CSC207 Phase 2 Report

Group 0162

Adam Paslawski, Daniel Dizon , Warren Liu

Albert Luo , Zexi Lv , Chelsea Mitchell

Adam Paslawsk , Fenil Patel , Insung Youn

Prepared for Shiva Saxena

Dec-7-2020

Contents

1 Introduction

Phase 2 has been a big improvement on phase 1. This is the result of quality feedback from our previous iteration that was implemented as well as some of our own design decisions whilst extending to meet the phase 2 specifications.

2 Design Patterns

We applied a series of design patterns to improve our codes adherence to SOLID Principles and Clean Architecture.

1. **User Manager, Builder**
    - The Problem: We needed to be able to construct users of many types with various combinations of attributes and the current solution used many constructors which left classes bloated.
    - How We Fixed it Implementing the builder pattern allowed for the flexibility we needed for building user representations without long parameter lists and many constructors.
2. **Database, Facade**
    - The Problem: The Database system has a series of complicated procedures for pulling data, parsing it, and converting it to usable objects for the program, we needed to hide the specifics of the implementation to make the code simple and easy to read.
    - How We Fixed it: A facade was used by implementing an interface for getting and setting database information while hiding the implementation.

3. **FeedBackControllerFactory, Factory**

   - The Problem: Feedback controllers were complicated to build and came in various types, potentially with more to come in the future. We needed to standardize the process to make our code open-closed.
   - How We Fixed it: Using a factory produced an abstract creation process where different types of Feedback Controllers can have different representations that are self defined.

4. **FeedBackManager: Dependency Inversion**

   - The Problem: We had a combined controller presenter which inhibited our codes adherence to the open-closed principle.
   - How We Fixed it We used an MVC design pattern which fetched presenter information via the FeedbackManager.

5. **UseCaseFactory, Factory**

   - The Problem: Use case classes are a core component of our system, and they need to be created in various ways upon application start up, constructing from the use case classes themselves left classes bloated and responsible to multiple actors.
   - How We Fixed it We created a factory for the creation of use cases to allow our code to be easily extended in the future and took the bloating out of the use case classes by offloading the responsibility of creation from them. This helps our code follow the Single Responsibility Principle.

3 Design Decisions

1. In phase one we had the presenter and controller amalgamated into one class. After realizing that it could cause issues upon extension of the program in the future, we fixed this by using the model view controller design pattern to separate the presenter and controller class. To do this, we moved the presenter functionality to the classes "DisplayInfo", "ErrorMessages", and "InfoPrompts". We strictly used the main controller menu class to take input from users. Prompting the user was done by calling one of the various methods in the presenter classes. This decision will allow for easier extension when compared to our phase 1 implementation because adding or changing presenter prompts (such as a change of language) is now a simple extension rather than requiring an overhaul to various controller classes.

2. During design of the database the decision was made to implement multiple small interfaces for database functionality as seen in the "DataBase" class. The reason for this was to apply the interface segregation principle since each interface has a specific purpose rather than combining these interfaces into one larger and more general interface.

3. We had a series of classes in phase 1 that were very long, this made the code difficult read and understand what was going on. We made a series of design decisions to better segregate functionality which resulted in large classes being split up into smaller classes that were specifically responsible to one actor. This helped produce code that better adhered to the Single Responsibility Principle and made it more readable for someone coming to work on the code in the future. An example of this can be seen in the "AvailabilityChecker" use case class which has the responsibility of

checking whether a user is available for a certain time slot or event, this was taken out of various user use case classes and put in its own class.

4. The application requires the parsing of user information upon shutting down the application to keep track of changes between sessions. In order to do this the "setUserManager" method directly depends on methods for converting user objects to a data format. Dependency injection was used here by injecting the method that parses user objects directly into the "setUserManager" method. A similar dependency injection strategy was used for storing room, message, and event information. The result is that other classes can simply ask to add an object to the database and pass in the objects themselves and the database class will handle all of the parsing, querying and inserting of the data to the relational database.

4 A Closing Note

Thanks for taking the time to read about our design and evaluate it, we are all really excited to see what you think. Big thanks to Shiva and the whole CSC207 teaching team, we all really enjoyed the course.