Phase 2 Design Document

Team Magic Conch Shell's Library Management System

Note1: Part 0 is a brief summary of what we have done after Phase1, for detailed explanations about them, please see Part 1 - 7. Part 8 is our progress report for phase 2.

Note2: In this document, all the new updates of phase 2 would be marked by a "*" sign.

Part 0. Updates and corrections made following feedbacks for Phase 1

(1) **<u>Implementation of Command design pattern – Part 1 & 3 (2)</u>**

We stated in our phase 1 design document that we are planning to add a new "shopping cart" feature to further expand our functionality. In phase 2, we add this function by implementing the Command design pattern in the **UserLoginManager** use case and add a new **Order** class.

(2) **<u>Implementing controller methods to connect our front-end html with back-end codes – Part 2 (2)</u>**

We continued to implement controller classes that are able to connect our html pages to back-end use cases. (***Updates after Monday's presentation: we have implemented all of the controller classes and our website is able to work now. Please see the README file for a demonstration of how to run the application to use our website**)

(3) **<u>Code Organizations – Part 4</u>**

We have improved our packaging strategy followed clean architecture by refactoring the code into packages named after the clean architecture layers according to the feedback.

(4) **<u>Use of github features – Part 6</u>**

After phase 1, we received the feedback that encouraged us to use more functions of github. Except the **pull request** that we are using continuously, our team have learned to use the github **Issues** to share questions and difficulties they encountered and assign works to each other.

(5) **<u>Add a section for code styling, documentation and testing – Part 7</u>**

In phase 2, we refactored and extracted helper methods for our duplicated codes in Gateway classes (IMongoDBbook, student and staffMethods).

Part 1. Updated Specification for Phase 2

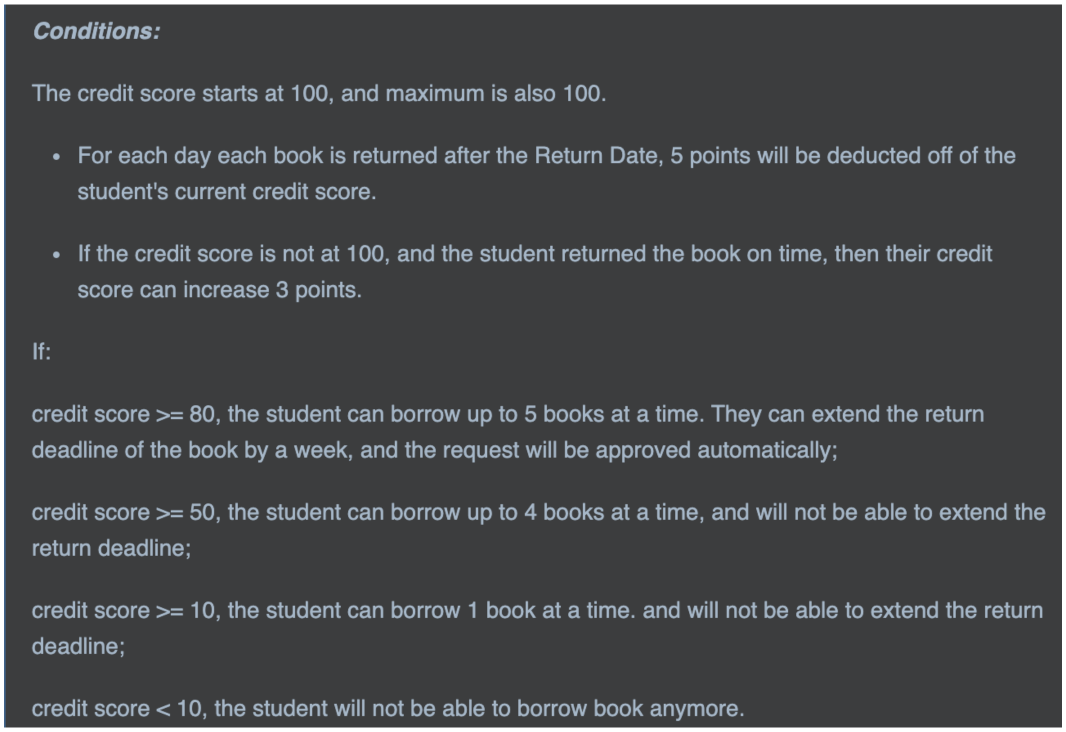
Create a library management system that allows students to borrow and return books with the help of a library staff.

Students need to register a student account in order to borrow and return books. A newly registered student account would have a default credit score of 100 and an empty current-borrowing record. Students are able to login using their username and password and can change their password. Each student is able to access their own credit scores and current-borrowing record. Each student account can borrow at most five books at a time, the number of books students can borrow depends on their credit score (detailed *conditions* attached below).

Each book is given a unique ID when storing into the library, along with their name, ISBN, published date, author, status (whether it is LENDED or not) its Return Date and its type. Different types of books may have different attributes, for example, Textbook would have an attribute "subject", which other types don't. All books would be stored in the inventory.

Each library staff would use a staff account that has different access than student accounts. Staff accounts are able to add new books and delete damaged books from inventory. Their accounts can also change the status of books when books are borrowed and returned. Besides, staff accounts are able to access all student accounts' credit scores and current-borrowing records, and can manually edit student accounts' credit scores.

If students wanted to borrow a book, they are able to use different Search algorithm to find the book they want first, for example, if they know about the id of the book that they want to borrow, they can just input the id to find the exact book. They are also able to search by type, ISBN and author, which would give them a list of books with that meet their requirements. After that, their credit scores and the status of the book that they want to borrow would be checked. ***After searching for the book, students are able to put the book they want in a shopping cart. When they finish searching, they are able to borrow all of the books in the shopping cart at once. If *conditions* are met, the books would be successfully borrowed, if not, students would be noticed by a message about which books are not borrowed.** Students can request to extend their return dates under certain *conditions*.



Conditions:

The credit score starts at 100, and maximum is also 100.

- For each day each book is returned after the Return Date, 5 points will be deducted off of the student's current credit score.
- If the credit score is not at 100, and the student returned the book on time, then their credit score can increase 3 points.

If:

credit score >= 80, the student can borrow up to 5 books at a time. They can extend the return deadline of the book by a week, and the request will be approved automatically;

credit score >= 50, the student can borrow up to 4 books at a time, and will not be able to extend the return deadline;

credit score >= 10, the student can borrow 1 book at a time. and will not be able to extend the return deadline;

credit score < 10, the student will not be able to borrow book anymore.

Part 2. Major design decisions

We have continued to implement and improve our major design decisions made in phase 1, which are MongoDB to store entities information and html web pages as GUI. For more details of these decisions, please see phase 1 design document. * **Here are some major updates on them:**

1. **In phase 1, we implemented MongoDB to store our data persistently (pull request #4 - #6). For phase two, we have continuously improved the database functions by writing test cases and debugging use case classes like DBbookManager and DBUserManager (pull request #20, #24). Now we are confident that our use case methods and database function correctly.**

2. **We also decided to use html pages under the Spring Boot framework and implement the design of the web pages. In phase 2, we are connecting our web pages to our back-end**

codes by writing controller classes (pull request #21, 40, 41, 43). We have implemented all of the controllers classes and now the users can use all functions of our program through the website GUI.

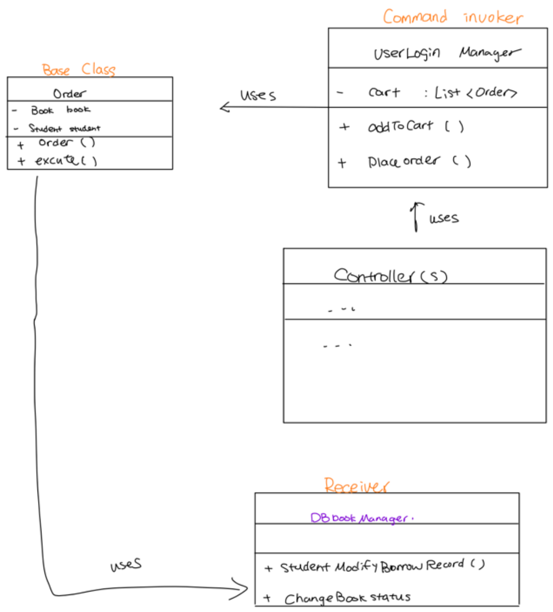
Part 3. Design Patterns

1. **Dependency-Injection** (Implemented at pull request #7 and was updated in later pull requests.)**:** With the implementation of com.bookSystem.mongoDBGateway, we refactored the code from phase 0 using the dependency-injection design pattern. We first created three interfaces for three gateway class, then we 'injected' these interfaces into our use cases, so that our use cases are able to call methods from gateway without the violation of clean architecture principles. More specifically, we split our original UserManager into two new use cases: DBUserManager and UserLoginManager, where DBUserManager manages all users stored in the database and UserLoginManager manage the user entity is currently login. These two use cases would have gateway user methods interfaces inject into their methods' parameter. Similarly, DBbookManager use methods from IMongoDBBookMethods in its parameters.

➢ **A scenario walk-through for this design pattern:** When a student wants to login, checkStudent() methods from the IDBUserManager would be called from the LoginWindow com.bookSystem.controller. Then, this method in the use case would call the gateway interface IMongoDBStudentMethods to check whether the username he inputted is in the student database, if yes, we then call other methods in the gateway interface for our use case to get all information of this student like credit score and borrowing record. After getting these information, we construct a new student entity with them and store it inside the UserLoginManager, return to the com.bookSystem.controller to let him log in.

2. ***Command** (Newly implemented at pull request #30, 31)**:** In phase 2, we add a new "shopping cart" feature to further expand our functionality. With this function, students can borrow/return several books at a time. In order to implement it, we need to use the command design pattern.

The implementation of this design pattern takes place in the useCase package. We add a new class named “Order” that contains an **execute()** method that would **check the status of the current student and the status of the book they want to borrow** to see **whether the action is successful or not**. If yes, it would return true and false otherwise. Then, the **addToCart()** in UserLoginManager is able to add this action into an Arraylist of Order that is stored in this use case. When the method **placeOrders()** in UserLoginManager is called, all the orders would be executed, which means that books in the shopping carts would be execute together, whether being lent successfully or not.

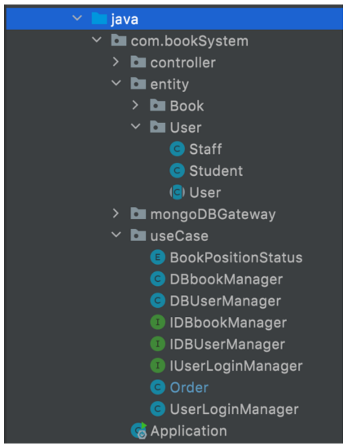


- ➢ **A scenario walk-through for this design pattern:** When a student wants to borrow books, after they use the search algorithm and find out the book they want to borrow, they can add it into the shopping cart, after that, the book would be passed to construct an **Order,** and be placed into the **cart** in UserLoginManager that stores Order objects. Each order would stand for one book. After the student select all the books they wants, they can click the borrow button, which would connect to the **placeOrders()** method in UserLoginManager and all of the orders in the cart would be execute at once, which returns the result that which books are successfully borrowed and which are not.

Part 4. Clean Architecture

In terms of clean architecture, we used the clean architecture by using the **clean architecture layer packaging strategy**, and ***we improved our packaging strategies in phase 2**.

Now we have one "com.bookSystem.entity" that contains Book and User entities, one "com.bookSystem.useCase" package, one "com.bookSystem.controller" package and one "com.bookSystem.mongoDBGateway" package for our codes.



```
java
  com.bookSystem
    controller
    entity
      Book
      User
        Staff
        Student
        User
    mongoDBGateway
    useCase
      BookPositionStatus
      DBbookManager
      DBUserManager
      IDBbookManager
      IDBUserManager
      IUserLoginManager
      Order
      UserLoginManager
    Application
```

***We followed the dependency of clean architecture, for example, our UserLoginManager use case constructs a User Entity which is the student or the staff that is currently login and contains methods to manipulate the entity. When students or staff want to login, the LoginController would call the DBUserManager use case to check whether the user is in the database. After checking and getting information of the user, the LoginController would call the UserLoginManager use case and use the information to construct the user. These are two examples of controller depends on use cases.**

Besides, as we mentioned in the last section, we used the **dependency injection design pattern** to **avoid the violation** of clean architecture principles. We have **gateway classes** like MongoDBBookMethods that manipulates book in the database, and we need **use case** methods in DBbookManager to call these methods. However, this would reverse the clean architecture dependency, so we created an interface for MongoDBBookMethods, and injected it into the use case.

Part 5. SOLID

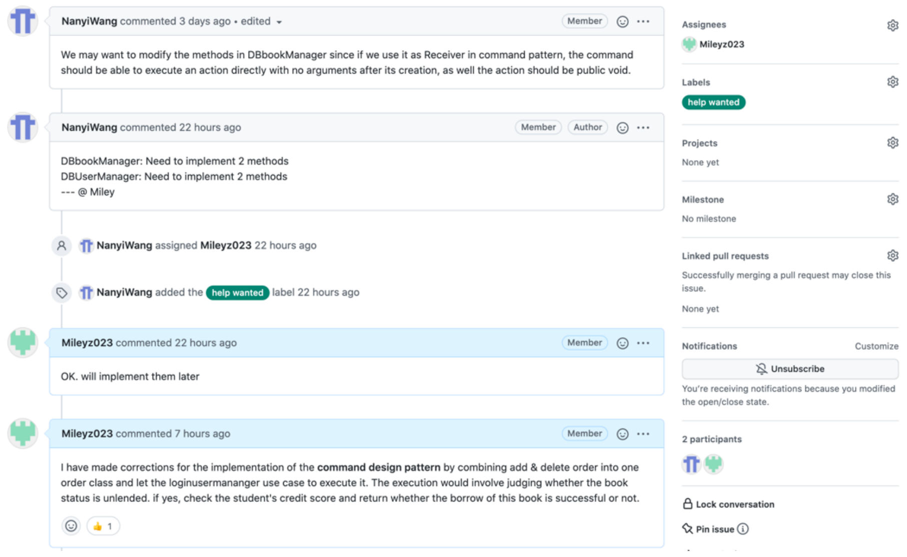
1. **SRP:** we split our original UserManager use case to DBUserManager and UserLoginManager because we changed the methods to store our data. The use case DBUserManager manages all users stored in the database and UserLoginManager manage the user entity is currently login. This follows the **Single Responsibility Principle** that each class should have only a single responsibility, instead of managing all users and the current user in the same class, we split them into two (pull request #7).
2. **ISP:** Our design to extract interfaces for use cases and gateways follows the **Interface Segregation Principle** that we created specific interfaces for other classes to call.

   ***For example, we extracted IMongoDBStudentMethods interface for MongoDBStudentMethods that manipulates student database, IMongoDBStaffMethods for MongoDBStaffMethods that manipulates staff database separately and they can be injected into DBUserManager use case so that this class can use gateway methods (pull request #7).**
3. **DIP:** Our implementation of dependency injection pattern by creating interfaces for gateways as an abstraction layer, so that use cases can call the methods into without violating the dependency. This follows the **Dependency Inversion Principle**. (*For more examples, please see Part3(1)) (pull request #7).

Part 6. Use of github features

*After phase 1, we received the feedback that encouraged us to use more functions of github. Except the **pull request** that we are using continuously, our team have learned to use the github **Issues** to share questions and difficulties they encountered and assign works to each other.

For example, when Vannie wanted to implement the command design pattern and encounter a problem that needs to implement several other methods, she then used the Issues to ask Miley for help (Issue #29).



NanyiWang commented 3 days ago • edited

We may want to modify the methods in DBbookManager since if we use it as Receiver in command pattern, the command should be able to execute an action directly with no arguments after its creation, as well the action should be public void.

NanyiWang commented 22 hours ago

DBbookManager: Need to implement 2 methods
DBUserManager: Need to implement 2 methods
--- @ Miley

NanyiWang assigned Mileyz023 22 hours ago

NanyiWang added the help wanted label 22 hours ago

Mileyz023 commented 22 hours ago

OK. will implement them later

Mileyz023 commented 7 hours ago

I have made corrections for the implementation of the **command design pattern** by combining add & delete order into one order class and let the loginusermananger use case to execute it. The execution would involve judging whether the book status is unlended. if yes, check the student's credit score and return whether the borrow of this book is successful or not.

Assignees: Mileyz023
Labels: help wanted
Projects: None yet
Milestone: No milestone
Linked pull requests: Successfully merging a pull request may close this issue. None yet
Notifications: You're receiving notifications because you modified the open/close state.
2 participants

Part 7. Code styling, documentation and testing

1. **Code styling:** In phase 1, we have the problem of duplicated codes and long method bodies in our Gateway Classes. Therefore, *****we refactored the codes by extracting out helper methods (Pull request #27, 28). For example, checkDb() and checkdatastored() are two helper methods in MongoDBBookMethods, which can be called by other methods in the same class and save the code. We have resolved most of the errors and warnings in Intellij.**
2. **Documentation:** We have documented all of the entity, use case, ***controller** and gateway classes. ***We also had brief documents for our test classes so that people who see them have the idea that what they are testing.**
3. **Testing:** In phase 1, we have covered tests for entities and use cases. ***In phase 2, we continued to implement more test cases to make sure that our back-end codes work well to support the implementation of our controllers and GUI. For example, we implement test cases for book subclasses (Pull request #27, 28). Besides, we add test cases for our newly implemented Order use case and new methods in UserLoginManager use case (Pull request #51).**

Part 8. Progress Report

1. A brief summary of what each group member has been working on since phase 1

Miley: Correct the implementation of Command design pattern, run tests and debugging codes for use cases methods, write design document & accessibility report.

Nicole: Write Controller classes for the student side in order to connect our back-end code with front-end, providing idea for command pattern.

Vannie: Implementation of Command design pattern, write accessibility report.

Stark: Write Controller classes for the staff side in order to connect our back-end code with front-end.

Hewitt: Refactor codes and write comment docs for mongoDBbook methods. Add test cases for book subclass and UserLoginManager.

Kris: Refactor & add comments for mongoDBstudent/staff methods.

2. Significant pull request for each group member:

Name	Link	Reason
Miley	https://github.com/CSC207-UofT/course-project-ask-the-magic-conch-shell/pull/7	This pull request is significant because it includes the implementation of the dependency injection design pattern, which created interfaces for the gateway methods to be called by use cases. This helped us to not violate the clean architecture dependency. Besides, in this pull request I separated the original UserManager into DBUserManager and UserLoginManager, which followed the Single Responsibility Principle in SOLID.
Nicole	https://github.com/CSC207-UofT/course-project-ask-the-	This pull request is significant since it creates multiple controller files and are implemented.

	magic-conch-shell/pull/43	Providing a good thymeleaf and spring boot usage examlple for my teammates. These controllers also give an base look on how the website intend to look like and work, as it allows users to now login and do some basic operations on their account.
Vannie	https://github.com/CSC207-UofT/course-project-ask-the-magic-conch-shell/pull/30#issue-1070273339	The pull request includes codes for Command Design Pattern, which we applied this design pattern to construct a shopping cart function for our library management system. This design pattern encapsulates a request as an object, thereby it is able to support undoable operations of a client.
Stark	https://github.com/CSC207-UofT/course-project-ask-the-magic-conch-shell/pull/40	This pull request is very important because the controller part is basically completed here. In addition, during this process, we discovered some entity naming problems and solved the problem that the controller could not connect to the front-end html due to the lack of some use case methods. At the same time, in the process of completing the controller, the front-end html is also improved.
Hewitt	https://github.com/CSC207-UofT/course-project-ask-the-magic-conch-shell/pull/8/commits (add search methods)	This adds an important searching function to our system which improves the convenience of finding a book.

Kris	https://github.com/CSC207-UofT/course-project-ask-the-magic-conch-shell/pull/4/files	This was the big breakthrough that finally connect Mongodb database into our code. Everyone on the team can now access to the database easily through gateway interfaces