Strategy Pattern

Example Problem Scenario

Suppose, you need to traverse the files and folders inside a given directory. So, you've decided to apply the BFS algorithm. Following is a sample pseudo-code solution to the problem:

```
public class Traversal {

    public static void main(String[] args) {
        Node sourceNode; //Input the source node

        Queue queue;
        queue.push(sourceNode);
        while (!queue.isEmpty()) {
            Node currentNode = queue.head(); //pops the head node and deletes it from queue
             Node[] children = currentNode.getChildrenNodes();
             for (Node node : children) {
                 print(node); //print the node in whatever manner we like
                 queue.push(node);
             }
         }
    }
}
```

However, some time later, suppose you decided to use DFS instead of BFS. So, you'll have to modify the code as follows:

```
public class Traversal {

    public static void main(String[] args) {
        Node sourceNode; //Input the source node
        traverse(sourceNode);
    }

    private static void traverse(Node currentNode) {
        Node[] children = currentNode.getChildrenNodes();
         for (Node node : children) {
             print(node); //print the node in whatever manner we like
             traverse(node);
         }
    }
}
```

Huge amount of change. This design has the following effects:

1. The class *Traversal* needs to be modified for the change in algorithm. But the class doesn't need to know anything about *how* the nodes are traversed.
2. The OCP (Open-Closed Principle[1]) of Object-Oriented Design Principles is violated.

How about the following solution?

```
interface TraversalAlgorithm {
    public void traverse(Node sourceNode);
}

class BFS implements TraversalAlgorithm {
    public void traverse(Node SourceNode) {
        Queue queue;
        queue.push(sourceNode);
        while (!queue.isEmpty()) {
            Node currentNode = queue.head(); //pops the head node and deletes it from queue
            Node[] children = currentNode.getChildrenNodes();
            for (Node node : children) {
                print(node); //print the node in whatever manner we like
```

[1] The OCP principle states that a class should be open for extension, but closed for modification.

```
                queue.push(node);
            }
        }
    }
}

class DFS implements TraversalAlgorithm {
    public void traverse(Node SourceNode) {
        Node[] children = sourceNode.getChildrenNodes();
        for (Node node : children) {
            print(node); //print the node in whatever manner we like
            traverse(node);
        }
    }
}

public class Traversal {
    public static void main(String[] args) {
        Node sourceNode; //Input the source node
        TraversalAlgorithm algorithm = new BFS();
        algorithm.traverse(sourceNode);
        //For DFS, just change BFS() to DFS()...
    }
}
```

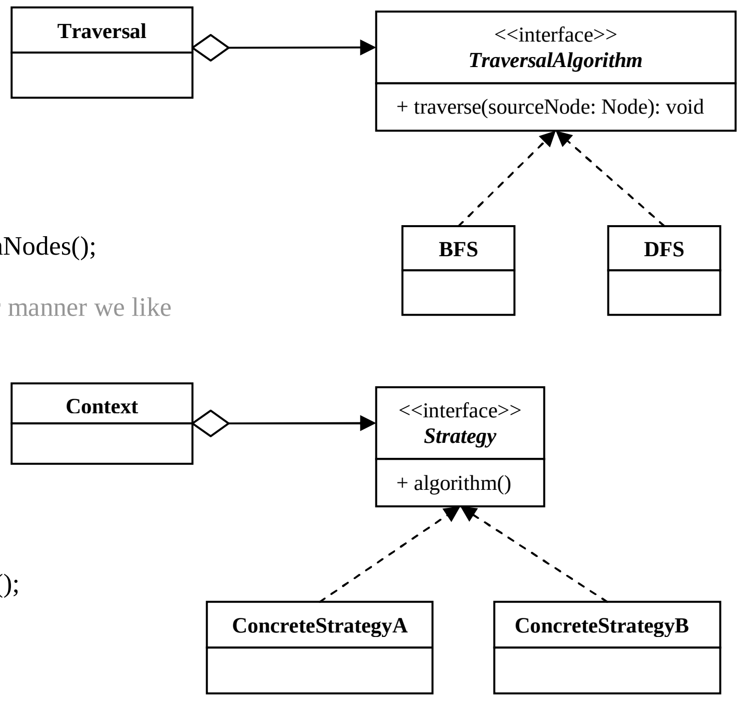


Figure: Strategy Pattern

Definition of Strategy Pattern

The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Example Use Cases:

1. Implementing different sort algorithms (quick sort, merge sort etc.).
2. You have to encrypt a file.

   For small files, you can use "in-memory" strategy, where the complete file is read and kept in memory (let's say for files less than 1 GB).

   For large files, you can use another strategy, where parts of the file are read in memory and partial encrypted results are stored in temporary files.

   These are two different strategies for the same task. The client code would look the same.
3. Bob needs to get to work in a morning. Now it's up to him how he goes about getting there. From the point of view of an outsider - Bob's boss for instance, it doesn't really matter how he gets to work, just that he arrives on time.

   When Bob starts his day he can choose the method by which he gets to work, based on what is available. He can assess if he has a car, if it is in a serviceable state and if the roads are clear. If not, Bob may decide to take the train and take into account his proximity from the station and the availability of a given train. But no matter what Bob's selected strategy may be, he is simply performing the action of getting to work.

Bridge Pattern

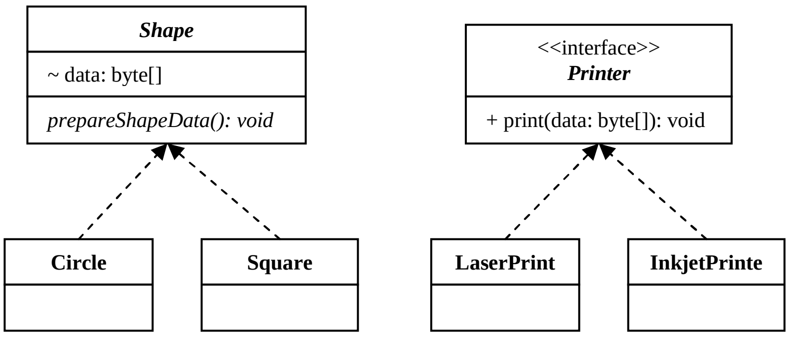
Example Problem Scenario

Suppose, you need to print some drawing shapes through a printer. Now, there can be various printers – laser, inkjet etc – whose printing algorithm differs. We can accommodate this using the strategy pattern. However, a shape itself has various forms – circle, square etc. So, for *m* shapes and *n* printers, there are *m* × *n* combinations. Is there any way of implementing this easily?

Solution Step 1

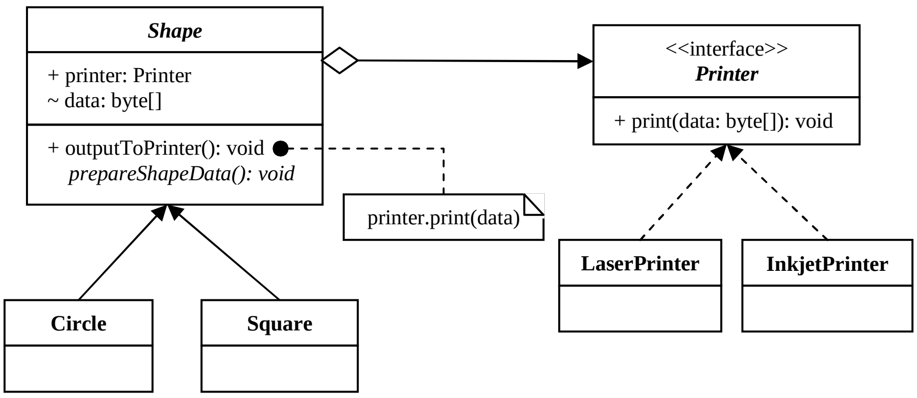
The various shapes and the various printers can be implemented *separately* using the strategy pattern. Below are the necessary classes:

```
interface Printer {
    public void print(byte[] data);
}

class LaserPrinter implements Printer {
    public void print(byte[] data) {
        //print data
    }
}

class InkjetPrinter implements Printer {
    public void print(byte[] data) {
        //print data
    }
}

abstract class Shape {
    protected byte[] data;

    abstract void prepareShapeData();
}

class Circle extends Shape {
    void prepareShapeData() {
        //Populate data with circle's pixels
    }
}

class Square extends Shape {
    void prepareShapeData() {
        //Populate data with square's pixels
    }
}
```



Solution Step 2 – Set up a bridge

```
abstract class Shape {
    public Printer printer;
    protected byte[] data;

    public void outputToPrinter() {
        printer.print(data);
    }

    abstract void prepareShapeData();
}
public class Client {
    public static void main(String[] args) {
        Shape shape = new Circle();
        shape.prepareShapeData();
        shape.printer = new LaserPrinter();
        shape.outputToPrinter();
    }
}
```



Definition of Bridge Pattern

The bridge pattern decouples an abstraction from its implementation so that the two can vary independently.

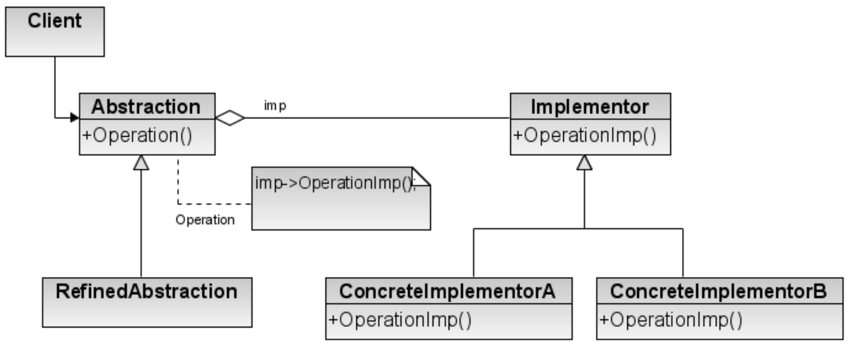


Figure: Bridge Pattern.

Example Use Cases

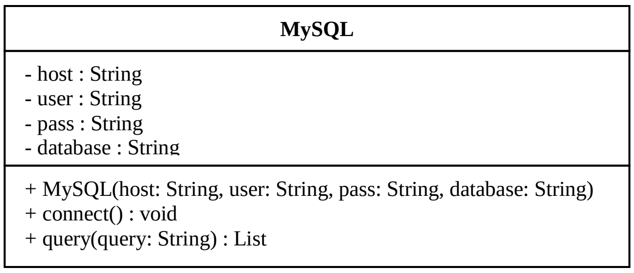
1. Switches are used to turn devices on and off. Devices can be of several types – light, fan etc. Again, switches can be of several types, too – normal switch, push-button switch, touch switch etc. Now, bridge pattern can be used to map any switch to any device.

Adapter / Wrapper Pattern

Example Problem Scenario

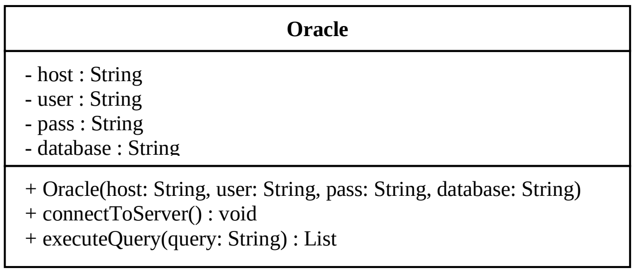
Suppose, you have the following code that executes database queries. It uses (ficticious) MySQL class APIs.

```java
public class Client {
   public static void main(String[] args) {
      MySQL mysql = new MySQL("localhost", "root", "", "someDB");
      mysql.connect();
      List result = mysql.query("select * from someTable");
   }
}

class MySQL {
   private String host, user, pass, database;

   MySQL(String host, String user, String pass, String database) {
      this.host = host;
      this.user = user;
      this.pass = pass;
      this.database = database;
   }

   public void connect() {
      //establish database connection with MySQL server
   }

   public List query(String query) {
      List result = new ArrayList<Object[]>();
      //fetch results from database and populate result list
      return result;
   }
}
```


MySQL
- host : String
- user : String
- pass : String
- database : String
+ MySQL(host: String, user: String, pass: String, database: String)
+ connect() : void
+ query(query: String) : List

Now, suppose you want to use Oracle class to use the Oracle database. The Oracle class is as follows:

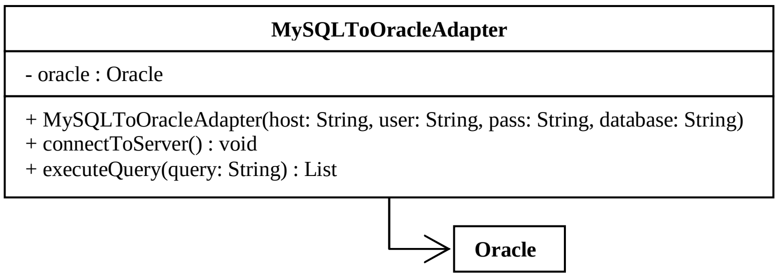
```
class Oracle {
  private String host, user, pass, database;

  Oracle(String host, String user, String pass, String database) {
    this.host = host;
    this.user = user;
    this.pass = pass;
    this.database = database;
  }

  public void connectToServer() {
    //establish database connection with MySQL server
  }

  public List executeQuery(String query) {
    List result = new ArrayList<Object[]>();
    //fetch results from database and populate result list
    return result;
  }
}
```



Oracle
- host : String
- user : String
- pass : String
- database : String
+ Oracle(host: String, user: String, pass: String, database: String)
+ connectToServer() : void
+ executeQuery(query: String) : List

Both the MySQL and Oracle classes do the same thing. However, their method names are different – the constructors, connect() vs. connectToServer() and query() vs. executeQuery(). Now, it's not possible to change the vendor Oracle class. We don't want to take the pain to change the method calls in Client code, either. In this case, we can simply provide a wrapper/adapter class which the Client code can use.

```
class MySQLToOracleAdapter {
  private Oracle oracle;

  MySQLToOracleAdapter(String host, String user, String pass, String database) {
    oracle = new Oracle(host, user, pass, database);
  }

  public void connect() {
    oracle.connectToServer();
  }

  public List query(String query) {
    return oracle.executeQuery(query);
  }
}
```



MySQLToOracleAdapter
- oracle : Oracle
+ MySQLToOracleAdapter(host: String, user: String, pass: String, database: String)
+ connectToServer() : void
+ executeQuery(query: String) : List
Oracle

The only change the Client needs now is just renaming MySQL to MySQLToOracleAdapter in the object instantiation statement.

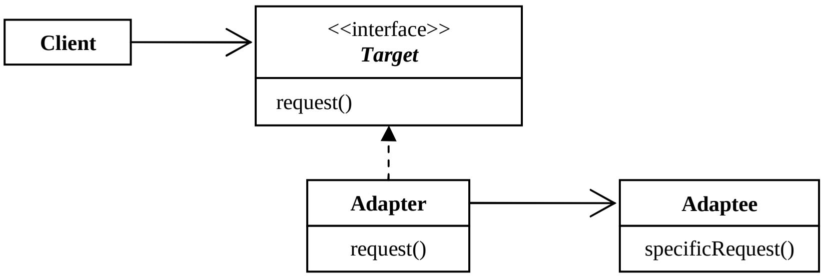
Definition of Adapter Pattern

The adapter pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Types of Adapter

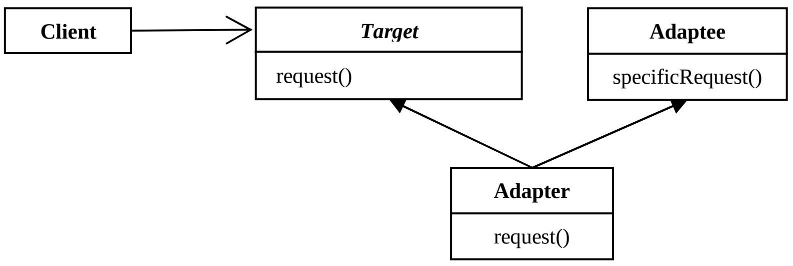
Object Adapter

This is the one we have been discussing.



Class Adapter

In class adapter, the Adapter subclasses Target and Adaptee (whereas in object adapter, composition is used to pass requests to an Adaptee). Class adapter uses multiple inheritance.



Observer Pattern

Example Problem Scenario

Suppose you're writing an antivirus software. The antivirus has three components – real-time file protection, real-time internet monitoring, always-on firewall. Now, suppose the user can give a command to pause and resume the antivirus. Upon pausing the antivirus, all the components would halt; and upon resume, all the components would resume their monitoring tasks. Now the problem is, how would all those components get the notification of pause/resume?

A quick solution can be as follows which includes only the pause action for brevity:

```
class RealTimeFileProtection {
   public void notifyPause() {
      //pause current task
   }
}

class RealTimeInternetMonitor {
   public void notifyPause() {
      //pause current task
   }
}

class AntivirusState {
   RealTimeFileProtection fileProtection;
   RealTimeInternetMonitor internetMonitor;

   public void pause() {
      fileProtection.notifyPause();
      internetMonitor.notifyPause();
   }
}

public class Main {
   public static void main(String[] args) {
      RealTimeFileProtection fileProtection = new RealTimeFileProtection();
      RealTimeInternetMonitor internetMonitor = new RealTimeInternetMonitor();

      AntivirusState antivirusState = new AntivirusState();
      antivirusState.fileProtection = fileProtection;
      antivirusState.internetMonitor = internetMonitor;

      antivirusState.pause();
   }
}
```

This solution works great! However, there is one problem – if we add another component, e.g. always-on firewall which also needs to be notified of pause; then the AntivirusState class needs to be modified. This is a clear violation of OCP (Open-Closed Principle). So, what can be done so that we can create as many components as we wish and can notify them of pause *without changing the code of AntivirusState*? Let's take a look at the following solution:

```java
interface Observer {
    public void notifyPause();
}

class RealTimeFileProtection implements Observer {
    public void notifyPause() {
        //pause current task
    }
}

class RealTimeInternetMonitor implements Observer {
    public void notifyPause() {
        //pause current task
    }
}

class AntivirusState {
    List<Observer> observers = new ArrayList<Observer>();

    public void registerForNotification(Observer observer) {
        observers.add(observer);
    }

    public void pause() {
        fileProtection.notifyPause();
        internetMonitor.notifyPause();
    }
}

public class Main {
    public static void main(String[] args) {
        Observer fileProtection = new RealTimeFileProtection();
        Observer internetMonitor = new RealTimeInternetMonitor();

        AntivirusState antivirusState = new AntivirusState();
        antivirusState.registerForNotification(fileProtection);
        antivirusState.registerForNotification(internetMonitor);

        antivirusState.pause();
    }
}
```

To add, for example, Firewall class to the system and register it with the AntivirusState, we need to write the Firewall class and change the Main class as follows:

```java
class Firewall implements Observer {
    public void notifyPause() {
        //pause current task
    }
}

public class Main {
    public static void main(String[] args) {
        Observer fileProtection = new RealTimeFileProtection();
        Observer internetMonitor = new RealTimeInternetMonitor();
        Observer firewall = new Firewall();

        AntivirusState antivirusState = new AntivirusState();
        antivirusState.registerForNotification(fileProtection);
        antivirusState.registerForNotification(internetMonitor);
        antivirusState.registerForNotification(firewall);

        antivirusState.pause();
    }
}
```

We can further reduce modifications by passing the AntivirusState object to the constructor of the components while instantiating them:

```java
class Firewall implements Observer {
    public Firewall(AntivirusState antivirusState) {
        antivirusState.registerForNotification(this);
    }

    public void notifyPause() {
        //pause current task
    }
}

public class Main {
    public static void main(String[] args) {
        AntivirusState antivirusState = new AntivirusState();

        Observer fileProtection = new RealTimeFileProtection(antivirusState);
        Observer internetMonitor = new RealTimeInternetMonitor(antivirusState);
        Observer firewall = new Firewall(antivirusState);

        antivirusState.pause();
    }
}
```

However, this solution tightly-couples the AntivirusState with the components. To loosen this coupling, we can pass an interface to the components:

```java
interface Subject {
    public void registerForNotification(Observer observer);
}

class AntivirusState implements Subject {
    List<Observer> observers = new ArrayList<Observer>();

    public void registerForNotification(Observer observer) {
        observers.add(observer);
    }

    public void pause() {
        fileProtection.notifyPause();
        internetMonitor.notifyPause();
    }
}

class Firewall implements Observer {
    public Firewall(Subject subject) {
        subject.registerForNotification(this);
    }

    public void notifyPause() {
        //pause current task
    }
}

public class Observer_5 {
    public static void main(String[] args) {
        AntivirusState antivirusState = new AntivirusState();

        Observer fileProtection = new RealTimeFileProtection(antivirusState);
        Observer internetMonitor = new RealTimeInternetMonitor(antivirusState);
        Observer firewall = new Firewall(antivirusState);

        antivirusState.pause();
    }
}
```

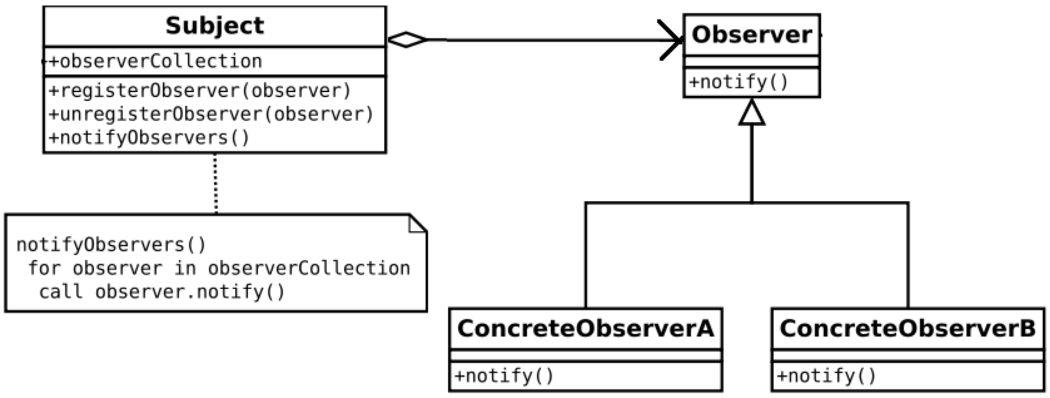


Figure: Observer Pattern (from Wikipedia)

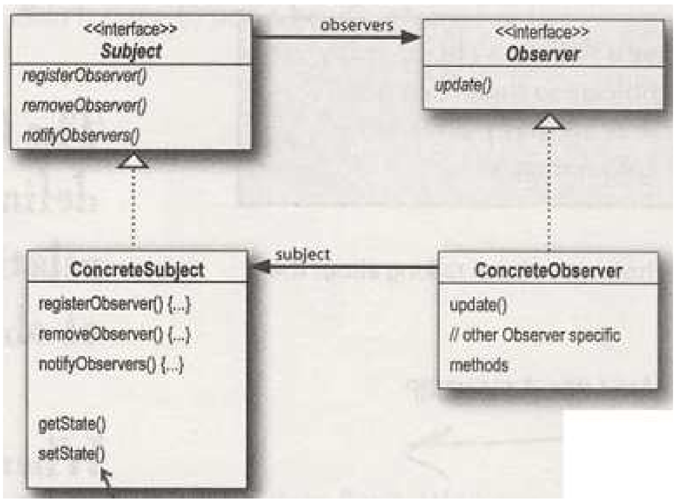


Figure: Observer Pattern (from Head First Design Patterns)

Definition of Observer Pattern

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

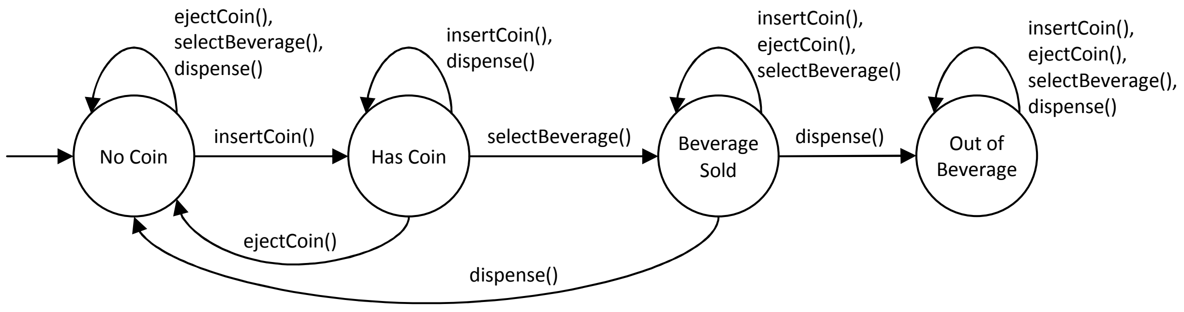
Uses of Observer Pattern

Distributed event notification system.

State Pattern

Example Problem Scenario

Suppose, you have to write a program for a beverage vending machine. The working process of the machine can be depicted by the following state diagram:


ejectCoin(), selectBeverage(), dispense()
insertCoin(), dispense()
insertCoin(), ejectCoin(), selectBeverage()
insertCoin(), ejectCoin(), selectBeverage(), dispense()
No Coin
insertCoin()
Has Coin
selectBeverage()
Beverage Sold
dispense()
Out of Beverage
ejectCoin()
dispense()

The program can be easily written as follows:

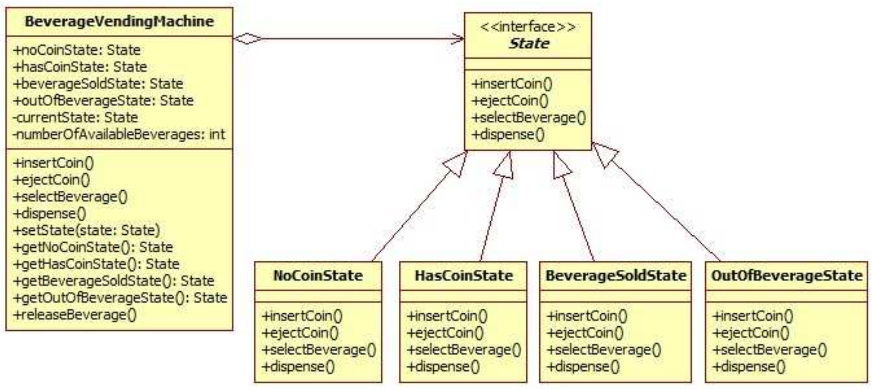
```
public class BeverageVendingMachine {
    final static int NO_COIN = 0;
    final static int HAS_COIN = 0;
    final static int BEVERAGE_SOLD = 0;
    final static int OUT_OF_BEVERAGE = 0;

    private int currentState = NO_COIN;
    private int numberOfAvailableBeverages;

    BeverageVendingMachine(int initialNumberOfBeverages) {
        numberOfAvailableBeverages = initialNumberOfBeverages;
        if (numberOfAvailableBeverages == 0) {
            currentState = OUT_OF_BEVERAGE;
        }
    }

    public void insertCoin() {
        if (currentState == NO_COIN) {
            currentState = HAS_COIN;
            System.out.println("You've inserted coin.");
        } else if (currentState == HAS_COIN) {
            System.out.println("You've already inserted a coin.");
        } else if (currentState == BEVERAGE_SOLD) {
            System.out.println("Please wait, we're giving you a beverage.");
        } else if (currentState == OUT_OF_BEVERAGE) {
            System.out.println("You can't insert a coin. All the beverages are sold out.");
        }
    }

    public void ejectCoin() {
        if (currentState == NO_COIN) {
            System.out.println("You haven't inserted any coin.");
        } else if (currentState == HAS_COIN) {
            currentState = NO_COIN;
            System.out.println("Coin returned.");
        } else if (currentState == BEVERAGE_SOLD) {
            System.out.println("Sorry, you've already selected a beverage.");
        } else if (currentState == OUT_OF_BEVERAGE) {
            System.out.println("You can't eject, you haven't inserted a coin yet.");
        }
    }

    public void selectBeverage() {
        if (currentState == NO_COIN) {
            System.out.println("You haven't inserted any coin.");
        } else if (currentState == HAS_COIN) {
```

```
      currentState = BEVERAGE_SOLD;
      System.out.println("You've selected a beverage.");
      dispense();
    } else if (currentState == BEVERAGE_SOLD) {
      System.out.println("Selecting twice doesn't get you another beverage!");
    } else if (currentState == OUT_OF_BEVERAGE) {
      System.out.println("Please insert a coin first.");
    }
  }

  public void dispense() {
    if (currentState == NO_COIN) {
      System.out.println("You need to pay first.");
    } else if (currentState == HAS_COIN) {
      System.out.println("Please select a beverage first.");
    } else if (currentState == BEVERAGE_SOLD) {
      System.out.println("Beverage comes rolling out the slot!");
      numberOfAvailableBeverages--;
      if (numberOfAvailableBeverages == 0) {
        currentState = OUT_OF_BEVERAGE;
      } else {
        currentState = NO_COIN;
      }
    } else if (currentState == OUT_OF_BEVERAGE) {
      System.out.println("No beverage dispensed.");
    }
  }
}
```

Now, suppose you'd like to add a lottery system to the machine. After the customer gets his beverage, a lottery would take place and the customer might win a free beverage. So, you'd have to add a new state WINNER to the system. After the dispense of the beverage, if the customer wins the lottery, the system would make transition towards the WINNER state; otherwise, the normal state transition would take place.

The problem is, adding a new state means adding a new *if* statement to all the methods. Is there any easier way to do it?

What we've done in the program above is handling all the states for each action. Let's think reversely – we'll handle all the actions for each state. The program is as follows:



```
interface State {
  public void insertCoin();
  public void ejectCoin();
  public void selectBeverage();
  public void dispense();
}
```

```
class HasCoinState implements State {
  BeverageVendingMachine machine;

  public HasCoinState(BeverageVendingMachine machine) {
    this.machine = machine;
  }

  public void insertCoin() {
    System.out.println("You've already inserted a coin.");
  }

  public void ejectCoin() {
    System.out.println("Coin returned.");
    machine.setState(machine.getNoCoinState());
  }
```

```java

    public void selectBeverage() {
        System.out.println("You've selected a beverage.");
        machine.setState(machine.getBeverageSoldState());
    }

    public void dispense() {
        System.out.println("Please select a beverage first.");
    }
}
```

```java
class BeverageSoldState implements State {
    BeverageVendingMachine machine;

    public BeverageSoldState(BeverageVendingMachine machine) {
        this.machine = machine;
    }

    public void insertCoin() {
        System.out.println("Please wait, we're giving you a beverage.");
    }

    public void ejectCoin() {
        System.out.println("Sorry, you've already selected a beverage.");
    }

    public void selectBeverage() {
        System.out.println("Selecting twice doesn't get you another beverage!");
    }

    public void dispense() {
        machine.releaseBeverage();
    }
}
```

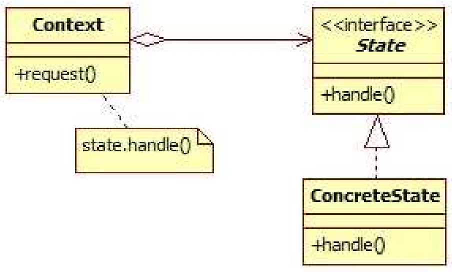
```java
class BeverageVendingMachine {
    State noCoinState;
    State hasCoinState;
    State beverageSoldState;
    State outOfBeverageState;
    private State currentState = noCoinState;
    private int numberOfAvailableBeverages;

    public BeverageVendingMachine(int initialNumberOfBeverages) {
        noCoinState = new NoCoinState(this);
        hasCoinState = new HasCoinState(this);
        beverageSoldState = new BeverageSoldState(this);
        outOfBeverageState = new OutOfBeverageState(this);

        numberOfAvailableBeverages = initialNumberOfBeverages;
        if (numberOfAvailableBeverages == 0) {
            currentState = outOfBeverageState;
        }
    }

    public void setState(State state) {
        currentState = state;
    }

    public State getNoCoinState() {
        return noCoinState;
    }

    public State getHasCoinState() {
        return hasCoinState;
    }

    public State getBeverageSoldState() {
        return beverageSoldState;
    }

    public State getOutOfBeverageState() {
        return outOfBeverageState;
    }

    /* Actions */
    public void insertCoin() {
        currentState.insertCoin();
    }

    public void ejectCoin() {
        currentState.ejectCoin();
    }

    public void selectBeverage() {
        currentState.selectBeverage();
    }

    public void dispense() {
        currentState.dispense();
    }

    public void releaseBeverage() {
        System.out.println("Beverage comes rolling out the slot!");
        numberOfAvailableBeverages--;
        if (numberOfAvailableBeverages == 0) {
            currentState = outOfBeverageState;
        } else {
            currentState = noCoinState;
        }
    }
}
```

Now, adding the WINNER state is easy:

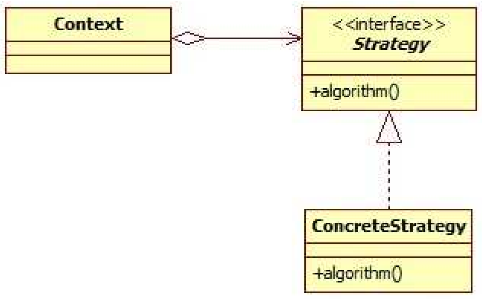
```
class BeverageVendingMachine {
    State noCoinState;
    State hasCoinState;
    State beverageSoldState;
    State outOfBeverageState;
    State winnerState;

    public State getWinnerState() {
        return winnerState;
    }

    //also initialize winnerState in constructor
}

class WinnerState implements State {
    public void dispense() {
        //do lottery and transition to other state
    }

    //constructors, instance variables, other action error messages the same
}
```

Definition of State Pattern

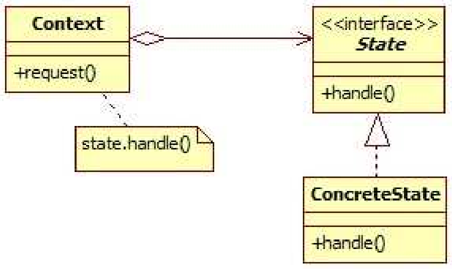
The state pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.



Difference between Strategy and State



Strategy Pattern



State Pattern

The class diagrams of both the patterns are the same, but they differ in their *intent*.

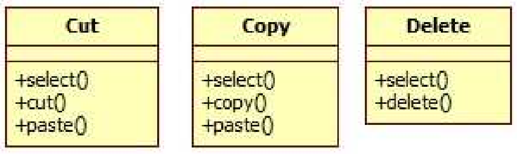
In general, think of the Strategy Pattern as a flexible alternative to subclassing; if you use inheritance to define the behavior of a class, then you're stuck with that behavior even if you need to change it. With Strategy, you can change the behavior by composing with a different object.

Think of the State Pattern as an alternative to putting lots of conditionals in your context; by encapsulating the behaviors within state objects, you can simply change the state object in context to change its behavior.

Command Pattern

Example Problem Scenario

Suppose, you are developing a Word Editor. You have Cut, Copy and Delete classes as follows:



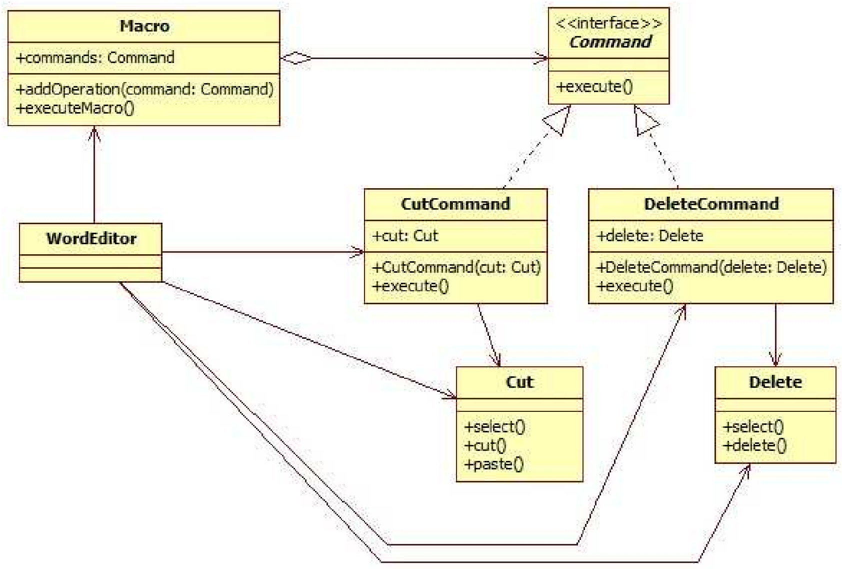
Now, you want to develop a Macro operation which would include all the above operations. But the method names are diversified. How would the Macro operation execute the methods in any of the operation classes?

As the Macro class doesn't know the exact method name of the operation class, we can do the following:

1. Define a common interface method execute() which the Macro class would call on each operation class.
2. As the operation classes don't implement the execute() method and we don't want to change the operation classes, we need to use some sort of adapter classes.

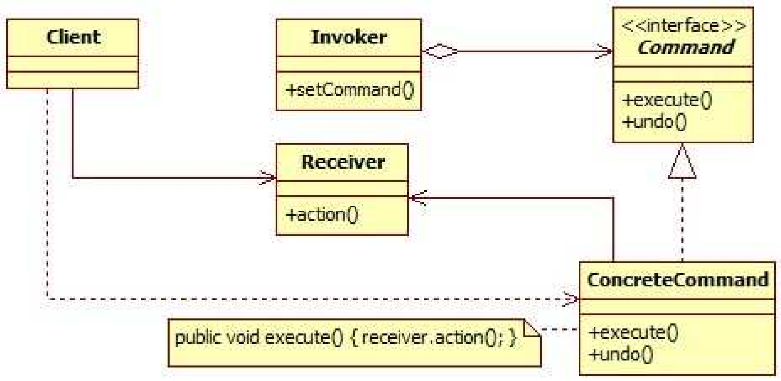
The solution can be as follows (we've omitted the Copy operation for brevity):

```java
class Cut {
    void select() {}
    void cut() {}
    void paste() {}
}

class Delete {
    void select() {}
    void delete() {}
}

interface Command {
    public void execute();
}

class CutCommand implements Command {
    private Cut cut;

    public CutCommand(Cut cut) {
        this.cut = cut;
    }

    public void execute() {
        cut.select();
        cut.cut();
        cut.paste();
    }
}

class DeleteCommand implements Command {
    private Delete delete;

    public DeleteCommand(Delete delete) {
        this.delete = delete;
    }

    public void execute() {
        delete.select();
        delete.delete();
    }
}

class Macro {
    List<Command> commands = new ArrayList<Command>();

    public void addOperation(Command command) {
        commands.add(command);
    }

    public void executeMacro() {
        for (Command command : commands) {
            command.execute();
        }
    }
}

class WordEditor {
    public static void main(String[] args) {
        Macro macro = new Macro();
        macro.addOperation(new CutCommand(new Cut()));
        macro.addOperation(new DeleteCommand(new Delete()));
        macro.executeMacro();
    }
}
```



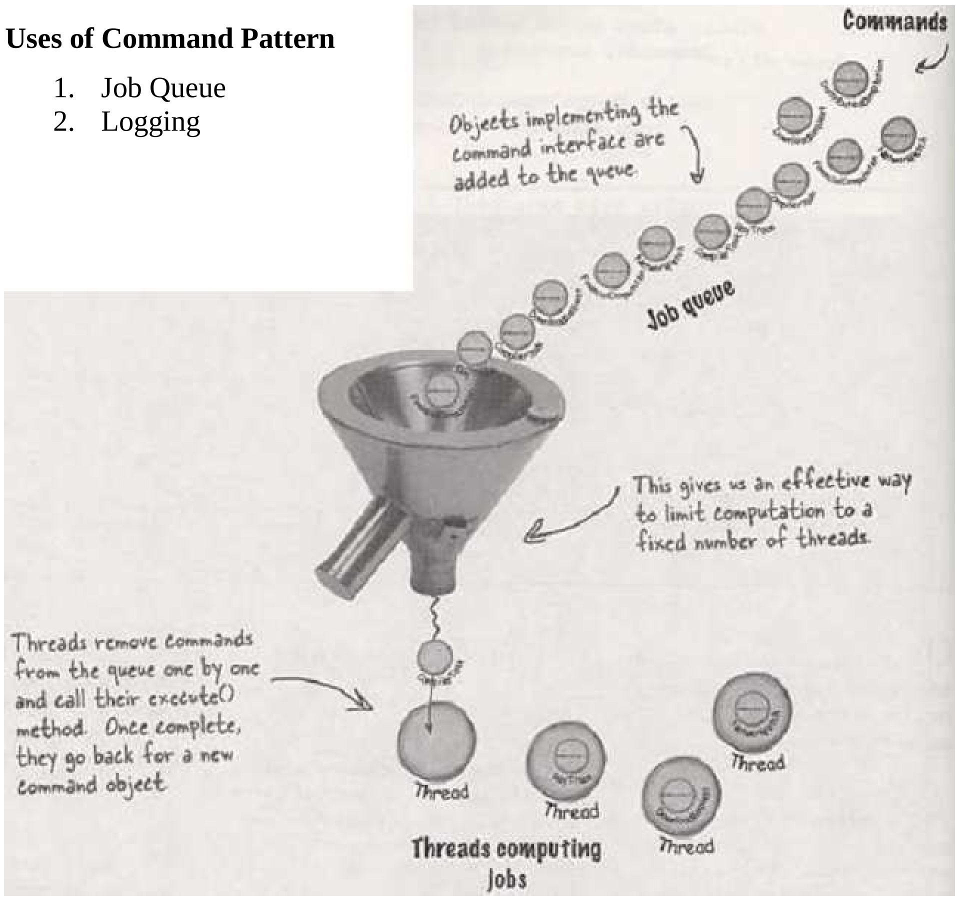
Definition of Command Pattern

The command pattern encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.

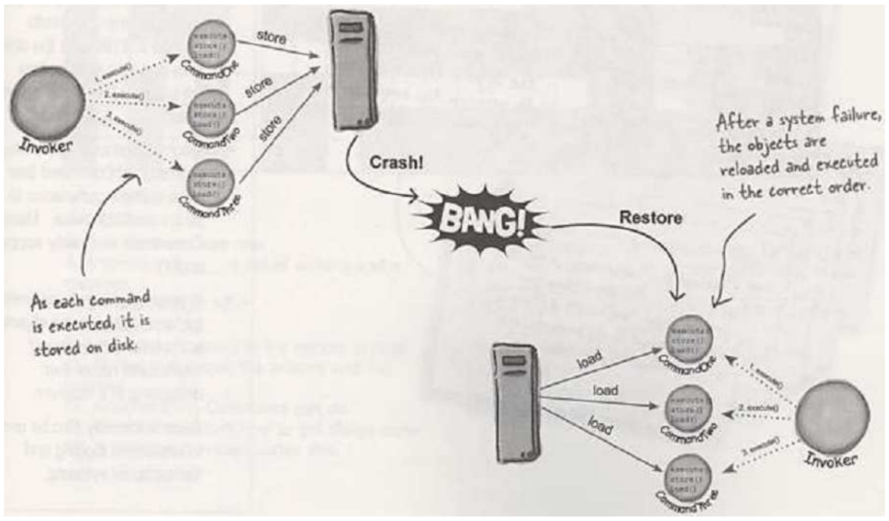


Uses of Command Pattern

1. Job Queue
2. Logging


Commands
Objects implementing the command interface are added to the queue.
Job queue
This gives us an effective way to limit computation to a fixed number of threads.
Threads remove commands from the queue one by one and call their execute() method. Once complete, they go back for a new command object.
Thread
Thread
Thread
Thread
Threads computing jobs

Job Queue


store
store
store
Invoker
Crash!
BANG!
Restore
After a system failure, the objects are reloaded and executed in the correct order.
As each command is executed, it is stored on disk.
load
load
load
Invoker

Logging