Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 5

GoF Design Patterns – Behavioral Part 2
Memento

■ Intent:
  □ Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

■ Applicability:
  □ Use the Memento pattern when
    ■ a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, \textit{and}
    ■ a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.
Memento: Structure

```
[Originator]
SetMemento(Memento m)
CreateMemento()

[State]

return new Memento(state)

[Multiple Associations]

[Originator]

[State]

state

[Memento]
GetState()
SetState()

[state]

state = m->GetState()

[Caretaker]
```
Memento: Collaboration

Diagram showing aCaretaker, anOriginator, and aMemento with actions CreateMemento(), SetState(), GetState(), and SetMemento(aMemento).
Memento: Consequences

- **Preserving encapsulation boundaries.** The pattern shields other objects from potentially complex Originator internals.

- **It simplifies Originator.** Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done.

- **Using mementos might be expensive.** Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return many mementos.

- **Defining narrow and wide interfaces.** It may be difficult in some languages to ensure that only the originator can access the memento's state.

- **Hidden costs in caring for mementos.**
Observer

Intent:
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Observer: Applicability

- Use the Observer pattern when
  - an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - a change to one object requires changing others, and you don't know how many objects need to be changed.
  - an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.
Observer: Structure
Observer: Collaboration
Observer: Consequences

✓ Abstract coupling between Subject and Observer.

✓ Support for broadcast communication. The notification is broadcast automatically to all interested objects that subscribed to it.

✗ Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

✗ A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
State

- **Intent:**
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
State: Applicability

Use the State pattern when

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state.
State: Structure

[Diagram showing the State pattern with classes and methods]
State: Consequences

- It localizes state-specific behavior and partitions behavior for different states. New states and transitions can be added easily by defining new subclasses.

- It makes state transitions explicit.

- State objects can be shared.
**Strategy**

- **Intent:**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Strategy: Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.
Strategy: Structure
Strategy: Consequences

- Families of related algorithms.
- An alternative to subclassing.
- Strategies eliminate conditional statements.
- A choice of implementations. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
- Increased number of objects.
Visitor

- **Intent:**
  - Represent an operation to be performed on the elements of an object structure; lets you define a new operation without changing the classes of the elements on which it operates.
Visitor: Applicability

- Use the Visitor pattern when
  - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure.
Visitor: Structure

- **Visitor**
  - VisitConcreteElementA(ConcreteElementA)
  - VisitConcreteElementB(ConcreteElementB)

- **ConcreteVisitor1**
  - VisitConcreteElementA(ConcreteElementA)
  - VisitConcreteElementB(ConcreteElementB)

- **ConcreteVisitor2**
  - VisitConcreteElementA(ConcreteElementA)
  - VisitConcreteElementB(ConcreteElementB)

- **Client**

- **ObjectStructure**

- **Element**
  - Accept(Visitor)

- **ConcreteElementA**
  - Accept(Visitor v)
  - OperationA()
  - v->VisitConcreteElementA(this)

- **ConcreteElementB**
  - Accept(Visitor v)
  - OperationB()
  - v->VisitConcreteElementB(this)
Visitor: Collaborations
Visitor: Consequences

- **Visitor makes adding new operations easy.**
- **A visitor gathers related operations and separates unrelated ones.**

- Adding new ConcreteElement classes is hard.
- **Breaking encapsulation.** The pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.
Reference