Design Patterns

Design patterns represent the best practices used by experienced developers. Design patterns are solutions to commons (or general) problems that software developers faced during development. These solutions were obtained by trial and error by numerous developers over quite a substantial period of time.

The Design patterns provide best solutions to certain problems faced during software development.

Design patterns are primarily based on the following principles:

I. Program to an interface not an implementation
II. Favor object composition over inheritance

Design Patterns have two main usages in software development such as Common platform and Best practices for developers.

Types of Design Patterns

1) **Creational Design patterns**
   These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator.

2) **Structural patterns**
   These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

3) **Behavioral Patterns**
   These design patterns are specifically concerned with communication between objects.

4) **J2EE Patterns**
   These design patterns are specifically concerned with the presentation tier.

Factory Pattern

Create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation:

```
public interface Shape {
        public void draw();
}

public class Rectangle implements Shape {
        @Override
        public void draw() {
                System.out.println("Inside Rectangle::draw() method.");
```

ASPIRE-K.RAMESH

```
	}
}

public class Square implements Shape {
	@Override
	public void draw() {
		System.out.println("Inside Square::draw() method.");
	}
}
public class Circle implements Shape {
	  @Override
	  public void draw() {
	    System.out.println("Inside Circle::draw() method.");
	  }
}

//Create a Factory to generate object of concrete class based on given information.
public class ShapeFactory {
	public Shape getShape(String shapeType) {
		if (shapeType == null) {
			return null;
		}
		if (shapeType.equalsIgnoreCase("CIRCLE")) {
			return new Circle();

			/* //Create obj using reflection mechanism
			Class cobj = null;
			Circle c = null;
			try {
				cobj = Class.forName("Circle");
				c = (Circle)cobj.newInstance();
			} catch (Exception e) {
				e.printStackTrace();
			}
			return c;*/
		} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
			return new Rectangle();

		} else if (shapeType.equalsIgnoreCase("SQUARE")) {
			return new Square();
		}
		return null;
```

ASPIRE-K.RAMESH

```
	}
}

public class FactoryPatternTest {
	public static void main(String[] args) {
		ShapeFactory shapeFactory = new ShapeFactory();

		// Get an object of Circle and call its draw method.
		Shape shape1 = shapeFactory.getShape("CIRCLE");
		shape1.draw(); //call draw method of Circle

		// Get an object of Rectangle and call its draw method.
		Shape shape2 = shapeFactory.getShape("RECTANGLE");
		shape2.draw(); //call draw method of Rectangle

		// Get an object of Square and call its draw method.
		Shape shape3 = shapeFactory.getShape("SQUARE");
		shape3.draw(); // call draw method of circle
	}
}
```

Abstract Factory Pattern

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as **factory of factories**.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Implementation:

```
package edu.aspire.shape;
public interface Shape {
	public void draw();
}

package edu.aspire.shape;
public class Rectangle implements Shape {
	@Override
	public void draw() {
		System.out.println("Inside Rectangle::draw() method.");
	}
}

package edu.aspire.shape;
```

ASPIRE-K.RAMESH

```
public class Square implements Shape {
	@Override
	public void draw() {
		System.out.println("Inside Square::draw() method.");
	}
}

package edu.aspire.shape;
public class Circle implements Shape {
	@Override
	public void draw() {
		System.out.println("Inside Circle::draw() method.");
	}
}

package edu.aspire.color;
public interface Color {
	void fill();
}

package edu.aspire.color;
public class Blue implements Color {
	@Override
	public void fill() {
		System.out.println("Inside Blue::fill() method.");
	}
}

package edu.aspire.color;
public class Green implements Color {
	@Override
	public void fill() {
		System.out.println("Inside Green::fill() method.");
	}
}

package edu.aspire.color;
public class Red implements Color {
	@Override
	public void fill() {
		System.out.println("Inside Red::fill() method.");
	}
```

ASPIRE-K.RAMESH

```
}

package edu.aspire.pattern.factory;
import edu.aspire.shape.Circle;
import edu.aspire.shape.Rectangle;
import edu.aspire.shape.Shape;
import edu.aspire.shape.Square;
//Create a Factory to generate object of concrete class based on given information.
public class ShapeFactory {
	public Shape getShape(String shapeType) {
		if (shapeType == null) {
			return null;
		}
		if (shapeType.equalsIgnoreCase("CIRCLE")) {
			return new Circle();

		} else if (shapeType.equalsIgnoreCase("RECTANGLE")) {
			return new Rectangle();

		} else if (shapeType.equalsIgnoreCase("SQUARE")) {
			return new Square();
		}
		return null;
	}
}

package edu.aspire.pattern.factory;
import edu.aspire.color.Blue;
import edu.aspire.color.Color;
import edu.aspire.color.Green;
import edu.aspire.color.Red;
public class ColorFactory {
	public Color getColor(String color) {
		if (color == null) {
			return null;
		} else if (color.equalsIgnoreCase("RED")) {
			return new Red();
		} else if (color.equalsIgnoreCase("GREEN")) {
			return new Green();
		} else if (color.equalsIgnoreCase("BLUE")) {
			return new Blue();
		}
```

ASPIRE-K.RAMESH

```
                return null;
        }
}

package edu.aspire.pattern.abstractfactory;
import edu.aspire.pattern.factory.ColorFactory;
import edu.aspire.pattern.factory.ShapeFactory;
public class AbstractFactory {
        public ShapeFactory getShapeFactory() {
                return new ShapeFactory();
        }

        public ColorFactory getColorFactory() {
                return new ColorFactory();
        }
}

package edu.aspire.test;
import edu.aspire.color.Color;
import edu.aspire.pattern.abstractfactory.AbstractFactory;
import edu.aspire.pattern.factory.ColorFactory;
import edu.aspire.pattern.factory.ShapeFactory;
import edu.aspire.shape.Shape;

public class AbstractFactoryTest {
        public static void main(String[] args) {
                AbstractFactory abstractFactory = new AbstractFactory();

                ShapeFactory shapeFactory = abstractFactory.getShapeFactory();
                Shape shape1 = shapeFactory.getShape("CIRCLE");
                shape1.draw();

                ColorFactory colorFactory = abstractFactory.getColorFactory();
                Color color = colorFactory.getColor("RED");
                color.fill();
        }
}
```

ASPIRE-K.RAMESH

Singleton Pattern

This pattern involves a single class which is responsible to create an object while making sure that **only single object gets created**. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Rules:

1) Constructor should be private.
2) Declare static field and static method.

Implementation:

```
package edu.aspire;
public class SingleObject {
	// Create an object of SingleObject
	private static SingleObject instance;
	// Make the constructor private so that this class cannot be instantiated from outsize
	private SingleObject() {
		System.out.println("*********");
	}

	// Get the only object available
	public static SingleObject getInstance() {
		if (instance == null) {
			instance = new SingleObject();
		}
		return instance;
	}

	public void showMessage() {
		System.out.println("Hello World!");
	}
}

package edu.asire.test;
import edu.aspire.SingleObject;
public class SingletonPatternTest {
	public static void main(String[] args) {
		// illegal construct
		// Compile Time Error: The constructor SingleObject() is not visible
		// SingleObject object = new SingleObject();
```

ASPIRE-K.RAMESH

```
                // Get the only object available
                SingleObject object = SingleObject.getInstance();
                SingleObject object2 = SingleObject.getInstance();

                // show the message
                object.showMessage();
                object2.showMessage();
        }
}
```

O/P

```
********
Hello World!
Hello World!
```

Prototype Pattern

Prototype pattern refers to creating duplicate objects while keeping performance in mind.

This pattern involves implementing a prototype interface which tells to create a **clone** of the current object. This pattern is used when creation of object directly is costly. We can cache the object, returns its clone on next request.

Implementation:

```
package edu.aspire.prototype;
public abstract class Shape implements Cloneable{
        private String id;
        protected String type;
        abstract void draw();
        public String getType() { return type; }
        public String getId() { return id; }
        public void setId(String id) { this.id = id; }

        public Object clone() {
                Object clone = null;
                try {
                        clone = super.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }
                return clone;
        }
}
```

ASPIRE-K.RAMESH

```
package edu.aspire.prototype;
public class Circle extends Shape {
	public Circle() {
		System.out.println("***inside circle constructor****");
		type = "Circle";
	}
	@Override
	public void draw() {
		System.out.println("Inside Circle::draw() method.");
	}
}

package edu.aspire.prototype;
public class Rectangle extends Shape {
	public Rectangle() {
		System.out.println("***inside rectangle constructor****");
		type = "Rectangle";
	}
	@Override
	public void draw() {
		System.out.println("Inside Rectangle::draw() method.");
	}
}

package edu.aspire.prototype;
public class Square extends Shape {
	public Square() {
		System.out.println("***inside square constructor****");
		type = "Square";
	}
	@Override
	public void draw() {
		System.out.println("Inside Square::draw() method.");
	}
}

package edu.aspire.prototype;
import java.util.Hashtable;
/**A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns
*their clone when requested.
*/
public class ShapeCache {
```

ASPIRE-K.RAMESH

```
	private static Hashtable<String, Shape> shapeMap = new Hashtable<String, Shape>();
	public static void loadCache() {
		Circle circle = new Circle();
		circle.setId("1");
		shapeMap.put(circle.getId(), circle);

		Square square = new Square();
		square.setId("2");
		shapeMap.put(square.getId(), square);

		Rectangle rectangle = new Rectangle();
		rectangle.setId("3");
		shapeMap.put(rectangle.getId(), rectangle);
	}

	public static Shape getShape(String shapeId) {
		Shape cachedShape = shapeMap.get(shapeId);
		return (Shape) cachedShape.clone();
	}
}

package edu.aspire.test;
import edu.aspire.prototype.Shape;
import edu.aspire.prototype.ShapeCache;
public class PrototypePatternTest {
	public static void main(String[] args) {
		ShapeCache.loadCache();

		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("1")).getType());
		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("1")).getType());

		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("2")).getType());
		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("2")).getType());

		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("3")).getType());
		System.out.println("Shape : " + ((Shape) ShapeCache.getShape("3")).getType());
	}
}
```

O/P

```
***inside circle constructor****
***inside square constructor****
***inside rectangle constructor****
```

ASPIRE-K.RAMESH

```
Shape : Circle
Shape : Circle
Shape : Square
Shape : Square
Shape : Rectangle
Shape : Rectangle
```

Facade Pattern

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities.

This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system.

Implementation:

```
package edu.aspire.sp;
public interface Shape {
	public void draw();
}

package edu.aspire.sp;
public class Circle implements Shape {
	@Override
	public void draw() {
		System.out.println("Circle::draw()");
	}
}

package edu.aspire.sp;
public class Rectangle implements Shape {
	@Override
	public void draw() {
		System.out.println("Rectangle::draw()");
	}
}

package edu.aspire.sp;
public class Square implements Shape {
	@Override
```

ASPIRE-K.RAMESH

```
	public void draw() {
		System.out.println("Square::draw()");
	}
}

package edu.aspire.pattern.facade;
import edu.aspire.sp.Circle;
import edu.aspire.sp.Rectangle;
import edu.aspire.sp.Shape;
import edu.aspire.sp.Square;
/**
 * ShapeMaker class uses the concrete classes to delegate user calls to these classes. FacadePatternTest,
 *our test class, will use ShapeMaker class to show the results.
 */
public class ShapeMaker {
	private Shape circle;
	private Shape rectangle;
	private Shape square;

	public ShapeMaker() {
		circle = new Circle();
		rectangle = new Rectangle();
		square = new Square();
	}

	public void drawCircle() {  circle.draw(); }
	public void drawRectangle() {  rectangle.draw(); }
	public void drawSquare() {  square.draw(); }
}

package edu.aspire.test;
import edu.aspire.pattern.facade.ShapeMaker;
public class FacadePatternTest {
	public static void main(String[] args) {
		ShapeMaker shapeMaker = new ShapeMaker();

		shapeMaker.drawCircle();
		shapeMaker.drawRectangle();
		shapeMaker.drawSquare();
	}
}
```

ASPIRE-K.RAMESH

Composite Pattern

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy. This type of design pattern comes under structural pattern as this pattern creates a tree structure for group of objects.

This pattern creates a class that contains group of its own objects. This class provides ways to modify its group of same objects.

Implementation:

```
package edu.aspire.pattern.composite;
import java.util.ArrayList;
import java.util.List;
public class Employee {
	private String name;
	private String dept;
	private int salary;
	private List<Employee> subordinates;

	public Employee(String name, String dept, int sal) {
		this.name = name;
		this.dept = dept;
		this.salary = sal;
		subordinates = new ArrayList<Employee>();
	}
	public void add(Employee e) {
		subordinates.add(e);
	}
	public void remove(Employee e) {
		subordinates.remove(e);
	}
	public List<Employee> getSubordinates() {
		return subordinates;
	}
	public String toString() {
		return ("Employee :[ Name : " + name + ", Dept : " + dept + ", Salary :" + salary + " ]");
	}
}

package edu.aspire.test;
import edu.aspire.pattern.composite.Employee;
public class CompositePatternDemo {
	public static void main(String[] args) {
```

ASPIRE-K.RAMESH

```
        Employee ceo= new Employee("John", "CEO", 30000);

        Employee headSales = new Employee("Robert", "Head Sales", 20000);
        Employee headMarketing = new Employee("Michel", "Head Marketing", 20000);
        ceo.add(headSales);
        ceo.add(headMarketing);

        Employee salesExecutive1 = new Employee("Richard", "Sales", 10000);
        Employee salesExecutive2 = new Employee("Rob", "Sales", 10000);
        headSales.add(salesExecutive1);
        headSales.add(salesExecutive2);

        Employee clerk1 = new Employee("Laura", "Marketing", 10000);
        Employee clerk2 = new Employee("Bob", "Marketing", 10000);
        headMarketing.add(clerk1);
        headMarketing.add(clerk2);

        // print all employees of the organization
        System.out.println(ceo);
        for (Employee headEmployee : ceo.getSubordinates()) {
            System.out.println(headEmployee);

            for (Employee employee : headEmployee.getSubordinates()) {
                System.out.println(employee);
            }
        }
    }
}
```

Command Pattern

Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

Implementation:
We have created a Stock class which acts as a request. We have created an interface Order which is acting as a command. We have concreted command classes BuyStock and SellStock implementing Order interface which will do actual command processing. A class Broker is created which acts as an invoker object. It can take and place orders.

ASPIRE-K.RAMESH

Broker object uses command pattern to identify which object will execute which command based on the type of command. CommandPatternDemo, our demo class, will use Broker class to demonstrate command pattern.

```
package edu.aspire;
public class Stock {
	private String name = "ABC";
	private int quantity = 10;
	public void buy() {
		System.out.println("Stock [ Name: " + name + ",  Quantity: " + quantity + " ] bought");
	}
	public void sell() {
		System.out.println("Stock [ Name: " + name + ",  Quantity: " + quantity + " ] sold");
	}
}
```

```
package edu.aspire;
public interface Order {
	void execute();
}
```

```
package edu.aspire;
public class BuyStock implements Order {
	private Stock abcStock;
	public BuyStock(Stock abcStock) {
		this.abcStock = abcStock;
	}
	public void execute() {
		abcStock.buy();
	}
}
```

```
package edu.aspire;
public class SellStock implements Order {
	private Stock abcStock;
	public SellStock(Stock abcStock) {
		this.abcStock = abcStock;
	}
	public void execute() {
		abcStock.sell();
	}
}
```

ASPIRE-K.RAMESH

```
package edu.aspire.pattern.command;
import java.util.ArrayList;
import java.util.List;
import edu.aspire.Order;
public class Broker {
	private List<Order> orderList = new ArrayList<Order>();
	public void takeOrder(Order order) {
		orderList.add(order);
	}
	public void placeOrders() {
		for (Order order : orderList) {
			order.execute();
		}
		orderList.clear();
	}
}

package edu.aspire.test;
import edu.aspire.BuyStock;
import edu.aspire.SellStock;
import edu.aspire.Stock;
import edu.aspire.pattern.command.Broker;
public class CommandPatternTest {
	public static void main(String[] args) {
		Stock abcStock = new Stock();

		Order buyStockOrder = new BuyStock(abcStock);
		Order sellStockOrder = new SellStock(abcStock);

		Broker broker = new Broker();

		broker.takeOrder(buyStockOrder);
		broker.takeOrder(sellStockOrder);

		broker.placeOrders();
	}
}
```

ASPIRE-K.RAMESH

Data Access Object Pattern

Data Access Object Pattern or DAO pattern is used to separate database operations from high level business services. Following are the participants in Data Access Object Pattern.

I. **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object(s).
II. **Data Access Object concrete class** - This class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.
III. **Model class or Data class** - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

Implementation:

```
package edu.aspire.model;
public class Student { //data class
	private int rollNo;
	private String name;

	public Student(int rollNo, String name) {
		this.rollNo = rollNo;
		this.name = name;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public int getRollNo() {
		return rollNo;
	}
	public void setRollNo(int rollNo) {
		this.rollNo = rollNo;
	}
}

package edu.aspire.dao;
import java.util.List;
import edu.aspire.model.Student;
public interface StudentDao { //dao interface
	public void create(Student s);
```

ASPIRE-K.RAMESH

```
	public Student read(int rollNo);
	public void update(Student student);
	public void delete(Student student);

	//finder methods
	public List<Student> getAllStudents();
}

package edu.aspire.dao;
import java.util.ArrayList;
import java.util.List;
import edu.aspire.model.Student;
public class StudentDaoImpl implements StudentDao {
	// list is working as a database
	List<Student> students;
	public StudentDaoImpl() {
		students = new ArrayList<Student>();
		Student student1 = new Student(1, "Robert");
		Student student2 = new Student(2, "John");
		students.add(student1);
		students.add(student2);
	}

	@Override
	public void create(Student s) {
		Student student = new Student(3, "Millar");
		students.add(student);
	}

	@Override
	public Student read(int rollNo) {
		return students.get(rollNo);
	}
	@Override
	public void update(Student student) {
		System.out.println(student.getRollNo());
		students.get(student.getRollNo()).setName(student.getName());
		System.out.println("Student: Roll No " + student.getRollNo() + ", updated in the database");
	}

	@Override
	public void delete(Student student) {
```

ASPIRE-K.RAMESH

```
            students.remove(student.getRollNo());
            System.out.println("Student: Roll No " + student.getRollNo() + ", deleted from database");
        }

        @Override
        public List<Student> getAllStudents() {
            return students;
        }
}

package edu.aspire.test;
import edu.aspire.dao.StudentDao;
import edu.aspire.dao.StudentDaoImpl;
import edu.aspire.model.Student;
public class DaoPatternDemo {
        public static void main(String[] args) {
            StudentDao studentDao = new StudentDaoImpl();

            //print all students
            for (Student student : studentDao.getAllStudents()) {
                System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
            }

            //update student
            Student student =studentDao.getAllStudents().get(0);
            student.setName("Michael");
            studentDao.update(student);

            //get the student
            studentDao.read(1);
            System.out.println("Student: [RollNo : " + student.getRollNo() + ", Name : " + student.getName() + " ]");
        }
}
```

ASPIRE-K.RAMESH

Service Locator Pattern

The service locator design pattern is used when we want to locate various services using JNDI lookup. Considering high cost of looking up JNDI for a service, Service Locator pattern makes use of caching technique. For the first time a service is required, Service Locator looks up in JNDI and caches the service object. Further lookup of same service via Service Locator is done in its cache which improves the performance of application to great extent.

Implementation:

```
package edu.aspire.impl;
public interface Service {
	public String getName();
	public void execute();
}
package edu.aspire.impl;
public class ServiceImpl1 implements Service {
	public void execute() {
		System.out.println("Executing Service1");
	}
	@Override
	public String getName() {
		return "Service1";
	}
}

package edu.aspire.impl;
public class ServiceImpl2 implements Service {
	public void execute() {
		System.out.println("Executing Service2");
	}
	@Override
	public String getName() {
		return "Service2";
	}
}

package edu.aspire.context;
import edu.aspire.impl.ServiceImpl1;
import edu.aspire.impl.ServiceImpl2;
public class InitialContext {
	public Object lookup(String jndiName) {
		if (jndiName.equalsIgnoreCase("SERVICE1")) {
			System.out.println("Looking up and creating a new Service1 object");
			return new ServiceImpl1();
		} else if (jndiName.equalsIgnoreCase("SERVICE2")) {
			System.out.println("Looking up and creating a new Service2 object");
			return new ServiceImpl2();
		}
		return null;
	}
}
```

ASPIRE-K.RAMESH

```
package edu.aspire.context;
import java.util.ArrayList;
import java.util.List;
import edu.aspire.impl.Service;
public class Cache {
	private List<Service> services;
	public Cache() {
		services = new ArrayList<Service>();
	}
	public Service getService(String serviceName) {
		for (Service service : services) {
			if (service.getName().equalsIgnoreCase(serviceName)) {
				System.out.println("Returning cached  " + serviceName + " object");
				return service;
			}
		}
		return null;
	}
	public void addService(Service newService) {
		boolean exists = false;
		for (Service service : services) {
			if (service.getName().equalsIgnoreCase(newService.getName())) {
				exists = true;
			}
		}
		if (!exists) {
			services.add(newService);
		}
	}
}

package edu.aspire.pattern.locator;
import edu.aspire.context.Cache;
import edu.aspire.context.InitialContext;
import edu.aspire.impl.Service;
public class ServiceLocator {
	private static Cache cache;

	static {
		cache = new Cache();
	}
```

ASPIRE-K.RAMESH

```
	public static Service getService(String jndiName) {
		Service service = cache.getService(jndiName);
		if (service != null) {
			return service;
		}
		InitialContext context = new InitialContext();
		Service service1 = (Service) context.lookup(jndiName);
		cache.addService(service1);
		return service1;
	}
}

package edu.aspire.test;
import edu.aspire.impl.Service;
import edu.aspire.pattern.locator.ServiceLocator;
public class ServiceLocatorPatternDemo {
	public static void main(String[] args) {
		Service service = ServiceLocator.getService("Service1");
		service.execute();
		service = ServiceLocator.getService("Service2");
		service.execute();
		service = ServiceLocator.getService("Service1");
		service.execute();
		service = ServiceLocator.getService("Service2");
		service.execute();
	}
}
```

ASPIRE-K.RAMESH