**The effects of using design patterns and architectural patterns in a Movie Recommendation System**

*Team Members:*

* Mădălina Franț
* David Nițu

1. Scenario Overview

This project implements a Movie Recommendation System in Java, featuring a minimal web-based interface. Users can input their movie preferences, including likes, dislikes, preferred actors, genres, and directors. Based on this input, the system generates recommendations, supporting two modes:

* **Classic Recommendation**: A single user's preferences are considered to generate recommendations.
* **Movie Night**: Two users enter their preferences, and the recommendations are tailored to accommodate both.

The project explores the impact of using design patterns and architectural patterns on software quality by comparing two implementations: one employing structured patterns and another without structured design, including anti-patterns.

1. State of the Art

The following design patterns and architectural patterns are relevant to this scenario:

* **Chain of Responsibility**: A behavioral pattern that allows multiple objects to handle a request, passing it along a chain until one processes it.
* **Strategy**: A behavioral pattern that enables interchangeable algorithms for a task, allowing dynamic changes at runtime.
* **Singleton**: A creational pattern that ensures a class has only one instance, providing global access to it.
* **Layered Architecture**: A structural approach that divides a system into distinct layers, each with specific responsibilities, promoting separation of concerns.

1. Implementation Details

The recommendation system:

* Allows users to specify preferences:
  + Movie Likes and Dislikes: Lists indicating movies the user likes and dislikes.
  + Preferred Attributes: Genres, actors, and directors.
* Provides movie suggestions based on:
  + The entered preferences for a single user (Classic Recommendation).
  + A combination of preferences for two users (Movie Night).

In the unstructured version, all the code is written in a single file, while in the structured version the code is separated into packages based on the functionality provided.

**Design Patterns:**

In the version optimized with design patterns, the following patterns are applied:

* **Chain of Responsibility**: To process user preferences in a modular way, allowing each preference type (genres, actors, directors) to be handled by dedicated handlers.
* **Strategy**: To define and apply different recommendation strategies dynamically based on user inputs, such as liked and disliked movies.
* **Singleton**: To ensure that only one instance of the movie database is created and shared across the system.

In the unstructured version, design patterns are intentionally avoided, and anti-patterns such as Spaghetti Code or God Object are introduced to compare their impact on maintainability and scalability.

**Architectural Patterns:**

The Layered Architecture is employed in the structured implementation to:

* Separate concerns into distinct layers (data access layer, business logic / service layer, and presentation layer).
* Ensure modularity and simplify testing.

The unstructured version does not use this architecture, highlighting the difficulties of working with tightly-connected, monolithic systems.

1. Metrics and Measurement Setup

To assess the impact of design and architectural patterns, the following metrics are used to compare the two implementations:

* **Execution Time**: Measure the time taken to generate recommendations, providing insights into the efficiency of each version.
* **Memory Usage**: Record memory consumption under similar workloads to evaluate memory efficiency.
* **Code Complexity**: Measure code complexity to assess maintainability and error risk.
* **Code Readability** (Lines of Code): Evaluate the clarity and readability of code, which influences ease of understanding and maintenance.
* **Modularity and Reusability**: Assess the modularity of components and the ease with which code can be reused in other projects.
* **Scalability**: Quantify the effort required to introduce new features (e.g. additional recommendation types) in each version, reflecting the system’s scalability.

The following testing conditions have been created (number of preferences the user inputs) :

* 1: 1 liked movie
* 2: 3 liked movies
* 3: 5 liked movies
* 4: 10 liked movies
* 5: 10 liked movies, 10 disliked movies, 5 actors, 5 genres, 5 directors

1. Results

* **Execution Time** (in ms):

| **Test / Version** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| 1 | **303** | 380 |
| 2 | **342** | 425 |
| 3 | 709 | **522** |
| 4 | 782 | **595** |
| 5 | 867 | **673** |

* **Memory Usage** (in MB):

| **Test / Version** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| 1 | **30** | 287 |
| 2 | **28** | 315 |
| 3 | **274** | 280 |
| 4 | **153** | 280 |
| 5 | **230** | 278 |

* **Code Complexity**:
* Unstructured Version: The code is difficult to understand and modify due to the lack of design patterns. There are many dependencies between components, which increase the risk of errors and make the code less maintainable.
* Structured Version: With the use of design and architectural patterns, the code is more modular and structured. This improves the maintainability and extensibility of the system, as different components can be modified without affecting the entire system.

| **Measurement** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| Number of files (java) | 1 | **14** |
| Number of classes | 2 | **12** |
| Number of interfaces | 0 | **2** |

* **Code Readability** (Lines of Code):
* Unstructured Version: Contains approximately 300 lines of code in a single file. The lack of clear structure leads to repetitive code and decreased readability.
* Structured Version: The use of design and architectural patterns and packages has helped reduce redundancy and improve code readability.

| **Measurement** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| Lines of code (total) | **~300** | ~600 |
| Lines of code per file (avg) | ~300 | **~40** |

* **Modularity and Reusability**:
* Unstructured Version: Components are tightly coupled, and reusing code in other projects or scenarios would be difficult without significant refactoring.
* Structured Version: The code is modular, with well-defined components that can be easily reused in other projects or extended with minimal effort.

| **Measurement** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| Number of packages | 1 | **7** |
| Number of files per package | 1 | **1-4** |

* **Scalability**:
* Unstructured Version: Scaling the system is challenging due to the lack of modularity and organization. Adding new features could introduce bugs or require significant refactoring.
* Structured Version: The system is more scalable due to the use of patterns that allow for easier additions and modifications. New features, such as additional recommendation modes or preference filters, can be implemented with minimal disruption.

| **Measurement** | **Anti-patterns** | **With-patterns** |
| --- | --- | --- |
| Effort to add new functionality | High | **Low** |

1. Scenario-Specific Discussion

The structured version of the Movie Recommendation System, using design patterns such as Chain of Responsibility, Strategy, Singleton, and Layered Architecture, is more modular, maintainable, and scalable compared to the unstructured version. The Chain of Responsibility and Strategy patterns helped modularize recommendation logic, allowing for easy expansion. Singleton ensured efficient database resource management, while Layered Architecture simplified testing and maintenance by separating concerns.

In contrast, the unstructured version lacked modularity, with tightly coupled components making it harder to extend and maintain, and increasing the risk of errors.

1. Scenario Conclusion

The results demonstrate that while the version with design patterns uses slightly more memory and has marginally higher execution times in simpler cases, it outperforms the unstructured version as the complexity of user input increases. The structured approach offers significant benefits in terms of maintainability, scalability, and performance for larger, more complex datasets. Overall, design patterns improve the system’s ability to handle growth and future changes, making them highly beneficial for systems expected to evolve over time.

1. References

<https://docs.spring.io/spring-boot/index.html>

<https://www.thymeleaf.org/documentation.html>

<https://refactoring.guru/design-patterns>