SC2002 Object Oriented Design & Programming

Assignment Submission

Declaration of Original Work for CE/CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature/Date
Le Nguyen Bao Huy (U2322337G)	SC2002	SCS3	[illegible] 19/11/2024
Bui Gia Nhat Minh (U2320021L)	SC2002	SCS3	[illegible] 19/11/2024
Dao Hai Nam (U2323251H)	SC2002	SCS3	[illegible] 19/11/2024
Nguyen Tung Lam (U2323471J)	SC2002	SCS3	[illegible] 19/11/2024
Vu Thao Nguyen (U2322303E)	SC2002	SCS3	[illegible] 19/11/2024

Table of content

1. Design Considerations

To better understand the examples given below, we recommend you to open the UML diagram file at the same time.

1.1. Approach

The HMS application is the implementation of a hospital appointment booking system, which is designed and programmed using an object-oriented approach. The system includes 4 main stakeholders: Patients, Doctor, Pharmacist, and Administrator, with various functions mirroring the actions of these stakeholders in real-life hospitals. Overall, we have split our projects into 3 main parts: The `system` and `ui` package handles the input and output display; the `system.service` and `manager` package handles the interactions between the respective users and the models; and lastly the models themselves (`appointment`, `inventory`, `prescription`, `user.model` and `user. repository` packages) to store and load the data, as well as processing requests from the system service internally within the class. Besides these, we also have utility packages like the `utils`, `common` and `common.id` package to help the project to be more organized overall.

1.2. Principles used

<u>Object - Oriented Model Concepts used</u>

Abstraction

We have used multiple interfaces such as `IModel (common package)` and `IManager (manager package)`, and abstract classes such as `AbstractRepository (common package)` and `AbstractManager (manager package)` to capture the common methods and attributes that need to be implemented. These common methods ensure that the concrete classes adhere to consistent contracts. These abstraction methods also hide the code's implementation, making these classes less prone to change when new features are added.

Encapsulation

We have practiced encapsulation by using the class `ManagerContext (manager package)`. Through this class, other classes are provided with the necessary methods and data of other classes, and they need to learn exactly how it is implemented. This will promote security as it hides the name of the methods in the original class, but it makes interaction between classes easier.

Moreover, we have utilized the encapsulation in the start screen of our program. When a user goes into the `Main` class to run the app, they can only see a class `App` being called and run, without knowing in detail how the code was implemented. This provides security, as well as convenience for the user.

Inheritance

We have used inheritance to group the common methods of many classes into one superclass, reducing the repetitive codes. To further enhance this, we used dynamic binding to group the methods of different classes with different data types for parameters, but the same structure overall. Examples of these can be found in the class `UserRepository (user.repository package), AbstractRepository, AbstractManager.`

Polymorphism

We have used polymorphism throughout the assignment. One example from our code would be the `ManagerContext` class, which used the interface `IManager` to call out the classes in the manager context, as well as using different classes' methods dynamically. Here is a code example:

(In `ManagerContext`)

```
        public void addManager(Class<?> managerClass, IManager
manager) {
            manager.initialize(); // Initialize the manager if it
hasn't been initialized yet
        managers.put(managerClass, manager);
    }
```

Here we have used a parameter of type `IManager` to save any kind of Manager that implements this interface into the dictionary. We have upcasted an arbitrary Manager to IManager to dynamically save all of them into the dictionary.

Another method in this ManagerContext class also demonstrates this polymorphism well:

(In `ManagerContext`)

```
    public void save() {
        managers.values().forEach(IManager::save);
    }
```

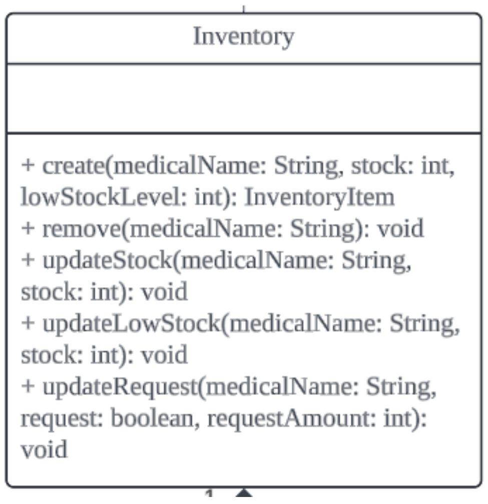
Here we have called the method `save` inside an upcasted `IManager` object stated above. This `save` will be executed by the relative Manager class. For example, if we have put in `UserManager(manager package)` then this method will be executed by this `UserManager` class, demonstrating well the working of polymorphism. This is especially useful when we consider the open-closed and dependency inversion principles which we will mention below.

<u>Object - Oriented Design Principles followed</u>

Single Responsibility Principle

Several classes in our design have shown mindfulness to the Single Responsibility Principle, where these classes have only one reason to change. Following are some of the examples:

The `Inventory` class `(inventory package).`


Inventory
+ create(medicalName: String, stock: int, lowStockLevel: int): InventoryItem
+ remove(medicalName: String): void
+ updateStock(medicalName: String, stock: int): void
+ updateLowStock(medicalName: String, stock: int): void
+ updateRequest(medicalName: String, request: boolean, requestAmount: int): void

This class works very similarly to a usual list, where its responsibility is to add, remove, and modify the item of type `InventoryItem` only. The only reason that this class might change is that the attributes of `InventoryItem` are altered. This applies to other repositories as well. Apart from these examples, most of the other classes of our design have followed this principle based on our interpretation.

Open - Closed Principle

This principle states that the software program should be open for extension, but closed for any modification to the current code.

One example that follows this principle in our project would be the use of generic types in `UserRepository` where they can be dynamically bound to support new object types, as mentioned in the **Inheritance** section above. This ensures that when we introduce a new type of object, like `Visitor` for example, we are able to add onto the current implementation without changing the logic of the current `UserRepository` class. In this example, we only need to create a new concrete class `VisitorRepository` to inherit from `UserRepository` because it is an abstract class.

Liskov Substitution Principle

Throughout our design, we have handled carefully all of the sub-class to make sure none of the methods they implement might incur new errors that the superclass does not have. All of the classes that are prone to errors such as those that are handling the I/O process, we put them as the "high-level" classes that do not inherit from any classes, but only implement interfaces. Some examples of such classes are `AbstractManager, UserManager,` handling the file IO from the database, and `LoginSystem (system package),` handling the data input from the user.

For all other classes, whenever there is a conditional statement, we design it so that it will not raise a fatal error that will throw an exception. Rather, if unintended input is given, then a default output will be returned. We have done this by testing multiple cases, especially edge cases, and catching all the respective exceptions that might be thrown, to prevent the system from crashing.

To make sure the subclass does everything that the superclass does, many of our subclasses inherit from abstract classes or implement interfaces, and only implement the abstract methods of the superclass. For example, `PatientRepository (user.repository package)` inherits `{abstract} UserRepository<Patient>.`

In very rare cases where we inherit from a concrete class or need to overwrite the methods, we have examined carefully to make sure no functionality of the superclass is lost.

Interface Segregation Principle

For every interface in our design, we have only put in the most common methods that every class that implements the interface will use. For example, our `IModel` interface only has 2 methods: `hydrate()` and `serialize()`. This serves as a load and save method to import and export data from and to an external source outside an object. Every Model object needs to be loaded with data saved in external files (CSV for example) to fully function, its data later needs to be saved into the external files so it is non-volatile.

Other interfaces like `IManager`, etc, similarly, only have an initialize, load, and save function, which every object must do before working. Overall, all of our interfaces are small enough and contain only the most necessary methods. This ensures that no class is implementing a method they do not need to satisfy interface implementation.

Dependency Inversion Principle

We have tried to design so that classes depend on the interface of other classes but not the concrete class itself. Some classes in our implementation follow this design. For example, as mentioned above in the polymorphism part, the class `ManagerContext` uses the interface of other Manager classes (i.e. `IManager)`. This helped the `ManagerContext` to save and call other Manager classes more efficiently. Moreover, this design helped `ManagerContext` be less prone to change when there are more types of models and managers added to the system.

Object - Oriented Design Patterns followed

Singleton Pattern

The Singleton Pattern is a design pattern that ensures a class has only one instance and a global reference to it. We have followed this design pattern throughout our project with the `Manager` and `InputHandler` classes, where for the manager of each type, we only create 1 repository throughout. This ensures the data accessed by different users are consistent with each other.

Factory Method Pattern

The Factory Method Pattern is a creational design pattern that provides an abstract method to create an object in a superclass, but subclasses can create different types of objects. In our design, the

`UserRepository` class demonstrates this method well where each of its subclasses can create an empty model of their type. This was also mentioned in the Inheritance and Open-closed principle above.

1.3. Assumptions made

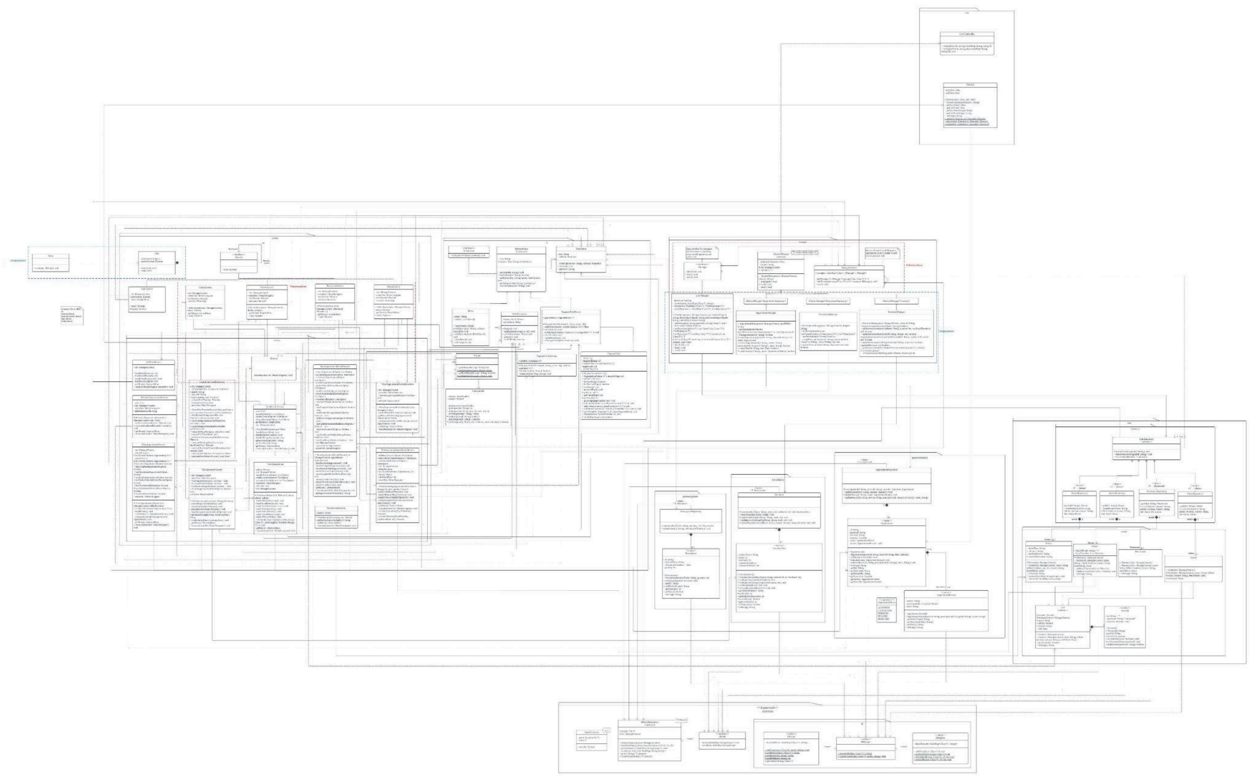
- No database tools (e.g. My SQL, MS Access) are permitted so the data saved in CSV files are assumed to follow the code's legal format.
- This system does not support multiple users at the same time.
- We used SimpleDateFormat to store the date of birth. If a user puts in an invalid date (e.g. 32/3/2003), we will assume and automatically calculate the legal date ourselves. In this example, the D.O.B of the user will be 1/4/2003. Only when the format is wrong there is an error message.
- The data of the users' profiles is all that the hospital needs. Other data (e.g. home address) are not needed.
- Each appointment starts in a half-an-hour bin (they can start at 10:00 or 10:30, but not 10:15, etc), and the length of each appointment is a multiple of 0.5 hours.

1.4. Additional features

- **Paginated list feature:** The items don't show as a big list, but show as many pages. We implemented this for lists of appointments and lists of users. This helped to make the user interface more organized
- **Choosable list feature:** Each item on the list can be selected to perform further action, while also adding/removing/modifying like the normal list item. We have designed this using a dynamic menu class `SimpleMenu.` This enabled us to add/remove choices from the list and change the menu during runtime without having to hardcode it.
- **Filter feature:** A systematic filter feature has been embedded in the back-end to help implement functions such as: All users can use filters to check out which appointments are pending/canceled/accepted/rejected/finished, Admin can use filters to check out each type of staff, etc.
- **Unique Id:** A unique ID manager package is added to ensure that no users/ appointment/ prescription/ item ID clashes with each other.

2. UML Diagram

For a clearer image, refer to the separate image file in the same "Report" folder.



Explanations:

User I/O handling

When a user starts the app, it will initiate itself by filling with saved data from external files via the `CSVFileHandler` class, then, `LoginSystem` class will be called to authorize the user, after which the user will be redirected to their respective role system (e.g. `PatientSystem` for patients). Depending on their choices, they will be redirected to different services and managers to process their request (e.g. `ViewProfileService` if viewing profile is wanted).

Interaction and data handling

In our project, we have designed the manager package as a common environment for all objects to interact with each other. In this package, there are different manager classes such as `AppointmentManager`, `InventoryManager`, `PrescriptionManager`, and `UserManager`. They manage each of the key models in our project (`Appointment`, `Inventory`, `Prescription`, `User`, respectively). They are then integrated into a common `HashMap` inside `ManagerContext`. Whenever an object needs to seek data or interact with other classes, it will call the appropriate manager from context `HashMap` to get the appropriate action. For example, `ViewingInventoryService` will call the `InventoryManager` via `ctx.getManager(Inventory.class)` to get access to the `Inventory` class, etc.

3. Testing

For full code output images, refer to the "Test output" folder in the "Report" folder. Note that for test cases we are referring to those used in this table and not in the assignment file.

#	**Test case description**	**Results and notes**
1	All users log in with the correct password, change the password, and then log in again. They should be allowed access and the new password should be used to log in.	Everything is working as intended
2	All users log in with the wrong password or userID or blank userID/password. They should not be able to log in and a printed warning should appear.	Everything is working as intended
3	All users should be able to view their own profile to see their general information	Everything is working as intended
4	Patients view their own medical records like Blood Type, Conditions, and Past medications and also view their personal information such as Name, Gender, Contact number, Day of birth	Patients can see their personal information or medical records, but not at the same time
5	Patients can update their profile information, but not their medical records.	Everything is working as intended
6	Patients can view available time slots to set appointments with doctors. The slots must include the Doctor's name, date, and time.	Everything is working as intended
7	Patients can choose a timeslot to schedule the appointment. This time slot will become unavailable to other patients.	Everything is working as intended
8	Patients can reschedule their appointments. The old time slot must open, while the new time slot becomes unavailable to other patients.	Everything is working as intended
9	Patients can cancel the appointment. This time slot will become available to other patients.	Everything is working as intended
10	Patients can view their upcoming appointments. This must also include information such as the doctor's name, date, and time.	Everything is working as intended
11	Patients can view the outcome of completed appointments. They should be able to see the services received, notes from doctors, and all prescriptions given to them.	Everything is working as intended
12	Patients try to view the upcoming/finished appointments when they have not had any appointments before. A message saying they don't have any appointment should appear, without interrupting any other services.	Everything is working as intended
13	Doctors can view the medical records of all patients they serve. All past medical records should appear with relevant information.	Everything is working as intended
14	Doctors can update the medical records of a patient and it will be reflected in the medical record when the patient views it.	Everything is working as intended
15	Doctors can both view and set their busy schedules so patients are not able to choose those time slots (default: free all day, add busy time slots)	Everything is working as intended. Hard to find the option. (In View Upcoming Appointment)
16	Doctors can accept or decline the appointment request from the patient	Everything is working as intended
17	Doctors can conclude the outcome of an appointment. After which they can no longer update the medical record of that appointment and the status is reflected as "finished". They should still be able to view the outcome records.	Everything is working as intended

18	Doctors can choose to filter the appointments to see which one is “pending”, “canceled”, “accepted”, “rejected”, or “finished”. After filtering, these appointments should still hold relevant information	Everything is working as intended
19	Pharmacists can view the prescriptions in the appointments’ outcome records	Everything is working as intended
20	Pharmacists can view the appointment details	Everything is working as intended
21	Pharmacists can update the prescription status as dispensed or not	Everything is working as intended
22	Pharmacists can view the items inside the inventory with all their relevant details	Everything is working as intended
23	Pharmacists have the option to request more items. This request can be seen by the Administrator	Everything is working as intended
24	Administrators can view all users in the system and their information	Everything is working as intended
25	Administrators can disable or enable every account in the system.	Everything is working as intended
26	Administrators can view all appointments and their details	Everything is working as intended
27	Administrators can view all items in inventory, update stock, resolve requests, and change low stock alert	Everything is working as intended
28	All users can log out. All data saved on the system is not lost when logging into another account to see common data	Everything is working as intended

4. Reflection

Difficulties encountered

The most significant challenge we encountered was the complexity of the project. Because there were a lot of requirements for different objects to adhere to, we needed to think of our design to ensure functionality and coherence carefully. On top of that, we needed to make sure every member in the group understood the strategy the group was taking, to be able to generate code that does not raise any error but works in harmony with others. Due to the sheer size of the project, this was a considerable obstacle.

To address this, we have spent a lot of our time thinking of the design and coming up with an appropriate UML diagram - the skeleton of the whole project. Moreover, we had constant updates on our group chat, as well as group meetings in our lab session to get everyone on the same page. We have also trained ourselves to use naming conventions, commenting, and documenting codes whenever we are done with coding our parts.

However, our emphasis on design and back-end development was at the expense of output generation, making the debugging step more difficult. As a result, we had to revise parts of our initial design, causing some classes (such as Model System and System Services) to temporarily violate design principles like the Open-Closed Principle and Dependency Inversion Principle by relying on concrete implementations rather than abstractions. A few classes did not strictly follow the Single-Responsibility Principle.

We are currently addressing these concerns by speeding up the debugging process to give time for refactoring. Fortunately, because the majority of our classes were developed with these concepts in mind from the start, incorporating the problematic classes back into a compliant framework should be a pretty simple process.

Knowledge learnt from the course

Throughout the course of this project, we have learned how to design and build a functioning application. We have learnt how to apply OOP foundational concepts such as Abstraction, Inheritance, Encapsulation, and Polymorphism and how to adhere to the SOLID design principle, as well as learning about new design patterns such as Singleton and Factory Method patterns that was not taught in class. We learnt that to make a working program, especially for those that can extend their application, coding is not the only important part, but designing and thinking of the approach is just as important.

Beyond coding, we became proficient with Github and Git, learning how to create pull requests, merge branches, and push and pull changes, among other crucial activities. To maintain the integrity of the common codebase, we also devised effective ways to resolve merge disputes. Working collaboratively, we explored advanced Git practices, including creating branches for feature development, reviewing and approving pull requests, and using these tools to facilitate teamwork. These practices allowed us to work asynchronously while maintaining a stable and functional codebase.

Further Improvements Suggestion

Our application can be further improved by introducing more packages and interfaces to break down the classes, to further organize the code, while making the classes follow the Single Responsibility Principle more strictly, preventing future features or discoveries of bugs from disrupting this adherence.

As the system extends further, new types of users can be added to the system such as: Visitor, Nurses, Office Staff, Cleaning staff, etc. With the current implementation with most of the system following design principle, these new classes with new actions and requirements can be added seamlessly.

We can add the export feature to more types of files, like HTML or XLSX, to let patients print out their medical records, doctors print out their working schedules, administrators do relevant office work, etc.

Lastly, we can upload the system to a local network/ online web to let multiple users access and use this system concurrently. Most of our architecture already supports this, where we use a common system service and a manager center to process all requests. But for now, only one user can log in at a time because our app is an offline application.

5. Github Repository Link

Github Repository Link: https://github.com/minhbgn/SC2002-Assignment/