R5.A.08 : Qualité de développement
(I.Borne)

Cours 8

- Intention de construction : Memento
- Intention de responsabilité : Mediator

Classification des patterns selon leur intention

Intention	Patterns
Interfaces	Adapter, Facade, Composite, Bridge
Responsability	Singleton, Observer, Mediator, Proxy, Chain of Responsability, Flyweight
Construction	Builder, Factory Method, Abstract Factory, Prototype, memento
Operations	Template Method, State, Strategy, Command, Interpreter
Extensions	Decorator, Iterator, Visitor

Intention de construction

- Construction ordinaire
  - Constructeur Java
  - Aspect important : collaboration que l'on peut orchestrer parmi les constructeurs
- Collaboration de superclasse
  - Chainage des constructeurs
- Collaboration dans une classe
  - Interaction des constructeurs d'une classe

Le patron Memento

Intention

Sans violer l'encapsulation, capturer et externaliser l'état interne d'un objet afin que l'objet puisse être restauré dans cet état ultérieurement.

Motivation

Parfois il est nécessaire d'enregistrer l'état interne d'un objet, notamment quand on veut implémenter des mécanismes de undo.

Mais les objets encapsulent leur état, le rendant inaccessible aux autres objets et impossible à sauver à l'extérieur.

Exemple : Choix d'options supplémentaires lors de l'achat en ligne d'un véhicule neuf.

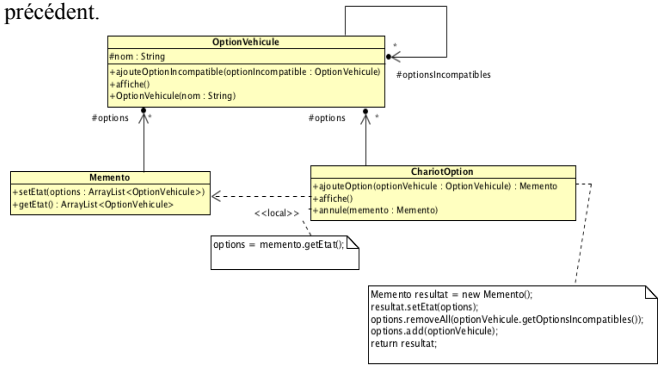
Les options sont ajoutées au chariot du client. Il existe des options incompatibles.

On désire ajouter une option d'annulation de la dernière opération effectuée

dans le chariot, puis l'étendre pour gérer un historique des états du chariot et pouvoir revenir dans n'importe quel état.

Exemple de motivation : options d'un véhicule

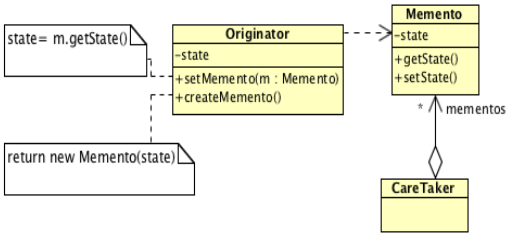
Le pattern Memento consiste à mémoriser les états du chariot dans un objet Memento. Lors de l'ajout d'une option, le chariot crée un memento, l'initialise avec son état, retire les options incompatibles avec cette nouvelle option, ajoute l'option et renvoie le memento. Seul le chariot peut mémoriser son état dans le memento et y restaurer un état précédent.



Quand utiliser le patron Memento

- L'état interne d'un objet doit être enregistré en externe afin que l'objet puisse être restauré ultérieurement dans cet état. L'encapsulation de l'objet ne doit pas être violée.
- Le problème est qu'un objet bien conçu est encapsulé de sorte que sa représentation (structure de données) soit cachée à l'intérieur de l'objet et ne soit pas accessible depuis l'extérieur de l'objet.
- Une partie de l'état interne d'un objet doit être sauvée afin qu'il puisse être restauré dans cet état ultérieurement.

Solution du pattern Memento



Participants

Le modèle Memento est implémenté avec trois objets : l'*originator* (auteur), un *caretaker* (gardien) et un *memento*.

- **Memento** : classe des objets qui mémorisent l'état interne des objets d'origine. Il possède deux interfaces : une interface complète destinée aux objets de l'originator et un interface réduite pour les objets du *CareTaker* qui n'ont pas le droit d'accéder à l'état interne des objets *Originator*.

- ***Originator (chariotOption)*** est la classe des objets qui possèdent un état interne et créent un memento pour mémoriser leur état interne qu'ils peuvent restaurer à partri d'un memento.

- Le ***caretaker*** est responsable de la gestion des mementos et n'accède pas à l'état interne des objets de l'*Originator*.

Bénéfices et défauts de Memento

- Préserver les frontières de l'encapsulation : l'originator a seulement une référence de l'objet Mémento et aucune façon d'accéder à l'information à l'intérieur.
- Simplifier l'Originator :
  - Il est plus facile de donner la responsabilité de gestion des différents états aux clients, pour que l'Originator ait juste besoin de créer et d'utiliser des mémentos au lieu de garder trace de multiples états
- Utiliser des mementos peut être cher : les mémentos coûtent très cher à créer si chaque élément de l'état de l'originator doit être stocké dans le mémento
- Stockage de mémento : le careTaker est responsable de la gestion du cycle de vie après la réception du mémento de l'originator, cependant il ne connaît pas la taille du mémento.

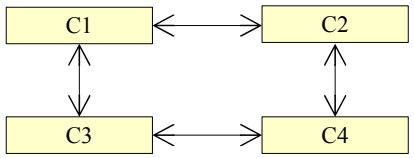
Intention de responsabilité

If your intent to	Apply the pattern
Centralize responsability in a single instance of a class	=> Singleton
Decouple an object from awareness of which other objects depend on it	=> Observer
Centralize responsability in a class that oversees how a set of other objects interact	=> Mediator
Let a object act on behalf of another objet	=> Proxy
Allow a request to escalate up a chain of objects until one handles it	=> Chain of Responsibility
Centralize responsability to shared fine-grained objects	=>Flyweight

Le patron Mediator

- Intention
  - Le pattern Mediator a pour but de construire un objet, le médiateur, dont la vocation est la gestion et le contrôle des interactions dans un ensemble d'objets sans que ses éléments doivent se connaître mutuellement.
- Motivation
  - On nous demande de développer un logiciel pour un nouveau modèle de lave-linge complètement automatique. La machine ajuste tous les réglages en fonction du type de vêtement et de leur composition.
  - Les classes importantes : la classe Machine qui a un tambour, un capteur pour vérifier la température, un moteur ...

Situation actuelle

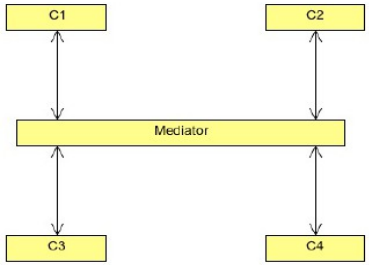


Les classes contiennent des références de chaque autre. La conception devient très couplé te difficile à maintenir.

Pour découpler ces objets nous avons besoin d'un médiateur qui contectera un objet pour le compte d'un autre permettra un couplage faible.

La solution avec un médiateur

- Le médiateur aide à réduire la complexité des classes
- Il encapsule l'interaction avec les objets et les rend indépendant de chaque autre.

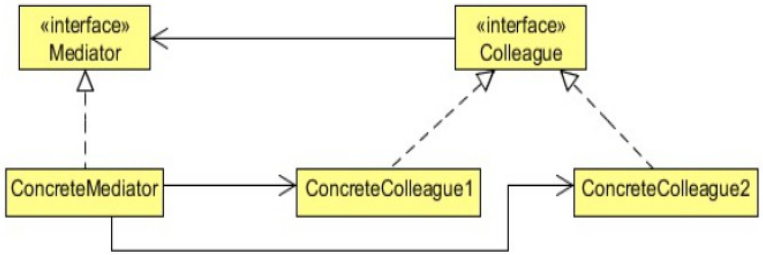


Domaines d'application

Le pattern Mediator est utilisé dans les cas suivants :

- un système est formé d'un ensemble d'objets basé sur une communication complexe conduisant à associer de nombreux objets entre eux ;
- les objets d'un système sont difficiles à réutiliser car ils possèdent de nombreuses associations avec d'autres objets ;
- la modularité d'un système est médiocre, obligeant dans le cas d'une adaptation d'une partie du système à écrire de nombreuses sous-classes.
- Utiliser un mediator signifie que le code doit résider à une seule place, ce qui plus facile à maintenir

Structure du Mediator



Mediator : participants

- **Mediator** : définit l'interface pour la communication entre les objets Colleague
- **ConcreteMediator** : implémente l'interface mediator et coordonne la communication entre les objets Colleague.
- **Colleague** : est soit une interface, soit la classe abstraite des colègues qui introduit leurs attributs, associations et méthodes communes.
- **ConcreteColleague** : chaque colleague connait son objet Mediator et communique avec avec les autres colleagues via son médiateur .

Exemple du lave-linge

```
public interface MachineMediator {
      public void start();
      public void wash();
      public void open();
      public void closed();
      public void on();
      public void off();
      public boolean checkTemperature(int temp);
}
```

Les classes *colleague* auront une méthode pour affecter le médiateur.

```
public interface Colleague {
    public void setMediator(MachineMediator mediator);
}
```



```
public class Button implements Colleague {
    private MachineMediator mediator;

    public void setMediator(MachineMediator mediator){
        this.mediator = mediator;
    }

    public void press(){
        System.out.println("Button pressed.");
        mediator.start();
    }
}
```