Design Patterns Part 1

CSC02A2

Outline

Outline

Design Patterns

Design Patterns

Design Patterns:

A general, repeatable solution to a commonly occurring problem.

There are three general categories of design pattern:

- `Behavioural` - deal with interactions between classes
- `Sturctural` - deal with the composition or arrangement of classes
- `Creational` - deal with the instantiation of class objects

Behavioural Design Patterns

Behavioural Design Patterns

Behavioural Design Patterns are concerned with algorithms and the assignment of responsibilities between objects.

Describe:

- Patterns of objects or classes, and
- Petterns of communication between them.

Let You:

- Shift focus away from flow control and
- Concentrate on how objects are interconnected.

Visitor

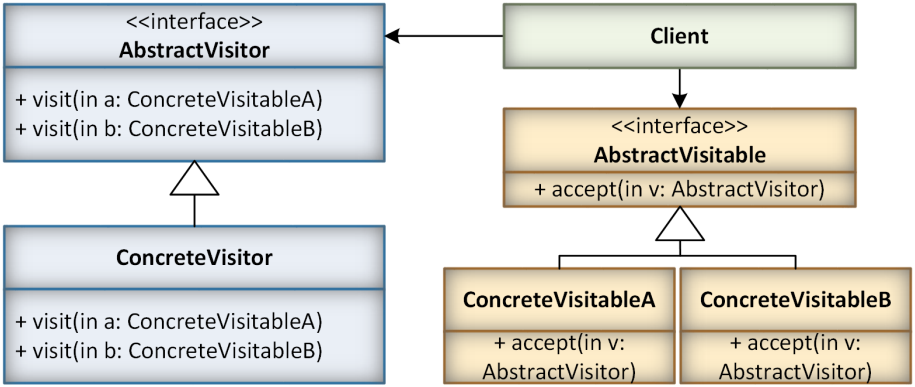
Overview of the Visitor Design Pattern

`Visitor` represents an operation to be performed on the elements of an object structure.
Lets you define new operations without changing the classes of the elements on which it operates.

Used When:

- An object structure contains many classes of ojects with differing interfaces.
- Many distinct and unrelated operations need to be performed on objects in the object structure.
- Classes seldom change but you want to define new operations.

Visitor Structure



Visitor Participants I

AbstractVisitor

- declares a `visit()` operation for every `ConcreteVisitable` in the object structure.
- The operation's name and signature identifies the concrete class that sends the `visit()` request to the visitor.
- The visitor determines which `ConcreteVisitable` to visit.
- The visitor directly accesses the `ConcreteVisitable` through its interface.

ConcreteVisitor

- Implements each of the `visit()` operations in the `AbstractVisitor`.
- The `ConcreteVisitor` provides the context for the algorithm and stores its local state.
- This state often accumulates results during the traversal (visiting `ConcreteVisitables`) of the structure.

Visitor Participants II

AbstractVisitable

- Defines an `accept()` operation that takes a visitor as an argument.

ConcreteVisitable

- Implements an `accept()` operation that takes a visitor as an argument.

Client

- Can enumerate its elements or store them in a collection such as an ArrayList.
- May provide a high-level interface to allow the `ConcreteVisitor` to visit its `ConcreteVisitables`.

Visitor Problem

- We often want to perform a variety of different actions (interpretations) upon a collection of different kinds of objects.
- The encapsulation principle of object orientation would suggest that we embed the action we want performed within the objects themselves.
- The principle of Separation of Concerns however would lean towards *separating* the *collection* from its *interpretation*.
- This however would require testing the elements in a collection during iteration (e.g. using `instanceof(c)`) followed by an appropriate casting and then method invocation (polymorphism).
- The approach adopted has an effect on the modularity of the system.
- In either case, an *interpretation* rule for ever *kind* must be provided and the addition of a new *interpretation* for one *kind* must be implemented for all *kinds*.

The Difference:

- The separation of concerns approach makes it easy to add new *interpretations* but not new *kinds*.
- The object oriented approach allows for the easy addition of *kinds* but not *interpretations*.

Visitor Example

Very Important for Practical Test

Very Important for Practical X (The Game)

Time for examples...

Visitor Code I

AbstractVisitor class:

```
1 public interface AbstractVisitor {
2   //One abstract method for each concreteVisitable
3   public void visit(ConcreteVisitableA a);
4   public void visit(ConcreteVisitableB b);
5 }
```

ConcreteVisitor class:

```
1 public class ConcreteVisitor implements AbstractVisitor{
2   public void visit(ConcreteVisitableA a){
3     System.out.println("Do Something Count: " + a.
          getDoSomethingCount());
4   }
5   public void visit(ConcreteVisitableB b){
6     System.out.println("Do Something Else Count: " + b.
          getDoSomethingElseCount());
7   }
8 }
```


Visitor Code II

AbstractVisitable class:

```
1  public interface AbstractVisitable {
2    //Accepts from the AbstractVisitor interface
3    public void accept(AbstractVisitor v);
4  }
```

ConcreteVisitableA class:

```
1  public class ConcreteVisitableA implements AbstractVisitable{
2    private int doSomethingCounter = 0;
3    public void doSomething(){doSomethingCounter++;}
4    public int getDoSomethingCount(){return doSomethingCounter;}
5    //Implements method from interface
6    public void accept(AbstractVisitor v){
7      doSomething();
8      v.visit(this);
9    }
10 }
```


Visitor Code III

ConcreteVisitableB class:

```
1  public class ConcreteVisitableB implements AbstractVisitable{
2    private int doSomethingElseCounter = 0;
3    public void doSomethingElse(){doSomethingElseCounter++;}
4    public int getDoSomethingElseCount(){return
         doSomethingElseCounter;}
5    //Implements method from interface
6    public void accept(AbstractVisitor v){
7      doSomethingElse();
8      v.visit(this);
9    }
10 }
```


Visitor Code IV

`Client` class:

```
1  public class Client {
2    public static void main(String[] args){
3      //Create ConcreteVisitor
4      ConcreteVisitor concreteVisitor = new ConcreteVisitor();
5      //Create ConcreteVisitables
6      ConcreteVisitableA concreteVisitableA = new ConcreteVisitableA
           ();
7      ConcreteVisitableB concreteVisitableB = new ConcreteVisitableB
           ();
8      //Test visiting the various ConcreteVisitables (using accept())
9      concreteVisitableA.accept(concreteVisitor);
10     concreteVisitableA.accept(concreteVisitor);
11     concreteVisitableA.accept(concreteVisitor);
12     concreteVisitableB.accept(concreteVisitor);
13     concreteVisitableB.accept(concreteVisitor);
14   }
15 }
```


Visitor Code Output

Output:

```
Do Something Count: 1
Do Something Count: 2
Do Something Count: 3
Do Something Else Count: 1
Do Something Else Count: 2
```


Observer

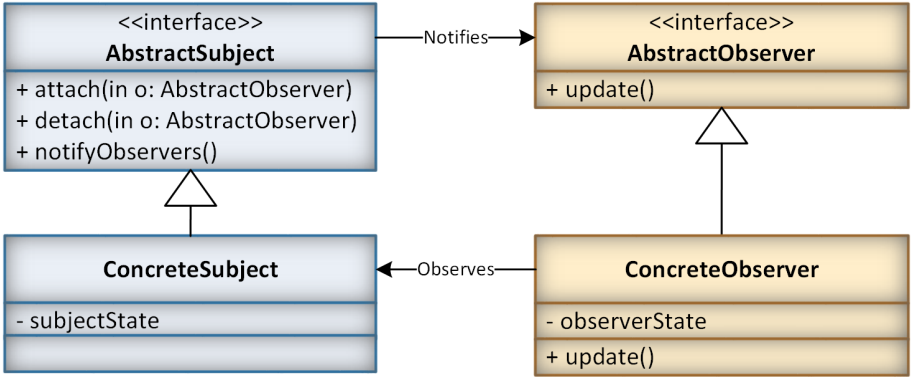
Observer Design Pattern

`Observer` (also known as `Dependents` or `Publish-Subscribe`) defines a one-to-many dependancy between objects so that when one object changes state, all its dependents are notified and updated automatically.

Used When:

- When one aspect of an abstraction is dependent on another. (Encapsuling each aspect into a separate object lets you vary and reuse each independently.)
- When a change to one object requires changing others and you dont know how many objects need changes.
- When an object should be able to notify other objects of changes without making assumptions about who these object are. (No tight couplings between objects.)

Observer Structure



Observer Participants I

AbstractSubject

- Knows about its `AbstractObservers`.
- Provides and interface for attaching and detaching `AbstractObserver` objects.

ConcreteSubject

- Stores the state of interest to `ConcreteObserver` objects.
- Sends a notification to its `AbstractObservers` when its state changes.

Observer Participants II

AbstractObserver

- Defines an `update()` interface for objects that should be notified of changes in an `AbstractSubject`'s state.

ConcreteObserver

- Maintains a reference to a `ConcreteSubject` objects.
- Stores state that must be consistent with the `ConcreteSubject`'s.
- Implements the `AbstractObserver`'s `update()` interface to keep its state consistent with the subject's.

Observer Example I

Consider a messaging application where nothing happens until a new message is *published* then all subscribed components need to be notified of the message.

In this case, the Observer pattern allows us to specify which components must *subscribe* to the message feed (*subject*) to get notified when a new message comes in.

The Observer then observes the *subject* for any changes and notifies components when a new message is published.

(Hence why Observer is also called publish-subscribe.)

Observer Code I

`AbstractSubject` class:

```
1 public interface AbstractSubject {
2   public void attach(AbstractObserver obs);
3   public void detach(AbstractObserver obs);
4   public void notifyObservers();
5   public Object getUpdate(AbstractObserver obs);
6 }
```


ConcreteSubject class:

```
1  import java.util.ArrayList;
2  import java.util.List;
3  public class ConcreteSubject implements AbstractSubject{
4    private List<AbstractObserver> observers = new ArrayList<>();
5    private boolean changed;
6    private String subjectState;//The internal state that is observed

7    public void attach(AbstractObserver obs){
8      if(obs != null && !observers.contains(obs)){observers.add(obs)
           ;}}
9    public void detach(AbstractObserver obs){observers.remove(obs);}
10   public void notifyObservers(){
11     List<AbstractObserver> observersLocal = null;
12     if(!changed){return;}
13     observersLocal = new ArrayList<>(this.observers);
14     this.changed = false;
15     for(AbstractObserver obs: observersLocal){obs.update();}  }
16   public Object getUpdate(AbstractObserver obs){return this.
         subjectState; }
17   public void changeState(String msg){
18     System.out.println("Changing state to: "+msg);
19     this.subjectState = msg;
20     this.changed = true;
21     notifyObservers();}
22 }
```


Observer Code I

`AbstractObserver` class:

```
1 public interface AbstractObserver {
2   public void update();
3   public void setAbstractSubject(AbstractSubject sub);
4 }
```


Observer Code II

ConcreteObserver class:

```
1  public class ConcreteObserver implements AbstractObserver{
2    private String name;  //For testing
3    private AbstractSubject subject;
4    public ConcreteObserver(String name){this.name = name;  }
5    public void update(){
6      String msg = (String) subject.getUpdate(this);
7      if(msg == null){
8        System.out.println(name + " has not changed state");
9      }else{
10       System.out.println(name + " has changed state to: " + msg);}}
11   public void setAbstractSubject(AbstractSubject sub){this.subject
         = sub;}
12 }
```


Observer Code III

`Client` class:

```
1  public class Client {
2    public static void main(String[] args){
3      //Create the ConcreteSubject
4      ConcreteSubject subject = new ConcreteSubject();
5      //Create the COncreteObservers
6      AbstractObserver obs1 = new ConcreteObserver("Test1");
7      AbstractObserver obs2 = new ConcreteObserver("Test2");
8      //Register the ConcreteObservers to the ConcreteSubject
9      subject.attach(obs1);
10     subject.attach(obs2);
11     //Attach the observers to the subject
12     obs1.setAbstractSubject(subject);
13     obs2.setAbstractSubject(subject);
14     //Test: Check for update
15     obs1.update();
16     //Test: Change state
17     subject.changeState("Testing 123");
18   }
19 }
```


Command Code Output

Output:

```
Test1 has not changed state
Changing state to: Testing 123
Test1 has changed state to: Testing 123
Test2 has changed state to: Testing 123
```


Command

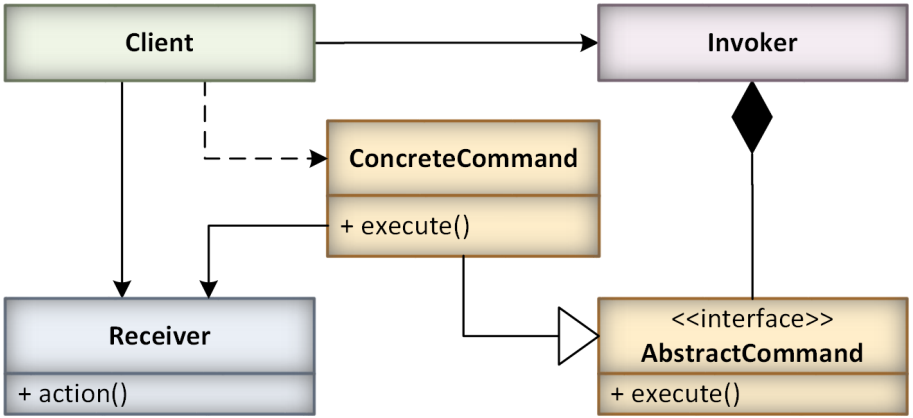
Command Design Pattern

`Command` (also known as `Action` or `Transaction`) encapsulates a request as an object, thereby letting you parameterise clients with different requests, queue or log requests, and support undoable operations.

Used When:

- You want to parameterise objects by an action to perform.
- Specify, queue, and execute requests at different times. (Commands can have a different lifetime to the original request.)
- Support the undoing of actions. (State can be stored in the command for later undo actions.)
- Support logging of changes as the commands themselves can be stored and repeated if need be.
- You want to structure a system around high-level operations built upon primitive operations.

Command Structure



Command Participants I

AbstractCommand

- Declares an interface for executing operations.

ConcreteCommand

- Defines a binding between a `Receiver` object and an action.
- Implements an `execute()` method by invoking the corresponding operation(s) on the `Receiver`.

Receiver

- Knows how to perform the operations associated with a `Command`.
- Any class may serve as a `Receiver`.

Command Participants II

Invoker

- Asks the `Command` to carry out the request.

Client

- Creates a `ConcreteCommand` oject and sets its `Receiver`.
- Specifies which `Commands` must be carried out.

Example

Consider a *light* that you want to turn *on* and *off* *remotely*. Using the Command Design Pattern:

- The *light* can be viewed as a `Receiver` that "receives" commands (on/off).
- The *on* and *off* actions are a type of `Command` that can be issued to the receiver.
- The *remote control* that you use to issue commands to the receiever can be viewed as the `Invoker`, as it "invokes" a particular command.
- The `Client` is the program which directs when the remote is used to initiate the on/off action.

Command Code I

AbstractCommand interface:

```
1 public interface AbstractCommand {
2   public void execute();
3 }
```

ConcreteCommand class:

```
1  public class ConcreteCommandTurnOn implements AbstractCommand{
2    //Reference to Receiver
3    LightReceiver receiver;
4    public ConcreteCommandTurnOn(LightReceiver receiver){
5      this.receiver = receiver;
6    }
7    public void execute(){
8      receiver.turnOn();
9    }
10 }
```


Command Code II

ConcreteCommand class:

```java
public class ConcreteCommandTurnOff implements AbstractCommand{
  //Reference to Receiver
  LightReceiver receiver;
  public ConcreteCommandTurnOff(LightReceiver receiver){
    this.receiver = receiver;
  }
  public void execute(){
    receiver.turnOff();
  }
}
```


Command Code III

Receiver class:

```
1  public class LightReceiver {
2    private boolean on = false;
3    public void turnOn(){
4      on = true;
5    }
6    public void turnOff(){
7      on = false;
8    }
9    //Accessor
10   public boolean getLightState(){
11     return on;
12   }
13 }
```


Command Code IV

Invoker class:

```
1 public class RemoteControlInvoker {
2   private AbstractCommand command;
3   public void setCommand(AbstractCommand command){
4     this.command = command;
5   }
6   public void pressButton(){
7     command.execute();
8   }
9 }
```


Client class:

```java
public class Client {
  public static void main(String[] args){
    //Invoker
    RemoteControlInvoker invoker = new RemoteControlInvoker();
    //Receiver
    LightReceiver receiver = new LightReceiver();
    //Commands
    ConcreteCommandTurnOn  turnLightsOn  = new
        ConcreteCommandTurnOn(receiver);
    ConcreteCommandTurnOff turnLightsOff = new
        ConcreteCommandTurnOff(receiver);
    //Switch lights ON
    invoker.setCommand(turnLightsOn);
    invoker.pressButton();//Executes Command
    System.out.printf("The lights are on: %b \n", receiver.
        getLightState());
    //Switch lights OFF
    invoker.setCommand(turnLightsOff);
    invoker.pressButton();//Executes Command
    System.out.printf("The lights are on: %b \n", receiver.
        getLightState());
  }
}
```


Command Code Output

Output:

```
The lights are on: true
The lights are on: false
```