École Nationale des Sciences Appliquées d'Agadir

Génie Informatique

Rapport TP: State Builder Patterns

Gestion d'états d'un compte bancaire

Réalisé par :

BOUANANI

Encadré par :
Prof. Encadrant

Année Universitaire : 2025-2026

Date : 18 janvier 2026

Table des matières

1 Introduction

Ce TP a pour objectif de mettre en œuvre deux patrons de conception (Design Patterns) fondamentaux : le patron **State** et le patron **Builder**. Nous appliquons ces concepts à un cas concret de gestion de comptes bancaires. Un compte peut avoir différents états (Normal, Bloqué, Découvert) influençant son comportement (retraits, dépôts). De plus, nous implémentons un gestionnaire de transitions flexible utilisant le pattern Builder pour définir dynamiquement les règles de passage d'un état à un autre.

2 Le Patron State (État)

2.1 Concept

Le patron State permet à un objet de modifier son comportement lorsque son état interne change. L'objet donnera l'impression de changer de classe. Dans notre cas, la classe `Compte` délègue les opérations `retirer` et `deposer` à un objet implémentant l'interface `EtatCompte`.

2.2 Implémentation

2.2.1 L'interface EtatCompte

Cette interface définit le contrat que tous les états doivent respecter.

```
public interface EtatCompte {
    void retirer(double montant, Compte compte) throws Exception;
    void deposer(double montant, Compte compte);
    String getEtatName();
}
```

Code 1 – Interface EtatCompte

2.2.2 Les États Concrets

Nous avons implémenté trois états :

- **CompteNormal** : État standard avec autorisation de retrait simple.
- **CompteDecouvert** : Affiche un avertissement lors des retraits.
- **CompteBloque** : Interdit formellement les retraits (lève une exception).

Exemple de `CompteBloque` :

```
public class CompteBloque implements EtatCompte {
    @Override
    public void retirer(double montant, Compte compte) throws Exception {
        System.out.println("Activit  refus e : Compte Bloqu  !");
        throw new Exception("Op ration refus e : Compte Bloqu  !");
    }
    // ... m thode deposer et getEtatName
}
```

Code 2 – Classe CompteBloque

3 Gestion des Transitions et Pattern Builder

3.1 EtatTransitionManager

Pour éviter de coder les transitions "en dur" dans les classes d'état (ce qui créerait un couplage fort), nous avons créé une classe `EtatTransitionManager`. Cette classe centralise la logique de changement d'état.
Elle utilise une liste de règles (`Transition`) définies par :

— L'état source.
— L'état cible.
— Une condition (`Predicate<Compte>`).

```java
public EtatCompte checkTransition(Compte compte) {
    EtatCompte etatActuel = compte.getEtat();
    for (Transition t : transitions) {
        if (t.source.equals(etatActuel.getClass()) && t.condition.test(compte))
    {
            try {
                return t.cible.getDeclaredConstructor().newInstance();
            } catch (Exception e) {
                // Gestion d'erreur...
            }
        }
    }
    return etatActuel;
}
```

Code 3 – Vérification des transitions dans EtatTransitionManager

3.2 Le Builder

Pour faciliter la configuration des transitions, nous utilisons une classe interne statique `Builder`. Cela permet de définir les règles de manière fluide et lisible.

```java
EtatTransitionManager manager = new EtatTransitionManager.Builder()
    .addTransition(CompteNormal.class, CompteDecouvert.class, c -> c.getSolde()
    < 0)
    .addTransition(CompteDecouvert.class, CompteNormal.class, c -> c.getSolde()
    > 0)
    .build();
```

Code 4 – Usage du Builder

4 Classe Compte : L'Intégration

La classe `Compte` est le contexte qui maintient une référence vers l'état courant et le gestionnaire de transitions.
À chaque opération (`retirer` ou `deposer`) :

1. La requête est déléguée à l'état actuel.
2. Le solde est mis à jour.
3. La méthode `verifierChangementEtat()` est appelée pour interroger le `manager`.

```java
public void retirer(double montant) {
    try {
        etat.retirer(montant, this);
        this.solde -= montant;
        verifierChangementEtat();
    } catch (Exception e) {
        System.out.println("Erreur : " + e.getMessage());
    }
}
```

Code 5 – Méthode retirer de la classe Compte

5 Conclusion

Ce TP nous a permis de comprendre l'intérêt des design patterns pour créer des architectures souples et maintenables.

- Le pattern **State** nous a permis de respecter le principe *Open/Closed* : nous pouvons ajouter de nouveaux états sans modifier la classe `Compte`.
- Le pattern **Builder** couplé au `TransitionManager` a externalisé la logique de transition, rendant le système de règles dynamique et facilement configurable.