Ministry of Education, Culture and Research
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automation

Software Design
Techniques and Mechanisms

REPORT

Laboratory Work #0 - SOLID

Student:
Vornic Daniela,
FAF-222

Teacher:
Furdui Alexandru,
univ. assist.

1 Tasks

1. Implement 2 SOLID letters in a simple project.

2 Theoretical Notes

The SOLID principles are five design principles in object-oriented programming and design aimed at making software designs more understandable, flexible, and maintainable. The acronym SOLID stands for:

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OCP)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**ependency Inversion Principle (DIP)

Let's explore each principle in detail:

Single Responsibility Principle (SRP)

This principle affirms that a class should have only one reason to change, meaning it should have only one job or responsibility. The main benefits of this principle are:

- Improved code organization and clarity
- Easier maintenance and updates
- Reduced coupling between different parts of an application

For example, instead of having a single class handle user input, data processing, and output, these responsibilities should be separated into distinct classes.

Open-Closed Principle (OCP)

The Open-Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means:

- You should be able to extend a class's behavior without modifying it
- Use interfaces and abstract classes to allow new functionality to be added with minimal changes to existing code

This principle encourages the use of abstraction and polymorphism to create flexible, extensible systems.

Liskov Substitution Principle (LSP)

This principle states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In other words:

- Subclasses should extend the capabilities of the parent class, not narrow them
- Subclasses should not change the expected behavior of the parent class

This principle ensures that inheritance is used correctly and promotes the creation of well-structured class hierarchies.

Interface Segregation Principle (ISP)

The Interface Segregation Principle states that no client should be forced to depend on methods it does not use. It suggests:

- Breaking down larger interfaces into smaller, more specific ones
- Clients should only need to know about the methods that are of interest to them

This principle helps in creating more focused and cohesive interfaces, reducing the impact of changes and making the system easier to refactor, change, and redeploy.

Dependency Inversion Principle (DIP)

This principle states that high-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. The key points are:

- Use interfaces or abstract classes to define contracts for lower-level components
- High-level modules should import abstractions, not concretions
- This promotes loose coupling and flexibility in software design

By depending on abstractions, the system becomes more modular and easier to modify, test, and maintain.

While all SOLID principles contribute to creating robust and maintainable software, this project focuses primarily on implementing the Single Responsibility Principle (SRP) and the Dependency Inversion Principle (DIP).

3 Implementation

The implementation on GitHub

Project Idea

The project implements a Movie Recommender system that suggests movies based on a user's mood and situation. The system adheres to SOLID principles, particularly SRP and DIP, to create a modular and flexible design.

SOLID Principles Implementation

Single Responsibility Principle

The project demonstrates SRP through the use of focused interfaces, each defining a single, well-defined responsibility. This approach allows for the creation of distinct classes that implement these interfaces, each handling a specific aspect of the system. Here are examples of how SRP is applied:

```
public interface InputHandler {
  String getMood();
  String getSituation();
  int getPostRecommendationChoice();
  void close();
}

public interface MovieRecommender {
  String recommendMovie(String mood, String situation);
}

public interface RecommendationStorage {
  void addRecommendation(String mood, String situation, List<String> movies);
  List<String> getRecommendation(String mood, String situation);
}
```

For instance, `ConsoleInputHandler` would implement `InputHandler`, focusing only on console-based user interaction, while `PersonalizedMovieRecommender` would implement `MovieRecommender`, concentrating on the movie recommendation logic.

Additionally, there are enums to represent moods and situations, further adhering to SRP:

```
public enum Mood {
    HAPPY("happy"),
    SAD("sad"),
    EXCITED("excited"),
    ANXIOUS("anxious"),
    RELAXED("relaxed");

    private final String displayName;

    Mood(String displayName) {
        this.displayName = displayName;
    }

    public String getDisplayName() {
        return displayName;
    }
}

public enum Situation {
    AT_HOME("at home"),
```

```
    ON_VACATION("on vacation"),
    STUDYING("studying"),
    WITH_FRIENDS("with friends");
    // Constructor and getter similar to Mood enum
}
```

Dependency Inversion Principle

The project applies DIP through the use of interfaces and dependency injection. The `Main` class depends on abstractions rather than concrete implementations:

```
public class Main {
  public static void main(String[] args) {
    RecommendationStorage storage = new HashMapRecommendationStorage();
    MovieRecommender recommender = new PersonalizedMovieRecommender(storage);
    InputHandler inputHandler = new ConsoleInputHandler();
    // ...
  }
}
```

This approach allows for easy swapping of implementations without affecting the core logic of the program.

Interfaces

As previously stated, the project uses several interfaces to define clear contracts for different components of the system:

- `InputHandler`: Manages user input and interaction
- `MovieRecommender`: Recommends movies based on mood and situation
- `RecommendationStorage`: Manages storage and retrieval of movie recommendations

Functionality

The main functionality of the system includes:

1. Getting user's mood and situation through numbered options
2. Recommending a movie based on the input
3. Allowing users to get another recommendation, change their mood/situation, or quit

As such, this system is composed of several classes, each implementing a specific interface and providing distinct functionality:

ConsoleInputHandler (implements InputHandler)

This class manages user interactions through the console:

- `getMood()`: Prompts the user to select their mood from a numbered list.

- `getSituation()`: Asks the user to choose their current situation from predefined options.
- `displayRecommendation()`: Displays the movie recommendation based on the provided mood and situation.
- `getPostRecommendationChoice()`: Presents options to the user after a recommendation (try again, change mood/situation, or quit).
- `close()`: Closes the scanner to prevent resource leaks.

PersonalizedMovieRecommender (implements MovieRecommender)

The PersonalizedMovieRecommender class is responsible for generating movie recommendations based on the user's mood and situation.

Key features:

- Uses a RecommendationStorage object to retrieve movie recommendations.
- Implements the `recommendMovie(String mood, String situation)` method:
  - Fetches a list of movie recommendations for the given mood and situation.
  - If no specific recommendation is found, falls back to a default recommendation for the mood.
  - Randomly selects and returns a movie from the available recommendations.
  - Returns a default movie ("Forrest Gump") if no recommendations are available.
- Initializes the recommendation data, populating the storage with predefined mood-situation-movie mappings.

HashMapRecommendationStorage (implements RecommendationStorage)

Manages the storage and retrieval of movie recommendations:

- `addRecommendation(String mood, String situation, List<String> movies)`: Adds a list of movies for a specific mood-situation combination.
- `getRecommendation(String mood, String situation)`: Retrieves the list of movies for a given mood-situation pair.

Main

Manages the overall flow of the application:

- Initializes the necessary components (RecommendationStorage, MovieRecommender, InputHandler).
- Manages the main loop of the application, coordinating user input and movie recommendations.
- Handles the program flow based on user choices (continue, change mood/situation, or quit).

The main functionality is implemented in a loop in the `Main` class:

```
String mood = null;
String situation = null;

while (true) {
  if (mood == null || situation == null) {
    mood = inputHandler.getMood();
    if (mood.equals("quit"))
      break;

    situation = inputHandler.getSituation();
    if (situation.equals("quit"))
      break;
  }

  String recommendedMovie = recommender.recommendMovie(mood, situation);
  inputHandler.displayRecommendation(mood, situation, recommendedMovie);

  int choice = inputHandler.getPostRecommendationChoice();
  if (choice == 1) {
    continue;
  } else if (choice == 2) {
    mood = null;
    situation = null;
  } else {
    break;
  }
}

    inputHandler.close();
}
```

This implementation showcases how the system uses the defined interfaces and their implementing classes to create a modular movie recommendation system. The loop structure allows for repeated interactions, giving users the ability to get multiple recommendations or change their inputs as desired.

4 Conclusion

In conclusion, the implementation of SOLID principles has resulted in a system that is not only functional but also robust and adaptable to future changes. By focusing on SRP, I created classes and interfaces with clear, singular purposes, making the codebase easier to understand and modify. The application of DIP has ensured that our high-level modules depend on abstractions rather than concrete implementations. This project is a practical example of how applying these principles can lead to more maintainable and flexible software design.