**A SYNTHESE ON THE DIFFERENT TYPES OF DESIGN PATTERNS**

**CREATIONAL DESIGN PATTERNS**

The creational design patterns are a set of design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Here's a brief resume of each creational design pattern with examples:

**Singleton Pattern:** This pattern ensures that a class has only one instance and provides a global point of access to that instance.

Use: Ensures that a class has only one instance and provides a global point of access to that instance.

Example: A logging class in a software application that needs to maintain a single log instance across the entire application.

public class Singleton {

private static Singleton instance;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

// Other methods and properties

}

**Factory Method Pattern:** This pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate.

Use: Defines an interface for creating an object, but lets subclasses decide which class to instantiate.

Example: A vehicle factory that has multiple subclasses for creating different types of vehicles (Car, Truck, Motorcycle, etc.).

public interface Product {

void operation();

}

public class ConcreteProduct implements Product {

@Override

public void operation() {

System.out.println("Performing operation in ConcreteProduct");

}

}

public interface Creator {

Product createProduct();

}

public class ConcreteCreator implements Creator {

@Override

public Product createProduct() {

return new ConcreteProduct();

}

}

**Abstract Factory Pattern:** This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Use: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Example: An abstract factory for creating different types of furniture, such as chairs, tables, and sofas, each with variations based on material and style.

public interface AbstractFactory {

ProductA createProductA();

ProductB createProductB();

}

public class ConcreteFactory implements AbstractFactory {

@Override

public ProductA createProductA() {

return new ConcreteProductA();

}

@Override

public ProductB createProductB() {

return new ConcreteProductB();

}

}

**Builder Pattern:** This pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations

Use: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Example: Building a complex object like a computer, where the builder constructs the computer step by step, allowing different representations (e.g., gaming PC, office PC) to be created using the same construction process.

public class Product {

private String partA;

private String partB;

public void setPartA(String partA) {

this.partA = partA;

}

public void setPartB(String partB) {

this.partB = partB;

}

// Other methods and properties

}

public interface Builder {

void buildPartA();

void buildPartB();

Product getResult();

}

public class ConcreteBuilder implements Builder {

private Product product;

public ConcreteBuilder() {

this.product = new Product();

}

@Override

public void buildPartA() {

product.setPartA("Part A");

}

@Override

public void buildPartB() {

product.setPartB("Part B");

}

@Override

public Product getResult() {

return product;

}

}

**Prototype Pattern:** This pattern creates new objects by copying an existing object, known as the prototype.

Use: Creates new objects by copying an existing object (the prototype).

Example: Creating new instances of a complex object by cloning an existing instance, such as creating multiple instances of a customized character in a game.

These patterns provide flexible ways to create objects and manage their instantiation, allowing for better code organization and reusability.

public class ObjectPool {

private static final int MAX\_OBJECTS = 5;

private static List<ReusableObject> pool = new ArrayList<>();

static {

for (int i = 0; i < MAX\_OBJECTS; i++) {

pool.add(new ReusableObject());

}

}

public synchronized ReusableObject acquireObject() {

if (pool.isEmpty()) {

return new ReusableObject();

} else {

return pool.remove(0);

}

}

public synchronized void releaseObject(ReusableObject object) {

if (pool.size() < MAX\_OBJECTS) {

pool.add(object);

}

}

}

public class ReusableObject {

// Object properties and methods

}

**BEHAVIORAL DESIGN PATTERNS**

**Chain of Responsibility Pattern:**

Definition: This pattern allows an object to pass a request 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.

Example: An ATM system where the request to dispense cash is passed through a chain of handlers, each responsible for a different denomination of currency.

public abstract class Handler {

private Handler nextHandler;

public void setNextHandler(Handler nextHandler) {

this.nextHandler = nextHandler;

}

public void handleRequest(Request request) {

if (nextHandler != null) {

nextHandler.handleRequest(request);

}

}

}

public class ConcreteHandlerA extends Handler {

@Override

public void handleRequest(Request request) {

if (request.getType().equals(RequestType.TYPE\_A)) {

System.out.println("ConcreteHandlerA handles the request");

} else {

super.handleRequest(request);

}

}

}

public class ConcreteHandlerB extends Handler {

@Override

public void handleRequest(Request request) {

if (request.getType().equals(RequestType.TYPE\_B)) {

System.out.println("ConcreteHandlerB handles the request");

} else {

super.handleRequest(request);

}

}

}

public class Request {

private String type;

public Request(String type) {

this.type = type;

}

public String getType() {

return type;

}

}

public class RequestType {

public static final String TYPE\_A = "Type A";

public static final String TYPE\_B = "Type B";

}

// Usage

public class Client {

public static void main(String[] args) {

Handler handlerA = new ConcreteHandlerA();

Handler handlerB = new ConcreteHandlerB();

handlerA.setNextHandler(handlerB);

Request requestA = new Request(RequestType.TYPE\_A);

Request requestB = new Request(RequestType.TYPE\_B);

handlerA.handleRequest(requestA);

handlerA.handleRequest(requestB);

}

}

**Command Pattern:**

Definition: This pattern encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

Example: A menu system in a restaurant where customer orders are encapsulated as command objects, allowing the kitchen staff to execute the orders.

public interface Command {

void execute();

}

public class ConcreteCommand implements Command {

private Receiver receiver;

public ConcreteCommand(Receiver receiver) {

this.receiver = receiver;

}

@Override

public void execute() {

receiver.action();

}

}

public class Receiver {

public void action() {

System.out.println("Receiver executing action");

}

}

public class Invoker {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void executeCommand() {

command.execute();

}

}

**Interpreter Pattern:**

Definition: This pattern defines a grammar for interpreting a language and provides an interpreter to deal with sentences in the language.

Example: A regular expression engine that interprets and matches patterns in text.

**Iterator Pattern:**

Definition: This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Example: Iterating through elements of a list or collection without needing to know the internal structure of the collection.

public interface Iterator<T> {

boolean hasNext();

T next();

}

public class ConcreteIterator<T> implements Iterator<T> {

private T[] elements;

private int position;

public ConcreteIterator(T[] elements) {

this.elements = elements;

this.position = 0;

}

@Override

public boolean hasNext() {

return position < elements.length;

}

@Override

public T next() {

return elements[position++];

}

}

public interface Aggregate<T> {

Iterator<T> createIterator();

}

public class ConcreteAggregate<T> implements Aggregate<T> {

private T[] elements;

public ConcreteAggregate(T[] elements) {

this.elements = elements;

}

@Override

public Iterator<T> createIterator() {

return new ConcreteIterator<>(elements);

}

}

**Mediator Pattern:**

Definition: This pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly.

Example: An air traffic control system that coordinates the movement of aircraft without direct communication between individual aircraft.

**Memento Pattern:**

Definition: This pattern captures and externalizes an object's internal state so that the object can be restored to this state later.

Example: The "undo" feature in a word processor that allows users to revert to a previous state of the document.

**Observer Pattern:**

Definition: This pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Example: A weather station that notifies multiple displays when the weather conditions change.

import java.util.ArrayList;

import java.util.List;

public interface Observer {

void update();

}

public class ConcreteObserver implements Observer {

private String name;

public ConcreteObserver(String name) {

this.name = name;

}

@Override

public void update() {

System.out.println(name + " received an update!");

}

}

public interface Subject {

void attach(Observer observer);

void detach(Observer observer);

void notifyObservers();

}

public class ConcreteSubject implements Subject {

private List<Observer> observers = new ArrayList<>();

@Override

public void attach(Observer observer) {

observers.add(observer);

}

@Override

public void detach(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update();

}

}

}

**State Pattern:**

Definition: This pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

Example: A vending machine that changes its behavior based on its current state (e.g., idle, accepting coins, dispensing product).

public interface State {

void handleRequest();

}

public class ConcreteStateA implements State {

@Override

public void handleRequest() {

System.out.println("Handling request in State A");

}

}

public class ConcreteStateB implements State {

@Override

public void handleRequest() {

System.out.println("Handling request in State B");

}

}

public class Context {

private State state;

public void setState(State state) {

this.state = state;

}

public void handleRequest() {

state.handleRequest();

}

}

// Usage

public class Client {

public static void main(String[] args) {

Context context = new Context();

State stateA = new ConcreteStateA();

State stateB = new ConcreteStateB();

context.setState(stateA);

context.handleRequest();

context.setState(stateB);

context.handleRequest();

}

}

**Strategy Pattern:**

Definition: This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

Example: Sorting algorithms (e.g., bubble sort, merge sort) that can be selected and used interchangeably based on requirements.

public interface Strategy {

void execute();

}

public class ConcreteStrategyA implements Strategy {

@Override

public void execute() {

System.out.println("Executing strategy A");

}

}

public class ConcreteStrategyB implements Strategy {

@Override

public void execute() {

System.out.println("Executing strategy B");

}

}

public class Context {

private Strategy strategy;

public Context(Strategy strategy) {

this.strategy = strategy;

}

public void executeStrategy() {

strategy.execute();

}

}

**Template Method Pattern:**

Definition: This pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. It allows subclasses to redefine certain steps of an algorithm without changing its structure.

Example: A framework for building different types of reports where the overall structure is defined, but specific steps can be implemented differently for different types of reports.

public abstract class AbstractClass {

public void templateMethod() {

step1();

step2();

step3();

}

protected abstract void step1();

protected abstract void step2();

protected abstract void step3();

}

public class ConcreteClass extends AbstractClass {

@Override

protected void step1() {

System.out.println("ConcreteClass: Step 1");

}

@Override

protected void step2() {

System.out.println("ConcreteClass: Step 2");

}

@Override

protected void step3() {

System.out.println("ConcreteClass: Step 3");

}

}

**Visitor Pattern:**

Definition: This pattern represents an operation to be performed on elements of an object structure. It lets you define a new operation without changing the classes of the elements on which it operates.

Example: A document structure with different types of elements (e.g., paragraphs, headers) that can be visited by a spell-checking visitor or a formatting visitor.

These behavioural design patterns provide solutions for effective communication between objects and simplify the design by making objects more flexible and reusable.

**STRUCTURAL DESIGN PATTERNS**

**Adapter Pattern:**

Definition: This pattern allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code.

Example: Adapting a European electrical plug to fit into a North American electrical outlet using a plug adapter.

public interface Target {

void request();

}

public class Adaptee {

public void specificRequest() {

System.out.println("Adaptee's specific request");

}

}

public class Adapter implements Target {

private Adaptee adaptee;

public Adapter(Adaptee adaptee) {

this.adaptee = adaptee;

}

@Override

public void request() {

adaptee.specificRequest();

}

}

// Usage

public class Client {

public static void main(String[] args) {

Adaptee adaptee = new Adaptee();

Target target = new Adapter(adaptee);

target.request();

}

}

**Bridge Pattern:**

Definition: The bridge pattern decouples an abstraction from its implementation so that the two can vary independently. It involves an interface and a concrete implementation that can be changed easily without affecting each other.

Example: A shape drawing application where different shapes (abstraction) can be drawn using different drawing tools (implementation).

public interface Implementor {

void operationImpl();

}

public class ConcreteImplementorA implements Implementor {

@Override

public void operationImpl() {

System.out.println("Implementor A operation");

}

}

public class ConcreteImplementorB implements Implementor {

@Override

public void operationImpl() {

System.out.println("Implementor B operation");

}

}

public abstract class Abstraction {

protected Implementor implementor;

public Abstraction(Implementor implementor) {

this.implementor = implementor;

}

public abstract void operation();

}

public class RefinedAbstraction extends Abstraction {

public RefinedAbstraction(Implementor implementor) {

super(implementor);

}

@Override

public void operation() {

implementor.operationImpl();

}

}

// Usage

public class Client {

public static void main(String[] args) {

Implementor implementorA = new ConcreteImplementorA();

Implementor implementorB = new ConcreteImplementorB();

Abstraction abstractionA = new RefinedAbstraction(implementorA);

Abstraction abstractionB = new RefinedAbstraction(implementorB);

abstractionA.operation();

abstractionB.operation();

}

}

**Composite Pattern:**

Definition: The composite pattern composes objects into tree structures to represent part-whole hierarchies. It allows clients to treat individual objects and compositions of objects uniformly.

Example: Representing a company's organizational structure as a tree, with departments and employees as nodes.

import java.util.ArrayList;

import java.util.List;

public abstract class Component {

protected String name;

public Component(String name) {

this.name = name;

}

public abstract void operation();

public abstract void add(Component component);

public abstract void remove(Component component);

public abstract List<Component> getChildren();

}

public class Leaf extends Component {

public Leaf(String name) {

super(name);

}

@Override

public void operation() {

System.out.println("Leaf " + name + " operation");

}

@Override

public void add(Component component) {

// Leaf cannot add components

}

@Override

public void remove(Component component) {

// Leaf cannot remove components

}

@Override

public List<Component> getChildren() {

// Leaf does not have any children

return new ArrayList<>();

}

}

public class Composite extends Component {

private List<Component> children = new ArrayList<>();

public Composite(String name) {

super(name);

}

@Override

public void operation() {

System.out.println("Composite " + name + " operation");

for (Component component : children) {

component.operation();

}

}

@Override

public void add(Component component) {

children.add(component);

}

@Override

public void remove(Component component) {

children.remove(component);

}

@Override

public List<Component> getChildren() {

return children;

}

}

// Usage

public class Client {

public static void main(String[] args) {

Component leafA = new Leaf("Leaf A");

Component leafB = new Leaf("Leaf B");

Component composite = new Composite("Composite");

composite.add(leafA);

composite.add(leafB);

composite.operation();

}

}

**Decorator Pattern:**

Definition: The decorator pattern attaches additional responsibilities to an object dynamically. It provides a flexible alternative to subclassing for extending functionality.

Example: Adding toppings to a pizza without creating a new subclass for each combination of toppings.

public interface Component {

void operation();

}

public class ConcreteComponent implements Component {

@Override

public void operation() {

System.out.println("ConcreteComponent operation");

}

}

public abstract class Decorator implements Component {

protected Component component;

public Decorator(Component component) {

this.component = component;

}

@Override

public void operation() {

component.operation();

}

}

public class ConcreteDecoratorA extends Decorator {

public ConcreteDecoratorA(Component component) {

super(component);

}

@Override

public void operation() {

super.operation();

System.out.println("ConcreteDecoratorA operation");

}

}

public class ConcreteDecoratorB extends Decorator {

public ConcreteDecoratorB(Component component) {

super(component);

}

@Override

public void operation() {

super.operation();

System.out.println("ConcreteDecoratorB operation");

}

}

// Usage

public class Client {

public staticHere is the continuation of the Decorator Pattern example:

```java

public class Client {

public static void main(String[] args) {

Component component = new ConcreteComponent();

Component decoratorA = new ConcreteDecoratorA(component);

Component decoratorB = new ConcreteDecoratorB(decoratorA);

decoratorB.operation();

}

}

**Facade Pattern:**

Definition: The facade pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

Example: A computer user interface that provides simplified access to complex system operations.

public class SubsystemA {

public void operationA() {

System.out.println("SubsystemA: Operation A");

}

}

public class SubsystemB {

public void operationB() {

System.out.println("SubsystemB: Operation B");

}

}

public class SubsystemC {

public void operationC() {

System.out.println("SubsystemC: Operation C");

}

}

public class Facade {

private SubsystemA subsystemA;

private SubsystemB subsystemB;

private SubsystemC subsystemC;

public Facade() {

subsystemA = new SubsystemA();

subsystemB = new SubsystemB();

subsystemC = new SubsystemC();

}

public void operation() {

subsystemA.operationA();

subsystemB.operationB();

subsystemC.operationC();

}

}

// Usage

public class Client {

public static void main(String[] args) {

Facade facade = new Facade();

facade.operation();

}

}

**Flyweight Pattern:**

Definition: The flyweight pattern uses sharing to support large numbers of fine-grained objects efficiently. It is used when the quantity of objects to be managed is so large that it would be inefficient to store and manipulate them individually.

Example: Storing and reusing font objects in a word processing application to reduce memory usage.

import java.util.HashMap;

import java.util.Map;

public class FlyweightFactory {

private Map<String, Flyweight> flyweights = new HashMap<>();

public Flyweight getFlyweight(String key) {

if (flyweights.containsKey(key)) {

return flyweights.get(key);

} else {

Flyweight flyweight = new ConcreteFlyweight(key);

flyweights.put(key, flyweight);

return flyweight;

}

}

}

public interface Flyweight {

void operation();

}

public class ConcreteFlyweight implements Flyweight {

private String key;

public ConcreteFlyweight(String key) {

this.key = key;

}

@Override

public void operation() {

System.out.println("ConcreteFlyweight with key: " + key);

}

}

// Usage

public class Client {

public static void main(String[] args) {

FlyweightFactory factory = new FlyweightFactory();

Flyweight flyweight1 = factory.getFlyweight("key1");

Flyweight flyweight2 = factory.getFlyweight("key2");

Flyweight flyweight3 = factory.getFlyweight("key1");

flyweight1.operation();

flyweight2.operation();

flyweight3.operation();

}

}

**Proxy Pattern:**

Definition: The proxy pattern provides a surrogate or placeholder for another object to control access to it.

Example: A remote proxy that acts as a local representative for an object in a different address space.

These structural design patterns provide solutions for organizing and managing relationships between classes and objects in a flexible and efficient manner.

public interface Image {

void display();

}

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading image: " + filename);

}

@Override

public void display() {

System.out.println("Displaying image: " + filename);

}

}

public class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

// Usage

public class Client {

public static void main(String[] args) {

Image image1 = new ProxyImage("image1.jpg");

Image image2 = new ProxyImage("image2.jpg");

image1.display(); // Real image is loaded and displayed

image1.display(); // Real image is already loaded, so it is displayed again

image2.display(); // Real image is loaded and displayed

}

}