420-301-VA: Programming Patterns

Lab 7: Software Design Patterns I

Computer Science & Technology
Vanier College

Introduction

In our previous labs, we focused on the Java Collections Framework, which provides structures for **storing** data efficiently. We now shift our focus to **Software Design Patterns**, which offer reusable templates for **organizing** code and solving common architectural problems.

A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. They are not finished designs but rather descriptions for how to solve a problem that can be used in many different situations. Using patterns helps create software that is more flexible, maintainable, and scalable.

This lab introduces three foundational patterns: Singleton (Creational), Adapter (Structural), and Model-View-Controller (MVC) (Behavioral).

Objectives

This lab provides a hands-on introduction to three classic design patterns. By the end of this lab, you should be comfortable with:

- Understanding the purpose of Creational, Structural, and Behavioral patterns.
- Implementing the **Singleton** pattern to ensure a class has only one instance and provides a global access point.
- Recognizing the need for thread-safety in a Singleton implementation.
- Applying the **Adapter** pattern to make two incompatible interfaces work together.
- Understanding the **Model-View-Controller (MVC)** pattern for separating application logic, data, and presentation.
- Building a simple application using the MVC architecture with multiple, distinct views.

Instructions

- Please try out the examples first to deepen your understanding.
- For each exercise, create the required classes and a main method to test your implementation.
- Copy-paste your formatted code for each exercise into a separate document, followed by screenshots of the console outputs to validate that your code worked as expected.
- You can work with your peers, but each person must write, execute, and document their own work.

1 Part 1: The Singleton Pattern (Creational)

1.1 Concept Review: Ensuring One Instance

The Singleton pattern is a **creational** pattern that ensures a class has only one instance and provides a global access point to that instance. This is useful for managing shared resources like a configuration manager, a logging service, or a connection pool, where having multiple instances would be inefficient or cause conflicts.

The classic implementation involves:

- A `private` static variable to hold the single instance.
- A `private` constructor to prevent other classes from instantiating it.
- A `public` static method (commonly named `getInstance()`) that creates the instance if it doesn't exist and returns it.

```java
public class Logger {
    // 1. The private static instance
    private static Logger instance;

    // 2. The private constructor
    private Logger() {
        // Initialization code, e.g., opening a log file
        System.out.println("Logger instance created.");
    }

    // 3. The public static getter method
    public static Logger getInstance() {
        if (instance == null) {
            // Lazy initialization
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("LOG: " + message);
    }
}

// --- In a main method ---
public class SingletonTest {
    public static void main(String[] args) {
        // Both variables will point to the *same* object
        Logger logger1 = Logger.getInstance();
        Logger logger2 = Logger.getInstance();

        logger1.log("First message from logger1.");
        logger2.log("Second message from logger2.");
        // Prove it's the same instance
        if (logger1 == logger2) {
            System.out.println("Both loggers are the same instance.");
        }
    }
}
```

Listing 1: Example: A simple (non-thread-safe) Logger

Exercise 1: Centralized Application Logger

A very common use for a Singleton is a logging service. In a large application, you want all parts of your code (e.g., the user service, the payment service, the database connector) to write their log messages to the *same file*. If each service created its own logger, they would conflict over file access, or you would have dozens of different log files.

Your task is to create a `ServiceLogger` Singleton that manages writing all application log messages to a single file, `app_log.txt`.

1. **Setup:** You will need to import `java.io.FileWriter` and `java.io.PrintWriter` for this exercise.

2. **Create the `ServiceLogger` class:**

   - Add a private static instance: `private static ServiceLogger instance;`
   - Add a private instance variable to manage the file: `private PrintWriter writer;`
   - **Private Constructor:** The constructor `private ServiceLogger()` should:
     - Attempt to open a `FileWriter` and `PrintWriter` for a file named `"app_log.txt"`. Use a `try-catch` block to handle the `IOException`.
     - In the `try` block, initialize the `PrintWriter` in "append" mode:
       `FileWriter fw = new FileWriter("app_log.txt", true);`
       `this.writer = new PrintWriter(fw);`
     - Print a console message so you know when it runs:
       `System.out.println("--- LOGGER INITIALIZED ---");`
   - `getInstance()` **Method:** Create the standard method:
     `public static ServiceLogger getInstance()` that uses lazy initialization (the `if (instance == null)` check).
   - `log()` **Method:** Create a public method
     `public void log(String serviceName, String message)`. This method should:
     - Get the current date/time (e.g., `new java.util.Date()`).
     - Write a formatted string to the file using the `writer`:
       `writer.println("["+ new java.util.Date()+ "]  [" + serviceName + "] "+ message);`
     - **Important:** Call `writer.flush();` to ensure the message is written to the file immediately and not just kept in a buffer.

3. **Test in `main()`:**

   - **Simulate a User Service:**
     - Get the logger instance: `ServiceLogger userLogger = ServiceLogger.getInstance();`
     - Log an action:
       `userLogger.log("UserService", "User with userid:student1 logged in.");`
   - **Simulate a Product Service:**
     - Get the logger instance again:
       `ServiceLogger productLogger = ServiceLogger.getInstance();`
     - Log another action:
       `productLogger.log("ProductService", "Product 'B-102' viewed.");`
   - **Verify:**
     - Check if both variables point to the same object:
       `System.out.println("Same logger instance: "+ (userLogger == productLogger));`
     - Close the logger's file writer so you can inspect the file: `userLogger.closeWriter();` (You'll need to add this simple public method to your logger class, which just calls `writer.close()`).

4. **Observe the Results:**

   - Look at your console output. You should see the `"LOGGER INITIALIZED"` message print *only once*, and `"Same logger instance: true"` should print at the end.
   - Open the `app_log.txt` file in your project's root directory. You should see both log messages, from both "services," correctly formatted and stored in the same file.

5. **Final Step (Thread-Safety):**

   - The `getInstance()` method you wrote is not **thread-safe**. If two threads call it at the exact same time, both might see `instance` as `null` and create two loggers, which would cause file-corruption issues.
   - To fix this, simply add the `synchronized` keyword to your `getInstance()` method signature. This ensures that only one thread can execute that method at a time, protecting your Singleton.

```java
public static synchronized ServiceLogger getInstance() {
    // ... implementation ...
}
```

2 Part 2: The Adapter Pattern (Structural)

2.1 Concept Review: Making Incompatible Interfaces Work Together

The Adapter pattern is a **structural** pattern that allows objects with incompatible interfaces to collaborate. It acts as a bridge, converting the interface of one class (the `Adaptee`) into an interface a client expects (the `Target`).

This is common when integrating new code with legacy systems or third-party libraries. Imagine you have a new component that returns data as JSON, but your existing application only understands XML. An adapter would be perfect here.

```java
// 1. The Target Interface (what the client expects)
interface Duck {
    public void quack();
    public void fly();
}

// 2. The Adaptee (the class we have)
class WildTurkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }
    public void flyShortDistance() {
        System.out.println("I'm flying a short distance");
    }
}

// 3. The Adapter (implements Target, wraps Adaptee)
class TurkeyAdapter implements Duck {
    WildTurkey turkey;

    public TurkeyAdapter(WildTurkey turkey) {
        this.turkey = turkey;
    }

    // Translate the method calls
    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        // A Duck's fly is different, so we adapt
        for(int i=0; i < 5; i++) {
            turkey.flyShortDistance();
        }
    }
}
```

```java
// --- In a main method ---
public class AdapterTest {
    public static void main(String[] args) {
        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says...");
        turkey.gobble();
        turkey.flyShortDistance();

        System.out.println("\nThe Duck (Adapter) says...");
        testDuck(turkeyAdapter);
    }

    // A client method that only accepts Ducks
    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Listing 2: Example: Adapting a Turkey to behave like a Duck

Exercise 2: Adapting a Third-Party JSON API

Imagine you are working on an e-commerce platform. Your application is built around an `IProductService` interface that expects product data in a specific `Product` object format. You just purchased a new, high-performance API, but it provides data in a raw JSON `String` format.

Your task is to create an adapter so your client code can use the new API without being modified.

1. **The Target:** Create an interface named `IProductService` with one method: `Product getProductById(int id)`.

2. **The Data Object:** Create a simple `Product` class with attributes `id` (`int`), `name` (`String`), and `price` (`double`). Include a constructor, getters, and a `toString()` method.

3. **The Adaptee:** Create a class `NewProductApi`. This class simulates the third-party service. It should have one method: `String fetchProductJson(int id)`. This method should just return a hard-coded JSON string. For example, if `id == 101`, it returns:

   ```
   { 'productId': 101, 'productName': 'SuperWidget', 'cost': 45.99 }
   ```

4. **The Adapter:** Create a class `ProductApiAdapter` that implements `IProductService`.

   - It must have a `NewProductApi` object as a private field (pass it in the constructor).
   - Implement the `getProductById(int id)` method.
   - Inside this method, you must:
     (a) Call the adaptee's `fetchProductJson(id)` method.
     (b) **(The "notch higher" part)** Parse the simple JSON string. You don't need a full JSON library; you can use `String.split()` or `String.replace()` to extract the ID, name, and price.
     (c) Create a new `Product` object using the extracted data.
     (d) Return the `Product` object.

5. **The Client:** In your `main` method, create a `NewProductApi` instance. Then, create a `ProductApiAdapter` instance, passing the API to it.

6. Finally, show that your client code can seamlessly get a `Product` object just by calling `adapter.getProductById(101)`. Print the resulting `Product` object.

3 Part 3: The Model-View-Controller (MVC) Pattern

3.1 Concept Review: Separating Concerns

The Model-View-Controller (MVC) pattern is a **behavioral** pattern that divides an application into three interconnected components. This separation of concerns makes applications much easier to manage, test, and scale.

- **Model**: Manages the application's data, logic, and rules. It is the "brain" of the application and holds the state. It knows nothing about the View or Controller.
- **View**: The user interface (UI) responsible for displaying the data from the Model. It is a "dumb" component that only presents information.
- **Controller**: Listens for user input (e.g., button clicks) and acts as the intermediary. It receives input, translates it into commands for the Model (e.g., "update your data"), and can tell the View to refresh.

```java
// 1. Model: Holds the data
class Student {
    private String name;
    private String id;

    // Getters and Setters...
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getId() { return id; }
    public void setId(String id) { this.id = id; }
}

// 2. View: Renders the data to the console
class StudentView {
    public void printStudentDetails(String studentName, String studentId) {
        System.out.println("--- STUDENT RECORD ---");
        System.out.println("Name: " + studentName);
        System.out.println("ID:   " + studentId);
        System.out.println("----------------------");
    }
}

// 3. Controller: Connects Model and View
class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student model, StudentView view) {
        this.model = model;
        this.view = view;
    }

    // Methods to update the model
    public void setStudentName(String name) {
        model.setName(name);
    }
    public void setStudentId(String id) {
        model.setId(id);
    }

    // Method to update the view
    public void updateView() {
        view.printStudentDetails(model.getName(), model.getId());
    }
}

// --- In a main method ---
public class MvcTest {
    public static void main(String[] args) {
        // Fetch student record (simulated)
        Student model = new Student();
        model.setName("Alice");
        model.setId("12345");

        // Create the view
        StudentView view = new StudentView();

        // Create the controller
        StudentController controller = new StudentController(model, view);

        // Display initial data
        controller.updateView();

        // Simulate user input changing the model
        controller.setStudentName("Bob");

        // Display updated data
        controller.updateView();
    }
}
```

Listing 3: Example: Simple MVC with a Console View

Exercise 3: MVC for a Simple E-Commerce Cart

In this exercise, you will build a simple console-based shopping cart. This task is a perfect fit for MVC:

- **Model**: Will hold the list of products and calculate the total price.
- **View**: Will be responsible **only** for printing the cart contents to the console.
- **Controller**: Will respond to user actions (simulated from `main`) like "add product" and "show cart".

This exercise follows the pattern from the example, where the controller *manually* tells the view to update after the model is changed.

1. **The Model (Data Object):**
   - Create a `Product` class with `private String name` and `private double price`.
   - Add a constructor and public getters for both fields.
   - **Important:** To use this class in a `Map`, you must override `equals(Object o)` and `hashCode()`. (You can ask your IDE to auto-generate these.)
2. **The Model (The logic):**
   - Create a `CartModel` class.
   - It must contain a `private Map<Product, Integer> items` (to store products and their quantities). Initialize it in the constructor.
   - Create a method `public void addProduct(Product p)`. This method should check if the product is already in the map. If not, `put` it with a quantity of 1. If it is, get the current quantity, increment it, and `put` it back.
   - Create a method `public Map<Product, Integer> getItems()` that returns the `items` map.
   - Create a method `public double getTotalPrice()`. This is the **business logic**. It should loop through the map's `entrySet()`, multiply each product's price by its quantity, and return the total sum.
3. **The View:**
   - Create a `CartView` class.
   - It should have *one* method:
     `public void displayCart(Map<Product, Integer> items, double total)`.
   - This method should loop through the map and print each item's details (name, quantity, price).
   - After the loop, it should print the final `total` price, formatted nicely.
   - **Note:** This class does *no calculations*. It only prints data it is given.

4. **The Controller:**
   - Create a `CartController` class.
   - It must have `private CartModel model` and `private CartView view` fields.
   - Create a constructor that accepts the model and view and assigns them.
   - Create a method `public void handleAddProduct(Product p)`. This method should call `model.addProduct(p)`.
   - Create a method `public void handleDisplayCart()`. This method is the "coordinator." It must:
     (a) Get the data from the model: `Map<Product, Integer> items = model.getItems();`
     (b) Get the calculated total from the model: `double total = model.getTotalPrice();`
     (c) Pass this data to the view: `view.displayCart(items, total);`

5. **Test in `main()`:**
   - Create your two `Product` objects (e.g., "Laptop" at 1200.00, "Mouse" at 45.00).
   - Create the `CartModel` and `CartView`.
   - Create the `CartController`, passing it the model and view.
   - **Simulate a user adding items and viewing the cart:**

     ```
     System.out.println("Adding Laptop...");
     controller.handleAddProduct(laptop);
     controller.handleDisplayCart();
     System.out.println("\nAdding Mouse...");
     controller.handleAddProduct(mouse);
     System.out.println("Adding another Laptop...");
     controller.handleAddProduct(laptop);
     controller.handleDisplayCart();
     ```
   - **Observe:** Your final output should show a cart with `"Laptop (2)"` and `"Mouse (1)"` and the correct total price, proving all three components worked together correctly.

Important: Override `equals()` and `hashCode()`

- **Why?** A `HashMap` (which you are using for `items`) does not use the `==` operator to compare keys. Instead, it relies on two methods: `hashCode()` and `equals()`.
- **The Process:**
  1. When you call `map.put(product, 1)`, the map first calls `product.hashCode()` to find which "bucket" or storage location to use.
  2. It then checks this bucket for any existing keys. To compare `product` with an existing key, it calls `product.equals(existingKey)`.
- **The Problem:** If you don't override these methods, the default behavior from the `Object` class is used. The default `hashCode()` is based on the object's memory address, and the default `equals()` is the same as `==` (it checks for the exact same object, not an equivalent one).
- **Example:** This means two separate objects:
  `new Product("Laptop", 1200)` and `new Product("Laptop", 1200)`, would have different hash codes and be considered "not equal". Your map would incorrectly store them as two completely separate entries instead of updating the quantity of the first one.
- **The Solution:** You must override `equals(Object o)` to define that two `Product` objects are the same if their `name` and `price` are the same. You must also override `hashCode()` to ensure that two "equal" objects will produce the same hash code.
- **(You can ask your IDE to auto-generate these methods based on the `name` and `price` fields.)**

(Optional) Exercise 4: MVC with Multiple Views (Observer Pattern)

The simple MVC example is good, but the controller has to manually call `updateView()`. A more robust MVC implementation uses the **Observer pattern**, where Views "subscribe" to the Model and are notified automatically when the Model changes.

Your task is to build a simple counter application with one Model, one Controller, and **two** different Views (a console view and a GUI view). Both views must update automatically when the counter is changed.

1. **Observer Interface:** Create an interface named `Observer` with a single method: `void update();`.

2. **Model** (`CounterModel`)**:**

   - Hold a private `int value`.
   - Hold a private `List<Observer>` to store its subscribers.
   - Create methods `addObserver(Observer o)` and `notifyObservers()`. The second method should loop through the list and call `update()` on each observer.
   - Create methods `public void increment()`, `public void decrement()`, and `public int getValue()`.
   - **Crucially:** `increment()` and `decrement()` must call `notifyObservers()` after changing the value.

3. **Views (Implement** `Observer`**):**

   - `ConsoleView`:
   - Must implement `Observer`.
   - Keep a reference to the `CounterModel` (passed in constructor).
   - Its `update()` method should get the value from the model (`model.getValue()`) and print it to the console (e.g., "Console View: Value is now 5").
   - `GuiView` **(using Java Swing):**
   - Must implement `Observer`.
   - Keep a reference to the `CounterModel`.
   - Create a `JFrame`, a `JLabel` (to show the count), and two `JButtons` ("Increment" and "Decrement").
   - Its `update()` method should get the value from the model and call `label.setText("Count: "+ model.getValue())`.

4. **Controller** (`CounterController`)**:**

   - This controller's job is to handle *input* from the GUI.
   - It should take the `CounterModel` and `GuiView` in its constructor.
   - Add `ActionListeners` to the `GuiView`'s buttons.
   - The "Increment" button's action should call `model.increment()`.
   - The "Decrement" button's action should call `model.decrement()`.

5. **Main Method (`Application`):**
    - Create the `CounterModel`.
    - Create the `ConsoleView` (passing the model).
    - Create the `GuiView` (passing the model).
    - **Register the views:**
      Call `model.addObserver(consoleView)` and `model.addObserver(guiView)`.
    - Create the `CounterController` (passing the model and guiView).
    - Make the `JFrame` visible.
    - **Test:** When you click the buttons in the GUI, you should see the `JLabel` in the GUI *and* the text in the console update simultaneously.

References

- Refactoring.Guru: Design Patterns: An excellent modern resource with clear, visual explanations and examples for all major design patterns.
- Baeldung: Design Patterns in Java: A practical, Java-focused guide with code examples for creational, structural, and behavioral patterns.
- SourceMaking: Design Patterns: Another classic and comprehensive resource that covers patterns, anti-patterns, and refactoring techniques.
- Refactoring.Guru: Singleton: A deep dive into the Singleton pattern, including its trade-offs and thread-safety.
- Refactoring.Guru: Adapter: A clear explanation of the Adapter pattern, which is the focus of Part 2.
- Refactoring.Guru: MVC: Discusses the Model-View-Controller pattern, its components, and how they interact.
- Baeldung: SOLID Principles: Understanding SOLID (Single Responsibility, Open/Closed, etc.) is crucial for knowing *why* and *when* to use design patterns.
- *Design Patterns: Elements of Reusable Object-Oriented Software* (1994) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. This is the canonical "Gang of Four" (GoF) book that first cataloged these patterns.
- *Head First Design Patterns* by Eric Freeman & Elisabeth Robson. An extremely popular and student-friendly book that uses Java to explain patterns in a highly visual and intuitive way.