TP 2 Génie Logiciel
Questions Bonus

LOUALI Taher Ouasim

20 novembre 2025

1 Diagramme de Classes UML

Le diagramme de classes ci-dessous illustre l'architecture finale du système de gestion d'emploi du temps, intégrant les trois design patterns : Builder, Decorator et Observer.

1.1 Description du diagramme

1.2 Légende des patterns

— **Builder Pattern** : La classe `CoursBuilder` facilite la construction d'objets `Cours` complexes avec de nombreux paramètres.

— **Decorator Pattern** : La classe abstraite `CoursDecorator` et ses sous-classes (ex : `CoursEnLigne`) permettent d'ajouter dynamiquement des fonctionnalités aux cours.

— **Observer Pattern** : L'interface `Subject` et `Observer` permettent la notification automatique des changements d'emploi du temps.

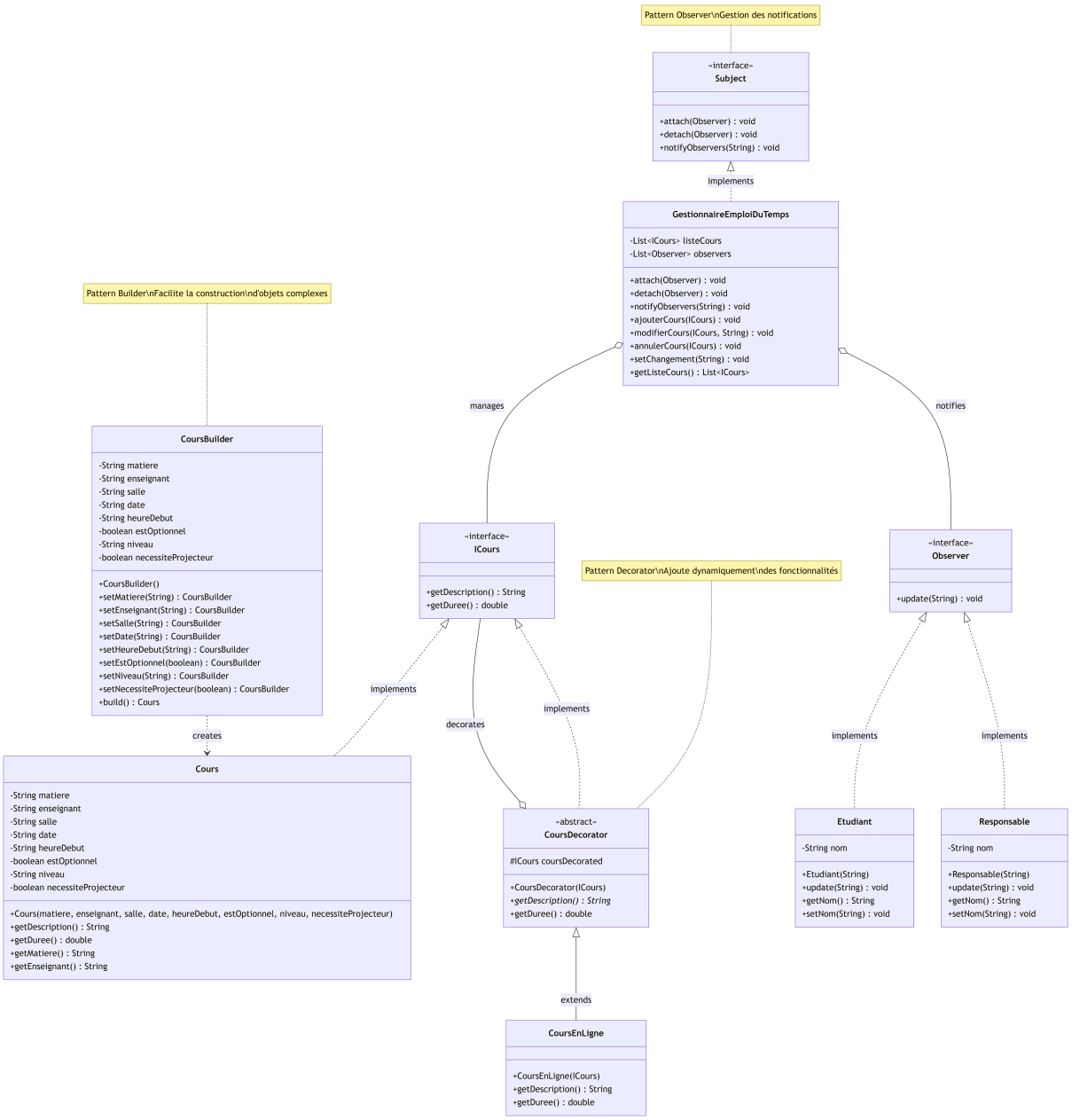

FIGURE 1 – Diagramme de classes UML du système de gestion d'emploi du temps

2 Analyse des Principes de Conception Logicielle (SOLID)

Cette section analyse le respect des cinq principes SOLID dans notre implémentation.

2.1 Principes Respectés

2.1.1 Single Responsibility Principle (SRP) ✓

Définition : Chaque classe doit avoir une seule responsabilité et une seule raison de changer.

Analyse : Ce principe est bien respecté dans notre code :

- `Cours` : Représente uniquement un cours avec ses attributs
- `CoursBuilder` : Responsable uniquement de la construction d'objets `Cours`
- `GestionnaireEmploiDuTemps` : Gère l'emploi du temps et les notifications
- `Etudiant` et `Responsable` : Observent et réagissent aux changements

Chaque classe a une responsabilité claire et distincte, ce qui facilite la maintenance et l'évolution du code.

2.1.2 Open/Closed Principle (OCP) ✓

Définition : Les entités logicielles doivent être ouvertes à l'extension mais fermées à la modification.

Analyse : Le pattern Decorator implémenté respecte parfaitement ce principe :

- On peut ajouter de nouvelles fonctionnalités (CoursEnAnglais, CoursMagistral) sans modifier la classe `Cours`
- Extension par composition plutôt que par modification
- Le système est extensible sans risque de régression

Exemple : Pour ajouter un cours magistral, il suffit de créer une nouvelle classe :

```
1 public class CoursMagistral extends CoursDecorator {
2     public String getDescription() {
3         return coursDecorated.getDescription() + "␣(Magistral)";
4     }
5 }
```

2.1.3 Liskov Substitution Principle (LSP) ✓

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

Analyse : Ce principe est respecté :

- `CoursEnLigne` peut remplacer n'importe quelle instance de `ICours`
- Les décorateurs préservent le contrat de l'interface `ICours`
- Aucune violation de contrat dans les sous-classes

2.1.4 4. Interface Segregation Principle (ISP) ✓

Définition : Les clients ne doivent pas dépendre d'interfaces qu'ils n'utilisent pas.
Analyse : Nos interfaces sont petites et ciblées :

- `ICours` : Seulement 2 méthodes essentielles
- `Observer` : Une seule méthode `update()`
- `Subject` : Trois méthodes liées à la gestion des observateurs

Aucune classe n'est forcée d'implémenter des méthodes inutiles.

2.2 Principe Partiellement Violé

2.2.1 Dependency Inversion Principle (DIP) ~

Définition : Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
Problème identifié : Dans la classe `GestionnaireEmploiDuTemps`, il y a une dépendance directe à `ArrayList` :

```
public class GestionnaireEmploiDuTemps implements Subject {
    private List<ICours> listeCours = new ArrayList<>();
    private List<Observer> observers = new ArrayList<>(); //
       Violation du DIP
}
```

Explication : Le code instancie directement la classe concrète `ArrayList`, créant un couplage fort avec une implémentation spécifique.
Solution recommandée : Utiliser l'injection de dépendances :

```
public class GestionnaireEmploiDuTemps implements Subject {
    private List<Observer> observers;
    private List<ICours> listeCours;

    // Injection via constructeur
    public GestionnaireEmploiDuTemps(List<Observer> observers,
                                     List<ICours> listeCours) {
        this.observers = observers;
        this.listeCours = listeCours;
    }

    // Ou constructeur par d faut
    public GestionnaireEmploiDuTemps() {
        this(new ArrayList<>(), new ArrayList<>());
    }
}
```

Avantages de cette approche :

- Facilite les tests unitaires (injection de mocks)
- Permet de changer d'implémentation facilement
- Respecte totalement le principe DIP

3 Conclusion

3.1 Points forts de l'architecture

- ✓ **Séparation des responsabilités** : Chaque classe a un rôle clair et unique
- ✓ **Utilisation appropriée des design patterns** : Builder, Observer et Decorator sont correctement implémentés
- ✓ **Code extensible** : Nouvelles fonctionnalités ajoutables sans modification du code existant
- ✓ **Interfaces bien définies** : Contrats clairs entre les composants
- ✓ **Maintenabilité** : Structure claire facilitant les évolutions futures

3.2 Point d'amélioration

- ∼ **Injection de dépendances** : Pour respecter totalement le DIP, privilégier l'injection de dépendances plutôt que l'instanciation directe des implémentations concrètes

3.3 Synthèse

L'architecture respecte globalement les principes SOLID avec 4 principes sur 5 totalement respectés. La violation mineure du DIP n'affecte pas significativement la qualité du code mais pourrait être améliorée pour une architecture encore plus robuste et testable.

Les trois design patterns sont correctement implémentés et répondent parfaitement aux problématiques du TP :

— Builder : Simplifie la construction d'objets complexes

— Decorator : Permet l'extension dynamique sans modification

— Observer : Assure la notification automatique des changements