Design patterns

Venkatesh

WDC

March 20, 2018

Outline

Structural Patterns

- Responsible for building simple and efficient class hierarchies and relations between different classes
- Class patterns use inheritance
- Object patterns use `object composition`

Adapter

- Also known as `Wrapper`
- Used to make existing classes work with others without modifying their source code

Adapter ...

solves problems like:

How can a class be reused that does not have an interface that a client requires?

Adapter ...

how to solve:

Define a separate Adapter class that converts the (incompatible) interface of a class (Adaptee) into another interface (Target) clients require.

Adapter ...

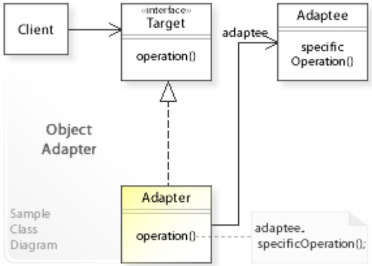
There are two ways to define an Adapter:

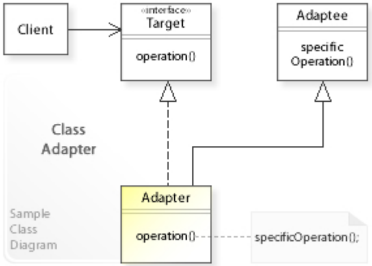
- `Class Adapter` : Uses inheritance to implement Target Interface
- `Object Adapter` : Uses object composition to implement Target Interface

Note:

Adapter is responsible for functionality the adapted class doesn't provide

Structure


Client
«interface»
Target
operation()
adaptee
Adaptee
specific Operation()
Object Adapter
Adapter
operation()
adaptee. specificOperation();
Sample Class Diagram


Client
«interface»
Target
operation()
Adaptee
specific Operation()
Class Adapter
Adapter
operation()
specificOperation();
Sample Class Diagram

Sample Code

Class adapter vs Object adapter

class adapter

- adapts Adaptee to Target by commiting to a concrete Adapter class
    - It won't work, if we want to adapt a class and all its subclasses
- lets Adapter override some of Adaptee's behavior
- no additional pointer indirection is needed

Class adapter vs Object adapter

object adapter

- lets a single Adapter work with many Adaptees - Adaptee itself and all of its subclasses
- harder to override Adaptee behavior

Bridge

- Also Known as `"Handle/Body"`
- "Decouple an abstraction from its implementation so that the two can vary independently"

Bridge ...

solves problems like:

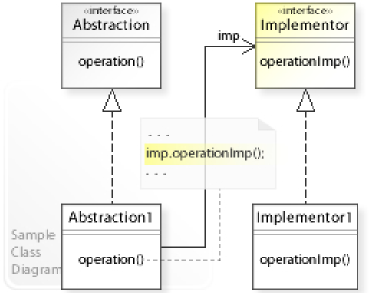
A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

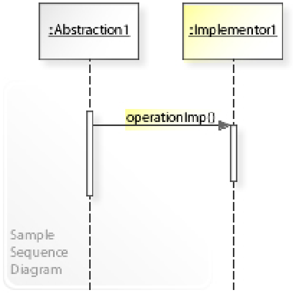
Bridge ...

how to solve:

- Separate an abstraction (Abstraction) from its implementation (Implementor) by putting them in separate class hierarchies
- Implement the Abstraction in terms of (by delegating to) an Implementor object

Structure


«interface»
Abstraction
operation()
«interface»
Implementor
operationImp()
imp
...
imp.operationImp();
...
Abstraction1
operation()
Implementor1
operationImp()
Sample Class Diagram


:Abstraction1
:Implementor1
operationImp()
Sample Sequence Diagram

Sample Code

Proxy

- Also known as `Surrogate`
- Provide a surrogate or placeholder for another object to control access to it

Proxy ...

solves problems like:

- How can the access to an object be controlled?
- How can additional functionality be provided when accessing an object?

Proxy ...

how to solve:

Define a separate Proxy object that

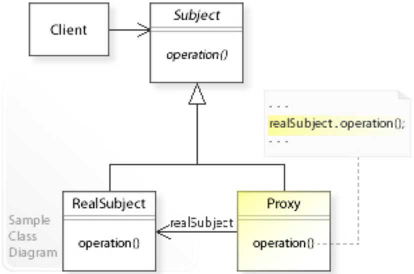
- can be used as substitute for another object (Subject)
- Work through a Proxy object to control the access to an object

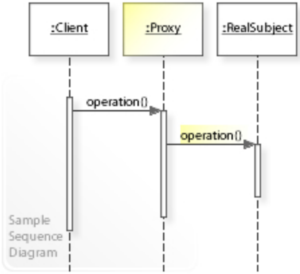
Proxy...

Common Kinds of Proxies

- remote proxy : hide complex network communication details
- virtual proxy : defer expensive objects creation until needed
- protection proxy : check access rights for sensitive objects
- smart reference : performs additional actions when object is accessed

Structure


Client
Subject
operation()
. . .
realSubject.operation();
. . .
RealSubject
operation()
realSubject
Proxy
operation()
Sample
Class
Diagram


:Client
:Proxy
:RealSubject
operation()
operation()
Sample
Sequence
Diagram

Sample Code

Adapter vs Proxy

- An Adapter provides a different interface to the object it adapts
- A proxy provides the same interface as its subject

Adapter vs Bridge

- Adapter pattern is applied to systems `after they're designed`
- Bridge is used `up-front in design` to let abstractions and implementations vary independently

Behavioral Patterns

- Concerned with
  - algorithms
  - assignment of responsibilities between objects
  - communication between objects

Mediator

- Define an object that encapsulates how a set of objects interact
- Objects no longer communicate directly with each other
    - instead communicate through the mediator

Mediator ...

solves problems like:

- How can tight coupling between a set of interacting objects be avoided?
- How can the interaction between a set of objects be changed independently?

Mediator ...

how to solve:

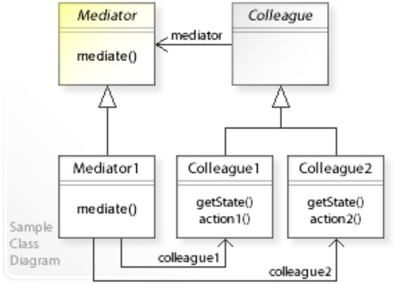
- Define a separate (mediator) object that encapsulates the interaction between a set of objects
- Objects delegate their interaction to a mediator object
  - instead of interacting with each other directly

Mediator ...

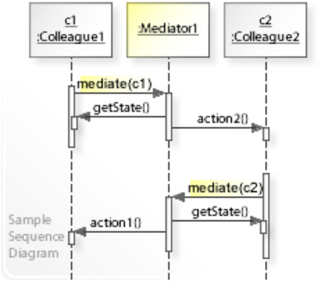
Consequences

- It decouples colleagues : promotes loose coupling between colleagues
- It centralizes control : makes the mediator itself a monolith

Structure



Mediator
mediate()
mediator
Colleague
Mediator1
mediate()
Colleague1
getState()
action1()
Colleague2
getState()
action2()
colleague1
colleague2
Sample Class Diagram



c1 :Colleague1
:Mediator1
c2 :Colleague2
mediate(c1)
getState()
action2()
mediate(c2)
getState()
action1()
Sample Sequence Diagram

Sample Code

Memento

- Provides the ability to restore an object to its previous state (undo)

Memento ...

solves problems like:

- Without violating encapsulation,
- internal state of an object should be saved externally so that the object can be restored to this state later

Memento ...

how to solve:

Make an object (originator) itself responsible for

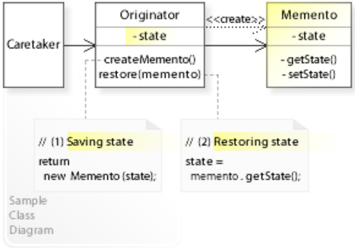
- saving its internal state to a (memento) object and
- restoring to a previous state from a (memento) object
- only the originator is allowed to access a memento

Memento ...

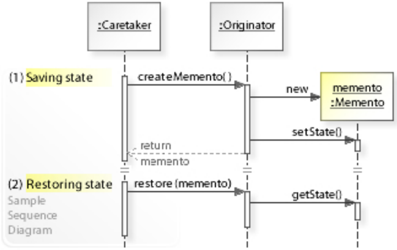
Usage

- A client (caretaker) can request a memento from the originator (to save the internal state of the originator)
- A client can pass a memento back to the originator (to restore to a previous state)
- Caretaker is responsible for deleting the mementos it cares for

Structure



Caretaker
Originator
- state
createMemento()
restore(memento)
<<create>>
Memento
- state
- getState()
- setState()
// (1) Saving state
return
new Memento (state);
// (2) Restoring state
state =
memento . getState();
Sample
Class
Diagram



:Caretaker
:Originator
(1) Saving state
createMemento( )
new
memento
:Memento
setState()
return
memento
(2) Restoring state
restore (memento)
getState()
Sample
Sequence
Diagram

Memento ...

Consequences

- Using mementos might be expensive : copy large amount of information
- Defining narrow and wide interfaces : difficult in some languages

Sample Code

```
class State;
class Memento;

class Originator {
public:
    Memento* CreateMemento();
    void restore(const Memento
    *);
    // ..
private:
    // internal data structures
    State* _state;
    // ...
};
```

```
class Memento {
public:
// narrow public interface
    virtual ~Memento();
private:
    friend class Originator;
    Memento(State *);
    State * getState();
    // ...
private:
    State * _state;
    // ...
};
```

References

Design Patterns - Erich Gamma, Richard helm, Ralph Johnson, John Vlissides

Thank You