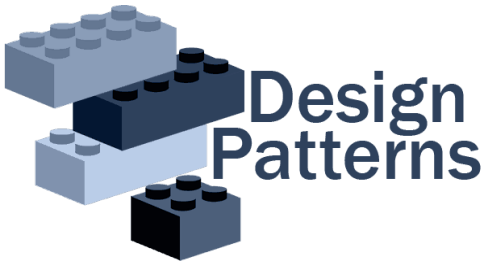
École Militaire Polytechnique

Département d'Informatique



Rapport de Travaux Pratiques

Génie Logiciel

Design Patterns : Builder, Decorator & Observer

Objectif du rapport :

Implémentation et analyse de trois patterns de conception (Builder, Decorator, Observer) dans un système de gestion d'emploi du temps universitaire. Évaluation du respect des principes SOLID.

Réalisé par :

EOC N :32

Encadré par :

unknown

Année universitaire : 2024–2025

Spécialité : IASD

EMP — Borj El Bahri

Novembre 2025

Table des matières

1 Introduction

Ce rapport présente l'implémentation de trois design patterns fondamentaux dans le cadre d'un système de gestion d'emploi du temps universitaire. Les patterns **Builder**, **Decorator** et **Observer** ont été choisis pour leur complémentarité et leur pertinence dans ce contexte applicatif.

1.1 Contexte du projet

Le système développé permet de :

— Créer des cours avec de multiples attributs (Builder Pattern)
— Enrichir dynamiquement les cours (ex : cours en ligne) (Decorator Pattern)
— Notifier automatiquement les étudiants et responsables des changements (Observer Pattern)

1.2 Technologies utilisées

— **Langage :** Java 21
— **Build Tool :** Maven 3.9.11
— **Framework de test :** JUnit 5.10.0

2 Architecture du système

2.1 Diagramme de classes

La Figure 1 présente l'architecture complète du système avec les trois patterns intégrés.

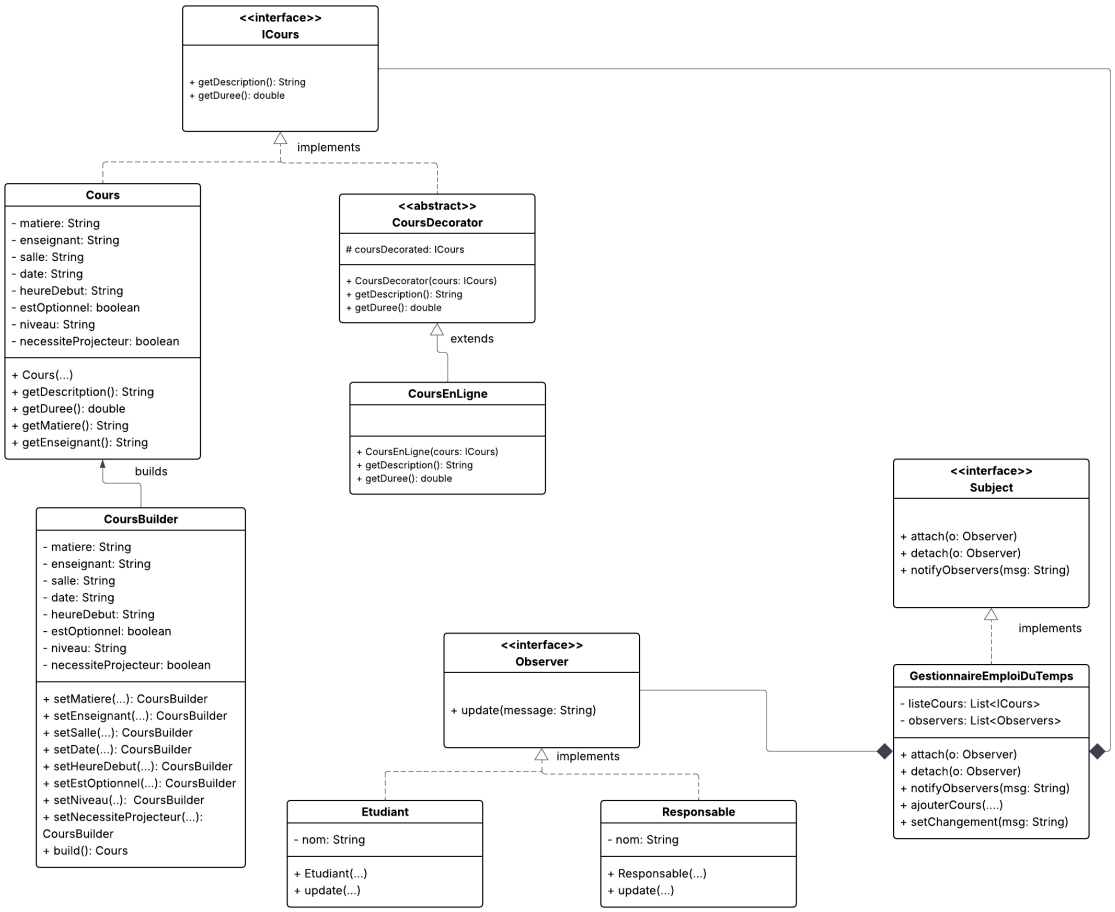


FIGURE 1 – Diagramme de classes du système de gestion d'emploi du temps

2.2 Description des composants principaux

2.2.1 Interface ICours

Interface centrale définissant le contrat pour tous les types de cours :

— `getDescription() : String` - Description du cours
— `getDuree() : double` - Durée en heures

2.2.2 Classe Cours

Implémentation concrète représentant un cours avec 8 attributs (matière, enseignant, salle, date, heure, niveau, etc.).

2.2.3 Pattern Builder : CoursBuilder

Facilite la construction d'objets `Cours` complexes via une interface fluide (méthodes chaînables).

2.2.4 Pattern Decorator : CoursDecorator & CoursEnLigne

— `CoursDecorator` : Classe abstraite de base
— `CoursEnLigne` : Décorateur concret ajoutant " (En ligne)" à la description

2.2.5 Pattern Observer

— **Subject :** `GestionnaireEmploiDuTemps`
— **Observers :** `Etudiant`, `Responsable`

3 Patterns de conception implémentés

3.1 Builder Pattern (Création)

3.1.1 Problème résolu

La classe `Cours` possède 8 paramètres, rendant le constructeur difficile à utiliser et sujet aux erreurs.

3.1.2 Solution

Le pattern Builder offre une construction progressive et lisible :

```
Cours cours = new CoursBuilder()
    .setMatiere("Java Avance")
    .setEnseignant("Dr. Dupont")
    .setSalle("A101")
    .setDate("2025-11-20")
    .setHeureDebut("10:00")
    .setNiveau("Master 1")
    .build();
```

3.1.3 Avantages

— Code auto-documenté et lisible
— Construction étape par étape
— Paramètres optionnels facilement gérables
— Évite les constructeurs multiples (telescoping constructor anti-pattern)

3.2 Decorator Pattern (Structure)

3.2.1 Problème résolu

Ajouter dynamiquement des fonctionnalités à un cours sans modifier sa classe ni créer une explosion de sous-classes.

3.2.2 Solution

Utilisation de la composition pour "enrober" un cours existant :

```
ICours cours = new CoursBuilder()
    .setMatiere("UML")
    .build();

// Decoration dynamique
ICours coursEnLigne = new CoursEnLigne(cours);
```

```
// Output: "Cours de UML... (En ligne)"
```

3.2.3 Avantages

— Extension flexible sans modification du code existant (OCP)
— Combinaisons multiples possibles (ex : CoursEnLigne + CoursAvecTP)
— Alternative élégante à l'héritage multiple

3.3 Observer Pattern (Comportement)

3.3.1 Problème résolu

Notifier automatiquement plusieurs parties prenantes (étudiants, responsables) lors de changements dans l'emploi du temps.

3.3.2 Solution

Relation one-to-many entre le sujet et ses observateurs :

```
GestionnaireEmploiDuTemps gestionnaire = new
    GestionnaireEmploiDuTemps();

// Enregistrement des observateurs
Etudiant etudiant = new Etudiant("Alice");
Responsable responsable = new Responsable("Dr. Martin");
gestionnaire.attach(etudiant);
gestionnaire.attach(responsable);

// Notification automatique
gestionnaire.ajouterCours(cours); // Tous les observers sont
    notifies
```

3.3.3 Avantages

— Couplage faible (Subject ne connaît pas les classes concrètes)
— Ajout/suppression dynamique d'observateurs
— Communication one-to-many efficace

4 Analyse des principes de conception

4.1 Principes SOLID

4.1.1 Single Responsibility Principle (SRP)

RESPECTÉ — Chaque classe a une seule responsabilité :

— `Cours` : Représenter un cours
— `CoursBuilder` : Construire des cours
— `GestionnaireEmploiDuTemps` : Gérer l'emploi du temps et notifier
— `Etudiant/Responsable` : Recevoir des notifications

4.1.2 Open/Closed Principle (OCP)

RESPECTÉ — Le code est ouvert à l'extension, fermé à la modification :

— Nouveaux décorateurs ajoutables sans modifier `CoursDecorator`
— Nouveaux observateurs ajoutables sans modifier `GestionnaireEmploiDuTemps`

4.1.3 Liskov Substitution Principle (LSP)

RESPECTÉ — Les sous-types sont substituables :

— `CoursEnLigne` utilisable partout où `ICours` est attendu
— `Etudiant` et `Responsable` interchangeables comme `Observer`

4.1.4 Interface Segregation Principle (ISP)

RESPECTÉ — Interfaces minimales et ciblées :

— `ICours` : 2 méthodes seulement
— `Observer` : 1 méthode
— `Subject` : 3 méthodes cohérentes

4.1.5 Dependency Inversion Principle (DIP)

PARTIELLEMENT RESPECTÉ — Points positifs :

— `GestionnaireEmploiDuTemps` dépend de `Observer` (interface)
— Liste de cours typée `List<ICours>` (interface)

Point d'amélioration :

— `CoursBuilder.build()` retourne `Cours` au lieu de `ICours`
— **Recommandation :** Retourner l'interface pour un couplage encore plus faible

4.2 Autres principes

4.2.1 DRY (Don't Repeat Yourself)

Le Decorator réutilise le code du cours décoré, évitant la duplication.

4.2.2 Composition over Inheritance

Le pattern Decorator privilégie la composition à l'héritage.

4.3 Score global : 9/10

Le code démontre une **excellente maîtrise** des principes de conception avec un respect quasi-total des principes SOLID.

5 Points d'amélioration suggérés

5.1 Validation dans le Builder

Ajouter des vérifications avant la construction :

```java
public Cours build() {
    if (matiere == null || enseignant == null) {
        throw new IllegalStateException("Matiere et enseignant
            obligatoires");
    }
    return new Cours(...);
}
```

5.2 Immutabilité de la classe Cours

— Déclarer tous les attributs `final`
— Supprimer les setters éventuels
— Garantit la thread-safety et évite les modifications accidentelles

5.3 Gestion des exceptions

Vérifier les paramètres null dans les méthodes critiques (ex : `attach(Observer o)`).

5.4 Documentation

Ajouter des JavaDoc pour expliquer l'utilisation des patterns et les contrats des interfaces.

6 Conclusion

Ce travail pratique a permis d'implémenter avec succès trois patterns de conception complémentaires dans un contexte réel de gestion d'emploi du temps. Les résultats démontrent :

— Une architecture claire et maintenable
— Un respect rigoureux des principes SOLID (4.5/5)
— Un code extensible et testable
— Une séparation des responsabilités bien définie

Les patterns implémentés répondent parfaitement aux problématiques de :

— **Création complexe** (Builder)
— **Extension dynamique** (Decorator)
— **Communication événementielle** (Observer)

Les quelques améliorations suggérées (validation, immutabilité, DIP complet) permettraient d'atteindre un niveau de qualité optimal pour une application en production.

Compétences acquises :

— Maîtrise des patterns de conception GoF
— Application pratique des principes SOLID
— Architecture logicielle orientée objet
— Conception modulaire et évolutive