Programmation Avancée et Application

Les Design Pattern

Jean-Guy Mailly

jean-guy.mailly@parisdescartes.fr

LIPADE - Université de Paris (Paris Descartes)
http://www.math-info.univ-paris5.fr/∽jmailly/

Les Design Pattern

1. Introduction

2. Patrons de création

Introduction

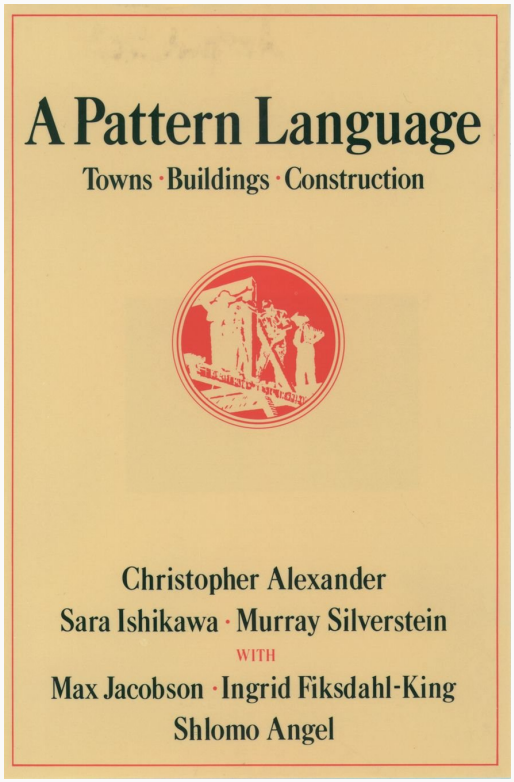
Design Pattern

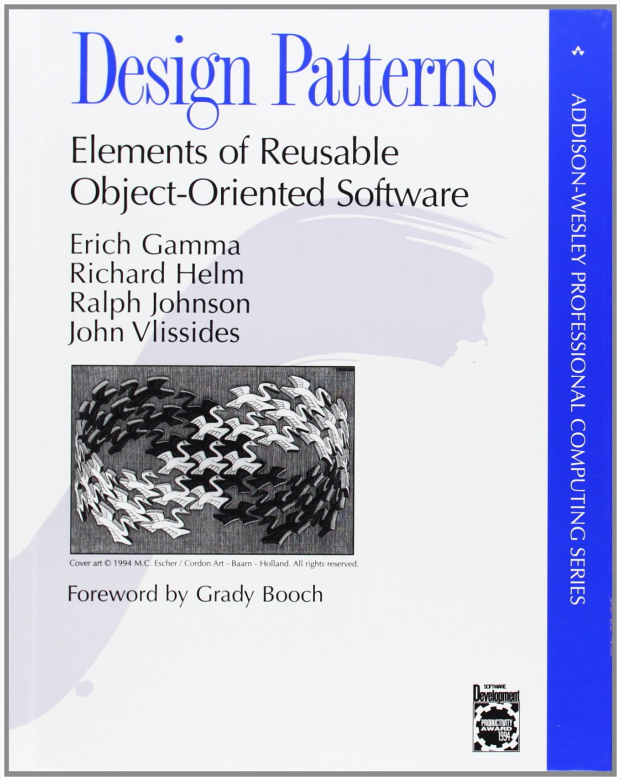
- En français : patrons de conception
- Intuition : différentes situations rencontrées par un développeur correspondent à un problème similaire, seul le contexte change
- Une solution similaire peut donc être appliquée
- Les design patterns forment un « catalogue » de solutions prêtes à l'emploi : l'expérience des générations précédentes de développeurs mise à disposition des nouveaux développeurs
- Nommage d'une structure de haut niveau qu'on ne peut pas exprimer directement sous forme de code $\Rightarrow$ Vocabulaire commun à tous les développeurs

Un peu de culture générale

- Origine du concept : architecture (1977 : *A Pattern Language : Towns, Buildings, Construction*, C. Alexander, S. Ishikawa et M. Silverstein)
- Années 80 : premières adaptations du concept au génie logiciel
- 1994 : Publication du *Gang of Four (GoF)*

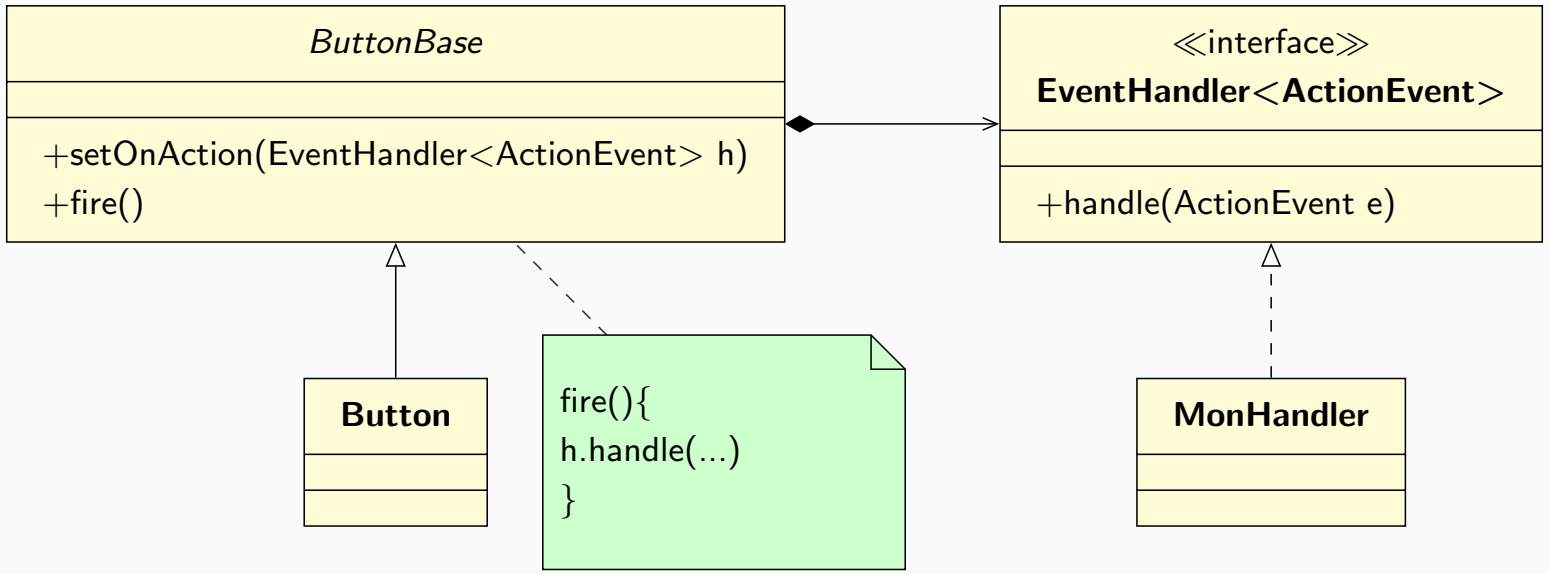
  Design Patterns : Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson et J. Vlissides





Premier exemple : gestion d'événements en JavaFX

- Rappel : lorsqu'un événement se produit dans une interface graphique, le composant lié à cet événement fait automatiquement appel à la méthode handle() de son Handler



- On peut avoir d'autres classes à la place du Button (*e.g.* CheckBox) ou de MonHandler (gestion différente d'un même événement)
- On retrouve la même structure pour les autres types d'événements (MouseEvent, KeyEvent,...)

Deuxième exemple : mises à jour et sauvegardes multiples (1/10 : Scénario)

Scénario simple :

- Notre application a besoin de sauvegarder les changements de l'état d'un objet dès qu'il se produit
  - Pour l'exemple, ce sera la modification d'un attribut de type **int** ou **double**
- Plusieurs modes de sauvegardes peuvent être implémentés, et utilisés au choix, voire même en parallèle
  - sauvegarde dans un fichier/une base de données
  - envoi par email
  - affichage sur la console
  - ...

Deuxième exemple : mises à jour et sauvegardes multiples (2/10 : La classe Donnees)

```
public abtract class Donnees {
  private List<Sauvegarde> sauvegardes ;

  public Donnees(){
    sauvegardes = new ArrayList<Sauvegarder>();
  }

  public void ajoutSauvegarde(Sauvegarde s){
    sauvegardes.add(s);
  }

  protected void signalerMAJ(){
    for(Sauvegarde s : sauvegardes) s.sauver(this) ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (3/10 : La classe DonneesInt)

```
public abtract class DonneesInt extends Donnees {
  private int donnees ;

  public DonneesInt(int d){
    super();
    donnees = d ;
  }
  public void miseAJour(int d){
    donnees = d ;
    signalerMAJ();
  }
  public String toString(){
    return System.currentTimeMillis()
                        + " : " + donnees ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (4/10 : La classe DonneesDouble)

```
public abtract class DonneesDouble extends Donnees {
  private double donnees ;

  public DonneesInt(double d){
    super();
    donnees = d ;
  }
  public void miseAJour(double d){
    donnees = d ;
    signalerMAJ();
  }
  public String toString(){
    return System.currentTimeMillis()
                        + " : " + donnees ;
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (5/10 : L'interface Sauvegarde)

```
public interface Sauvegarde {
  public void sauver(Donnees d) ;
}
```

- Première Sauvegarde concrète : sur la sortie standard

```
public class SauvegardeStdout implements Sauvegarde {
  public void sauver(Donnees d){
    System.out.println(d);
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (6/10 : Sauvegarde dans un fichier)

```
public class SauvegardeFichier implements Sauvegarde {
  private File fichier ;

  public SauvegardeFichier(String filename){
    fichier = new File(filename);
  }

  public void sauver(Donnees d){
    // Ouvrir le fichier
    // Ajouter d.toString() a la suite du contenu
    // Fermer le fichier
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (7/10 : Sauvegarde par email)

```
public class SauvegardeEmail implements Sauvegarde {
  private String email ;

  public SauvegardeEmail(String email){
    this.email = email ;
  }

  public void sauver(Donnees d){
    // Envoyer d.toString() par email
    // a l'adresse voulue
  }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (8/10 : Initialisation du système)

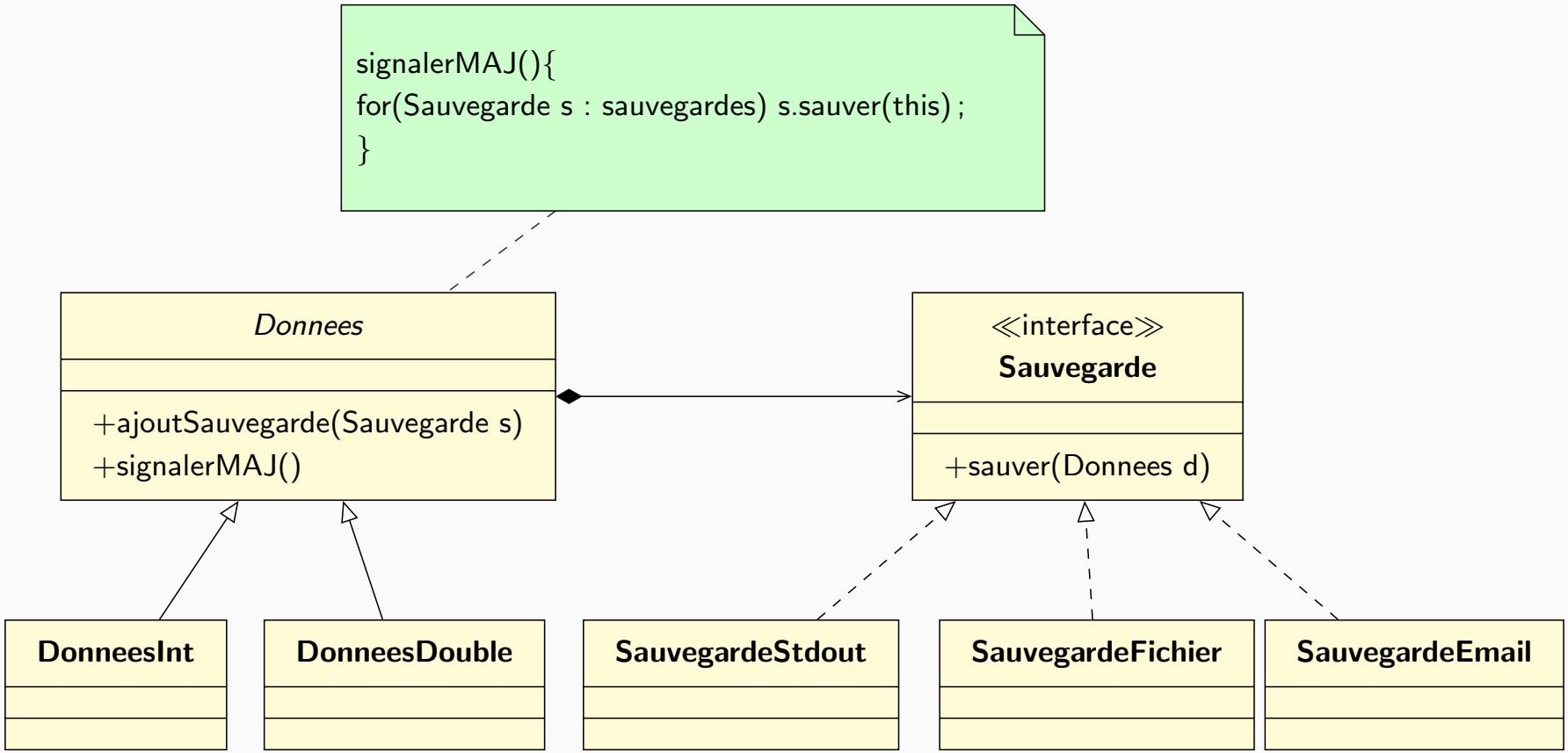
```
public class SystemeSauvegarde {
  public static void main(String[] args){
    List<Donnees> donnees = new ArrayList<Donnees>();
    donnees.add(new DonneesInt(0));
    donnees.add(new DonneesDouble(0));

    Sauvegarde stdout = new SauvegardeStdout();
    Sauvegarde fichier = new SauvegardeFichier(
                        "/chemin/vers/fichier.log");
    Sauvegarde email = new SauvegardeEmail(
                "sauvegarde@parisdescartes.fr");
```

Deuxième exemple : mises à jour et sauvegardes multiples (9/10 : Initialisation du système)

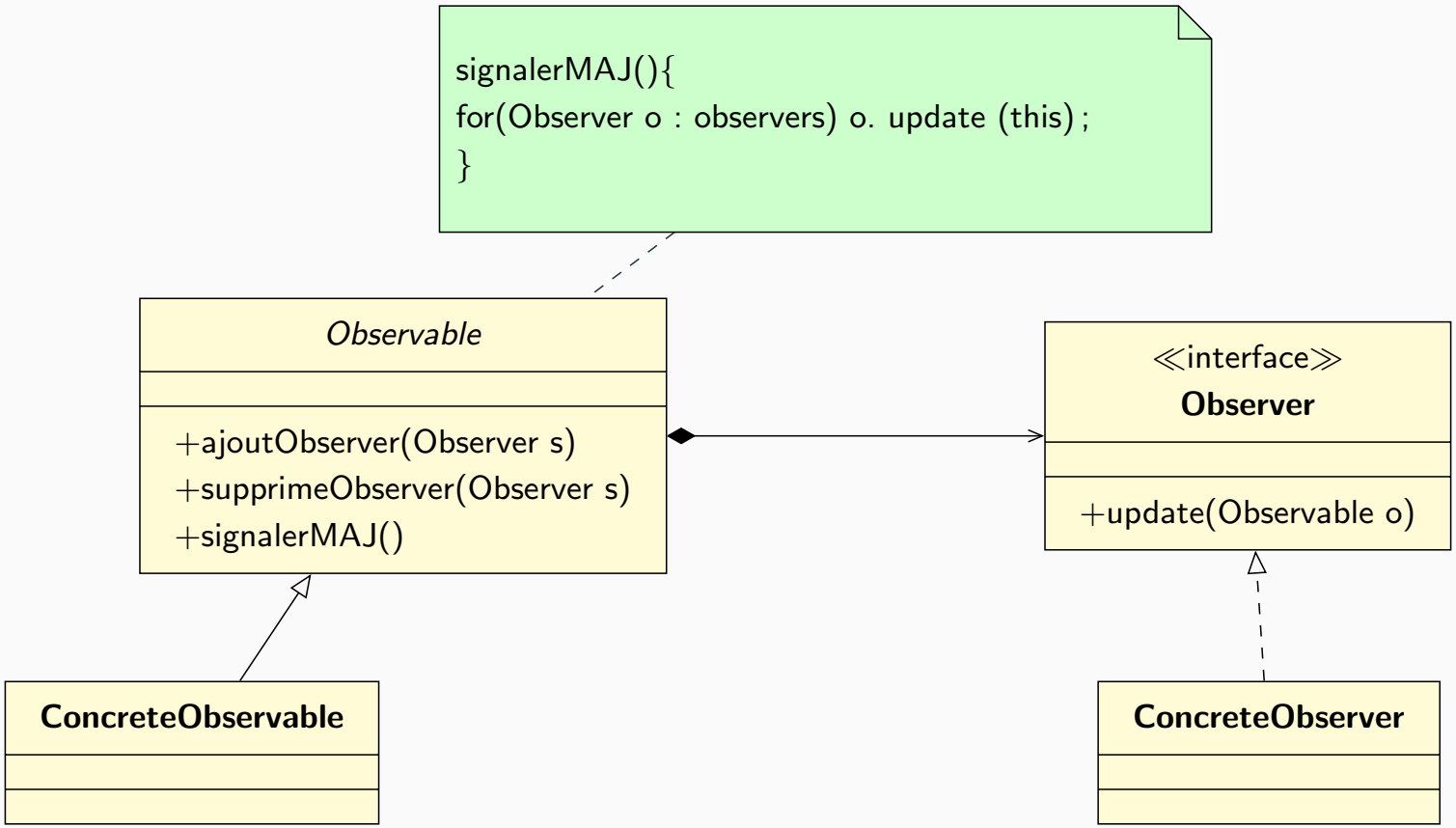
```
        for(Donnees d : donnees){
          d.ajouteSauvegarde(stdout);
          d.ajouteSauvegarde(fichier));
          d.ajouteSauvegarde(email);
        }
        // Suite des operations...
    }
}
```

Deuxième exemple : mises à jour et sauvegardes multiples (10/10 : Diagramme de classe)



Le design pattern Observer

- Ce patron de conception permet à des objets (les observateurs) d'agir en fonction de la modification de l'état d'un objet (l'observé)



Description minimale d'un design pattern

- Nom et classification
- Utilité

- Structure (UML)

Description minimale d'un design pattern

- Nom et classification $\rightarrow$ Observer, patron de comportement
- Utilité $\rightarrow$ permet à des objets (les observateurs) d'agir en fonction de la modification de l'état d'un objet (l'observé)
- Structure (UML) $\rightarrow$ voir slide précédent

Plusieurs types de patrons

- Patrons de création
  - Patrons qui permettent de créer des objets de manière adaptée à la situation
- Patrons de structure
  - Patrons qui concernent les relations entre différentes entités du programme
- Patrons de comportement
  - Patrons qui concernent la communication entre différents objets et l'adaptation de leur comportement (*e.g.* Observer)

Patrons de création

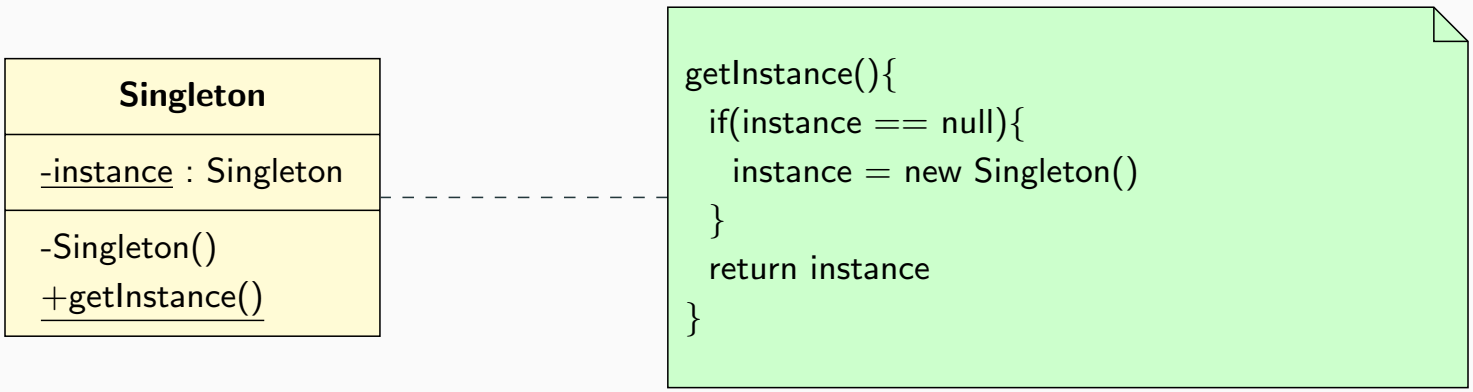
Les patrons de création

Ou *Creational design patterns*

- Abstraire le processus de création d'objets
- Rendre indépendante la façon dont les objets sont créés
- Encapsuler la connaissance de la classe concrète qui est instanciée
- Cacher ce qui est créé, quand, comment, par qui

Le Singleton

- On a besoin d'une classe dont on doit créer une seule instance, et cette instance doit être disponible n'importe où dans le programme
- Exemple d'application : gestion des ressources d'une application (file d'attente de l'imprimante), configuration des propriétés de l'application,...



Une file d'impression en Singleton

```
public class PrintQueue {
  private List<String> printQueue ;
  private static PrintQueue instance = null ;

  private PrintQueue(){
    printQueue = new LinkedList<String>();
  }

  public static PrintQueue getInstance(){
    if(instance == null)
      instance = new PrintQueue();
    return instance ;
  }

  // Methodes de la file : ajout, retrait,...
}
```

Une file d'impression en Singleton : la synchronisation

- La première version de PrintQueue pose problème si plusieurs threads peuvent l'utiliser : deux appels simultanés de getInstance() peuvent créer deux instances !

```
public class PrintQueue {
  // ...
  // Attributs et constructeur

  public static synchronized PrintQueue getInstance(){
    if(instance == null)
      instance = new PrintQueue();
    return instance ;
  }

  // Methodes de la file : ajout, retrait,...
}
```

Une file d'impression en Singleton : la synchronisation améliorée

- La synchronisation a un coût, on préfère éviter de la faire à chaque appel de getInstance()

```
public class PrintQueue {
  // Attributs et constructeur
  public static PrintQueue getInstance(){
    if(instance == null)
      synchronized(PrintQueue.class){
        if(instance == null)
          instance = new PrintQueue();
      }
    return instance ;
  }
  // Methodes de la file : ajout, retrait,...
}
```

- PrintQueue.**class** est une instance de Class<PrintQueue>