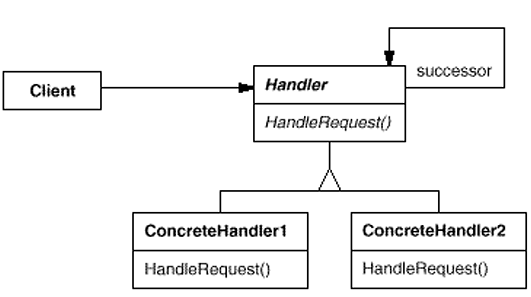
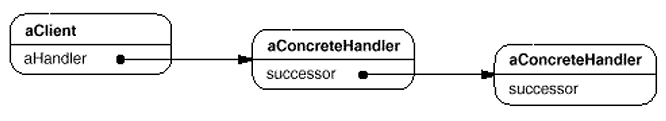
Chain of Responsibility

GOF : **Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.**

# **Structure Diagram**



Object Structure Diagam



**Java Code Example :** A loan has be to approved based upon the amount by various actors like LoanOfficer, BranchManager and Vice President of the Bank.

**LoanData.java**

**public** **class** LoanData {

**private** String purpose;

**private** **int** amount;

**public** LoanData(String purpose, **int** amount) {

**this**.purpose = purpose;

**this**.amount = amount;

}

**public** String getPurpose() {

**return** purpose;

}

**public** **int** getAmount() {

**return** amount;

}

}

**LoanApprover.java**

**public** **abstract** **class** LoanApprover {

**protected** LoanApprover successor;

**public** **void** setSuccessor(LoanApprover successor) {

**this**.successor = successor;

}

**public** **abstract** **void** approveLoan(LoanData data);

}

**LoanOfficer.java**

**public** **class** LoanOfficer **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

**if**( data.getAmount() < 1000 )

System.***out***.println("Loan approved by LoanOfficer ...");

**else**

successor.approveLoan(data);

}

}

**BranchManager.java**

**public** **class** BranchManager **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

**if**( data.getAmount() > 1000 && data.getAmount() < 10000 )

System.***out***.println("Loan approved by BranchManager ...");

**else**

successor.approveLoan(data);

}

}

**VicePresident.java**

**public** **class** VicePresident **extends** LoanApprover {

@Override

**public** **void** approveLoan(LoanData data) {

System.***out***.println("Loan approved by VicePresident ...");

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

LoanData data = **new** LoanData("Cultivation", 50000);

LoanApprover loanOfficer = **new** LoanOfficer();

LoanApprover manager = **new** BranchManager();

LoanApprover vp = **new** VicePresident();

//Form the chain

loanOfficer.setSuccessor(manager);

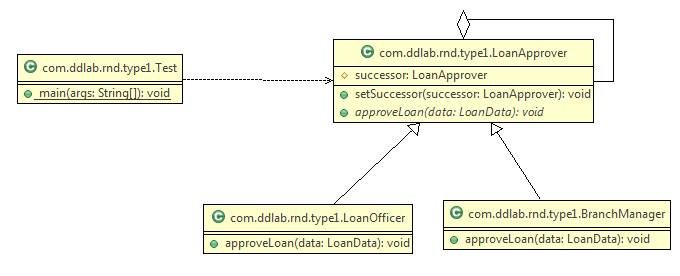
manager.setSuccessor(vp);

loanOfficer.approveLoan(data);

}

}

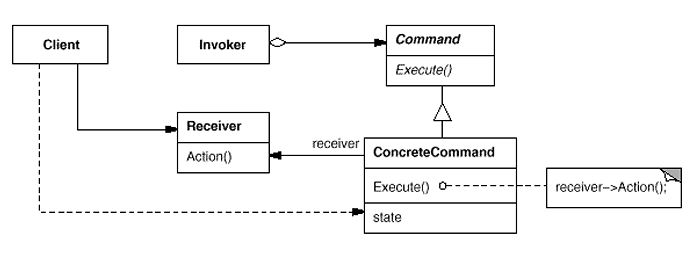
**UML Class Diagram** is given below.



Command Pattern

GOF: **Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations**.

# **Structure Diagram**

  
Java Code Example : To switch off and switch on light.

**Command.java**

**LightOnCommand.java**

//Concrete Command

**public** **class** LightOnConcreteCommand **implements** Command {

// reference to the light

LightReceiver light;

**public** LightOnConcreteCommand(LightReceiver light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOn();

}

}

//Command

**public** **interface** Command{

**public** **void** execute();

}

**LightReceiver.java**

//Receiver

**public** **class** LightReceiver {

**private** **boolean** on;

**public** **void** switchOn() {

on = **true**;

}

**public** **void** switchOff() {

on = **false**;

}

}

**LightOffConcreteCommand.java**

//Concrete Command

**public** **class** LightOffConcreteCommand **implements** Command {

// reference to the light

LightReceiver light;

**public** LightOffConcreteCommand(LightReceiver light) {

**this**.light = light;

}

**public** **void** execute() {

light.switchOff();

}

}

**RemoteController.java**

//Invoker

**public** **class** RemoteControlInvoker {

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command = command;

}

**public** **void** pressButton() {

command.execute();

}

}

**Client.java**

//Client

**public** **class** Client {

**public** **static** **void** main(String[] args) {

RemoteControlInvoker control = **new** RemoteControlInvoker();

LightReceiver light = **new** LightReceiver();

Command lightsOn = **new** LightOnConcreteCommand(light);

Command lightsOff = **new** LightOffConcreteCommand(light);

// switch on

control.setCommand(lightsOn);

control.pressButton();

// switch off

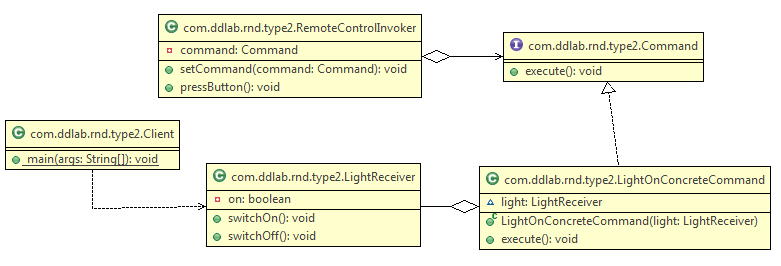
control.setCommand(lightsOff);

control.pressButton();

}

}

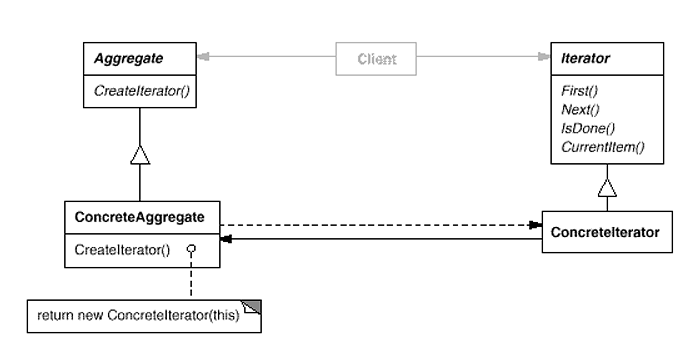
**UML Class Diagram** is given below.

****

Iterator Design Pattern

GOF : **Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation**.

# **Structure Diagram**

  
An example is given below.

import java.util.ArrayList;

import java.util.Iterator;

public class ArrayIterator implements Iterable {

**private Object[] elements;**

**private int size;**

**private int counter = 0;**

public ArrayIterator() {

**elements = new Object[20];**

}

**public void add( Object x ) {**

**elements[size++] = x;**

**}**

**@Override**

**public Iterator iterator() {**

**//reset the counter**

**//If you do not reset the counter, you will not be iterate once again**

**counter = 0;**

**return new MyIterator();**

**}**

**private class MyIterator implements Iterator {**

**@Override**

**public boolean hasNext() {**

**return counter < elements.length && elements[counter] != null ;**

**}**

**@Override**

**public Object next() {**

**return elements[counter++] ;**

**}**

@Override

public void remove() {

System.out.println("Don't want to delete item");

}

}

public static void main(String[] args) {

ArrayIterator arr = new ArrayIterator();

for( int i = 0 ; i < 10 ; i++ ) {

arr.add( new Integer(i));

}

Iterator itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next()); //0 1 2 3 4 5 6 7 8 9

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t" + itr.next()); //0 1 2 3 4 5 6 7 8 9

}

//In case of arraylist also, everytime, you get an iterator,

//the index is et to 0 so that it can be iterated.

ArrayList al = new ArrayList();

for( int i = 0 ; i < 10 ; i++ ) {

al.add( new Integer(i));

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next());

}

System.out.println("\n\n");

itr = arr.iterator();

while( itr.hasNext() ) {

System.out.print("\t"+itr.next());

}

}

}

The iterator pattern is a behavioral object design pattern. The iterator pattern allows for the traversal through the elements in a grouping of objects via a standardized interface. The code is given below.

public class Item {

String name;

float price;

public Item(String name, float price) {

this.name = name;

this.price = price;

}

public String toString() {

return name + ": $" + price;

}

}

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Menu {

List<Item> menuItems;

public Menu() {

menuItems = new ArrayList<Item>();

}

public void addItem(Item item) {

menuItems.add(item);

}

public Iterator<Item> iterator() {

return new MenuIterator();

}

class MenuIterator implements Iterator<Item> {

int currentIndex = 0;

@Override

public boolean hasNext() {

if (currentIndex >= menuItems.size()) {

return false;

} else {

return true;

}

}

@Override

public Item next() {

return menuItems.get(currentIndex++);

}

@Override

public void remove() {

menuItems.remove(--currentIndex);

}

}

}

import java.util.Iterator;

public class Demo {

public static void main(String[] args) {

Item i1 = new Item("spaghetti", 7.50f);

Item i2 = new Item("hamburger", 6.00f);

Item i3 = new Item("chicken sandwich", 6.50f);

Menu menu = new Menu();

menu.addItem(i1);

menu.addItem(i2);

menu.addItem(i3);

System.out.println("Displaying Menu:");

Iterator<Item> iterator = menu.iterator();

while (iterator.hasNext()) {

Item item = iterator.next();

System.out.println(item);

}

System.out.println("\nRemoving last item returned");

iterator.remove();

System.out.println("\nDisplaying Menu:");

iterator = menu.iterator();

while (iterator.hasNext()) {

Item item = iterator.next();

System.out.println(item);

}

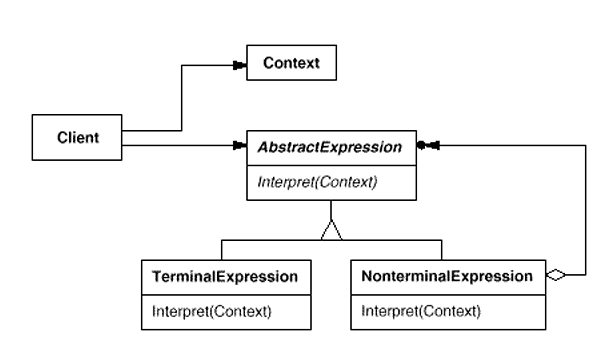
}

}

Interpreter Pattern

GOF : **Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.**

# **Structure Diagram**



Example : Evaluate an expression like “5 + 10”.

**Numbers.java**

**public** **class** Numbers **implements** Expression {

**private** **int** num;

**public** Numbers( **int** num ) {

**this**.num = num;

}

@Override

**public** **int** calc() {

**return** **this**.num;

}

}

The code is given below.

**Evaluator.java  
public** **class** Evaluator {

**Add.java**

**public** **class** Add **implements** Expression {

**private** Expression lhs;

**private** Expression rhs;

**public** Add(Expression lhs , Expression rhs) {

**this**.lhs = lhs;

**this**.rhs = rhs;

}

@Override

**public** **int** calc() {

**return** **this**.lhs.calc()+**this**.rhs.calc();

}

}

**Expression.java**

**public** **interface** Expression {

**int** calc();

}

**public** **int** evaluate( String statement ) {

String[] exps = statement.split(" ");

**int** leftOperand = Integer.*parseInt*( exps[0]);

**int** rightOperand = Integer.*parseInt*( exps[2]);

String operation = exps[1];

**return** **new** Add( **new** Numbers(leftOperand) , **new** Numbers(rightOperand) ).calc();

}

}

**Test.java**

**public** **class** Test {

**public** **static** **void** main(String[] args) {

String statement = "5 + 10";

Evaluator evalutator = **new** Evaluator();

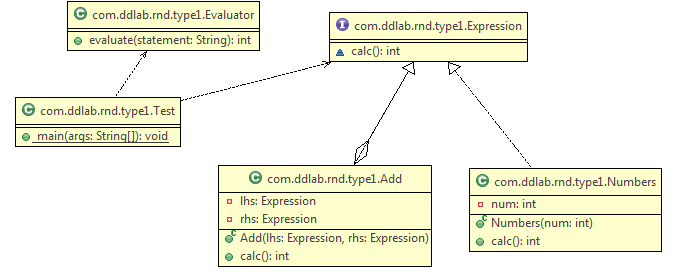
**int** result = evalutator.evaluate(statement);

System.***out***.println("Result :::"+result);

}

}

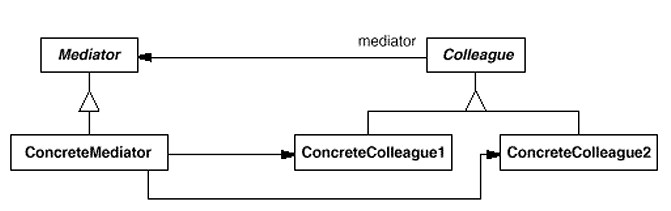
UML diagram is given below.

****

Mediator Pattern

GOF: **Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently**.

# **Structure Diagram**

  
**Java Code**

**Mediator.java**

//Mediator interface  
public interface Mediator {  
 public void send(String message, Colleague colleague);  
}

**Colleague.java**  
//Colleage interface  
public abstract class Colleague {  
 private Mediator mediator;  
  
 public Colleague(Mediator m) {  
 mediator = m;  
 }  
  
 //send a message via the mediator  
 public void send(String message) {  
 mediator.send(message, this);  
 }  
  
 //get access to the mediator  
 public Mediator getMediator() {  
 return mediator;  
 }  
 public abstract void receive(String message);  
}

**ConcreteColleague.java**public class ConcreteColleague extends Colleague {  
 public ConcreteColleague(Mediator mediator) {  
 super(mediator);  
 }  
  
 public void receive(String message) {  
 System.*out*.println("Colleague Received: " + message);  
 }  
}

**MobileColleague**

public class MobileColleague extends Colleague {  
 public MobileColleague(Mediator mediator) {  
 super(mediator);  
 }  
  
 public void receive(String message) {  
 System.*out*.println("Mobile Received: " + message);  
 }  
}

**ApplicationMediator.java**

import java.util.ArrayList;  
public class ApplicationMediator implements Mediator {  
  
 private ArrayList<Colleague> colleagues;  
  
 public ApplicationMediator() {  
 colleagues = new ArrayList<Colleague>();  
 }  
  
 public void addColleague(Colleague colleague) {  
 colleagues.add(colleague);  
 }  
  
 public void send(String message, Colleague originator) {  
 //let all other screens know that this screen has changed  
 for (Colleague colleague : colleagues) {  
 //don't tell ourselves  
 if (colleague != originator) {  
 colleague.receive(message);  
 }  
 }  
 }  
}

**Client.java**

public class Client {  
 public static void main(String[] args) {  
 ApplicationMediator mediator = new ApplicationMediator();  
  
 Colleague desktop = new ConcreteColleague(mediator);  
  
 Colleague mobile = new MobileColleague(mediator);  
  
 mediator.addColleague(desktop);  
 mediator.addColleague(mobile);  
  
  
 desktop.send("Hello World");  
 mobile.send("Hello");  
 }  
}

Create an “intermediary” that decouples “senders” from “receivers”

Producers are coupled only to the Mediator , Consumers are coupled only to the Mediator

The Mediator arbitrates the storing and retrieving of messages

http://en.wikipedia.org/wiki/Mediator\_pattern

The mediator pattern defines an object that encapsulates how a set of objects interact. This pattern is considered to be a behavioral pattern due to the way it can alter the program's running behavior. You can consider the example on Cosumer and Producer in multi threading.

public class Player1 extends Thread

{

private Object monitor = null;

private StatusHolder status = null;

public Player1( Object monitor , StatusHolder holder )

{

this.monitor = monitor;

this.status = holder;

}

public void run()

{

synchronized( monitor )

{

while( true )

{

try

{

System.out.println("Status for Player 1---->"+status.hasCompleted);

if( status.hasCompleted == false )

{

monitor.wait();

}

System.out.println("Player 1 is playing the game ...");

Thread.sleep(1000);

status.hasCompleted = false;

System.out.println("Player 1 has completed the turn and going to notify");

System.out.println("-----------------------END for Player 1-------------------------");

monitor.notify();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

}

}

public class Player2 extends Thread

{

private Object monitor = null;

private StatusHolder status = null;

public Player2( Object monitor , StatusHolder holder )

{

this.monitor = monitor;

this.status = holder;

}

public void run()

{

synchronized( monitor )

{

while( true )

{

try

{

System.out.println("Status for Player 2---->"+status.hasCompleted);

if( status.hasCompleted == true )

{

monitor.wait();

}

System.out.println("Player 2 is playing the game ...");

Thread.sleep(1000);

System.out.println("Player 2 has completed the turn and going to notify");

status.hasCompleted = true;

System.out.println("-----------------------END for Player 2-------------------------");

monitor.notify();

}

catch (Exception e)

{

e.printStackTrace();

}

}

}

}

}

public class StatusHolder //Mediator Design Pattern

{

public boolean hasCompleted = false;

}

public class TestGame {

public static void main(String[] args)

{

Object monitor = new Object();

StatusHolder holder = new StatusHolder();

new Player1(monitor, holder).start();

new Player2(monitor, holder).start();

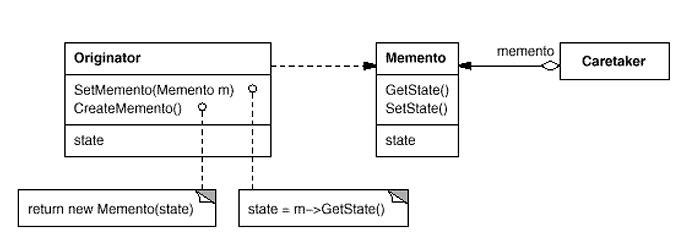
}

}

Memento Pattern

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

**Structure Diagram**

  
Java Code Example is given below.

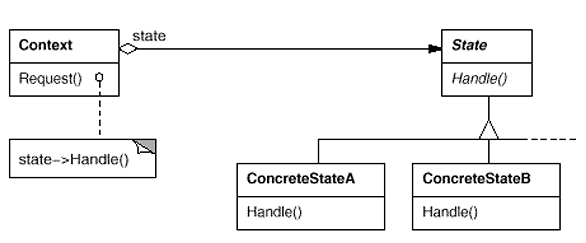
class Originator {  
 private String state;  
 // The class could also contain additional data that is not part of the  
 // state saved in the memento..  
  
 public void set(String state) {  
 System.*out*.println("Originator: Setting state to " + state);  
 this.state = state;  
 }  
  
 public Memento saveToMemento() {  
 System.*out*.println("Originator: Saving to Memento.");  
 return new Memento(this.state);  
 }  
  
 public void restoreFromMemento(Memento memento) {  
 this.state = memento.getSavedState();  
 System.*out*.println("Originator: State after restoring from Memento: " + state);  
 }  
  
 **public static class Memento {  
 private final String state;  
  
 public Memento(String stateToSave) {  
 state = stateToSave;  
 }  
  
 public String getSavedState() {  
 return state;  
 }  
 }**}

import java.util.ArrayList;  
import java.util.List;  
  
class Caretaker {  
 public static void main(String[] args) {  
 List<Originator.Memento> savedStates = new ArrayList<Originator.Memento>();  
   
 Originator originator = new Originator();  
 originator.set("State1");  
 originator.set("State2");  
 savedStates.add(originator.saveToMemento());  
 originator.set("State3");  
 // We can request multiple mementos, and choose which one to roll back to.  
 savedStates.add(originator.saveToMemento());  
 originator.set("State4");  
   
 originator.restoreFromMemento(savedStates.get(1));   
 }  
}

State Pattern

GOF : **Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.**

**Structure Diagram**

  
**Java Code Example**

**State.java**

public interface State {  
 public void doAction(Context context);  
}

**StartState.java**

public class StartState implements State {  
  
 public void doAction(Context context) {  
 System.*out*.println("Player is in start state");  
 context.setState(this);  
 }  
  
 public String toString() {  
 return "Start State";  
 }  
}

**StopState.java**

public class StopState implements State {  
 public void doAction(Context context) {  
 System.*out*.println("Player is in stop state");  
 context.setState(this);  
 }  
  
 public String toString() {  
 return "Stop State";  
 }  
}

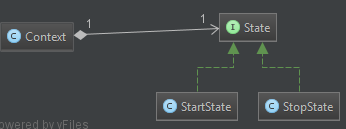
**Context.java**

public class Context {  
 private State state;  
  
 public Context() {  
 state = null;  
 }  
  
 public void setState(State state) {  
 this.state = state;  
 }  
  
 public State getState() {  
 return state;  
 }  
}

**StatePatternDemo.java**

public class StatePatternDemo {  
 public static void main(String[] args) {  
 Context context = new Context();  
 StartState startState = new StartState();  
 startState.doAction(context);  
  
 System.*out*.println(context.getState().toString());  
  
 StopState stopState = new StopState();  
 stopState.doAction(context);  
  
 System.*out*.println(context.getState().toString());  
 }  
}

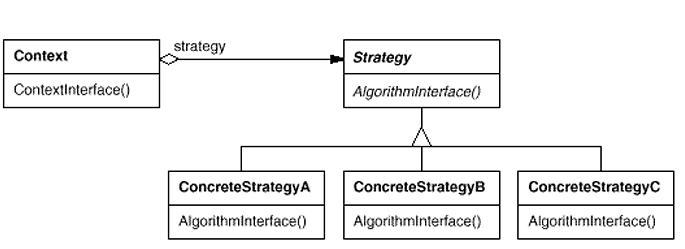
UML Diagram for the above is given below.



Strategy Pattern

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

# **Structure Diagram**

  
http://en.wikipedia.org/wiki/Strategy\_pattern

In computer programming, the strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime. Formally speaking, 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.

The example is given below.

// The classes that implement a concrete strategy should implement this.

// The Context class uses this to call the concrete strategy.

interface IStrategy {

int execute(int a, int b);

}

// Implements the algorithm using the strategy interface

class ConcreteStrategyAdd implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategyAdd's execute()");

return a + b; // Do an addition with a and b

}

}

class ConcreteStrategySubtract implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategySubtract's execute()");

return a - b; // Do a subtraction with a and b

}

}

class ConcreteStrategyMultiply implements IStrategy {

public int execute(int a, int b) {

System.out.println("Called ConcreteStrategyMultiply's execute()");

return a \* b; // Do a multiplication with a and b

}

}

// Configured with a ConcreteStrategy object and maintains a reference to a Strategy object

class Context {

private IStrategy strategy;

// Constructor

public Context(IStrategy strategy) {

this.strategy = strategy;

}

public int executeStrategy(int a, int b) {

return strategy.execute(a, b);

}

}

// Test application

class StrategyExample {

public static void main(String[] args) {

Context context;

// Three contexts following different strategies

context = new Context(new ConcreteStrategyAdd());

int resultA = context.executeStrategy(3,4);

context = new Context(new ConcreteStrategySubtract());

int resultB = context.executeStrategy(3,4);

context = new Context(new ConcreteStrategyMultiply());

int resultC = context.executeStrategy(3,4);

System.out.println("Result A : " + resultA );

System.out.println("Result B : " + resultB );

System.out.println("Result C : " + resultC );

}

}

Another example is given below.

**CompressionStrategy.java**  
import java.io.File;  
import java.util.ArrayList;  
  
//Strategy Interface  
public interface CompressionStrategy {  
 public void compressFiles(ArrayList<File> files);  
}

**CompressionContext.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class CompressionContext {  
 private CompressionStrategy strategy;  
  
 public CompressionContext(CompressionStrategy strategy) {  
 this.strategy = strategy;  
 }  
  
 //use the strategy  
 public void createArchive(ArrayList<File> files) {  
 strategy.compressFiles(files);  
 }  
}

**RarCompressionStrategy.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class RarCompressionStrategy implements CompressionStrategy {  
 public void compressFiles(ArrayList<File> files) {  
 //using RAR approach  
 System.*out*.println("Compressing files using ZIP strategy ...");  
 }  
}

**ZipCompressionStrategy.java**

import java.io.File;  
import java.util.ArrayList;  
  
public class ZipCompressionStrategy implements CompressionStrategy {  
 public void compressFiles(ArrayList<File> files) {  
 //using ZIP approach  
 System.*out*.println("Compressing files using ZIP strategy ...");  
 }  
}

**Client.java**

import java.io.File;  
import java.util.ArrayList;  
import java.util.List;  
  
public class Client {  
 public static void main(String[] args) {  
 CompressionContext ctx = new CompressionContext(new ZipCompressionStrategy());  
 //we could assume context is already set by preferences  
  
 //get a list of files...  
 ArrayList<File> files = new ArrayList<File>();  
 File file1 = new File("a.doc");  
 File file2 = new File("b.doc");  
 files.add(file1);  
 files.add(file2);  
  
 ctx.createArchive(files);  
 }  
}

UML Diagram

