Islamic University of Technology

Board Bazar, Gazipur, Dhaka, 1704, Bangladesh

Final Project

Project Report

SWE 4301 - Object Oriented Concepts II

Khalid Hasan Ador

ID : 210042102

BSc in Software Engineering

Dept of Computer Science and Engineering

January 19, 2024

Contents

Chapter 1

Idea of the Project

The project is a movie review system, written in c. Where users can add movie to their liking and review them.

A user can add Review to existing movies or can add new ones. As there is no admin, this is more like a personal review system.

The project also maintains the record of all the Users, Movies And Reviews. Besides a user cannot review the same movie twice. When a user reviews more than 10 movies, his status is upgraded to ”Critic”.

The idea behind the project is to leverage the principles of Object-Oriented Programming (OOP) to create a flexible and scalable application. The application adheres to the SOLID principles, ensuring that it is well-structured and easy to maintain. This design choice allows for the easy addition of new movies without the need for significant code restructuring.

1.1 Key Features

1. **Add New User:** New Users can be added but 2 users cannot have the same username.
2. **Add New Movie:** With this function users can add new movies to review. For now they can add only 3 kinds of movies but as the project follows OOP principles, other kinds of movies can be added very easily.
3. **Add Review:** With this function users can review existing movies. But they cannot review the same movie more than once.
4. **Print Movies:** This function allows users to see the list of all the movies available in the system.
5. **Print Users:** This function prints all the users.
6. **Print Reviews:** This function allows users to see all the Reviews.
7. **Average Rating:** If any user wishes to know the average rating of any particular movie, they can use this feature.

1.2 UML Diagram

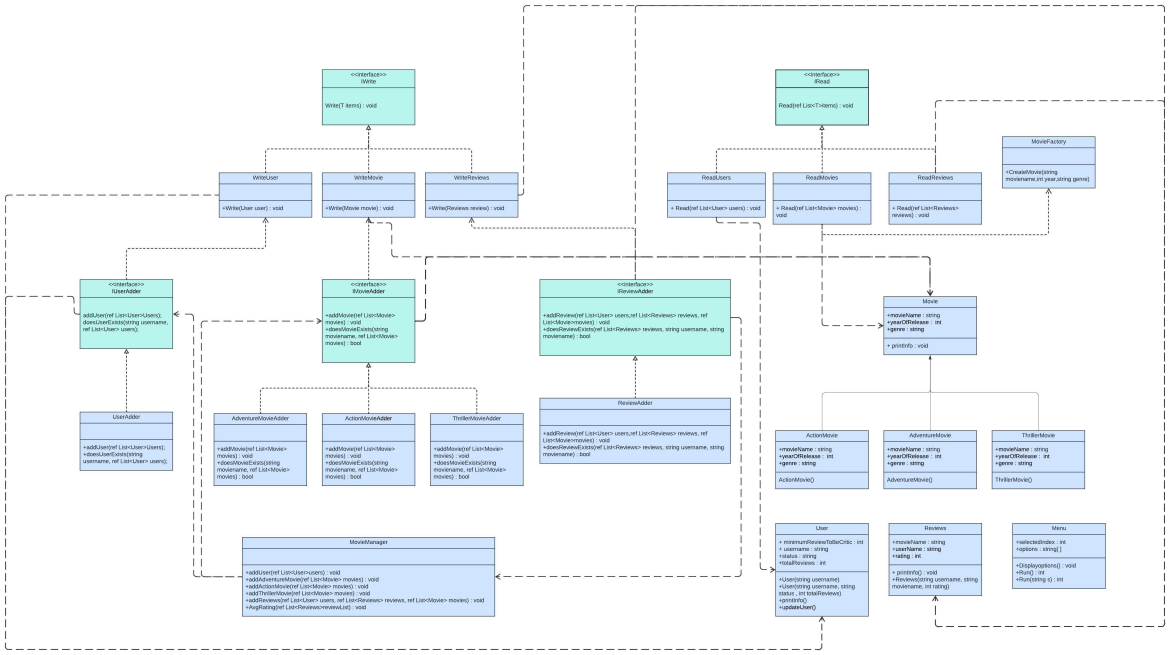

Figure 1.1: UML Diagram

Chapter 2

Use Of SOLID

The project strictly follows the SOLID Principles.

2.1 Single Responsibility Principle

Single Responsibility Principle or SRP states that a class should have only one reason to change, meaning it should have only one job or responsibility. It is one of the 5 principles of SOLID in object-oriented-programming. SRP is used when we want to avoid classes becoming overly complex and difficult to maintain. It's particularly useful in larger code-bases where it's crucial to keep everything modular and understandable. If a class starts to take on responsibilities that are not central to its primary purpose, it might be time to refactor the code and apply SRP. So, SRP maintains high cohesion of class and violation of SRP results in a code smell called ”Large Class”.

In the project, each class is designed to fulfill a specific role. For example, the MovieAdder class only has one responsibility and that is to add Movies. It does so by the addMovie() method. There are no other methods in that class. So the class has only one reason to change.

All other classes like: User, Movie, Review, Moviemanager follows SRP too. Though Moviemanager has multiple methods but it only works as a Delegator for all the methods except one. So it has only one reason to change.

2.2 Open Close Principle

The Open/Closed Principle (OCP) is another crucial aspect of the SOLID principles in object oriented programming. It states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” This means that you should be able to add new functionality or behavior to a system without changing existing code. This principle ensures that the introduction of new features does not introduce new bugs in the existing features. In the context of my project, this principle is effectively implemented in the design of the Movie class.

OCP is achieved by combination of composition, polymorphism and DIP. Overthinking of OCP results in ”Speculative Generality” and ”Needless Complexity”. Violation of OCP results in ”Conditional Statements” and ”Rigidity”.

The Movie class is ”Open for extension”, which means new types of movies can be added to application by simply extending the Movie class. This is evident in the AdventureMovie, ActionMovie and ThrillerMovie classes, which extend the Movie class to add functionalities specific to those classes.

The class is also ”Closed for modification”. This means adding a new type of Movie does not require any changes to the existing code in the Movie class.

In conclusion, the application of the Open/Closed Principle in the project contributes to a maintainable code-base. It allows for easy expansion of the application’s features while minimizing the risk of introducing bugs in the existing code.

2.3 Liskov Substitution Principle

The Liskov Substitution Principle or LSP is another integral part of the SOLID. It states that ”Sub-types must be substitutable by its parent type. If there is an object O_1 of type S and there is another object O_2 of type T, such that there is a program P, and it accepts any object of type T, we can use any object of type S, then S is a sub-type of T.

LSP imposes rules on inheritance. Violation of LSP is latent violation of OCP. ”Refused Bequest” is a special case of violation. LSP is used when we have a base class and one or more derived classes. It’s particularly useful when we want to ensure that a derived class does not affect the behavior of the base class. If a derived class causes issues when used in place of the base class, it might be time to consider applying the Liskov Substitution Principle.

In the context of my project, the Movie class is extended by AdventureMovie, ActionMovie and ThrillerMovie classes. These derived classes are subt-ypes of the Movie class. According to LSP, these subtypes must be substitutable for their base type. This means that the AdventureMovie, ActionMovie and ThrillerMovie classes can be used anywhere the Movie class is expected, without altering the correctness of the program. This ensures that the program remains functional even as new types of movies are added, demonstrating the flexibility and scalability of the application’s design.

In conclusion, the application of the Liskov Substitution Principle in the project contributes to a robust and maintainable code-base. It allows for easy expansion of the application’s features while ensuring the integrity of the program. This design approach is a testament to the power of the SOLID principles in creating effective object-oriented software.

2.4 Interface Segregation Principle

The Interface Segregation Principle (ISP) is a design principle in object-oriented programming that states that no client should be forced to depend on interfaces they do not use. This means that larger interfaces should be split into smaller ones, so that clients only need to know about the methods that are of interest to them.

ISP is used when a class does not use all the methods of an interface they inherit from. If a class is forced to implement an interface which has methods that are not relevant to its behavior, then it might be time to consider applying the Interface Segregation Principle.

In simpler terms, this principle suggests that a class should only be required to implement the methods that are relevant to its behavior and should not be forced to implement unnecessary methods from an interface. It promotes the idea of creating specific, focused interfaces rather than large, monolithic ones.

In terms of my project, it follows ISP perfectly. Classes like AdventureMovieAdder, ActionMovieAdder and ThrillerMovieAdder implements IMovieAdder. But they do use the methods of that interface. So the design follows ISP.

Similarly, ReadUsers, ReadMovies and ReadReviews classes implement IRead interface and uses the Read() method.

ReviewAdder implements IReviewAdder and uses addReview() method. UserAdder implements IUserAdder and uses addUser() method. WriteUser, WriteMovie and WriteReview implements IWrite and uses Write() method.

So, the project follows the ISP quite thoroughly. Applying the Interface Segregation Principle often results in interfaces that are tailored to the specific needs of the classes that implement them, promoting a more modular and maintainable code-base. This principle is particularly relevant in scenarios where interfaces are used to define contracts between different parts of a software system, such as in object-oriented programming languages like Java or C.

2.5 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) is a design principle in object-oriented programming that advocates for the decoupling of high-level modules from low-level modules, making systems more reusable, maintainable, and flexible. It states that high-level modules should not depend on low-level modules, but both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions.

DIP is used when we want to reduce the dependency between high-level and low-level classes, making our code more reusable and easier to change. If a high-level class is directly

dependent on a low-level class, then it might be time to consider applying the Dependency Inversion Principle.

DIP imposes use of Abstraction and reduces coupling. DIP is implemented by DI or Dependency Injection. Violation of DIP results in ”Fragility”.

In my project, the DIP is effectively implemented across the system. From the high-level Main class to other classes depend on abstractions rather than concrete implementations. This design choice promotes flexibility and makes the system more adaptable to future changes.

For instance, MovieManager class depends on abstract IuserAdder, IMovieAdder and IReviewAdder class not the concrete UserAdder, AdventureMovieAdder, ActionMovieAdder , ThrillerMovieAdder or the ReviewAdder class.

This adherence to the DIP throughout the project ensures that high-level modules do not depend on low-level modules; both depend on abstractions. Moreover, abstractions do not depend on details; instead, details depend on abstractions. This design approach contributes to a robust, flexible, and maintainable code-base that is well-prepared to accommodate future enhancements.

Chapter 3

Code Smells

3.1 Duplicated Code

Duplication is a code smell where the same code structure is repeated in more than one place in the program. It increases the size of the code and makes it harder to maintain. There may be different kinds of duplication like:

1. **Literal Duplication:** Writing exactly the same code again and again.
2. **Semantic Duplication:** Doing the same thing but writing in a different way.
3. **Data Duplication:** Keeping the same data in different places.
4. **Conceptual Duplication:** Objective same but algorithm different.
5. **Structural Duplication:** Condition on same value again and again.

In my project i have made all the decisions necessary to avoid Duplicated Code.

3.2 Inappropriate Naming

Inappropriate Naming refers to the use of unclear or misleading naming for variables, methods, classes, or other elements in the code. It makes the code harder to understand and maintain.

In my project, I have placed a strong emphasis on code quality and readability. One of the key aspects of this is adhering to good naming practices, which is crucial for ensuring that the code is easily understandable and maintainable.

All method names in our project follow the camel case convention. This means that the first letter of each word in the method name is capitalized, except for the first word. For example, a method to calculate the average score might be named calculateAverageScore. This naming convention makes the method names more readable and provides a clear indication

of what each method does.

Similarly, all class names start with a capital letter for each word, a convention often referred to as Pascal case. This helps to distinguish class names from method and variable names, making the code easier to read and understand.

In addition to methods and classes, we have also ensured that all variables are well-typed. This means that the type of each variable is clearly indicated, making the code safer and more predictable. It also helps to prevent errors, as it allows the compiler or interpreter to check that the operations performed on the variables are valid for their type.

Furthermore, we have avoided abstract naming for modules and attributes. Each module and attribute has a descriptive and meaningful name that clearly indicates its purpose or the value it holds. This practice helps to avoid the "Inappropriate Naming" code smell, where unclear or misleading names make the code difficult to understand and maintain.

3.3 Other Code Smells

1. Dead Code
2. Black Sheep
3. Long Method
4. Large Class
5. Conditional Complexity
6. Feature Envy
7. Primitive Obsession
8. Conditional Statements
9. Lazy Class
10. Speculative Generality
11. Refused Bequest
12. Bad Comments
13. Data Clumps
14. Shotgun Surgery
15. Temporary Field
16. Incomplete Library Class

In my project, I have made a concerted effort to avoid code smells. Code smells are characteristics in the source code that possibly indicate a deeper problem. They are not bugs, they do not prevent the program from functioning but they can negatively impact readability, scalability, and maintainability, and thus slow down development.

We have adopted several strategies to avoid code all other smells:

3.3.1 Adherence to SOLID Principles

By adhering to SOLID principles, we have avoided several common code smells. For example, the Single Responsibility Principle helps avoid the 'god class' smell, where a class knows too much or does too much.

3.3.2 Use of Design Patterns

The use of design patterns like Factory Method Pattern has helped me avoid code smells related to object creation and global access.

3.3.3 Proper Abstraction

By properly using abstraction, such as abstract classes and interfaces, we have avoided 'inappropriate intimacy' code smell, where one class uses the internal fields and methods of another class.

3.3.4 Consistent Formatting

Consistent code formatting and organization practices have been followed to avoid 'misplaced method' or 'data clumps' smells.

Chapter 4

Links and References

4.1 GitHub Link

The project's source code was maintained effectively by GitHub. The repository link of the project:

https://github.com/KHALID9029/OOP_2_Project

4.2 References

1. https://github.com/topics/solid-principles-examples
2. https://www.geeksforgeeks.org/design-patterns-in-object-oriented-progr
3. https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
4. https://refactoring.guru/refactoring/smells
5. https://www.lucidchart.com/pages/