1. **Adapter Pattern**

**Example 1.**

The famous adapter classes in Java API are WindowAdapter,ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter and MouseMotionAdapter. As you know, WindowListner interface has seven methods. Whenever your class implements such interface, you have to implements all of the seven methods. WindowAdapter class implements WindowListener interface and make seven empty implementation. When you class subclass WindowAdapter class, you may choose the method you want without restrictions. The following give such an example.

public interface Windowlistener {

 public void windowClosed(WindowEvent e);

 public void windowOpened(WindowEvent e);

 public void windowIconified(WindowEvent e);

 public void windowDeiconified(WindowEvent e);

 public void windowActivated(WindowEvent e);

 public void windowDeactivated(WindowEvent e);

 public void windowClosing(WindowEvent e);

}

public class WindowAdapter implements WindowListner{

 public void windowClosed(WindowEvent e){}

 public void windowOpened(WindowEvent e){}

 public void windowIconified(WindowEvent e){}

 public void windowDeiconified(WindowEvent e){}

 public void windowActivated(WindowEvent e){}

 public void windowDeactivated(WindowEvent e){}

 public void windowClosing(WindowEvent e){}

}

Here is a test program

import javax.swing.\*;

import java.awt.event.\*;

class Test extends JFrame {

 public Test () {

 setSize(200,200);

 setVisible(true);

 addWindowListener(new Closer());

 }

 public static void main(String[] args) {

 new Test();

 }

 class Closer extends WindowAdapter {

 public void windowClosing(WindowEvent e) {

 System.exit(0);

 }

 }

}

**Example 2:**  To reuse classes and make new class compatible with existing ones. For example, A clean system is already designed, you want to add more job in, the Extra interface uses adapter pattern to plug in the existing system.

interface Clean {

 public void makeClean();

}

class Office implements Clean{

 public void makeClean() {

 System.out.println("Clean Office");

 }

}

class Workshop implements Clean{

 public void makeClean() {

 System.out.println("Clean Workshop");

 }

}

interface Extra extends Clean{

 public void takeCare();

}

class Facility implements Extra{

 public void makeClean() {

 System.out.println("Clean Facility");

 }

 public void takeCare() {

 System.out.println("Care has been taken");

 }

}

**In order to reuse Workshop and Office classes, we create an adapter interface Extra and add new job takeCare in the system.**

class Test {

 static void Jobs (Extra job) {

 if (job instanceof Clean)

 ((Clean)job).makeClean();

 if (job instanceof Extra)

 ((Extra)job).takeCare();

 }

 public static void main(String[] args) {

 Extra e = new Facility();

 Jobs(e);

 Clean c1 = new Office();

 Clean c2 = new Workshop();

 c1.makeClean();

 c2.makeClean();

 e.makeClean();

 }

}

|  C:\> java TestClean FacilityCare has been takenClean OfficeClean WorkshopClean Facility |
| --- |

**Example 3**

By composition, we can achieve adapter pattern. It is also called wrapper. For example, a Data class has already been designed and well tested. You want to adapt such class to your system. You may declare it as a variable and wrapper or embed it into your class.

//well-tested class

class Data {

 public void add(Info){}

 public void delete(Info) {}

 public void modify(Info){}

 //...

}

//Use Data class in your own class

class AdaptData {

 Data data;

 public void add(Info i) {

 data.add(i);

 //more job

 }

 public void delete(Info i) {

 data.delete(i);

 //more job

 }

 public void modify(Info i) {

 data.modify(i);

 //more job

 }

 //more stuff here

 //...

}

1. **Bridge Pattern**

**Example 1**

If you have a question database, you may want to develop a program to display it based on the user selection. The following is a simple example to show how to use a Bridge pattern to decouple the relationship among the objects.

import java.util.\*;

//abstraction

interface Question {

 public void nextQuestion();

 public void priorQuestion();

 public void newQuestion(String q);

 public void deleteQuestion(String q);

 public void displayQuestion();

 public void displayAllQuestions();

}

//implementation

class QuestionManager {

 protected Question questDB; //instantiate it later

 public String catalog;

 public QuestionManager(String catalog) {

 this.catalog = catalog;

 }

 public void next() {

 questDB.nextQuestion();

 }

 public void prior() {

 questDB.priorQuestion();

 }

 public void newOne(String quest) {

 questDB.newQuestion(quest);

 }

 public void delete(String quest) {

 questDB.deleteQuestion(quest);

 }

 public void display() {

 questDB.displayQuestion();

 }

 public void displayAll() {

 System.out.println("Question Catalog: " + catalog);

 questDB.displayAllQuestions();

 }

}

//further implementation

class QuestionFormat extends QuestionManager {

 public QuestionFormat(String catalog){

 super(catalog);

 }

 public void displayAll() {

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

 super.displayAll();

 System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~");

 }

}

//decoupled implementation

class JavaQuestions implements Question {

 private List<String> questions = new ArrayList<String>();

 private int current = 0;

 public JavaQuestions() {

 //load from a database and fill in the container

 questions.add("What is Java? ");

 questions.add("What is an interface? ");

 questions.add("What is cross-platform? ");

 questions.add("What is UFT-8? ");

 questions.add("What is abstract? ");

 questions.add("What is Thread? ");

 questions.add("What is multi-threading? ");

 }

 public void nextQuestion() {

 if( current <= questions.size() - 1 )

 current++;

 }

 public void priorQuestion() {

 if( current > 0 )

 current--;

 }

 public void newQuestion(String quest) {

 questions.add(quest);

 }

 public void deleteQuestion(String quest) {

 questions.remove(quest);

 }

 public void displayQuestion() {

 System.out.println( questions.get(current) );

 }

 public void displayAllQuestions() {

 for (String quest : questions) {

 System.out.println(quest);

 }

 }

}

class TestBridge {

 public static void main(String[] args) {

 ***QuestionFormat questions = new QuestionFormat("Java Language");***

 ***questions.questDB = new JavaQuestions();***

 //questDB is a reference variable of type Question which is

 //found in QuestionManager class

 //can be hooked up with other question class

 //questions.questDB = new CsharpQuestions();

 //questions.questDB = new CplusplusQuestions();

 questions.display();

 questions.next();

 questions.newOne("What is object? ");

 questions.newOne("What is reference type?");

 questions.displayAll();

 }

}

//need jdk1.5 to compile

| C:\> javac TestBridge.javaC:\> java TestBridgeWhat is Java?~~~~~~~~~~~~~~~~~~~~~~~~Question Catalog: Java LanguageWhat is Java?What is an interface?What is cross-platform?What is UFT-8?What is abstract?What is Thread?What is multi-threading?What is object?What is reference type?~~~~~~~~~~~~~~~~~~~~~~~~C:\> |
| --- |

 C:\ Command Prompt

**Note that the JavaQuestion class can be launched independently and work as its own system. Here we just show you how to use Bridge pattern to decouple the interface from its implementation.**

**Example 2**

public class BridgeExample {

public static void main(String[] args) {

Shape[] shapes = new Shape[2];

shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());

shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

for (Shape shape : shapes) {

shape.resizeByPercentage(2.5);

shape.draw();

}

}

}

/\*\* "Abstraction" \*/

public interface Shape {

public void draw();

public void resizeByPercentage(double pct);

}

/\*\* "Refined Abstraction" \*/

public class CircleShape implements Shape {

private double x, y, radius;

private DrawingAPI drawingAPI;

public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {

this.x = x;

this.y = y;

this.radius = radius;

this.drawingAPI = drawingAPI;

}

// Implementation specific

public void draw() {

drawingAPI.drawCircle(x, y, radius);

}

// Abstraction specific

public void resizeByPercentage(double pct) {

radius \*= pct;

}

}

/\*\* "Implementor" \*/

public interface DrawingAPI {

public void drawCircle(double x, double y, double radius);

}

/\*\* "ConcreteImplementor" 1/2 \*/

public class DrawingAPI1 implements DrawingAPI {

public void drawCircle(double x, double y, double radius) {

System.out.printf("API1.circle at %f:%f radius %f\n", x, y,

radius);

}

}

/\*\* "ConcreteImplementor" 2/2 \*/

public class DrawingAPI2 implements DrawingAPI {

public void drawCircle(double x, double y, double radius) {

System.out.printf("API2.circle at %f:%f radius %f\n", x, y,

radius);

}

}

When BridgeExample is executed the result is:

c:>API1.circle at 1,000000:2,000000 radius 7,5000000

c:>API2.circle at 5,000000:7,000000 radius 27,5000000



1. **Composite Pattern**

**Example 1**

For example, General Manager may have several employees and some of employees are Managers which have several employees. To illustrate such issue, we design a simple Manager class.

class Employee {

 String name;

 double salary;

 Employee(String n, double s){

 name = n;

 salary = s;

 }

 String getName() {

 return name;

 }

 double getSalary() {

 return salary;

 }

 public String toString() {

 return "Employee " + name;

 }

}

class Manager {

 Manager mgr;

 Employee[] ely;

 String dept;

 Manager(Manager mgr,Employee[] e, String d ) {

 this(e, d);

 this.mgr = mgr;

 }

 Manager(Employee[] e, String d) {

 ely = e;

 dept =d;

 }

 String getDept() {

 return dept;

 }

 Manager getManager() {

 return mgr;

 }

 Employee[] getEmployee() {

 return ely;

 }

 public String toString() {

 return dept + " manager";

 }

}

class Test {

 public static void main(String[] args) {

 Employee[] e1 = {new Employee("Aaron", 50),

 new Employee("Betty", 60)};

 Manager m1 = new Manager(e1, "Accounting");

 Employee[] e2 = {new Employee("Cathy", 70),

 new Employee("Dan", 80),

 new Employee("Eliz", 90)};

 Manager m2 = new Manager(m1, e2, "Production");

 System.out.println(m2);

 Employee[] emp = m2.getEmployee();

 if (emp != null)

 for (int k = 0; k < emp.length; k++)

 System.out.println(" "+emp[k]+" Salary: $"+ emp[k].getSalary());

 Manager m = m2.getManager();

 System.out.println(" " + m);

 if (m!= null) {

 Employee[] emps = m.getEmployee();

 if (emps != null)

 for (int k = 0; k < emps.length; k++)

 System.out.println(" " + emps[k]+" Salary: $"+ emps[k].getSalary());

 }

 }

}

| C:\> java TestProduction managerEmployee Cathy Salary: $70.0Employee Dan Salary: $80.0Employee Eliz Salary: $90.0Accounting managerEmployee Aaron Salary: $50.0Employee Betty Salary: $60.0 |
| --- |

 C: Command Prompt

**Example 2**

The example demonstrates how to use the Composite pattern to calculate the time required to complete a project or some part of a project. The example has four principal parts:

* Deliverable – A class that represents an end product of a completed Task.
* Project – The class used as the root of the composite, representing the entire project.
* ProjectItem – This interface describes functionality common to all items that can be part of a project. The getTimeRequired method is defined in this interface.
* Task – A class that represents a collection of actions to perform. The task has a collection of ProjectItem objects.

**// ProjectItem.java**

import java.io.Serializable;

public interface ProjectItem extends Serializable{

public double getTimeRequired();

}

**// Deliverable.java**

import java.io.Serializable;

public interface **Deliverable** extends Serializable{

public double getTimeRequired();

}

**// Project.java**

import java.util.ArrayList;

import java.util.Iterator;

public class Project implements ProjectItem{

private String name;

private String description;

private ArrayList projectItems = new ArrayList();

public Project(){ }

public Project(String newName, String newDescription){

name = newName;

description = newDescription;

}

public String getName(){ return name; }

public String getDescription(){ return description; }

public ArrayList getProjectItems(){ return projectItems; }

public double getTimeRequired(){

double totalTime = 0;

Iterator items = projectItems.iterator();

while(items.hasNext()){

ProjectItem item = (ProjectItem)items.next();

totalTime += item.getTimeRequired();

}

return totalTime;

}

public void setName(String newName){ name = newName; }

public void setDescription(String newDescription){

description = newDescription; }

public void addProjectItem(ProjectItem element){

if (!projectItems.contains(element)){

projectItems.add(element);

}

}

public void removeProjectItem(ProjectItem element){

projectItems.remove(element);

}

}

**// Task.java**

import java.util.ArrayList;

import java.util.Iterator;

public class Task implements ProjectItem{

private String name;

private String details;

private ArrayList projectItems = new ArrayList();

private Contact owner;

private double timeRequired;

public Task(){ }

public Task(String newName, String newDetails, Contact newOwner, double newTimeRequired){

name = newName;

details = newDetails;

owner = newOwner;

timeRequired = newTimeRequired;

}

public String getName(){ return name; }

public String getDetails(){ return details; }

public ArrayList getProjectItems(){ return projectItems; }

public Contact getOwner(){ return owner; }

public double getTimeRequired(){

double totalTime = timeRequired;

Iterator items = projectItems.iterator();

while(items.hasNext()){

ProjectItem item = (ProjectItem)items.next();

totalTime += item.getTimeRequired();

 }

return totalTime;

}

public void setName(String newName){ name = newName; }

public void setDetails(String newDetails){ details = newDetails; }

public void setOwner(Contact newOwner){ owner = newOwner; }

public void setTimeRequired(double newTimeRequired){ timeRequired = newTimeRequired; }

public void addProjectItem(ProjectItem element){

if (!projectItems.contains(element)){

projectItems.add(element);

}

}

public void removeProjectItem(ProjectItem element){

projectItems.remove(element);

}

}

… for the complete code Refer the Applied Java patterns book

**Example 3**

The next Java example illustrates the use of decorators using coffee making scenario. In this example, the scenario only includes cost and ingredients.

// The Coffee Interface defines the functionality of Coffee implemented by decorator

public interface Coffee {

 public double getCost(); // returns the cost of the coffee

 public String getIngredients(); // returns the ingredients of the coffee

}

// implementation of a simple coffee without any extra ingredients

public class SimpleCoffee implements Coffee {

 public double getCost() {

 return 1;

 }

 public String getIngredients() {

 return "Coffee";

 }

}

The following classes contain the decorators for all Coffee classes, including the decorator classes themselves..

// abstract decorator class - note that it implements Coffee interface

abstract public class CoffeeDecorator implements Coffee {

 protected final Coffee decoratedCoffee;

 protected String ingredientSeparator = ", ";

 public CoffeeDecorator(Coffee decoratedCoffee) {

 this.decoratedCoffee = decoratedCoffee;

 }

 public double getCost() { // implementing methods of the interface

 return decoratedCoffee.getCost();

 }

 public String getIngredients() {

 return decoratedCoffee.getIngredients();

 }

}

// Decorator Milk that mixes milk with coffee

// note it extends CoffeeDecorator

public class Milk extends CoffeeDecorator {

 public Milk(Coffee decoratedCoffee) {

 super(decoratedCoffee);

 }

 public double getCost() { // overriding methods defined in the abstract superclass

 return super.getCost() + 0.5;

 }

 public String getIngredients() {

 return super.getIngredients() + ingredientSeparator + "Milk";

 }

}

// Decorator Whip that mixes whip with coffee

// note it extends CoffeeDecorator

public class Whip extends CoffeeDecorator {

 public Whip(Coffee decoratedCoffee) {

 super(decoratedCoffee);

 }

 public double getCost() {

 return super.getCost() + 0.7;

 }

 public String getIngredients() {

 return super.getIngredients() + ingredientSeparator + "Whip";

 }

}

// Decorator Sprinkles that mixes sprinkles with coffee

// note it extends CoffeeDecorator

public class Sprinkles extends CoffeeDecorator {

 public Sprinkles(Coffee decoratedCoffee) {

 super(decoratedCoffee);

 }

 public double getCost() {

 return super.getCost() + 0.2;

 }

 public String getIngredients() {

 return super.getIngredients() + ingredientSeparator + "Sprinkles";

 }

}

Here's a test program that creates a Coffee instance which is fully decorated (i.e., with milk, whip, sprinkles), and calculate cost of coffee and prints its ingredients:

public class Main

{

 public static void main(String[] args)

 {

 Coffee c = new SimpleCoffee();

 System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

 c = new Milk(c);

 System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

 c = new Sprinkles(c);

 System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

 c = new Whip(c);

 System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

 // Note that you can also stack more than one decorator of the same type

 c = new Sprinkles(c);

 System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());

 }

}

The output of this program is given below:

Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
Cost: 2.4; Ingredients: Coffee, Milk, Sprinkles, Whip
Cost: 2.6; Ingredients: Coffee, Milk, Sprinkles, Whip, Sprinkles

1. **Decorator pattern**

A JScrollPane object can be used to decorate a JTextArea object or a JEditorPane object. A window can be decorated with different borders like BevelBorder, CompoundBorder, EtchedBorder TitledBorder etc. These border classes working as decorators are provided in Java API.

Decorator pattern can be used in a non-visual fashion. For example, BufferedInputStream, DataInputStream, and CheckedInputStream are decorating objects of FilterInputStream class. These decorators are standard Java API classes.

To illustrate a simple decorator pattern in non-visual manner, we design a class that prints a number. We create a decorator class that adds a text to the Number object to indicate that such number is a random number. Of course we can subclass the Number class to achieve the same goal. But the decorator pattern provides us an alternative way.

import java.util.Random;

class Number {

public void print() {

System.out.println(new Random().nextInt());

}

}

class Decorator {

public Decorator() {

//add a description to the number printed

System.out.print("Random number: ");

new Number().print();

}

}

class SubNumber extends Number{

public SubNumber() {

super();

System.out.print("Random number: ");

print();

}

}

class Test {

public static void main(String[] args) {

new Decorator();

new SubNumber();

}

}

java Test

Random number: 145265744

Random number: 145265755

1. **Façade Pattern**

JDBC design is a good example of Façade pattern. A database design is complicated. JDBC is used to connect the database and manipulate data without exposing details to the clients. Security of a system may be designed with Façade pattern. Clients' authorization to access information may be classified. General users may be allowed to access general information; special guests may be allowed to access more information; administrators and executives may be allowed to access the most important information. These subsystems may be generalized by one interface. The identified users may be directed to the related subsystems.

interface General {

 public void accessGeneral();

}

interface Special extends General {

 public void accessSpecial();

}

interface Private extends General {

 public void accessPrivate();

}

class GeneralInfo implements General {

 public void accessGeneral() {

 //...

 }

 //...

}

class SpecialInfo implements Special{

 public void accessSpecial() {

 //...

 }

 public void accessGeneral() {}

 //...

}

class PrivateInfo implements Private, Special {

 public void accessPrivate() {

 // ...

 }

 public void accessSpecial() {

 //...

 }

 public void accessGeneral() {

 // ...

 }

 //...

}

class Connection {

 //...

 if (user is unauthorized) throw new Exception();

 if (user is general) return new GeneralInfo();

 if (user is special) return new SpecialInfo();

 if (user is executive) return new PrivateInfo();

 //...

}

The above code example illustrates that the whole system is not exposed to the clients. It depends on the user classification.

Mr. SudHakar Chavali proposes a better design, similar to the above, but avoids repeated code. Look at code below.

interface General {

 public void accessGeneral();

}

interface Special extends General {

 public void accessSpecial();

}

interface Private extends General {

 public void accessPrivate();

}

class GeneralInfo implements General {

 public void accessGeneral() {

 //...

 }

 //...

}

class SpecialInfo extends GeneralInfo implements Special{

 public void accessSpecial() {

 //...

 }

}

class PrivateInfo extends SpecialInfo implements Private {

 public void accessPrivate() {

 // ...

 }

 //...

}

To avoid repeated code, SpecialInfo become subclass of GeneralInfo and PrivateInfo becomes subclass of SpecialInfo. When a person is exposed to special information, that person is allowed to access general information also. When a person is exposed to private information, that person is allowed to access general information and special information also.

1. **Flyweight Pattern**

Trying to use objects at very low levels of granularity is nice, but the overhead may be prohibitive.

public class Gazillion1 {

private static int num = 0;

private int row, col;

public Gazillion1( int maxPerRow ) {

row = num / maxPerRow;

col = num % maxPerRow;

num++;

}

void report() {

System.out.print( " " + row + col );

}

}

public class NoFlyweight {

public static final int ROWS = 6, COLS = 10;

public static void main( String[] args ) {

Gazillion1[][] matrix = new Gazillion1[ROWS][COLS];

for (int i=0; i < ROWS; i++)

for (int j=0; j < COLS; j++)

matrix[i][j] = new Gazillion1( COLS );

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

for (int j=0; j < COLS; j++)

matrix[i][j].report();

System.out.println();

 }

}

}

When NoFlyweight is executed we generate an excessive number of objects and the resulting output is:

c:> 00 01 02 03 04 05 06 07 08 09

c:>10 11 12 13 14 15 16 17 18 19

c:>20 21 22 23 24 25 26 27 28 29

c:>30 31 32 33 34 35 36 37 38 39

c:>40 41 42 43 44 45 46 47 48 49

c:>50 51 52 53 54 55 56 57 58 59

Introducing Flyweight means that we remove the non-shareable state from the class, and making the client supply it when needed. This places more responsibility on the client, but, considerably fewer instances of the Flyweight class are now created. Sharing of these instances is facilitated by introducing a Factory class that maintains a pool of existing Flyweights with a reusable state. In the following example, the "row" state is considered shareable (within each row anyways), and the "col" state has been externalized (it is supplied by the client when report() is called).

class Gazillion2 {

private int row;

public Gazillion2( int theRow ) {

row = theRow;

System.out.println( "Actual ctor: " + row );

}

void report( int theCol ) {

System.out.print( " " + row + theCol );

}

}

class FlyweightFactory {

private Gazillion2[] pool;

public FlyweightFactory( int maxRows ) {

pool = new Gazillion2[maxRows];

}

public Gazillion2 getFlyweight( int theRow ) {

if (pool[theRow] == null)

pool[theRow] = new Gazillion2( theRow );

return pool[theRow];

}

}

public class Flyweight {

public static final int ROWS = 6, COLS = 10;

public static void main( String[] args ) {

FlyweightFactory theFactory = new FlyweightFactory( ROWS );

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

for (int j=0; j < COLS; j++)

theFactory.getFlyweight( i ).report( j );

System.out.println();

}

}

}

When Flyweight is executed we generate, from a client perspective the same (excessive) amount of instances but in reality we only generate one instance per row and reuse that instance several times. This is indicated in the output below with the text “Actual instance no. x”:

c:>Actual instance no. 0

c:>00 01 02 03 04 05 06 07 08 09

c:>Actual instance no. 1

c:>10 11 12 13 14 15 16 17 18 19

c:>Actual instance no. 2

c:>20 21 22 23 24 25 26 27 28 29

c:>Actual instance no. 3

c:>30 31 32 33 34 35 36 37 38 39

c:>Actual instance no. 4

c:>40 41 42 43 44 45 46 47 48 49

c:>Actual instance no. 5

c:>50 51 52 53 54 55 56 57 58 59

1. **Proxy Pattern**

When loading a large image, you may create some light object to represent it until the image is loaded completely. Usually a proxy object has the same methods as the object it represents. Once the object is loaded, it passes on the actual object. For example,

abstract class Graphic {

 public abstract void load();

 public abstract void draw();

 ...

}

class Image extends Graphic{

 public void load() {

 ...

 }

 public void draw() {

 ...

 }

 ...

}

class ImgProxy extends Graphic {

 public void load() {

 if(image == null) {

 image = new Image(filename);

 }

 ...

 public void draw() {

 ...

 }

 ...

}