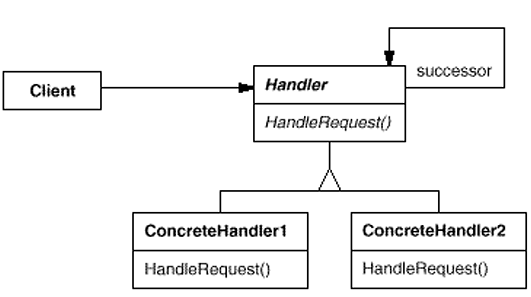
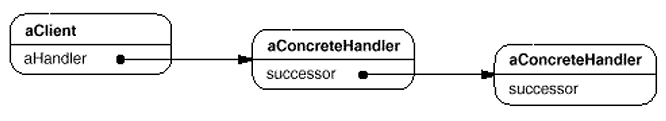
Chain of Responsibility

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);

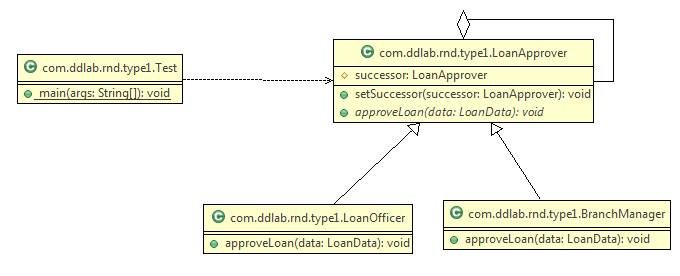
manager.setSuccessor(vp);

loanOfficer.approveLoan(data);

}

}

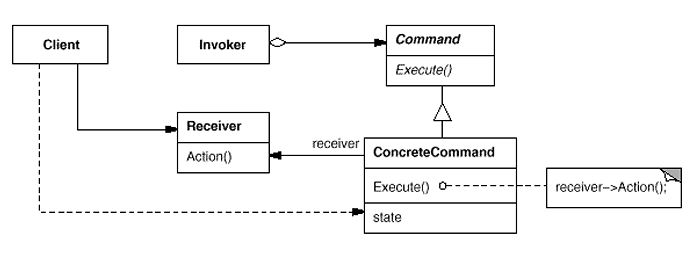
**UML Class Diagram** is given below.



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

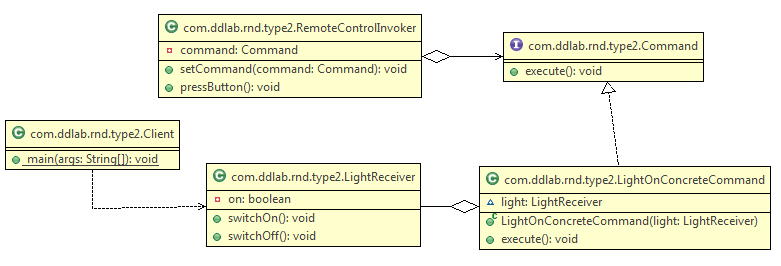
control.setCommand(lightsOff);

control.pressButton();

}

}

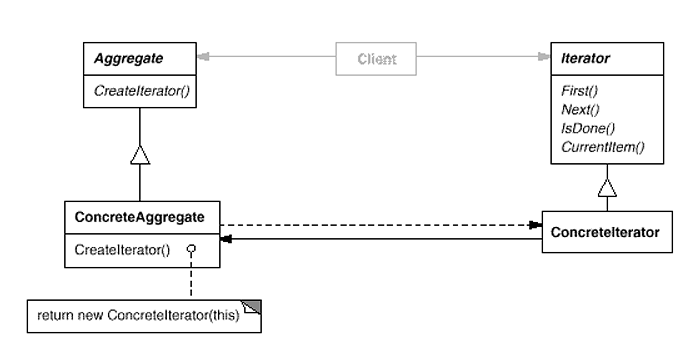
**UML Class Diagram** is given below.

****

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);

}

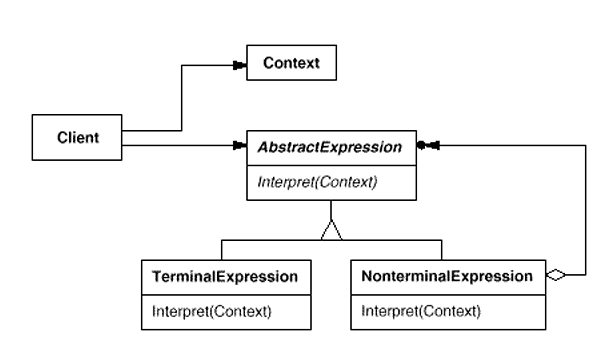
}

}

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();

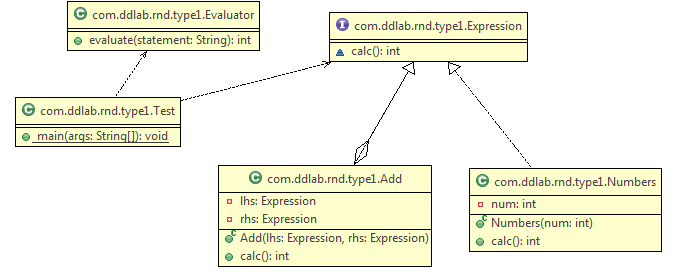
**int** result = evalutator.evaluate(statement);

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

}

}

UML diagram is given below.

****