# DESIGN PATTERNS (RIDDHI DUTTA)

## CREATIONAL PATTERNS :

### SINGLETON PATTERN :

In object-oriented programming, a singleton class is a class that can have only one object (an instance of the class) at a time.

After first time, if we try to instantiate the Singleton class, the new variable also points to the first instance created. So whatever modifications we do to any variable inside the class through any instance, it affects the variable of the single instance created and is visible if we access that variable through any variable of that class type defined.

To design a singleton class:

1. Make constructor as private.
2. Write a static method that has return type object of this singleton class.

class Mother

{

private String name;a

private static Mother m = null;

private Mother(String n)

{

name = n;

}

public Mother getMother(String n)

{

if(m==null)

m = new Mother(n);

return m;

}

}

For a child class there can be only one mother instance hence Mother class is a singleton class. We keep the constructor private and make a getMother() method which will be called from outside to instantiate a new Mother object. If an object of Mother class is already created before that same instance will be returned else a new Mother object will be returned.

### 

### FACTORY DESIGN PATTERN AND ABSTRACT FACTORY :

Factory method is a [creational design pattern](https://www.geeksforgeeks.org/design-patterns-set-1-introduction/), i.e., related to object creation. In Factory pattern, we create object without exposing the creation logic to client and the client use the same common interface to create new type of object.

Factory method creates the object and returns it to the client(one who calls the factory method) depending on the parameter passed. So if we have to insert a new object type we will add code to the factory method and not in the client. The object type is the concrete product and it’s base class is the product.

**Let's first create an interface for a product to be manufactured by thefactory.**

**/\*\***

**\* Product interface**

**\*/**

**public interface GeometricShape {**

**void draw();**

**}**

**The following are the implementations of the above interface:**

**/\*\***

**\* Concrete Product**

**\*/**

**public class Line implements GeometricShape {**

**@Override**

**public void draw() {**

**System.out.println("Line Drawn.");**

**}**

**}**

**/\*\***

**\* Concrete Product**

**\*/**

**public class Rectangle implements GeometricShape {**

**@Override**

**public void draw() {**

**System.out.println("Rectangle is drawn.");**

**}**

**}**

**/\*\***

**\* Concrete product**

**\*/**

**public class Circle implements GeometricShape{**

**@Override**

**public void draw() {**

**System.out.println("Circle is drawn.");**

**}**

**}**

**I have added the following enums to name the shapes:**

**public enum ShapeType {**

**LINE,**

**CIRCLE,**

**RECTANGLE,**

**TRIANGLE**

**}**

**Now, let's create a factory that provides the product (in this case, GeometricShape):**

**/\*\***

**\* Concrete Product**

**\*/**

**public abstract class ShapeFactory {**

**public static GeometricShape getShape(ShapeType name) {**

**GeometricShape shape = null;**

**switch (name) {**

**case LINE:**

**shape = new Line();**

**break;**

**case CIRCLE:**

**shape = new Circle();**

**break;**

**case RECTANGLE:**

**shape = new Rectangle();**

**break;**

**}**

**return shape;**

**}**

**}**

**The client for the application provides the name of the shape required as follow.**

**/\*\***

**\* Client**

**\*/**

**public class Application {**

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

**//requests for circle shape**

**GeometricShape circle = ShapeFactory.getShape(ShapeType.CIRCLE);**

**if (circle != null) {**

**circle.draw();**

**} else {**

**System.out.println("This shape can not be drawn.");**

**}**

**//requests non existent shape**

**GeometricShape triangle = ShapeFactory.getShape(ShapeType.TRIANGLE);**

**if (triangle != null) {**

**triangle.draw();**

**} else {**

**System.out.println("This shape can't be drawn");**

**}**

**}**

**}**

**The output of the program is:**

**Circle is drawn.**

**This shape can't be drawn**

## BUILDER PATTERN

Builder pattern aims to “Separate the construction of a complex object from its representation so that the same construction process can create different representations.” It is used to construct a complex object step by step and the final step will return the object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

Below is the coded solution of problem we discussed above. This uses a additional class UserBuilder which helps us in building desired User object with all mandatory attributes and combination of optional attributes, without loosing the immutability.

public class User

{

//All final attributes

private final String firstName; // required

private final String lastName; // required

private final int age; // optional

private final String phone; // optional

private final String address; // optional

private User(UserBuilder builder) {

this.firstName = builder.firstName;

this.lastName = builder.lastName;

this.age = builder.age;

this.phone = builder.phone;

this.address = builder.address;

}

//All getter, and NO setter to provde immutability

public String getFirstName() {

return firstName;

}

public String getLastName() {

return lastName;

}

public int getAge() {

return age;

}

public String getPhone() {

return phone;

}

public String getAddress() {

return address;

}

@Override

public String toString() {

return "User: "+this.firstName+", "+this.lastName+", "+this.age+", "+this.phone+", "+this.address;

}

public static class UserBuilder

{

private final String firstName;

private final String lastName;

private int age;

private String phone;

private String address;

public UserBuilder(String firstName, String lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

public UserBuilder age(int age) {

this.age = age;

return this;

}

public UserBuilder phone(String phone) {

this.phone = phone;

return this;

}

public UserBuilder address(String address) {

this.address = address;

return this;

}

//Return the finally consrcuted User object

public User build() {

User user = new User(this);

validateUserObject(user);

return user;

}

private void validateUserObject(User user) {

//Do some basic validations to check

//if user object does not break any assumption of system

}

}

}

And below is the way, we will use the UserBuilder in our code:

public static void main(String[] args) {

User user1 = new User.UserBuilder("Lokesh", "Gupta")

.age(30)

.phone("1234567")

.address("Fake address 1234")

.build();

System.out.println(user1);

User user2 = new User.UserBuilder("Jack", "Reacher")

.age(40)

.phone("5655")

//no address

.build();

System.out.println(user2);

User user3 = new User.UserBuilder("Super", "Man")

//No age

//No phone

//no address

.build();

System.out.println(user3);

}

Output:

User: Lokesh, Gupta, 30, 1234567, Fake address 1234

User: Jack, Reacher, 40, 5655, null

User: Super, Man, 0, null, null

## BEHAVIOURAL PATTERNS :

### Chain of Responsibility :

**Chain of Responsibility** is a behavioral design pattern that lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

The Handler declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.

The Base Handler is an optional class where you can put the boilerplate code that’s common to all handler classes.

Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.

// The handler interface declares a method for building a chain

// of handlers. It also declares a method for executing a

// request.

interface ComponentWithContextualHelp is

method showHelp()

// The base class for simple components.

abstract class Component implements ComponentWithContextualHelp is

field tooltipText: string

// The component's container acts as the next link in the

// chain of handlers.

protected field container: Container

// The component shows a tooltip if there's help text

// assigned to it. Otherwise it forwards the call to the

// container, if it exists.

method showHelp() is

if (tooltipText != null)

// Show tooltip.

else

container.showHelp()

// Containers can contain both simple components and other

// containers as children. The chain relationships are

// established here. The class inherits showHelp behavior from

// its parent.

abstract class Container extends Component is

protected field children: array of Component

method add(child) is

children.add(child)

child.container = this

// Primitive components may be fine with default help

// implementation...

class Button extends Component is

// ...

// But complex components may override the default

// implementation. If the help text can't be provided in a new

// way, the component can always call the base implementation

// (see Component class).

class Panel extends Container is

field modalHelpText: string

method showHelp() is

if (modalHelpText != null)

// Show a modal window with the help text.

else

super.showHelp()

// ...same as above...

class Dialog extends Container is

field wikiPageURL: string

method showHelp() is

if (wikiPageURL != null)

// Open the wiki help page.

else

super.showHelp()

// Client code.

class Application is

// Every application configures the chain differently.

method createUI() is

dialog = new Dialog("Budget Reports")

dialog.wikiPageURL = "http://..."

panel = new Panel(0, 0, 400, 800)

panel.modalHelpText = "This panel does..."

ok = new Button(250, 760, 50, 20, "OK")

ok.tooltipText = "This is an OK button that..."

cancel = new Button(320, 760, 50, 20, "Cancel")

// ...

panel.add(ok)

panel.add(cancel)

dialog.add(panel)

// Imagine what happens here.

method onF1KeyPress() is

component = this.getComponentAtMouseCoords()

component.showHelp()

### Command :

* Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.
* Promote "invocation of a method on an object" to full object status
* An object-oriented callback

The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parametrized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.

**Check list**

1. Define a Command interface with a method signature like execute().
2. Create one or more derived classes that encapsulate some subset of the following: a "receiver" object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to execute().

interface Command {

void execute();

}

class DomesticEngineer implements Command {

public void execute() {

System.out.println("take out the trash");

}

}

class Politician implements Command {

public void execute() {

System.out.println("take money from the rich, take votes from the poor");

}

}

class Programmer implements Command {

public void execute() {

System.out.println("sell the bugs, charge extra for the fixes");

}

}

public class CommandDemo {

public static List produceRequests() {

List<Command> queue = new ArrayList<>();

queue.add(new DomesticEngineer());

queue.add(new Politician());

queue.add(new Programmer());

return queue;

}

public static void workOffRequests(List queue) {

for (Object command : queue) {

((Command)command).execute();

}

}

public static void main( String[] args ) {

List queue = produceRequests();

workOffRequests(queue);

}

}

Output

take out the trash

take money from the rich, take votes from the poor

sell the bugs, charge extra for the fixes

### STRATEGY

* Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
* Capture the abstraction in an interface, bury implementation details in derived classes.

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on trade-offs between cost, convenience, and time.

**interface Strategy {**

**void solve();**

**}**

**// 2. Bury implementation**

**@SuppressWarnings("ALL")**

**abstract class StrategySolution implements Strategy {**

**// 3. Template Method**

**public void solve() {**

**start();**

**while (nextTry() && !isSolution()) {}**

**stop();**

**}**

**abstract void start();**

**abstract boolean nextTry();**

**abstract boolean isSolution();**

**abstract void stop();**

**}**

**class FOO extends StrategySolution {**

**private int state = 1;**

**protected void start() {**

**System.out.print("Start ");**

**}**

**protected void stop() {**

**System.out.println("Stop");**

**}**

**protected boolean nextTry() {**

**System.out.print("NextTry-" + state++ + " ");**

**return true;**

**}**

**protected boolean isSolution() {**

**System.out.print("IsSolution-" + (state == 3) + " ");**

**return (state == 3);**

**}**

**}**

**// 2. Bury implementation**

**abstract class StrategySearch implements Strategy {**

**// 3. Template Method**

**public void solve() {**

**while (true) {**

**preProcess();**

**if (search()) {**

**break;**

**}**

**postProcess();**

**}**

**}**

**abstract void preProcess();**

**abstract boolean search();**

**abstract void postProcess();**

**}**

**@SuppressWarnings("ALL")**

**class BAR extends StrategySearch {**

**private int state = 1;**

**protected void preProcess() {**

**System.out.print("PreProcess ");**

**}**

**protected void postProcess() {**

**System.out.print("PostProcess ");**

**}**

**protected boolean search() {**

**System.out.print("Search-" + state++ + " ");**

**return state == 3 ? true : false;**

**}**

**}**

**// 4. Clients couple strictly to the interface**

**public class StrategyDemo {**

**// client code here**

**private static void execute(Strategy strategy) {**

**strategy.solve();**

**}**

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

**Strategy[] algorithms = {new FOO(), new BAR()};**

**for (Strategy algorithm : algorithms) {**

**execute(algorithm);**

**}**

**}**

**}**

**Output**

**start nextTry-1 isSolution-false nextTry-2 isSolution-true stop**

**preProcess search-1 postProcess preProcess search-2**

### Adapter : -

* Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
* Wrap an existing class with a new interface.
* Impedance match an old component to a new system

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

The Adapter could also be thought of as a "wrapper".

class SquarePeg {

private double width;

public SquarePeg(double width) {

this.width = width;

}

public double getWidth() {

return width;

}

public void setWidth(double width) {

this.width = width;

}

}

/\* The NEW \*/

class RoundHole {

private final int radius;

public RoundHole(int radius) {

this.radius = radius;

System.out.println("RoundHole: max SquarePeg is " + radius \* Math.sqrt(2));

}

public int getRadius() {

return radius;

}

}

// Design a "wrapper" class that can "impedance match" the old to the new

class SquarePegAdapter {

// The adapter/wrapper class "has a" instance of the legacy class

private final SquarePeg squarePeg;

public SquarePegAdapter(double w) {

squarePeg = new SquarePeg(w);

}

// Identify the desired interface

public void makeFit(RoundHole roundHole) {

// The adapter/wrapper class delegates to the legacy object

double amount = squarePeg.getWidth() - roundHole.getRadius() \* Math.sqrt(2);

System.out.println( "reducing SquarePeg " + squarePeg.getWidth() + " by " + ((amount < 0) ? 0 : amount) + " amount");

if (amount > 0) {

squarePeg.setWidth(squarePeg.getWidth() - amount);

System.out.println(" width is now " + squarePeg.getWidth());

}

}

}

public class AdapterDemoSquarePeg {

public static void main( String[] args ) {

RoundHole roundHole = new RoundHole( 5 );

SquarePegAdapter squarePegAdapter;

for (int i = 6; i < 10; i++) {

squarePegAdapter = new SquarePegAdapter((double)i);

// The client uses (is coupled to) the new interface

squarePegAdapter.makeFit(roundHole);

}

}

}

Output

RoundHole: max SquarePeg is 7.0710678118654755

reducing SquarePeg 6.0 by 0.0 amount

reducing SquarePeg 7.0 by 0.0 amount

reducing SquarePeg 8.0 by 0.9289321881345245 amount

width is now 7.0710678118654755

reducing SquarePeg 9.0 by 1.9289321881345245 amount

width is now 7.0710678118654755

### Decorator : -

* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
* Client-specified embellishment of a core object by recursively wrapping it.
* Wrapping a gift, putting it in a box, and wrapping the box.

The client is always interested in CoreFunctionality.doThis(). The client may, or may not, be interested in OptionalOne.doThis() and OptionalTwo.doThis(). Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.

Before

class A {

public void doIt() {

System.out.print('A');

}

}

class AwithX extends A {

public void doIt() {

super.doIt();

doX();

}

private void doX() {

System.out.print('X');

}

}

class aWithY extends A {

public void doIt() {

super.doIt();

doY();

}

public void doY() {

System.out.print('Y');

}

}

class aWithZ extends A {

public void doIt() {

super.doIt();

doZ();

}

public void doZ() {

System.out.print('Z');

}

}

class AwithXY extends AwithX {

private aWithY obj = new aWithY();

public void doIt() {

super.doIt();

obj.doY();

}

}

class AwithXYZ extends AwithX {

private aWithY obj1 = new aWithY();

private aWithZ obj2 = new aWithZ();

public void doIt() {

super.doIt();

obj1.doY();

obj2.doZ();

}

}

public class DecoratorDemo {

public static void main(String[] args) {

A[] array = {new AwithX(), new AwithXY(), new AwithXYZ()};

for (A a : array) {

a.doIt();

System.out.print(" ");

}

}

}

After

interface I {

void doIt();

}

class A implements I {

public void doIt() {

System.out.print('A');

}

}

abstract class D implements I {

private I core;

public D(I inner) {

core = inner;

}

public void doIt() {

core.doIt();

}

}

class X extends D {

public X(I inner) {

super(inner);

}

public void doIt() {

super.doIt();

doX();

}

private void doX() {

System.out.print('X');

}

}

class Y extends D {

public Y(I inner) {

super(inner);

}

public void doIt() {

super.doIt();

doY();

}

private void doY() {

System.out.print('Y');

}

}

class Z extends D {

public Z(I inner) {

super(inner);

}

public void doIt() {

super.doIt();

doZ();

}

private void doZ() {

System.out.print('Z');

}

}

public class DecoratorDemo {

public static void main( String[] args ) {

I[] array = {new X(new A()), new Y(new X(new A())),

new Z(new Y(new X(new A())))};

for (I anArray : array) {

anArray.doIt();

System.out.print(" ");

}

}

}

Output

AX AXY AXYZ