Visitor Pattern

Comment accepter la flexibilité sans déranger l'ordre existant

Loïc Herman, Samuel Roland, Massimo Stefani, et Timothée Van Hove

16 juin 2024

HEIG-VD — MCR

Sommaire

Contexte

Intention

Le visiteur est un modèle de conception **comportemental** qui permet de définir **de nouvelles opérations** sur un groupe d'objets **sans apporter de modifications** à la structure de l'objet existante.

Ce modèle favorise l'application de l'**OCP** en facilitant l'extension des fonctionnalités sans nécessiter de modifications des classes existantes.

Motivation

- Jeu vidéo : Un joueur interagit avec des PNJ, des objets et des obstacles.
- Chaque type d'objet nécessite des interactions différentes : parler aux PNJ, ramasser des objets, contourner des obstacles.
- Implémenter ces interactions dans les classes PNJ, objet et obstacle peut rendre le code complexe et difficile à maintenir, surtout lorsque le jeu évolue.
- Le modèle Visiteur permet d'externaliser la logique d'interaction dans des classes de visiteurs séparées.
- L'ajout de nouveaux types d'interactions est facilité sans modifier la logique de jeu existante.

Définition

Diagramme général

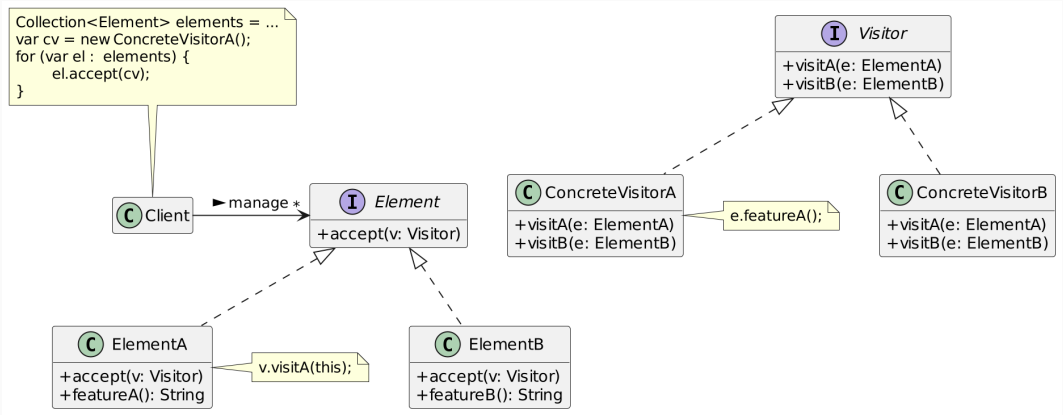

Figure 1 : Diagramme général du pattern Visiteur

Avantages

- Permet de séparer les algorithmes de traitement des objets, favorisant ainsi la modularité et la maintenabilité du code.
- Facilite l'ajout de nouveaux algorithmes sans modifier les classes des objets, ce qui respecte le principe d'ouverture/fermeture du principe SOLID.
- Favorise le traitement polymorphique des objets. Les objets peuvent être traités de manière différente en fonction du visiteur, évitant ainsi l'utilisation des 'instanceof'. Il utilise une technique appelée double répartition (double dispatch), qui aide à lancer la bonne méthode sans s'encombrer avec des blocs conditionnels.

Inconvénients

- Engendre une complexité accrue dans le code, notamment en raison de la nécessité de créer des classes de visiteurs pour chaque algorithme.
- Dans certains cas, le pattern Visiteur peut violer le principe d'encapsulation, car il nécessite que les classes à visiter exposent une méthode 'accept' pour accepter le visiteur.
- Si vous devez ajouter de nouveaux types d'objets à visiter, vous devrez modifier l'interface Visitor pour inclure une méthode pour chaque nouveau type. Ce qui peut être difficile dans une grande structure.

Exemple

Système d'interactions

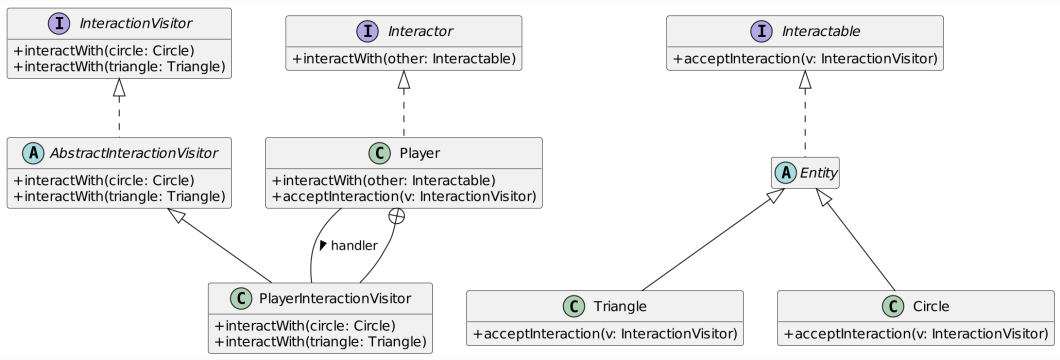

Figure 2 : Modèle pour les interactions joueur-objets

Interface commune aux entités

```
public interface Interactable {

    /// When checking if interactable is in cell
    DiscreteCoordinates getCoordinates();

    /// Accept interaction from interactor
    void acceptInteraction(InteractionVisitor v);
}

public interface Interactor {

    /// When checking if interactor is in cell
    DiscreteCoordinates getCoordinates();

    /// When an interaction can be triggered
    void interact(Interactable other);
}
```

Définition du visiteur

```java
/// Shared contract for the behaviour handlers
public interface InteractionVisitor {

    void interactWith(Circle circle);

    void interactWith(Triangle triangle);
}

/// Support class for building visitors
public abstract class AbstractInteractionVisitor
    implements InteractionVisitor {

    @Override
    public void interactWith(Circle circle) {
    }

    // [...]
}
```

Séquence

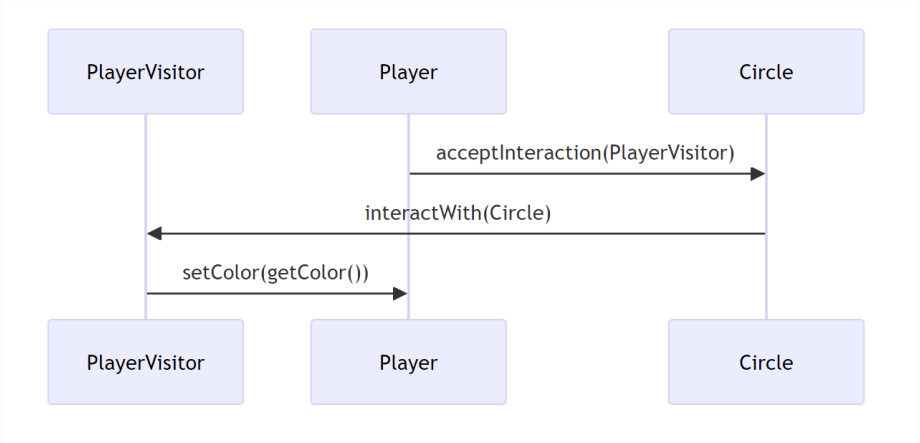

Figure 3 : Séquence des interactions

Formation des pièces

```java
public class Circle implements Interactable {
    // [...]

    @Override
    public void acceptInteraction(InteractionVisitor v) {
        // Simply accept the interaction
        v.interactWith(this);
    }
}
```

Création des premières interactions

```
public class Player implements Interactor {
    private PlayerVisitor handler = new PlayerVisitor();

    @Override
    public void interact(Interactable other) {
        // Simply try to interact
        other.acceptInteraction(handler);
    }

    private class PlayerVisitor extends AbstractInteractionVisitor {
        @Override
        public void interactWith(Circle circle) {
            setColor(circle.getColor());
        }

        // [...]
    }
}
```

Démonstration

Ajout d'une fonctionalité

Un pac-man est apparu, on aimerait ajouter une interaction

- Ajout de la méthode *interactWith(Pacman p)* dans le visitor
- Définition de l'interaction pour le joueur dans son visiteur

Ajout d'une fonctionalité

Un pac-man est apparu, on aimerait ajouter une interaction

- Ajout de la méthode *`interactWith(Pacman p)`* dans le visitor
- Définition de l'interaction pour le joueur dans son visiteur

→ Sans modifier la nouvelle entité ou le joueur concret, on peut ajouter sans difficulté une nouvelle interaction avec lui.

Questions?