TP Génie Logiciel - Design Patterns

Gestion de l'Emploi du Temps - Design Patterns

KHEDROUCHE DJASSIM
SIAD

20 novembre 2025

Table des matières

1 Introduction

Ce document analyse le respect des principes SOLID dans l'implémentation des trois design patterns (Builder, Observer, Decorator) pour le système de gestion d'emploi du temps.

1.1 Les principes SOLID

Les principes SOLID sont cinq principes de conception orientée objet :

— **S**ingle Responsibility Principle (SRP)
— **O**pen/Closed Principle (OCP)
— **L**iskov Substitution Principle (LSP)
— **I**nterface Segregation Principle (ISP)
— **D**ependency Inversion Principle (DIP)

2 Analyse par principe

2.1 Single Responsibility Principle (SRP)

« Une classe ne devrait avoir qu'une seule raison de changer. »

2.1.1 Respect du principe

Respecté

Justification :

- ✓ **Classe** `Cours` : Responsabilité unique = représenter un cours avec ses attributs
- ✓ **Classe** `CoursBuilder` : Responsabilité unique = construire des objets `Cours`
- ✓ **Classe** `GestionnaireEmploiDuTemps` : Responsabilité unique = gérer les cours et notifier les observateurs
- ✓ **Classes** `Etudiant` **et** `Responsable` : Responsabilité unique = recevoir et traiter les notifications
- ✓ **Décorateurs** (`CoursEnLigne`, `CoursEnAnglais`, `CoursMagistral`) : Chacun ajoute une seule fonctionnalité spécifique

Exemple de code respectant le SRP :

```
public class CoursBuilder {
    // Responsabilité unique : construire des Cours
    public Cours build() {
        return new Cours(matiere, enseignant, ...);
    }
}
```

2.2 Open/Closed Principle (OCP)

« Les entités logicielles doivent être ouvertes à l'extension, mais fermées à la modification. »

2.2.1 Respect du principe

Respecté

Justification :

- ✓ **Pattern Decorator** : Permet d'ajouter de nouvelles fonctionnalités (ex : `CoursHybride`, `CoursAvecTP`) sans modifier les classes existantes
- ✓ **Pattern Observer** : On peut ajouter de nouveaux types d'observateurs (ex : `Parent`, `Administration`) sans modifier `GestionnaireEmploiDuTemps`
- ✓ **Interface** `ICours` : Permet l'extension via de nouvelles implémentations

Démonstration :

```
// Extension sans modification
public class CoursHybride extends CoursDecorator {
    @Override
    public String getDescription() {
        return coursDecorated.getDescription() + " (Hybride)";
    }
}
// Aucune modification des classes existantes requise !
```

2.3 Liskov Substitution Principle (LSP)

« Les objets d'une classe dérivée doivent pouvoir remplacer les objets de la classe de base sans altérer le comportement du programme. »

2.3.1 Respect du principe

Respecté

Justification :

- ✓ Tous les décorateurs (`CoursEnLigne`, `CoursEnAnglais`, `CoursMagistral`) peuvent remplacer un `ICours` sans problème
- ✓ Les classes `Etudiant` et `Responsable` peuvent remplacer `Observer` de manière transparente
- ✓ Le comportement des méthodes reste cohérent dans toutes les sous-classes

Test de substitution :

```
ICours cours = new Cours(...);
ICours coursDecore = new CoursEnLigne(cours);
// coursDecore peut être utilisé partout où ICours est attendu
System.out.println(coursDecore.getDescription()); // Fonctionne !
```

2.4 Interface Segregation Principle (ISP)

« Aucun client ne devrait dépendre de méthodes qu'il n'utilise pas. »

2.4.1 Respect du principe

Respecté

Justification :

- ✓ **Interface** `ICours` : Contient uniquement 2 méthodes essentielles (`getDescription()`, `getDuree()`)
- ✓ **Interface** `Observer` : Une seule méthode `update(String)` - interface minimale
- ✓ **Interface** `Subject` : 3 méthodes cohérentes et nécessaires pour tous les sujets observables
- ✓ Aucune interface ne force l'implémentation de méthodes inutiles

Conception minimaliste :

```
public interface Observer {
    void update(String message); // Une seule méthode nécessaire
}
```

2.5 Dependency Inversion Principle (DIP)

« Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions. »

2.5.1 Respect du principe

Respecté

Justification :

- ✓ `GestionnaireEmploiDuTemps` dépend de l'abstraction `Observer`, pas des classes concrètes `Etudiant`/`Responsable`
- ✓ `CoursDecorator` dépend de l'interface `ICours`, pas de la classe concrète `Cours`
- ✓ Utilisation systématique d'interfaces pour le couplage

Exemple d'inversion de dépendance :

```
public class GestionnaireEmploiDuTemps {
    private List<Observer> observers; // Dépend de l'abstraction
    // Pas de : private List<Etudiant> etudiants;

    public void attach(Observer o) { // Accepte toute implémentation
        observers.add(o);
    }
}
```

3 Violations potentielles et améliorations

3.1 Violation mineure identifiée

Violation potentielle du SRP

Problème identifié : La classe `GestionnaireEmploiDuTemps` a **deux responsabilités** :

1. Gérer la liste des cours (ajout, modification)
2. Gérer les observateurs et les notifications

3.2 Solution proposée

Séparer en deux classes :

```
// Classe 1 : Gestion pure des observateurs
public class NotificationManager implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer o) { observers.add(o); }

    @Override
    public void detach(Observer o) { observers.remove(o); }

    @Override
    public void notifyObservers(String message) {
        for (Observer o : observers) {
            o.update(message);
        }
    }
}

// Classe 2 : Gestion des cours
public class GestionnaireEmploiDuTemps {
    private List<ICours> listeCours = new ArrayList<>();
    private NotificationManager notificationManager;

    public GestionnaireEmploiDuTemps() {
        this.notificationManager = new NotificationManager();
    }

    public void ajouterCours(ICours cours) {
        listeCours.add(cours);
        notificationManager.notifyObservers(
            "Nouveau cours : " + cours.getDescription()
        );
    }
```

```
    // Délégation aux observateurs
    public void attach(Observer o) {
        notificationManager.attach(o);
    }
}
```

4 Diagramme de classes UML

Le diagramme de classes suivant illustre l'architecture complète du système de gestion d'emploi du temps, mettant en évidence les trois design patterns implémentés.

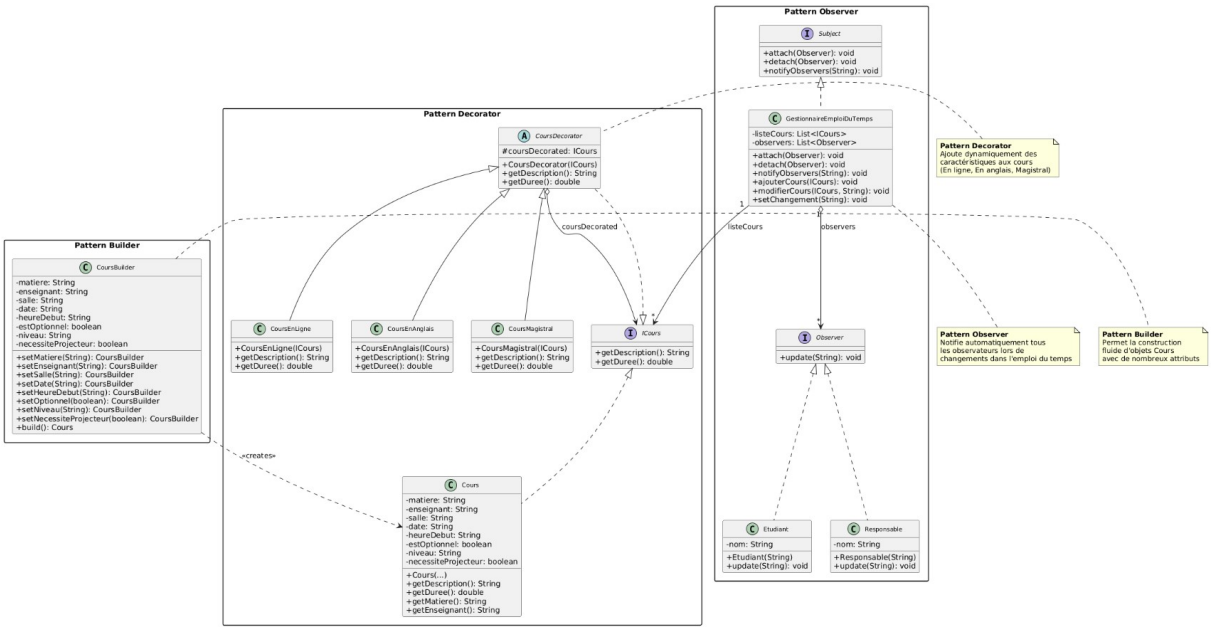

FIGURE 1 – Diagramme de classes UML complet du système

4.1 Légende et relations

Le diagramme utilise les notations UML standard :

— **Ligne pleine avec triangle vide (△)** : Héritage (extends)

— **Ligne pointillée avec triangle vide (△)** : Implémentation (implements)

— **Ligne avec losange vide (◇→)** : Agrégation

— **Ligne pointillée avec flèche (- - →)** : Dépendance (¡¡creates¿¿)

4.2 Organisation en packages

Le système est organisé en trois packages correspondant aux trois patterns :

1. **Pattern Builder** : Construction fluide d'objets `Cours`
2. **Pattern Observer** : Système de notification automatique
3. **Pattern Decorator** : Extension dynamique des fonctionnalités

5 Synthèse et tableau récapitulatif

Principe	Statut	Commentaire
SRP	∼	Globalement respecté, mais `GestionnaireEmploiDuTemps` pourrait être séparé
OCP		Excellente utilisation du pattern Decorator et Observer
LSP		Toutes les substitutions fonctionnent correctement
ISP		Interfaces minimalistes et cohérentes
DIP		Dépendances vers les abstractions, pas les implémentations

TABLE 1 – Respect des principes SOLID

6 Conclusion

L'implémentation des trois design patterns (Builder, Observer, Decorator) respecte globalement les principes SOLID. Les points forts sont :

— **Extensibilité** : Facile d'ajouter de nouveaux types de cours, décorateurs ou observateurs

— **Découplage** : Utilisation systématique d'interfaces

— **Maintenabilité** : Code clair avec des responsabilités bien définies

La seule amélioration suggérée concerne la séparation de `GestionnaireEmploiDuTemps` en deux classes distinctes pour un respect strict du SRP.