observateurs peuvent être *définies*.
- Elles implémentent toutes la même interface,
- Elles jouent toutes le même rôle vis-a-vis de la DonneesMeteo.

Utilisation/Exécution

```
public class TestMeteo {
  public static void main(String[] args) {

    DonneesMeteo d = new DonneesMeteo() ;

    Affichage ac = new AffichageConditions(d) ;

    Affichage ap = new AffichagePrevision(d) ;

    d.changementsMesures(3, 65, 1200) ;
  }
}
```



Utilisation/Exécution

```
public class TestMeteo {
  public static void main(String[] args) {

    DonneesMeteo d = new DonneesMeteo() ;

    Affichage ac = new AffichageConditions(d) ;

    Affichage ap = new AffichagePrevision(d) ;

    d.changementMesures(3, 65, 1200) ;
  }
}
```

Objectifs remplis

- Les afficheurs sont notifiés dès que la `DonneesMeteo` change d'état,
- La liste des afficheurs peut être modifiée dynamiquement lors de l'éxécution,
- De nouveaux afficheurs peuvent être développés sans toucher au code existant.

Deux stratégies : Pousser ou tirer les données

```
public class DonneesMeteo implements Sujet {

  public void notifierObservateurs() {
    for( Observateur o : this.observateurs) {
      o.actualiser( this.temp, this.press, this.humi);
    }
  }
}
public class AffichageConditions implements Observateur {

  public void actualiser( Double t, Double h, Double p ) {
    [...]
    [...]
  }
}
```

Pousser les données

- La classe implémentant le `Sujet` concret (*i.e.* `DonneesMeteo`) qui *"pousse"* les données vers la classe implémentant l'`Observateur` concret (*i.e.* `AffichageConditions`).
- Les données sont passées en paramètre de la méthode `actualiser(donnees_poussees)`.

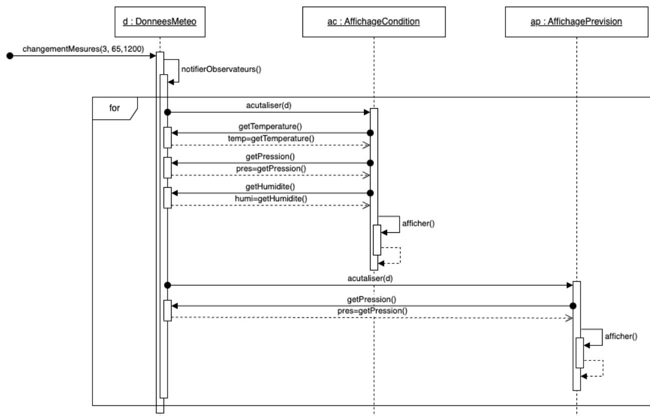
Deux stratégies : Pousser ou tirer les données

```
public class DonneesMeteo implements Sujet {

  public void notifierObservateurs() {
    for( Observateur o : this.observateurs) {
      o.actualiser( this);
    }
  }
}
public class AffichageConditions implements Observateur {

  public void actualiser( Sujet s ) {
    if ( s instanceof DonneesMeteo){
      ((DonneeMeteo) s).getTemperature();
      ((DonneeMeteo) s).getPression();
      ((DonneeMeteo) s).getHumidite();
    }
  }
}
```

Tirer les données

- La classe implémentant l'`Observateur` concret (*i.e.* `AffichageConditions`) *"tire"* les données dont elle a besoin, **et seulement celles dont elle a besoin,**
- Les données sont récupérées par l'appel aux `getters` de la classe concrète implémentant le `Sujet` (*i.e.* `DonneesMeteo`).

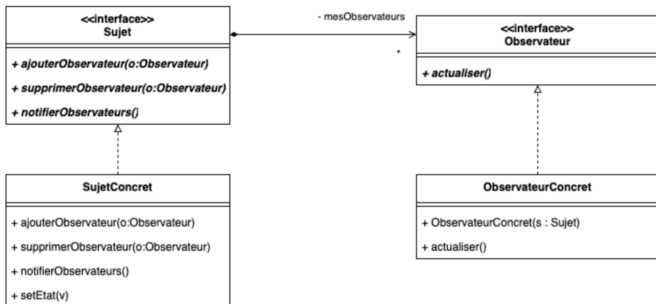
Deux stratégies : Pousser ou tirer les données



Tirer les données

AfficheurPrevision n'a besoin que de la pression contrairement à AfficheurCondition qui a besoin des 3 mesures.

Le patron de conception Observateur



Le patron de conception Observateur

Règles de construction

- Les contrats déclarés par les interfaces `Sujet` et `Observateur` *sont toujours les mêmes*,
- Les instances d'`ObservateurConcret` sont liés par composition avec l'instance de `SujetConcret`,
- Le constructeur de l'`ObservateurConcret` prend en paramètre une référence vers l'instance de `SujetConcret` auprès de laquelle elle est enregistrée comme `observateur`.
- Les implémentations des classes concrètes n'utilisent, autant que faire se peut, que les types abstraits introduits par les interfaces,

Implémentation du patron

```
public interface Sujet {
  public void ajouterObservateur( Observateur o);
  public void supprimerObservateur( Observateur o);
  public void notifierObservateurs();
}
```

```
public interface Observateur {
  public void actualiser(Double t, Double h, Double p);
}
```

Définition des types et des contrats

Les interfaces définissent :

- des types qui seront utilisés dans les déclarations des implémentations des classes concrètes,
- des contrats standardisés également appelés dans les classes concrètes.

Implémentation du patron

```
public class SujetConcret {

  private ArrayList<Observateur> observateurs ;

  public void ajouterObservateur( Observateur o) this.observateurs.add(o); }

  public void supprimerObservateur( Observateur o) this.observateurs.remove(o); }

   public void notifierObservateurs(){
     for (Observateur o : this.observateurs) {
       o.actualiser(this.temp, this.humi, this.pres);
  } }

   public void setEtat(...){
     // mise a jour de l'etat
     this.notifierObservateurs();
  }
}
```

Gestion et notification des observateurs

- la variable d'instance `observateurs` permet de stocker une collection de références vers des instances implémentant l'interface `Observable`,
- la liste des `observateurs` est ajustable dynamiquement (`ajouterObservateur()`, `supprimerObservateur()`).

Implémentation du patron

```
public class SujetConcret {

  private ArrayList<Observateur> observateurs ;

  public void ajouterObservateur( Observateur o){ this.observateurs.add(o); }

  public void supprimerObservateur( Observateur o){ this.observateurs.remove(o); }

  public void notifierObservateurs(){
    for (Observateur o : this.observateurs) {
      o.actualiser(this.temp, this.humi, this.pres);
  } }

  public void setEtat(...){
    // mise a jour de l'etat
    this.notifierObservateurs();
  }
}
```

Gestion et notification des observateurs

- La méthode `notifierObservateurs()` invoque la méthode `actualiser()` sur tous les objets de la liste `observateurs`,

Implémentation du patron

```
public class SujetConcret {

  private ArrayList<Observateur> observateurs ;

  public void ajouterObservateur( Observateur o){ this.observateurs.add(o); }

  public void supprimerObservateur( Observateur o){ this.observateurs.remove(o); }

  public void notifierObservateurs()\{
    for (Observateur o : this.observateurs) \{
      o.actualiser(this.temp, this.humi, this.pres);
  } }

  public void setEtat(...){
    // mise a jour de l'etat
    this.notifierObservateurs();
  }
}
```

Gestion et notification des observateurs

- toute modification de l'état `setEtat()` de l'instance `SujetConcret` entraine un appel de la méthode `notifierObservateurs()`.

Implémentation du patron

```
public class ObservateurConcret implements Observateur {
  public ObservateurConcret(Sujet s) {
    s.ajouterObservateur(this) ;
  }
  public void actualiser() {
    // actualisation de l'ObservateursConcret
  }
}
```

Enregistrement et actualisation des observateurs

- L'`ObservateurConcret` est enregistré comme `Observateur` de l'instance de la classe `SujetConcret` `s`.
- L'enregistrement peur être pris en charge par le constructeur de l'`ObservateurConcret` mais pas obligatoirement.

Principes de conception

Principe généraux mis en œuvre

- Minimisation du couplage entre implémentations,
- Favoriser la composition,
- Mise à jour dynamique des liaisons,
- Choix de composition à l'exécution et non à l'implémentation.

Principe SOLID Inversion des dépendances

- Les types utilisés dans les classes sont majoritairement des types *interface*.
- si l'on opte pour *tirer* les données alors on doit accepter de rompre ce principe.

Principes de conception

Principe SOLID Ségrégation des interfaces

- Les responsabilités indépendantes sont définis abstraitement dans des interfaces séparées.

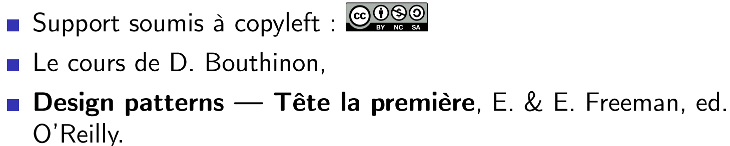
Principe SOLID Single responsability

- Les `SujetConcrets` ont pour responsabilité de notifier leurs observateurs de tout changement d'état,
- Les `ObservateurConcrets` ont pour responsabilité de réaliser une mise à jour dès qu'ils sont notifiés d'un changement d'état.

Principe SOLID Open/Closed

- Sans modifier l'implémentation des classes existantes on peut ajouter de nouveau type concrets d'`Observateurs` et de `Sujets`.

Credits



- Support soumis à copyleft : BY NC SA
- Le cours de D. Bouthinon,
- **Design patterns — Tête la première**, E. & E. Freeman, ed. O'Reilly.