Week 1. Revision of Object-Oriented Programming Concepts Week 2. Creational Design Patterns

**Aim/Objective**: Testing the ability to apply various design patterns effectively, as well as understanding their roles and interactions in a real-world context.

1. In a Java application that requires license validation, explain how you can use the Singleton Design Pattern to implement a License Manager. Describe the benefits of using a Singleton for managing the license state and validation process. Provide a sample implementation of a License Manager class that ensures only one instance is responsible for checking and managing license status throughout the application.

**Code : Class LicenseManager**

public class LicenseManager {

private static LicenseManager instance;

private boolean isLicenseValid;

private String licenseKey;

private LicenseManager() {

this.isLicenseValid = false;

this.licenseKey = null;

}

public static synchronized LicenseManager getInstance() {

if (instance == null) {

instance = new LicenseManager();

}

return instance;

}

public boolean validateLicense(String key) {

if (key != null && key.equals("VALID\_LICENSE\_KEY")) {

this.isLicenseValid = true;

this.licenseKey = key;

} else {

this.isLicenseValid = false;

}

return this.isLicenseValid;

}

public boolean isLicenseValid() {

return this.isLicenseValid;

}

public String getLicenseKey() {

return this.licenseKey;

}

public static void main(String[] args) {

LicenseManager licenseManager = LicenseManager.getInstance();

String sampleKey = "VALID\_LICENSE\_KEY";

boolean isValid = licenseManager.validateLicense(sampleKey);

System.out.println("License Valid: " + isValid);

System.out.println("Current License Key: " + licenseManager.getLicenseKey());

isValid = licenseManager.validateLicense("INVALID\_KEY");

System.out.println("License Valid after invalid attempt: " + isValid);

System.out.println("Current License Key: " + licenseManager.getLicenseKey());

}

}

**Output**

A screenshot of a computer program

Description automatically generated

1. In a Java application for handling banking transactions, how can the Singleton Design Pattern be used to create a BankingSystemManager? Explain how this pattern helps ensure there is only one instance managing banking operations like account balance and transactions. Provide a clear and simple example of a BankingSystemManager class that uses Singleton to guarantee a single instance and handle operations safely in a multi-threaded environment.

**Code : Class BankingSystemManager**

import java.util.ArrayList;

import java.util.List;

public class BankingSystemManager {

private static volatile BankingSystemManager instance;

private double balance;

private List<String> transactions;

private BankingSystemManager() {

balance = 0.0;

transactions = new ArrayList<>();

}

public static BankingSystemManager getInstance() {

if (instance == null) {

synchronized (BankingSystemManager.class) {

if (instance == null) {

instance = new BankingSystemManager();

}

}

}

return instance;

}

public synchronized void setInitialBalance(double initialBalance) {

balance = initialBalance;

transactions.add("Account created with balance: " + initialBalance);

}

public synchronized double getBalance() {

return balance;

}

public synchronized void performTransaction(double amount) {

balance += amount;

transactions.add("Transaction of " + amount + " | New Balance: " + balance);

}

public synchronized List<String> getTransactionHistory() {

return transactions;

}

public static void main(String[] args) {

BankingSystemManager manager = BankingSystemManager.getInstance();

manager.setInitialBalance(1000.00);

System.out.println("Initial Balance: " + manager.getBalance());

manager.performTransaction(-200.00);

manager.performTransaction(500.00);

System.out.println("Updated Balance: " + manager.getBalance());

System.out.println("\nTransaction History:");

for (String transaction : manager.getTransactionHistory()) {

System.out.println(transaction);

}

}

}

**Output**

A screen shot of a computer

Description automatically generated

1. In a reporting application, users need to generate reports in different formats such as PDF, Excel, and HTML. Each report format requires a different implementation for generating the report.
   1. Design a ReportFactory class that uses the Factory Design Pattern to create instances of the appropriate Report type based on user input.
   2. Provide the code implementation for the Report, PDFReport, ExcelReport, and HTMLReport classes.
   3. Explain how the Factory Design Pattern helps in managing the creation of different report types and how it enhances the flexibility and scalability of the reporting system.

**Code : Class ReportFactoryExample**

public class ReportFactoryExample {

abstract static class Report {

public abstract String generate();

}

static class PDFReport extends Report {

@Override

public String generate() {

return "Generating PDF report";

}

}

static class ExcelReport extends Report {

@Override

public String generate() {

return "Generating Excel report";

}

}

static class HTMLReport extends Report {

@Override

public String generate() {

return "Generating HTML report";

}

}

static class ReportFactory {

public Report createReport(String reportType) {

if (reportType.equalsIgnoreCase("PDF")) {

return new PDFReport();

} else if (reportType.equalsIgnoreCase("Excel")) {

return new ExcelReport();

} else if (reportType.equalsIgnoreCase("HTML")) {

return new HTMLReport();

} else {

throw new IllegalArgumentException("Unknown report type");

}

}

}

public static void main(String[] args) {

ReportFactory factory = new ReportFactory();

Report pdfReport = factory.createReport("PDF");

System.out.println(pdfReport.generate());

Report excelReport = factory.createReport("Excel");

System.out.println(excelReport.generate());

Report htmlReport = factory.createReport("HTML");

System.out.println(htmlReport.generate());

}

}

**Output**

A screenshot of a computer

Description automatically generated

1. You are tasked with implementing a calculator that evaluates postfix expressions (Reverse Polish Notation). The postfix expression will be provided as input, and the output should be the result of the expression.
   1. Define an abstract Operation class that represents a generic operation (e.g., addition, subtraction, multiplication, division) for evaluating postfix expressions. Implement concrete classes for each specific operation (e.g., Addition, Subtraction, Multiplication, Division).
   2. Create a OperationFactory class that uses the Factory Design Pattern to instantiate the appropriate Operation objects based on the operator encountered in the postfix expression.
   3. Implement a PostfixEvaluator class that utilizes the OperationFactory to evaluate a given postfix expression. The class should handle the input string (e.g., "7 3 - 2 1 + \*") and produce the correct output (e.g., 12).
   4. Explain how the Factory Design Pattern is utilized in your implementation to handle different operations dynamically and how it contributes to the design's flexibility and maintainability.

**Code : Class PostfixCalculator**

import java.util.Stack;

public class PostfixCalculator {

interface Operation {

double execute(double operand1, double operand2);

}

static class Factory {

static Operation getOperation(String operator) {

return switch (operator) {

case "+" -> (a, b) -> a + b;

case "-" -> (a, b) -> a - b;

case "\*" -> (a, b) -> a \* b;

case "/" -> (a, b) -> a / b;

default -> throw new IllegalArgumentException("Unknown operator");

};

}

}

public static double evaluate(String expression) {

Stack<Double> stack = new Stack<>();

for (String token : expression.split(" ")) {

if ("+-\*/".contains(token)) {

double b = stack.pop(), a = stack.pop();

stack.push(Factory.getOperation(token).execute(a, b));

} else {

stack.push(Double.parseDouble(token));

}

}

return stack.pop();

}

public static void main(String[] args) {

String expression = "7 3 - 2 1 + \*";

double result = evaluate(expression);

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

}

}

**Output**



1. In the context of developing a sophisticated vehicle manufacturing system that produces different types of vehicles such as cars, trucks, and motorcycles, each with its own distinct set of components and assembly processes, how can the Abstract Factory design pattern be utilized to manage the creation of these components?
   1. Design an abstract factory interface VehicleFactory for this system. What methods should this interface include to facilitate the creation of components specific to each type of vehicle?
   2. Implement concrete factories for each type of vehicle: CarFactory, TruckFactory, and MotorcycleFactory. Describe the specific components each factory should produce and how these components differ from one another.
   3. Discuss how the Abstract Factory pattern helps in achieving separation of concerns in the vehicle manufacturing system. What advantages does it offer in terms of scalability and flexibility when adding new vehicle types or components?
   4. Provide a code example of how a client class can utilize the VehicleFactory interface and its concrete implementations to assemble a vehicle. Demonstrate how the client can work with different factories to assemble different types of vehicles without changing the core logic.

**Code : Class VehicleAssemblyClient**

interface VehicleFactory {

String createEngine();

String createWheels();

String createChassis();

String createSeats();

}

class CarFactory implements VehicleFactory {

public String createEngine() {

return "CarEngine";

}

public String createWheels() {

return "CarWheels";

}

public String createChassis() {

return "CarChassis";

}

public String createSeats() {

return "CarSeats";

}

}

class TruckFactory implements VehicleFactory {

public String createEngine() {

return "TruckEngine";

}

public String createWheels() {

return "TruckWheels";

}

public String createChassis() {

return "TruckChassis";

}

public String createSeats() {

return "TruckSeats";

}

}

class MotorcycleFactory implements VehicleFactory {

public String createEngine() {

return "MotorcycleEngine";

}

public String createWheels() {

return "MotorcycleWheels";

}

public String createChassis() {

return "MotorcycleChassis";

}

public String createSeats() {

return "MotorcycleSeats";

}

}

public class VehicleAssemblyClient {

public static void assembleVehicle(VehicleFactory factory) {

System.out.println("Assembling vehicle with components:");

System.out.println("Engine: " + factory.createEngine());

System.out.println("Wheels: " + factory.createWheels());

System.out.println("Chassis: " + factory.createChassis());

System.out.println("Seats: " + factory.createSeats());

}

public static void main(String[] args) {

assembleVehicle(new CarFactory());

assembleVehicle(new TruckFactory());

assembleVehicle(new MotorcycleFactory());

}

}

**Output**

A screenshot of a computer program

Description automatically generated

1. Design a Logger Management System in Java that utilizes both the Singleton and Factory design patterns. In this system:
   1. **Singleton Pattern**: Ensure that there is only one instance of the LoggerManager class throughout the application. This class will manage the logging system and provide access to various types of loggers.
   2. **Factory Pattern**: Implement a factory method within the LoggerManager that creates and returns different types of loggers (e.g., FileLogger, ConsoleLogger) based on the specified type.

In your design:

1. Describe the role of the LoggerManager class and how the Singleton pattern is applied to it.
2. Explain how the Factory pattern is used within the LoggerManager to create different types of loggers.
3. Provide a code implementation for the LoggerManager class, the Logger interface, and at least two concrete implementations of loggers (e.g., FileLogger and ConsoleLogger).
4. Discuss how you would use this LoggerManager in a real-world application to ensure efficient and consistent logging.

**Code : Class LoggerApp**

interface Logger {

void log(String message);

}

class FileLogger implements Logger {

@Override

public void log(String message) {

System.out.println("Logging to file: " + message);

}

}

class ConsoleLogger implements Logger {

@Override

public void log(String message) {

System.out.println("Logging to console: " + message);

}

}

class LoggerManager {

private static LoggerManager instance;

private LoggerManager() {}

public static LoggerManager getInstance() {

if (instance == null) {

synchronized (LoggerManager.class) {

if (instance == null) {

instance = new LoggerManager();

}

}

}

return instance;

}

public Logger getLogger(String loggerType) {

if (loggerType.equalsIgnoreCase("file")) {

return new FileLogger();

} else if (loggerType.equalsIgnoreCase("console")) {

return new ConsoleLogger();

}

throw new IllegalArgumentException("Invalid logger type");

}

}

public class LoggerApp {

public static void main(String[] args) {

LoggerManager loggerManager = LoggerManager.getInstance();

Logger fileLogger = loggerManager.getLogger("file");

fileLogger.log("This is a file log message.");

Logger consoleLogger = loggerManager.getLogger("console");

consoleLogger.log("This is a console log message.");

}

}

**Output**

