Compte Rendu n°11 du Génie Logiciel

DERMOUCHE Mohammed Ryad
Université Paris Cité
L3 Informatique et Applications UFR Maths-Info

Partie 1 : Patron de Conception Mémento

1. Introduction

Le patron de conception **Mémento** est un patron comportemental qui permet de capturer et de sauvegarder l'état interne d'un objet sans exposer ses détails internes. Il est particulièrement utile dans les systèmes nécessitant la sauvegarde et la restauration de l'état d'objets à différents moments, tout en respectant le principe d'encapsulation.

2. Problème Résolu par le Mémento

Le Mémento répond aux problématiques suivantes :

- **Problème de sauvegarde/restauration** : Lorsque des objets doivent revenir à un état précédent (comme dans un éditeur avec une fonctionnalité d'annulation), il est nécessaire de capturer l'état interne sans exposer directement les détails de l'objet.
- **Problème d'historique** : Dans les applications comme les éditeurs de texte ou les jeux, il est essentiel de maintenir un historique des états successifs d'un objet.

3. Solution Proposée par le Mémento

Le patron propose une solution en séparant les responsabilités :

- Un objet **Originator** est responsable de créer et restaurer des états sous forme de **Mémentos**.
- Le **Caretaker** gère les *Mémentos*, les stocke, et permet leur restitution.
- Le **Mémento** encapsule les détails internes de l'état capturé, tout en restant inaccessible pour les autres objets.

Avantages :

- Respecte l'encapsulation.
- Permet une gestion facile de l'historique.

Inconvénients :

- Peut consommer beaucoup de mémoire si de nombreux états sont sauvegardés.

4. Exemple en Java

Voici un exemple simple de mise en œuvre du patron de conception **Mémento** dans un éditeur de texte :

```java
class TextEditor {
    private String content;

    public void setContent(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }

    // Cr e un memento pour sauvegarder l' tat
    public Memento save() {
        return new Memento(content);
    }

    // Restaure l' tat  depuis un memento
    public void restore(Memento memento) {
        this.content = memento.getContent();
    }

    // Classe interne Memento
    public static class Memento {
        private final String content;

        private Memento(String content) {
            this.content = content;
        }

        private String getContent() {
            return content;
        }
    }
}

class Caretaker {
    private final List<TextEditor.Memento> mementos = new
       ArrayList<>();

    public void addMemento(TextEditor.Memento memento) {
        mementos.add(memento);
    }

    public TextEditor.Memento getMemento(int index) {
        return mementos.get(index);
    }
}
```

Listing 1: Classe TextEditor et Memento

```java
public class MementoExample {
    public static void main(String[] args) {
        TextEditor editor = new TextEditor();
        Caretaker caretaker = new Caretaker();

        editor.setContent("Version 1");
        caretaker.addMemento(editor.save());

        editor.setContent("Version 2");
        caretaker.addMemento(editor.save());

        editor.setContent("Version 3");

        System.out.println(" tat  actuel : " + editor.getContent());

        editor.restore(caretaker.getMemento(0));
        System.out.println("Restaur     : " + editor.getContent());
    }
}
```

Listing 2: Utilisation du Mémento dans un éditeur de texte

5. Conclusion

Le Mémento est un patron idéal pour des systèmes nécessitant une gestion de l'historique ou des états successifs tout en respectant l'encapsulation. Cet exemple montre comment il peut être appliqué efficacement dans des applications comme un éditeur de texte.

Partie 2: Analyse générale des patrons de conception appliqués au projet de gestion de restaurant

1. Répartition des membres du groupe

Les membres de notre groupe ont travaillé sur différents patrons de conception appliqués au projet, comme suit :

- **DERMOUCHE Mohammed Ryad** : Mémento.
- **BOUCHELLAL Imad-Eddine** : Façade.
- **OUBERKA Mohamed** : Prototype.

2. Analyse des patrons de conception

2.1 Patron 1 : Façade

Objectif : Simplifier l'interaction avec les sous-systèmes complexes (comme la gestion des commandes, des paiements, ou du stock) en fournissant une interface unifiée via une classe **RestaurantFacade**.

Modification sur le diagramme de classe :

- Ajout d'une classe **RestaurantFacade** qui coordonne les interactions entre les classes principales telles que **Commande**, **Paiement**, **Stock**, et **Table**.

2.2 Patron 2 : Prototype

Objectif : Réduire la complexité de création d'objets similaires, notamment pour les classes **ArticleDuMenu** et **Commande**.

Modification sur le diagramme de classe :

- Ajout d'une méthode `clone()` dans **ArticleDuMenu** pour permettre le clonage d'articles.
- Implémentation du patron Prototype pour les objets pouvant être clonés.

2.3 Patron 3 : Mémento

Objectif : Sauvegarder et restaurer l'état des objets, comme les **Commande** ou **Table**, pour permettre un retour en arrière ou une annulation.

Modification sur le diagramme de classe :

- Ajout d'une classe **Memento** associée à la classe **Commande**.
- Ajout des méthodes `saveState()` et `restoreState(Memento m)` dans la classe **Commande**.

3. Diagramme de Classe Modifié

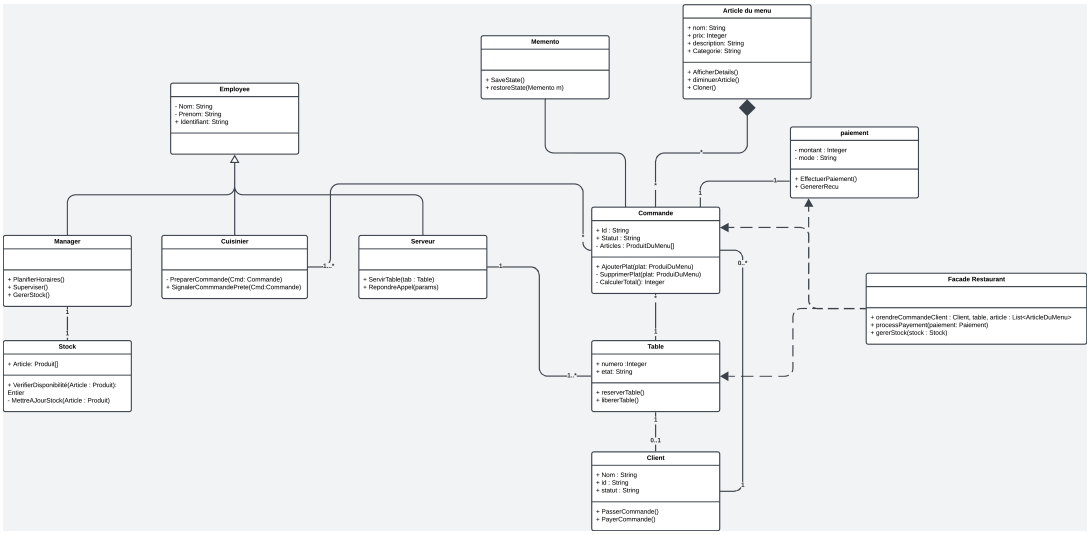

Figure 1: Diagramme de classe modifié avec les patrons de conception Façade, Prototype et Mémento.