**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

SingletonTest.java

public class SingletonTest {

    public static void main(String[] args) {

        // Test 1: Get first instance

        Logger logger1 = Logger.getInstance();

        logger1.info("First logger instance created");

        // Test 2: Get second instance

        Logger logger2 = Logger.getInstance();

        logger2.info("Second logger instance requested");

        // Test 3: Compare instances

        System.out.println("Are both instances the same? " + (logger1 == logger2));

        System.out.println("Logger1 hashcode: " + logger1.hashCode());

        System.out.println("Logger2 hashcode: " + logger2.hashCode());

        // Test 4: Use logger in different parts

        testInDifferentMethods();

        // Test 5: Multiple operations

        performMultipleOperations();

    }

    public static void testInDifferentMethods() {

        Logger methodLogger = Logger.getInstance();

        methodLogger.log("Testing from different method");

    }

    public static void performMultipleOperations() {

        Logger opLogger = Logger.getInstance();

        opLogger.info("Starting operations");

        opLogger.log("Operation 1 completed");

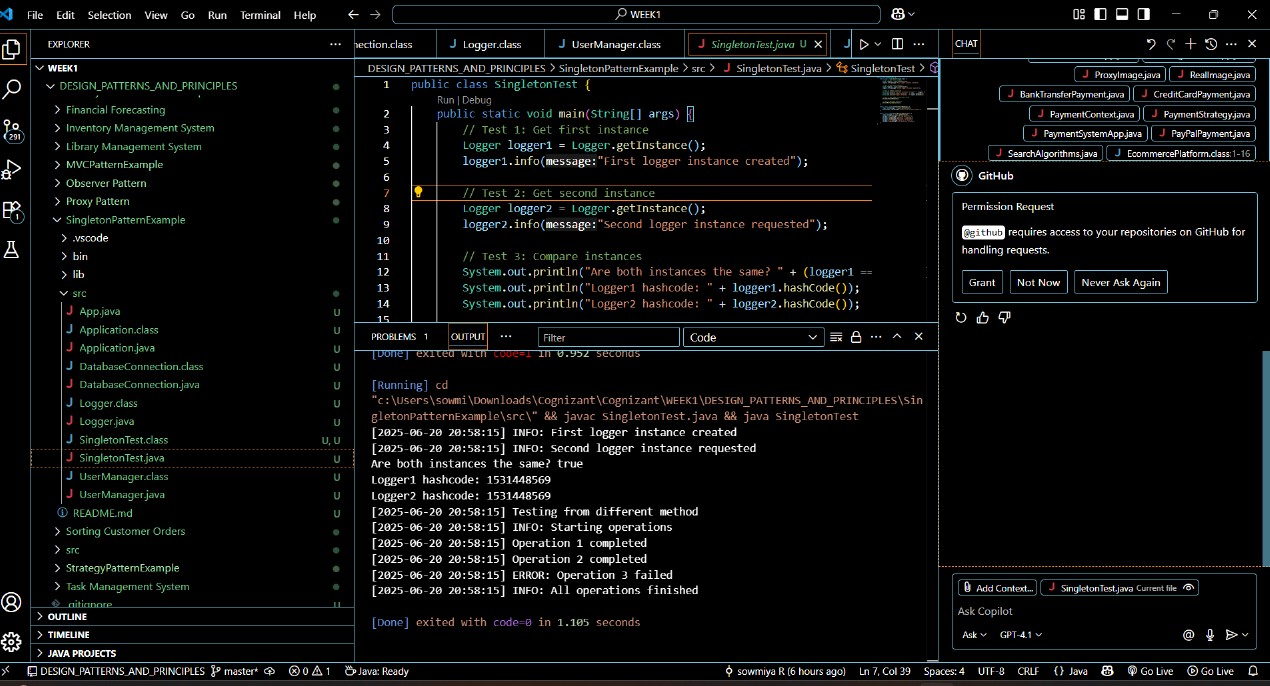
        opLogger.log("Operation 2 completed");

        opLogger.error("Operation 3 failed");

        opLogger.info("All operations finished");

    }

}



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

package FactoryMethodPatternExample;

// 1. Document interface - defines common behavior for all documents

interface Document {

void create();

void open();

void save();

void close();

String getDocumentType();

}

// 2. Concrete Document Classes

class WordDocument implements Document {

@Override

public void create() {

System.out.println("Creating a new Word document...");

}

@Override

public void open() {

System.out.println("Opening Word document with Microsoft Word...");

}

@Override

public void save() {

System.out.println("Saving Word document as .docx file...");

}

@Override

public void close() {

System.out.println("Closing Word document...");

}

@Override

public String getDocumentType() {

return "Word Document (.docx)";

}

}

class PdfDocument implements Document {

@Override

public void create() {

System.out.println("Creating a new PDF document...");

}

@Override

public void open() {

System.out.println("Opening PDF document with PDF reader...");

}

@Override

public void save() {

System.out.println("Saving PDF document as .pdf file...");

}

@Override

public void close() {

System.out.println("Closing PDF document...");

}

@Override

public String getDocumentType() {

return "PDF Document (.pdf)";

}

}

class ExcelDocument implements Document {

@Override

public void create() {

System.out.println("Creating a new Excel document...");

}

@Override

public void open() {

System.out.println("Opening Excel document with Microsoft Excel...");

}

@Override

public void save() {

System.out.println("Saving Excel document as .xlsx file...");

}

@Override

public void close() {

System.out.println("Closing Excel document...");

}

@Override

public String getDocumentType() {

return "Excel Document (.xlsx)";

}

}

// 3. Abstract Factory Class

abstract class DocumentFactory {

// Factory method - to be implemented by concrete factories

public abstract Document createDocument();

// Template method that uses the factory method

public Document processDocument() {

Document document = createDocument();

document.create();

return document;

}

}

// 4. Concrete Factory Classes

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

// 5. Document Type Enum for better type safety

enum DocumentType {

WORD, PDF, EXCEL

}

// 6. Factory Provider Class (Optional - for easier factory selection)

class DocumentFactoryProvider {

public static DocumentFactory getFactory(DocumentType type) {

switch (type) {

case WORD:

return new WordDocumentFactory();

case PDF:

return new PdfDocumentFactory();

case EXCEL:

return new ExcelDocumentFactory();

default:

throw new IllegalArgumentException("Unknown document type: " + type);

}

}

}

// 7. Test Class - Main Application

public class FactoryMethodPatternExample {

public static void main(String[] args) {

System.out.println("=== Factory Method Pattern Demo ===\n");

// Method 1: Using concrete factories directly

System.out.println("1. Creating documents using concrete factories:");

System.out.println("-".repeat(50));

// Create Word document

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.processDocument();

System.out.println("Document type: " + wordDoc.getDocumentType());

wordDoc.open();

wordDoc.save();

wordDoc.close();

System.out.println();

// Create PDF document

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.processDocument();

System.out.println("Document type: " + pdfDoc.getDocumentType());

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

System.out.println();

// Create Excel document

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.processDocument();

System.out.println("Document type: " + excelDoc.getDocumentType());

excelDoc.open();

excelDoc.save();

excelDoc.close();

System.out.println("\n" + "=".repeat(60) + "\n");

// Method 2: Using factory provider for cleaner code

System.out.println("2. Creating documents using factory provider:");

System.out.println("-".repeat(50));

DocumentType[] types = {DocumentType.WORD, DocumentType.PDF, DocumentType.EXCEL};

for (DocumentType type : types) {

try {

DocumentFactory factory = DocumentFactoryProvider.getFactory(type);

Document document = factory.createDocument();

System.out.println("Processing " + document.getDocumentType() + ":");

document.create();

document.open();

document.save();

document.close();

System.out.println();

} catch (IllegalArgumentException e) {

System.err.println("Error: " + e.getMessage());

}

}

System.out.println("=".repeat(60));

// Method 3: Demonstrating polymorphism

System.out.println("\n3. Demonstrating polymorphic behavior:");

System.out.println("-".repeat(50));

DocumentFactory[] factories = {

new WordDocumentFactory(),

new PdfDocumentFactory(),

new ExcelDocumentFactory()

};

for (DocumentFactory factory : factories) {

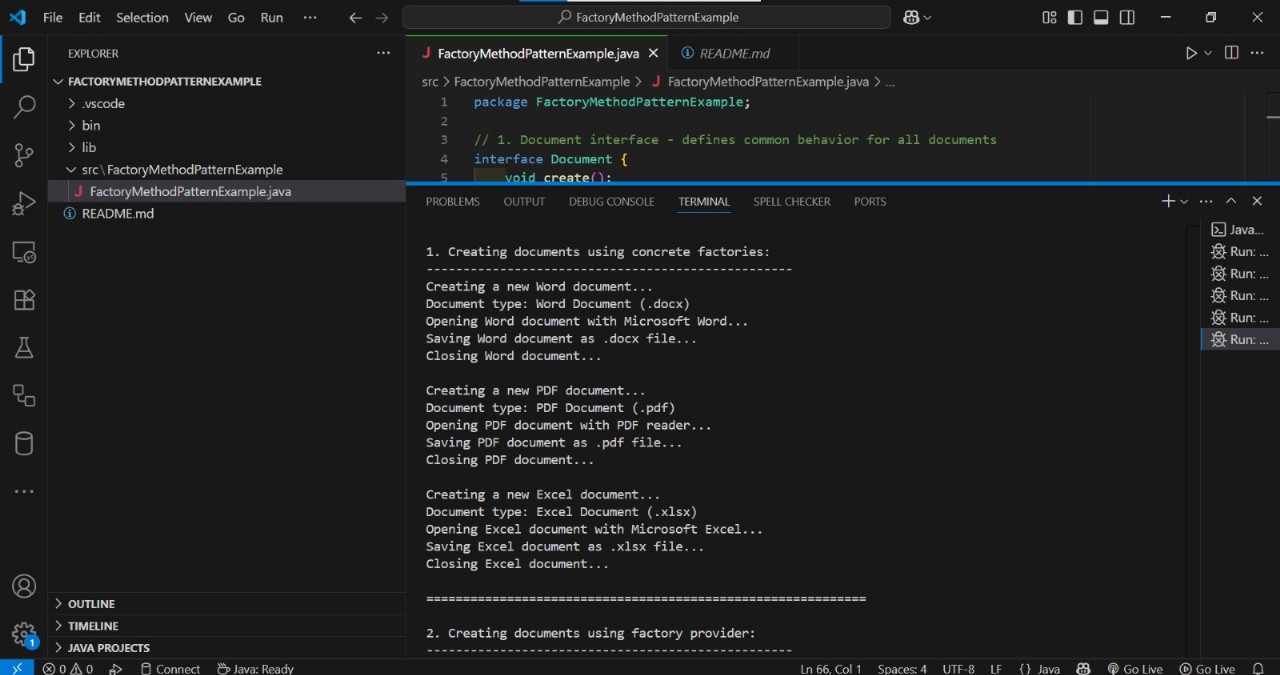
Document doc = factory.createDocument();

System.out.println("Created: " + doc.getDocumentType());

}

}

}



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

BuilderPatterExample.java

import java.util.ArrayList;

import java.util.List;

// 1. Product Class - Computer with multiple optional components

class Computer {

    // Required parameters

    private final String cpu;

    private final String motherboard;

    // Optional parameters

    private final String ram;

    private final String storage;

    private final String graphicsCard;

    private final String powerSupply;

    private final String coolingSystem;

    private final String caseType;

    private final List<String> peripherals;

    private final boolean isGamingPC;

    private final boolean hasWiFi;

    private final boolean hasBluetooth;

    private final String operatingSystem;

    private final int warrantyYears;

    // Private constructor - only accessible through Builder

    private Computer(Builder builder) {

        // Required parameters

        this.cpu = builder.cpu;

        this.motherboard = builder.motherboard;

        // Optional parameters

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.powerSupply = builder.powerSupply;

        this.coolingSystem = builder.coolingSystem;

        this.caseType = builder.caseType;

        this.peripherals = new ArrayList<>(builder.peripherals);

        this.isGamingPC = builder.isGamingPC;

        this.hasWiFi = builder.hasWiFi;

        this.hasBluetooth = builder.hasBluetooth;

        this.operatingSystem = builder.operatingSystem;

        this.warrantyYears = builder.warrantyYears;

    }

    // Getter methods

    public String getCpu() { return cpu; }

    public String getMotherboard() { return motherboard; }

    public String getRam() { return ram; }

    public String getStorage() { return storage; }

    public String getGraphicsCard() { return graphicsCard; }

    public String getPowerSupply() { return powerSupply; }

    public String getCoolingSystem() { return coolingSystem; }

    public String getCaseType() { return caseType; }

    public List<String> getPeripherals() { return new ArrayList<>(peripherals); }

    public boolean isGamingPC() { return isGamingPC; }

    public boolean hasWiFi() { return hasWiFi; }

    public boolean hasBluetooth() { return hasBluetooth; }

    public String getOperatingSystem() { return operatingSystem; }

    public int getWarrantyYears() { return warrantyYears; }

    // 2. Static nested Builder class

    public static class Builder {

        // Required parameters

        private String cpu;

        private String motherboard;

        // Optional parameters with default values

        private String ram = "8GB DDR4";

        private String storage = "256GB SSD";

        private String graphicsCard = "Integrated";

        private String powerSupply = "500W";

        private String coolingSystem = "Stock Cooler";

        private String caseType = "Mid Tower";

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

        private boolean isGamingPC = false;

        private boolean hasWiFi = true;

        private boolean hasBluetooth = false;

        private String operatingSystem = "Windows 11";

        private int warrantyYears = 1;

        // Constructor with required parameters

        public Builder(String cpu, String motherboard) {

            this.cpu = cpu;

            this.motherboard = motherboard;

        }

        // Builder methods for optional parameters

        public Builder ram(String ram) {

            this.ram = ram;

            return this;

        }

        public Builder storage(String storage) {

            this.storage = storage;

            return this;

        }

        public Builder graphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        public Builder powerSupply(String powerSupply) {

            this.powerSupply = powerSupply;

            return this;

        }

        public Builder coolingSystem(String coolingSystem) {

            this.coolingSystem = coolingSystem;

            return this;

        }

        public Builder caseType(String caseType) {

            this.caseType = caseType;

            return this;

        }

        public Builder addPeripheral(String peripheral) {

            this.peripherals.add(peripheral);

            return this;

        }

        public Builder peripherals(List<String> peripherals) {

            this.peripherals = new ArrayList<>(peripherals);

            return this;

        }

        public Builder isGamingPC(boolean isGamingPC) {

            this.isGamingPC = isGamingPC;

            return this;

        }

        public Builder hasWiFi(boolean hasWiFi) {

            this.hasWiFi = hasWiFi;

            return this;

        }

        public Builder hasBluetooth(boolean hasBluetooth) {

            this.hasBluetooth = hasBluetooth;

            return this;

        }

        public Builder operatingSystem(String operatingSystem) {

            this.operatingSystem = operatingSystem;

            return this;

        }

        public Builder warrantyYears(int warrantyYears) {

            this.warrantyYears = warrantyYears;

            return this;

        }

        // Build method that creates and returns Computer instance

        public Computer build() {

            // Optional: Add validation logic here

            validateComputer();

            return new Computer(this);

        }

        // Private validation method

        private void validateComputer() {

            if (cpu == null || cpu.trim().isEmpty()) {

                throw new IllegalStateException("CPU is required");

            }

            if (motherboard == null || motherboard.trim().isEmpty()) {

                throw new IllegalStateException("Motherboard is required");

            }

            if (warrantyYears < 0 || warrantyYears > 10) {

                throw new IllegalStateException("Warranty years must be between 0 and 10");

            }

        }

    }

    // toString method for easy display

    @Override

    public String toString() {

        StringBuilder sb = new StringBuilder();

        sb.append("Computer Configuration:\n");

        sb.append("├── CPU: ").append(cpu).append("\n");

        sb.append("├── Motherboard: ").append(motherboard).append("\n");

        sb.append("├── RAM: ").append(ram).append("\n");

        sb.append("├── Storage: ").append(storage).append("\n");

        sb.append("├── Graphics Card: ").append(graphicsCard).append("\n");

        sb.append("├── Power Supply: ").append(powerSupply).append("\n");

        sb.append("├── Cooling System: ").append(coolingSystem).append("\n");

        sb.append("├── Case Type: ").append(caseType).append("\n");

        sb.append("├── Operating System: ").append(operatingSystem).append("\n");

        sb.append("├── Gaming PC: ").append(isGamingPC ? "Yes" : "No").append("\n");

        sb.append("├── WiFi: ").append(hasWiFi ? "Yes" : "No").append("\n");

        sb.append("├── Bluetooth: ").append(hasBluetooth ? "Yes" : "No").append("\n");

        sb.append("├── Warranty: ").append(warrantyYears).append(" year(s)\n");

        sb.append("└── Peripherals: ");

        if (peripherals.isEmpty()) {

            sb.append("None");

        } else {

            sb.append(String.join(", ", peripherals));

        }

        return sb.toString();

    }

}

// 3. Pre-configured Computer Builder Classes (Optional Extension)

class GamingComputerBuilder {

    public static Computer.Builder getGamingBuilder() {

        return new Computer.Builder("Intel Core i7-13700K", "ASUS ROG Strix Z790")

                .ram("32GB DDR5")

                .storage("1TB NVMe SSD")

                .graphicsCard("NVIDIA RTX 4070")

                .powerSupply("750W 80+ Gold")

                .coolingSystem("Liquid Cooling AIO")

                .caseType("Full Tower RGB")

                .isGamingPC(true)

                .hasWiFi(true)

                .hasBluetooth(true)

                .warrantyYears(3);

    }

}

class OfficeComputerBuilder {

    public static Computer.Builder getOfficeBuilder() {

        return new Computer.Builder("Intel Core i5-13400", "MSI Pro B760M")

                .ram("16GB DDR4")

                .storage("512GB SSD")

                .powerSupply("450W 80+ Bronze")

                .caseType("Mini ITX")

                .hasWiFi(true)

                .warrantyYears(2);

    }

}

class BudgetComputerBuilder {

    public static Computer.Builder getBudgetBuilder() {

        return new Computer.Builder("AMD Ryzen 5 5600G", "ASRock B450M")

                .ram("8GB DDR4")

                .storage("256GB SSD")

                .powerSupply("400W")

                .caseType("Micro ATX")

                .warrantyYears(1);

    }

}

// 4. Test Class - Main Application

public class BuilderPatternExample {

    public static void main(String[] args) {

        System.out.println("=== Builder Pattern Demo - Computer Configuration ===\n");

        // Example 1: Basic computer with minimal configuration

        System.out.println("1. Basic Computer Configuration:");

        System.out.println("-".repeat(60));

        Computer basicComputer = new Computer.Builder("Intel Core i3-12100", "MSI H610M")

                .build();

        System.out.println(basicComputer);

        System.out.println();

        // Example 2: Gaming computer with extensive configuration

        System.out.println("2. High-End Gaming Computer:");

        System.out.println("-".repeat(60));

        Computer gamingComputer = new Computer.Builder("AMD Ryzen 9 7900X", "ASUS ROG Crosshair X670E")

                .ram("64GB DDR5-6000")

                .storage("2TB NVMe SSD + 4TB HDD")

                .graphicsCard("NVIDIA RTX 4090")

                .powerSupply("1000W 80+ Platinum")

                .coolingSystem("Custom Water Cooling Loop")

                .caseType("Full Tower Tempered Glass")

                .addPeripheral("Mechanical Gaming Keyboard")

                .addPeripheral("Gaming Mouse")

                .addPeripheral("4K Gaming Monitor")

                .addPeripheral("Gaming Headset")

                .isGamingPC(true)

                .hasWiFi(true)

                .hasBluetooth(true)

                .operatingSystem("Windows 11 Pro")

                .warrantyYears(5)

                .build();

        System.out.println(gamingComputer);

        System.out.println();

        // Example 3: Office computer using method chaining

        System.out.println("3. Office Workstation:");

        System.out.println("-".repeat(60));

        Computer officeComputer = new Computer.Builder("Intel Core i7-13700", "Dell OptiPlex Motherboard")

                .ram("32GB DDR4")

                .storage("1TB SSD")

                .graphicsCard("Intel UHD Graphics")

                .powerSupply("500W")

                .coolingSystem("Stock Intel Cooler")

                .caseType("Small Form Factor")

                .addPeripheral("Wireless Keyboard")

                .addPeripheral("Wireless Mouse")

                .addPeripheral("24-inch Monitor")

                .hasWiFi(true)

                .hasBluetooth(true)

                .operatingSystem("Windows 11 Enterprise")

                .warrantyYears(3)

                .build();

        System.out.println(officeComputer);

        System.out.println();

        // Example 4: Using pre-configured builders

        System.out.println("4. Pre-configured Computer Types:");

        System.out.println("-".repeat(60));

        // Gaming PC with additional customization

        Computer prebuiltGaming = GamingComputerBuilder.getGamingBuilder()

                .addPeripheral("RGB Mechanical Keyboard")

                .addPeripheral("Gaming Mouse Pad")

                .build();

        System.out.println("Gaming PC Configuration:");

        System.out.println(prebuiltGaming);

        System.out.println();

        // Office PC

        Computer officePC = OfficeComputerBuilder.getOfficeBuilder()

                .addPeripheral("Wireless Keyboard & Mouse Set")

                .build();

        System.out.println("Office PC Configuration:");

        System.out.println(officePC);

        System.out.println();

        // Budget PC

        Computer budgetPC = BudgetComputerBuilder.getBudgetBuilder()

                .build();

        System.out.println("Budget PC Configuration:");

        System.out.println(budgetPC);

        System.out.println();

        // Example 5: Demonstrating validation

        System.out.println("5. Builder Validation Demo:");

        System.out.println("-".repeat(60));

        try {

            // This will throw an exception due to invalid warranty years

            Computer invalidComputer = new Computer.Builder("Intel i5", "Generic Motherboard")

                    .warrantyYears(15) // Invalid warranty period

                    .build();

        } catch (IllegalStateException e) {

            System.out.println("Validation Error: " + e.getMessage());

        }

        try {

            // This will throw an exception due to empty CPU

            Computer invalidComputer2 = new Computer.Builder("", "Generic Motherboard")

                    .build();

        } catch (IllegalStateException e) {

            System.out.println("Validation Error: " + e.getMessage());

        }

        System.out.println("\n" + "=".repeat(80));

        System.out.println("Builder Pattern Benefits Demonstrated:");

        System.out.println("• Flexible object construction with optional parameters");

        System.out.println("• Method chaining for readable code");

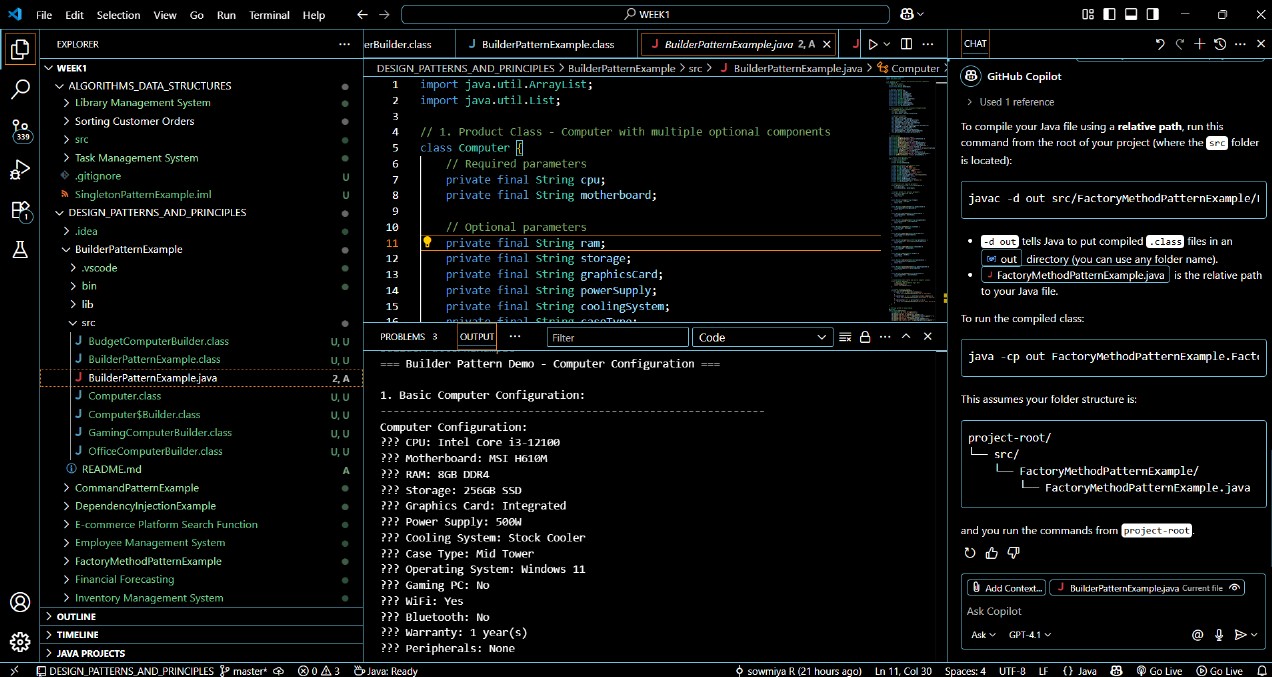
        System.out.println("• Immutable objects (thread-safe)");

        System.out.println("• Built-in validation");

        System.out.println("• Pre-configured builders for common configurations");

    }

}



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

ImageViewerApp.java

public class ImageViewerApp {

    public static void main(String[] args) {

        System.out.println("=== Proxy Pattern Example - Image Viewer ===\n");

        // Create proxy images (no actual loading happens here)

        System.out.println("1. Creating proxy images...");

        Image image1 = new ProxyImage("vacation\_photo.jpg");

        Image image2 = new ProxyImage("family\_portrait.png");

        Image image3 = new ProxyImage("landscape.gif");

        System.out.println("Proxy images created instantly (no loading yet)\n");

        // First display - triggers lazy loading

        System.out.println("2. First display of vacation\_photo.jpg:");

        image1.display();

        System.out.println();

        // Second display - uses cached image

        System.out.println("3. Second display of vacation\_photo.jpg (should be instant):");

        image1.display();

        System.out.println();

        // Display other images

        System.out.println("4. First display of family\_portrait.png:");

        image2.display();

        System.out.println();

        System.out.println("5. First display of landscape.gif:");

        image3.display();

        System.out.println();

        // Display cached images again

        System.out.println("6. Displaying cached images:");

        image1.display();

        System.out.println();

        image2.display();

        System.out.println();

        image3.display();

        System.out.println();

        // Demonstrate proxy status

        demonstrateProxyStatus();

    }

    private static void demonstrateProxyStatus() {

        System.out.println("7. Demonstrating proxy status:");

        ProxyImage proxyImage = new ProxyImage("new\_image.jpg");

        System.out.println("Created proxy for: " + proxyImage.getFilename());

        System.out.println("Is image loaded? " + proxyImage.isLoaded());

        System.out.println("\nDisplaying image for first time:");

        proxyImage.display();

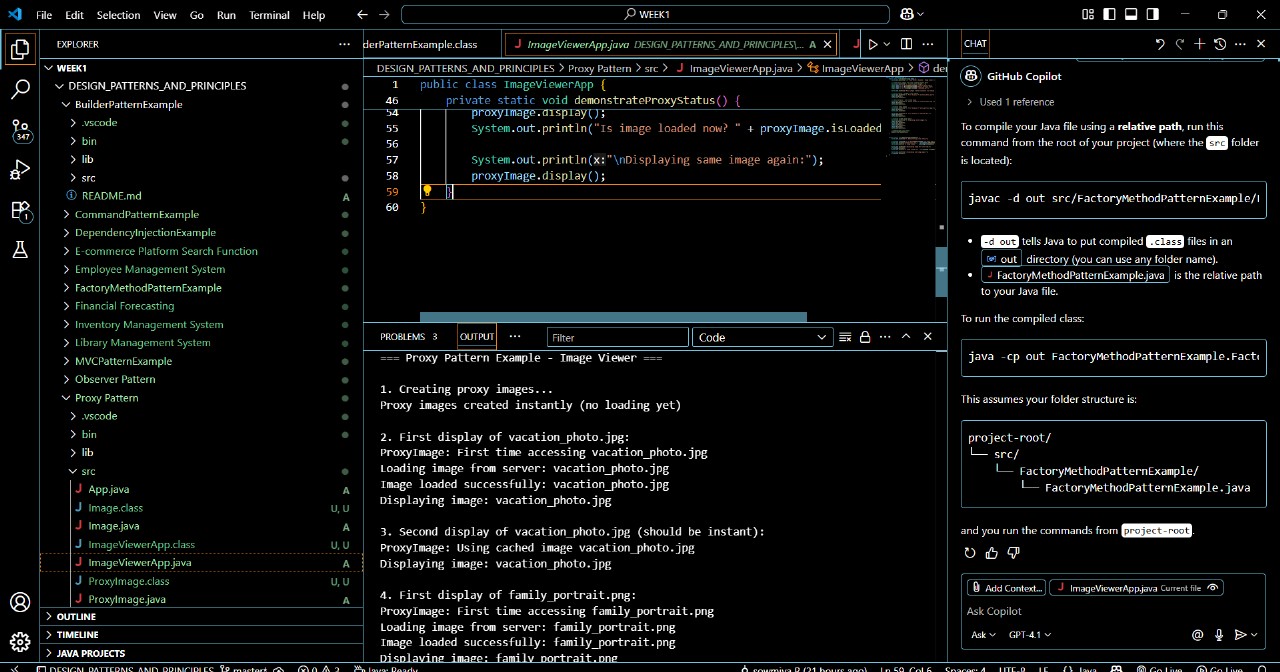
        System.out.println("Is image loaded now? " + proxyImage.isLoaded());

        System.out.println("\nDisplaying same image again:");

        proxyImage.display();

    }

}



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

StockMarketApp.java

import java.util.Map;

public class StockMarketApp {

    public static void main(String[] args) {

        System.out.println("=== Observer Pattern Example - Stock Market Monitor ===\n");

        // Create the subject (stock market)

        StockMarket stockMarket = new StockMarket();

        // Create observers

        System.out.println("1. Creating observers...");

        MobileApp mobileApp1 = new MobileApp("StockTrader Pro", "user123");

        MobileApp mobileApp2 = new MobileApp("InvestMobile", "investor456");

        WebApp webApp1 = new WebApp("TradingPortal.com", "session789");

        WebApp webApp2 = new WebApp("MarketWatch.net", "session101");

        EmailAlert emailAlert = new EmailAlert("trader@example.com", 5.0);

        // Register observers

        System.out.println("\n2. Registering observers...");

        stockMarket.registerObserver(mobileApp1);

        stockMarket.registerObserver(webApp1);

        stockMarket.registerObserver(emailAlert);

        // Test stock price updates

        System.out.println("\n3. Testing stock price updates...");

        stockMarket.setStockPrice("AAPL", 150.00);

        // Add more observers

        System.out.println("\n4. Adding more observers...");

        stockMarket.registerObserver(mobileApp2);

        stockMarket.registerObserver(webApp2);

        // More price updates

        stockMarket.setStockPrice("GOOGL", 2500.00);

        stockMarket.setStockPrice("AAPL", 155.50); // 3.67% increase

        // Test significant price change (triggers email alert)

        stockMarket.setStockPrice("TSLA", 800.00);

        stockMarket.setStockPrice("TSLA", 720.00); // 10% decrease

        // Test observer deregistration

        System.out.println("\n5. Testing observer deregistration...");

        stockMarket.deregisterObserver(mobileApp1);

        stockMarket.deregisterObserver(emailAlert);

        stockMarket.setStockPrice("GOOGL", 2450.00); // 2% decrease

        // Final stock update with fewer observers

        stockMarket.setStockPrice("AAPL", 160.00); // 2.9% increase

        // Display final summary

        System.out.println("\n6. Final Stock Summary:");

        displayStockSummary(stockMarket);

    }

    private static void displayStockSummary(StockMarket stockMarket) {

        System.out.println("Current Stock Prices:");

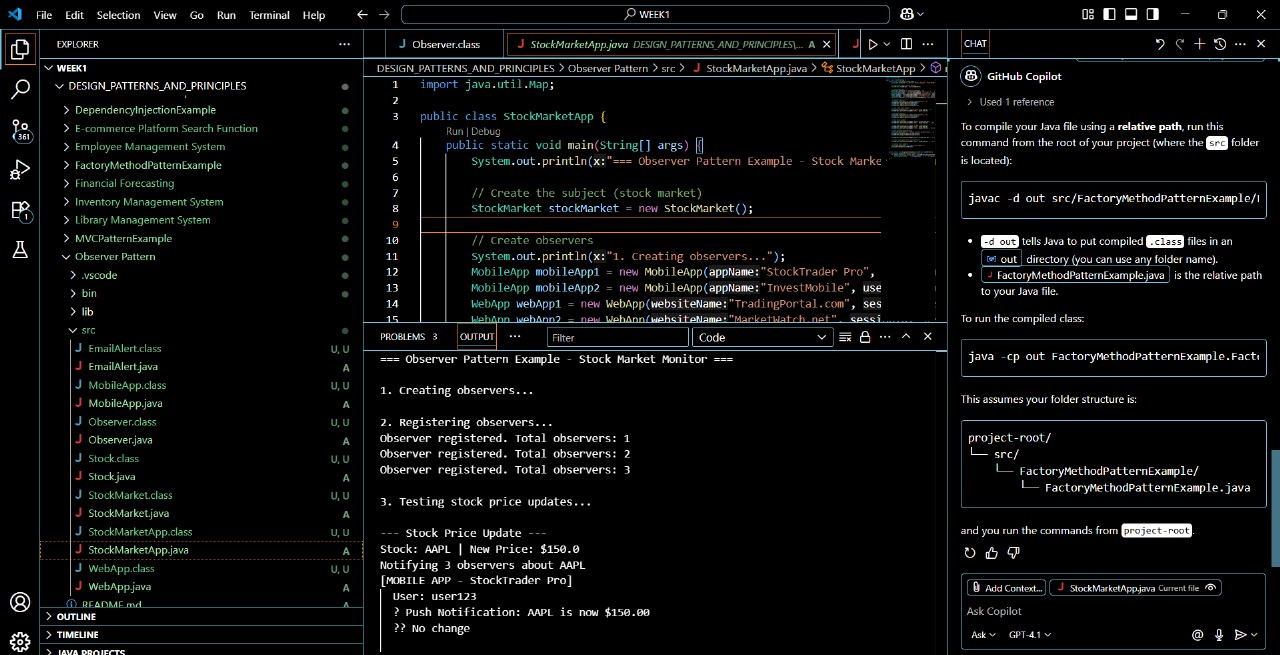
        for (Map.Entry<String, Double> entry : stockMarket.getAllStockPrices().entrySet()) {

            System.out.println("  " + entry.getKey() + ": $" + String.format("%.2f", entry.getValue()));

        }

    }

}



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

PaymentSystemApp.java

import java.util.Scanner;

public class PaymentSystemApp {

    public static void main(String[] args) {

        System.out.println("=== Strategy Pattern Example - Payment System ===\n");

        // Create payment context

        PaymentContext paymentContext = new PaymentContext("ORD-12345");

        paymentContext.setAmount(299.99);

        // Demonstrate different payment strategies

        demonstrateAutomaticPayments(paymentContext);

        // Interactive demo

        System.out.println("\n" + "=".repeat(50));

        System.out.println("Interactive Payment Demo");

        System.out.println("=".repeat(50));

        interactivePaymentDemo();

    }

      private static void demonstrateAutomaticPayments(PaymentContext context) {

        System.out.println("1. Testing Credit Card Payment:");

        System.out.println("-".repeat(40));

        // Create and test credit card payment

        PaymentStrategy creditCard = new CreditCardPayment(

            "1234567890123456",

            "John Doe",

            "12/25",

            "123"

        );

        context.setPaymentStrategy(creditCard);

        context.executePayment();

        System.out.println("\n2. Testing PayPal Payment:");

        System.out.println("-".repeat(40));

        // Create and test PayPal payment

        PaymentStrategy paypal = new PayPalPayment(

            "john.doe@example.com",

            "securepassword123"

        );

        context.setPaymentStrategy(paypal);

        context.executePayment();

        System.out.println("\n3. Testing Bank Transfer Payment:");

        System.out.println("-".repeat(40));

        // Create and test bank transfer payment

        PaymentStrategy bankTransfer = new BankTransferPayment(

            "12345678901",

            "123456789",

            "John Doe"

        );

        context.setPaymentStrategy(bankTransfer);

        context.executePayment();

    }

    private static void interactivePaymentDemo() {

        Scanner scanner = new Scanner(System.in);

        PaymentContext context = new PaymentContext("ORD-" + System.currentTimeMillis());

        // Get amount from user

        System.out.print("Enter payment amount: $");

        double amount = scanner.nextDouble();

        scanner.nextLine(); // consume newline

        context.setAmount(amount);

        while (true) {

            System.out.println("\nSelect Payment Method:");

            System.out.println("1. Credit Card");

            System.out.println("2. PayPal");

            System.out.println("3. Bank Transfer");

            System.out.println("4. Exit");

            System.out.print("Choice: ");

            int choice = scanner.nextInt();

            scanner.nextLine(); // consume newline

            PaymentStrategy strategy = null;

            switch (choice) {

                case 1:

                    strategy = createCreditCardStrategy(scanner);

                    break;

                case 2:

                    strategy = createPayPalStrategy(scanner);

                    break;

                case 3:

                    strategy = createBankTransferStrategy(scanner);

                    break;

                case 4:

                    System.out.println("Thank you for using our payment system!");

                    scanner.close();

                    return;

                default:

                    System.out.println("Invalid choice. Please try again.");

                    continue;

            }

            if (strategy != null) {

                context.setPaymentStrategy(strategy);

                boolean success = context.executePayment();

                if (success) {

                    System.out.println("\n Transaction completed successfully!");

                    break;

                } else {

                    System.out.println("\n Would you like to try a different payment method?");

                }

            }

        }

        scanner.close();

    }

    private static PaymentStrategy createCreditCardStrategy(Scanner scanner) {

        System.out.print("Enter card number (16 digits): ");

        String cardNumber = scanner.nextLine();

        System.out.print("Enter cardholder name: ");

        String name = scanner.nextLine();

        System.out.print("Enter expiry date (MM/YY): ");

        String expiry = scanner.nextLine();

        System.out.print("Enter CVV (3 digits): ");

        String cvv = scanner.nextLine();

        return new CreditCardPayment(cardNumber, name, expiry, cvv);

    }

    private static PaymentStrategy createPayPalStrategy(Scanner scanner) {

        System.out.print("Enter PayPal email: ");

        String email = scanner.nextLine();

        System.out.print("Enter password: ");

        String password = scanner.nextLine();

        return new PayPalPayment(email, password);

    }

  private static PaymentStrategy createBankTransferStrategy(Scanner scanner) {

        System.out.print("Enter account number: ");

        String accountNumber = scanner.nextLine();

        System.out.print("Enter routing number (9 digits): ");

        String routingNumber = scanner.nextLine();

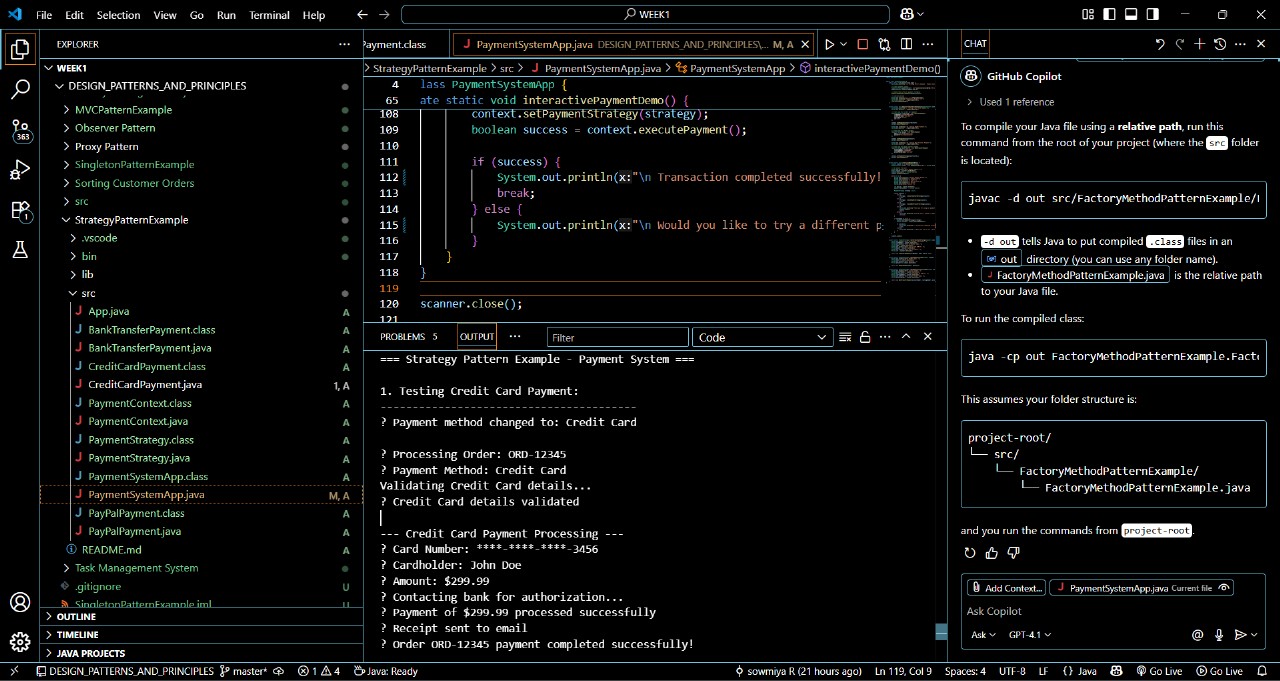
        System.out.print("Enter account holder name: ");

        String accountHolder = scanner.nextLine();

        return new BankTransferPayment(accountNumber, routingNumber, accountHolder);

    }

}



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

Student.java

// Source code is decompiled from a .class file using FernFlower decompiler.

class Student {

   private String name;

   private String id;

   private String grade;

   public Student(String var1, String var2, String var3) {

      this.name = var1;

      this.id = var2;

      this.grade = var3;

   }

   public String getName() {

      return this.name;

   }

   public String getId() {

      return this.id;

   }

   public String getGrade() {

      return this.grade;

   }

   public void setName(String var1) {

      this.name = var1;

   }

   public void setId(String var1) {

      this.id = var1;

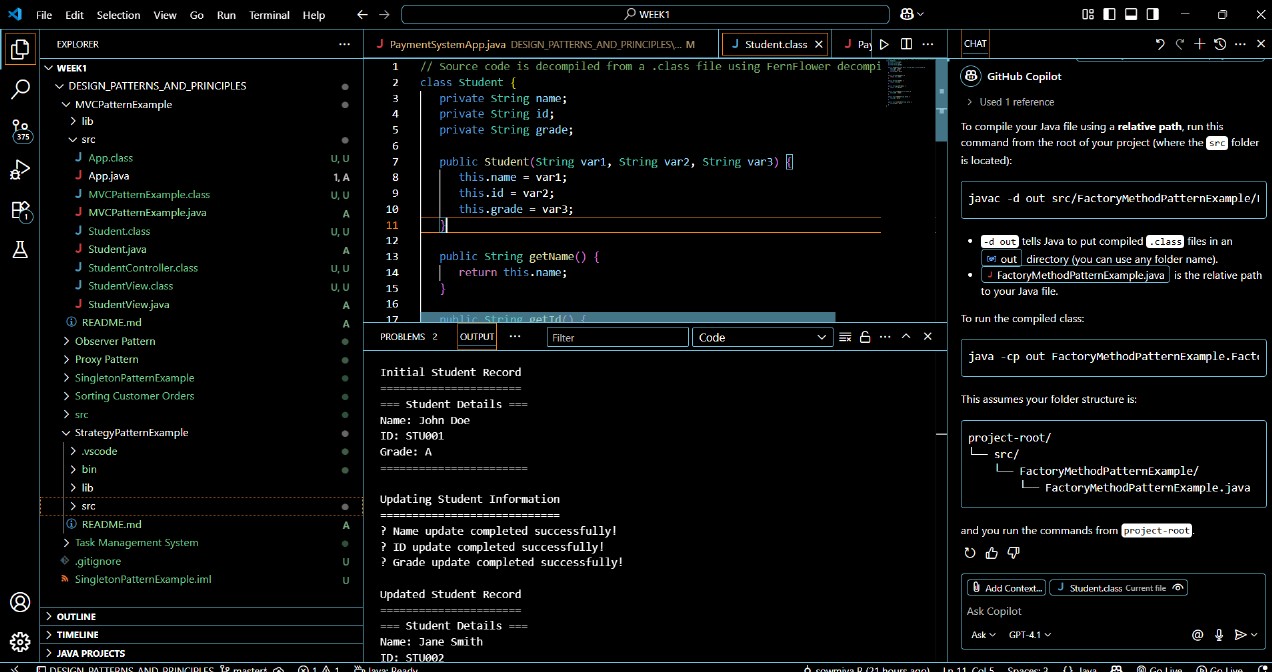
   }

   public void setGrade(String var1) {

      this.grade = var1;

   }

}



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

DependencyInjectionExample.java

import java.util.List;

public class DependencyInjectionExample {

    public static void main(String[] args) {

        System.out.println("=== Dependency Injection Example ===\n");

        // Demonstrate dependency injection with different implementations

        demonstrateWithRealRepository();

        System.out.println("\n" + "=".repeat(50) + "\n");

        demonstrateWithMockRepository();

        System.out.println("\n" + "=".repeat(50) + "\n");

        demonstrateBenefitsOfDI();

    }

    private static void demonstrateWithRealRepository() {

        System.out.println("--- Using Real Repository Implementation ---");

        // Step 1: Create the repository implementation

        CustomerRepository repository = new CustomerRepositoryImpl();

        // Step 2: Inject the repository into the service (Constructor Injection)

        CustomerService customerService = new CustomerService(repository);

        // Step 3: Use the service to perform operations

        System.out.println("\n1. Finding customer by ID:");

        Customer customer = customerService.getCustomerById("CUST001");

        if (customer != null) {

            System.out.println("Found: " + customer);

        }

        System.out.println("\n2. Finding non-existent customer:");

        Customer nonExistent = customerService.getCustomerById("CUST999");

        System.out.println("\n3. Getting all customers:");

        List<Customer> allCustomers = customerService.getAllCustomers();

        allCustomers.forEach(c -> System.out.println("- " + c.getName()));

        System.out.println("\n4. Creating new customer:");

        try {

            customerService.createCustomer("CUST005", "Charlie Wilson", "charlie@email.com", "555-0105");

        } catch (Exception e) {

            System.out.println("Error: " + e.getMessage());

        }

        System.out.println("\n5. Searching customers by name:");

        List<Customer> searchResults = customerService.searchCustomersByName("John");

        searchResults.forEach(c -> System.out.println("Search result: " + c.getName()));

    }

    private static void demonstrateWithMockRepository() {

        System.out.println("--- Using Mock Repository Implementation ---");

        // Step 1: Create a mock repository for testing

        CustomerRepository mockRepository = new MockCustomerRepository();

        // Step 2: Inject the mock repository into the service

        CustomerService customerService = new CustomerService(mockRepository);

        // Step 3: Use the service with mock data

        System.out.println("\n1. Finding customer with mock repository:");

        Customer mockCustomer = customerService.getCustomerById("TEST001");

        if (mockCustomer != null) {

            System.out.println("Mock customer: " + mockCustomer);

        }

        System.out.println("\n2. Getting all mock customers:");

        List<Customer> mockCustomers = customerService.getAllCustomers();

        mockCustomers.forEach(c -> System.out.println("- " + c.getName()));

    }

    private static void demonstrateBenefitsOfDI() {

        System.out.println("--- Benefits of Dependency Injection ---");

        System.out.println("\n✓ Loose Coupling:");

        System.out.println("  - CustomerService doesn't depend on concrete implementation");

        System.out.println("  - Can switch between real and mock repositories easily");

        System.out.println("\n✓ Testability:");

        System.out.println("  - Easy to inject mock objects for unit testing");

        System.out.println("  - Service logic can be tested in isolation");

        System.out.println("\n✓ Flexibility:");

        System.out.println("  - Can change repository implementation without modifying service");

        System.out.println("  - Supports multiple implementations (Database, File, Web Service, etc.)");

        System.out.println("\n✓ Maintainability:");

        System.out.println("  - Single Responsibility Principle followed");

        System.out.println("  - Dependencies are explicit and managed externally");

        System.out.println("\n✓ Reusability:");

        System.out.println("  - Service can be reused with different repository implementations");

        System.out.println("  - Repository implementations can be shared across services");

        // Demonstrate error handling

        System.out.println("\n--- Error Handling in Dependency Injection ---");

        try {

            CustomerService invalidService = new CustomerService(null);

        } catch (IllegalArgumentException e) {

            System.out.println("✓ Proper validation: " + e.getMessage());

        }

    }

}

