MyHealth EHR - Final Documentation & Testing Reports

Atmiya Patel & Ben Santhosh

1. Purpose

This document explains the design decisions and reasoning behind the use of specific software design patterns during the development of our Electronic Health Record (EHR) system. The specific design patterns we used are Singleton (Creational), Builder (Creational), and Chain of Responsibility (Behavioural). The document also details the system's overall architecture, challenges faced, solutions implemented, and the rationale for key design choices made to address system requirements and challenges.

2. System's Overall Architecture

The architecture of our Electronic Health Record (EHR) system is built to support scalability, maintainability, and efficiency. It is structured into a three-tier architecture: Presentation Layer, Application Layer, and Data Layer, with the integration of Object-Oriented Design (OOD) principles and design patterns. - https://www.ibm.com/topics/three-tier-architecture

1. Presentation Layer (User Interface)

    MainWindow: This is the main page which is used for the login, patient record search, and doctor record search. We used the UI select button to figure out whether the user is signing up as a patient or as a doctor.

    Sign-Up Window: This is the main page for when the user chooses to sign up. Through this, we are able to get the user's username, password and verification code using TextFields. We've used specific TextEdits to censor the password and verification code to add an additional layer of security.

    Patient Dashboard Window: This is the page that the patient has access to when they successfully sign up, which gives them their detailed medical history in a scrollable QTextEdit widget to accommodate for large datasets.

    Doctor Dashboard Window: This is the page that the doctor has access to when they sign up, which gives them access to search for patients, add records, and view medical histories.

    Add Patient Window: This is the page that allows the doctor to add a patient to their database.

Search Patient Window: This page allows doctors to search for their patients in their database. If a patient is found then, they can add a record for them or view their records.

Add Record Window: This page allows doctors to add a record for their patients in their database.

Responsibilities:

These classes provide an intuitive and user-friendly interface. They interact with the application layer in Qt to handle user actions (login, patient search, signup, etc) and mediates between user inputs and the backend logic.

2. Application Layer

Singleton Classes:

Doctor: This class maintains a single instance of the logged-in doctor to ensure global access to data throughout the system.

Patient: This class maintains a single instance of the logged-in patient to ensure global access to data throughout the system.

Builder Classes:

RecordBuilder: This class helps build the record object step-by-step, when the doctor chooses to add a record for a selected patient.

Record: The class whose object is built using the Builder class.

SignUp: Provides implementation of the SignUp class, which handles registration of doctors and patients in the system. It handles verification codes, checks username existence (for patients), and updates or inserts user credentials into the database. Also, it includes password validation to make sure it is 8 characters.

User: Provides the implementation of the User class, which represents the base class for different types of users in the system, such as doctors and patients.

DBManager, DBPatient, DBDoctor, RecordManager: Explained in the Data Layer section.

3. Data Layer (Database Management)

For this, the SQLite Database was used throughout the project. Tables for Doctors, Patients, and PatientRecords store user credentials, relationships, and medical histories were used.

DBManager Class: Provides a centralized interface for database operations and prepares secure queries.

DBPatient Class: This class handles the "Patients" table, which includes creating the table, inserting new patient records, and verifying if a patient's username exists in the table

DBDoctor Claass: This class handles the "Doctors" table, which includes creating the table, inserting new doctor records, and verifying if a doctor's username exists in the table

RecordManager: Initializes the RecordManager and connects to the SQLite database, which uses the Record table.

Responsibilities:

Maintains data consistencies using SQL queries, protects sensitive data, and successfully adheres to role-based access control, making sure only authorized users can access specific records.

Object-Oriented Design (OOD) Principles

1. Single Responsibility Principle (SRP)

Every class in the system is designed to have a specific, well-defined responsibility:

- DBDoctor and DBPatient manage database interactions for their respective roles.
- Doctor and Patient focus on handling user-specific session data.
- Each UI component (e.g., SearchPatientWindow, AddRecordWindow) is responsible only for the associated user interaction.

2. Open-Closed Principle (OCP)

The system is designed to be open for extension but closed for modification:

- Adding new user roles, such as a Hospital Head, requires extending the Chain of Responsibility pattern to include new handlers without modifying existing code.
- The use of Builder Pattern allows the creation of new types of records or data objects by extending the builder classes, and creating Director classes.

3. Low Coupling and High Cohesion

The architecture minimizes dependencies between components:

The DBManager acts as a mediator between the application layer and the database, ensuring the user role remains independent of their particular database operations.

The Singleton instances ensure shared session data is managed consistently without requiring tight coupling between UI components.

4. Inheritance and Polymorphism

The Doctor and Patient classes inherit from the base User class: Shared attributes like username and password are managed in User, while role-specific functionality is implemented in the subclasses.

Polymorphism allows the system to treat both Doctor and Patient objects uniformly when accessing shared methods.

The DBPatient and DBDoctor, RecordManager classes inherit from the base DBManager class. They share attributes, however, each of the DBPatient, DBDoctor, RecordManager have their own functionality depending on their tables.

Scalability, Maintainability, and Efficiency

1. Scalability

The use of modular classes and patterns like Chain of Responsibility ensures new roles or functionalities can be added seamlessly.

The entire architecture supports growth which includes additional managing of healthcare processes (e.g., appointment scheduling, analytics, etc).

2. Maintainability

Clear separation of concerns and adherence to OOD principles make the system easier to understand, modify, and debug. It can also be easy for the user to understand, and any new developers to understand the code. This modular design also helps make sure that individual components can be tested independently.

3. Efficiency

The Singleton Pattern reduces redundant object creation, improving performance. Usage of SQL queries ensures fast, secure database interactions. Also, scrollable widgets in the UI provide efficient data presentation for large datasets.

3. Pattern Justification

1. Singleton Pattern

- **Why it was chosen**: The Singleton pattern was used for the Doctor and Patient classes to ensure that only one instance of a logged-in user exists at any given time. This was necessary for our implementation as we needed to maintain only 1 global access to the user without unnecessary duplication or change to other users.
- **How using Singleton Pattern addresses challenges**:
    - **Challenge**: We needed to find a method to efficiently manage user identity across multiple windows and calls of methods.
    - **Solution**: By using the Singleton Design Pattern, we were able to ensure that all windows (SearchPatientWindow, AddRecordWindow, etc) have access to the same instance of the Doctor or Patient. This ensures consistent and secure handling of the currently logged-in user's data without having redundant parameters across multiple classes. This allows us to maintain much more simplicity in our code, making it a much more efficient implementation.
    - **Example Use**:
        - A doctor's credentials and actions (e.g., adding or viewing patient records) are done using the Singleton instance of the Doctor class.

2. Chain of Responsibility Pattern

- **Why it was chosen**: The Chain of Responsibility pattern was used for login verification in the DBManager class. Using this pattern, we were able to delegate the responsibility of validating credentials to the appropriate database table (Doctors or Patients) based on whether the user is logging in as a patient or a doctor.
- **How it addresses challenges**:
    - **Challenge**: A challenge that was faced was handling the login verification for two distinct user types in a modular, extensible way. We needed a way to incorporate modularity in our code and to have a concise way of handling the credentials verification.
    - **Solution**: By using Chain of Responsibility, the login process delegates the validation task to the correct handler (Doctors or Patients table). This method ensures a clean, maintainable design that can easily accommodate additional user types in the future such as the Hospital Head who wants to have access to all of the doctors and all the patients in the entire hospital.

    - **Example Use**:
        - In DBManager::verifyLogin, the system checks whether the user role is “DOCTOR” or “PATIENT” and then forwards the responsibility to the relevant method that we have in our modular classes (either DBPatient or DBDoctor) accordingly.

3. Builder Pattern

- **Why it was chosen**: The Builder pattern was used for constructing complex objects such as the Record where we built objects using a Builder. By using this pattern, we were able to separate the construction process from the representation, which further allowed flexibility and modularity in the creation and management of these objects. We wanted to create a system that is open for extension but closed for modification, which is why the Builder pattern was chosen. We wanted to have our system open to have different types of records such as “Urgent Records”, and “Records from past 6 months”, which we can just include a Director class to create.
- **How it addresses challenges**:
    - **Challenge**: We needed a way to create detailed and structured patient records without cluttering the code with long constructor calls or by having repeated configurations.
    - **Solution**: The Builder Pattern allowed us to create these complex objects step-by-step and in a controlled manner. We were then able to use the Builder to construct a way to display all the records on the UI. This way, a patient’s record could be built by adding data such as previous illnesses, medications, and treatments dynamically through the builder. This also made the code more readable and maintainable.
    - **Example Use:**
        - When a Doctor adds a record for a patient, it uses the Builder class to build an object of the Record, and pass it into the database for it to be retrieved later

4. Testing Report

DBManager: Key class that handles login verification. We emphasized security, so this class was extensively tested to ensure logging in to our system is thoroughly tested.

```
// Test 1: Successful login verification
TEST_F(DBManagerTest, VerifyLoginSuccess) {
    EXPECT_TRUE(dbManager->verifyLogin("test_user", "test_pass", "doctor"));
}

// Test 2: Login verification fails with incorrect password
TEST_F(DBManagerTest, VerifyLoginFailsIncorrectPassword) {
    EXPECT_FALSE(dbManager->verifyLogin("test_user", "wrong_pass", "doctor"));
}

// Test 3: Login verification fails with incorrect username
TEST_F(DBManagerTest, VerifyLoginFailsIncorrectUsername) {
    EXPECT_FALSE(dbManager->verifyLogin("wrong_user", "test_pass", "doctor"));
}

// Test 4: Login verification fails with incorrect role
TEST_F(DBManagerTest, VerifyLoginFailsIncorrectRole) {
    EXPECT_FALSE(dbManager->verifyLogin("test_user", "test_pass", "admin"));
}

// Test 5: Edge Case - Empty username
TEST_F(DBManagerTest, VerifyLoginEmptyUsername) {
    EXPECT_FALSE(dbManager->verifyLogin("", "test_pass", "doctor"));
}

// Test 6: Edge Case - Empty password
TEST_F(DBManagerTest, VerifyLoginEmptyPassword) {
    EXPECT_FALSE(dbManager->verifyLogin("test_user", "", "doctor"));
}
```

- **Test 1:** testing if login works with valid login credentials. EXPECT_TRUE because the login exists in the Doctor database. Test passed.
- **Test 2:** testing if login works with an incorrect password. EXPECT_FALSE because the login combination should not work with an incorrect password. Test passed.
- **Test 3:** testing if login works with an invalid username. EXPECT_FALSE because the login combination should not work with an incorrect username. Test passed.
- **Test 4:** testing if login works with an invalid role but correct username and password. EXPECT_FALSE because a valid role is required to log in. If the role is not “DOCTOR” or “PATIENT”, login fails. Role selection is done

through a required radio button on the login screen so specifying roles is simplified. Test passed.

- **Test 5:** testing if login works with an empty username. EXPECT_FALSE because accounts cannot be made with an empty username field. Test passed.
- **Test 6:** testing if login works with an empty password. EXPECT_FALSE because accounts cannot be made with an empty password field. Test passed.

SignUp: Key class that handles account creation. This class goes hand-in-hand with the DBManager class by creating and adding accounts to the database. Thus, extensive testing was done on this class to ensure accurate functionality.

```
// Test: Successful doctor registration
TEST_F(SignUpTest, RegisterDoctorSuccess) {
    EXPECT_TRUE(signUp->registerUser("doctor1", "password123", "DOCTOR"));
}

// Test: Doctor username already exists
TEST_F(SignUpTest, RegisterDoctorUsernameExists) {
    EXPECT_FALSE(signUp->registerUser("existing_doctor", "password123", "DOCTOR"));
}

// Test: Patient username already exists
TEST_F(SignUpTest, RegisterPatientUsernameExists) {
    EXPECT_FALSE(signUp->registerUser("existing_patient", "password123", "VALID_CODE"));
}

// Test: Successful patient registration
TEST_F(SignUpTest, RegisterPatientSuccess) {
    EXPECT_TRUE(signUp->registerUser("patient1", "password123", "VALID_CODE"));
}

// Test: Patient registration fails due to invalid signup code
TEST_F(SignUpTest, RegisterPatientInvalidCode) {
    EXPECT_FALSE(signUp->registerUser("patient1", "password123", "INVALID_CODE"));
}

// Test: Patient registration fails due to password already set
TEST_F(SignUpTest, RegisterPatientPasswordAlreadySet) {
    EXPECT_FALSE(signUp->registerUser("patient1", "password123", "VALID_CODE"));
}

// Test: Password validation success
TEST_F(SignUpTest, PasswordValid) {
    EXPECT_TRUE(signUp->passwordValid("strongPass"));
}

// Test: Password validation failure (too short)
TEST_F(SignUpTest, PasswordInvalidTooShort) {
    EXPECT_FALSE(signUp->passwordValid("short"));
}

// Test: Edge Case - Empty username
TEST_F(SignUpTest, RegisterUserEmptyUsername) {
    EXPECT_FALSE(signUp->registerUser("", "password123", "DOCTOR"));
}

// Test: Edge Case - Empty password
TEST_F(SignUpTest, RegisterUserEmptyPassword) {
    EXPECT_FALSE(signUp->registerUser("username", "", "DOCTOR"));
}

// Test: Edge Case - Empty signup code
TEST_F(SignUpTest, RegisterUserEmptySignUpCode) {
    EXPECT_FALSE(signUp->registerUser("username", "password123", ""));
}
```

- **Test 1:** testing if a valid doctor signup combination works. EXPECT_TRUE because this combination of username, password, and code should successfully create a doctor account.
- **Test 2:** testing if a doctor signup with duplicate username works. EXPECT_FALSE because duplicate usernames should not be allowed.
- **Test 3:** testing if a patient signup with duplicate username works. EXPECT_FALSE because duplicate usernames should not be allowed.
- **Test 4:** testing if a valid patient signup works. EXPECT_TRUE because this combination of username, password, and code should successfully create a patient account.
- **Test 5:** testing if an invalid code works for patient signup. EXPECT_FALSE because code must be valid to sign up as a patient.
- **Test 6:** testing if patient password can be set again with valid username and code. EXPECT_FALSE because once a password is set, that account cannot be signed up for again.
- **Test 7:** testing if password validation works. EXPECT_TRUE because this password satisfies requirement of being longer than 8 characters.
- **Test 8:** testing if password validation works. EXPECT_FALSE because this password does not satisfy requirements of being longer than 8 characters.
- **Test 9:** testing if users can sign up with empty username field. EXPECT_FALSE because username is a required field to successfully sign up.
- **Test 10:** testing if users can sign up with empty password field. EXPECT_FALSE because password is a required field to successfully sign up.
- **Test 11:** testing if users can sign up with empty code field. EXPECT_FALSE because signup code is a required field to successfully sign up.

5. Challenges and Solutions

1. Singleton Implementation

- **Challenge**: We needed to avoid unnecessary duplication of attributes already present in the base User class (e.g., username and password) when creating the Doctor and Patient singletons.
- **Solution**: We refactored the Doctor and Patient classes to inherit directly from User, reducing redundancy in the code. Additionally, static instances were initialized with dummy values to ensure that we have safe instantiation.

2. Maintaining Modularity Throughout our Program

- **Challenge**: We thought that coding everything inside DBManager would still result in us having low coupling and high cohesion. However, when we started coding, we realized that was not the best option we had.
- **Solution**: We split the database functionality into separate classes (DBDoctor and DBPatient, DBManager) to adhere to the Single Responsibility Principle. Using this, we were able to ensure that each class handles its specific function while sharing methods like SQL preparation that are used by all classes.

3. Implementing a Realistic & Secure System

- **First Challenge**: Since the purpose of the project is to tackle a real-life problem, our goal was to ensure that the implementation is as realistic and secure as it can be. We did not want anybody to be added as a doctor's patient without being added by the doctor first.
- **Solution**: We ensured that when a doctor signs up using the verification code "DOCTOR", they will only then be signed up as a Doctor. This code is a representation of a unique verification code that would be generated to ensure security. Also, in order for a patient to sign up and view their records, they must be in the database (added by the doctor first). We made sure this is the case by having a method to verify that their health card exists in the database before signing them up.
- **Second Challenge:** We needed to implement a security measure which prevents anybody from signing up as a patient for the first time. If someone got a hold of somebody else's health card, we wanted to make sure that they could not just sign up on their behalf and have access to their records. We wanted to create a security barrier that prevents this from happening.
- **Solution:** We created a textbox called "Verification Code", and the user must enter the doctor's username (that will only be given by the doctor to them) to successfully sign up. We implemented two checks, we made sure that a patient can successfully sign up if and only if the patient has already been added to the database by the doctor and the verification code matches the doctor's username that has been added to their patient log.

4. Passing Patient Information Dynamically

- **Challenge**: We wanted to allow doctors to dynamically access and interact with different patients' data while ensuring modularity.
- **Solution**: Patient data is dynamically fetched using the patientID passed to various windows. We made sure to check that the doctor can only view a patient added in the database by them. Combined with the Singleton pattern for the Doctor, this approach ensured modular access without tightly coupling the doctor's actions with specific patient data.

5. Creating Error Handling

- **Challenge:** Managing errors, such as incorrect login credentials or invalid health card numbers, without overwhelming the user was challenging.
- **Solution:** We implemented detailed and user-friendly error messages displayed in pop-up windows. The usage of Chain of Responsibility pattern helped solve errors to specific handlers, making the error-handling process modular and easy to manage.

6. Handling Large Database Queries Dynamically

- **Challenge:** We needed to efficiently fetch and display large sets of patient records and doctor reports from the database without overloading the system or causing any crashes or unwanted behaviour.
- **Solution:** We implemented lazy loading techniques to fetch only the data when it was needed instead of calling it all at once in the beginning. Combined with the Builder Pattern, this allowed us to construct detailed objects dynamically without retrieving all the data from the database at once

7. UI Integration with Scrollable Record Displays

- **Challenge**: We needed a way to show large amounts of data (patient records) in a readable format on the UI. At first, we displayed the records using a label, but we soon realized that if there are more than a certain amount of records added to the patient's record history, it won't all be displayed in a clean and legible format.
- **Solution**: We implemented a scrollable QTextEdit to display detailed records for the patient. Through this, we were able to ensure clarity when displaying the records.

8. Keeping Doctor and Patient Inheritance Instead of Only User

- **Challenge**: Upon receiving feedback questioning the necessity for multiple subclasses (Doctor and Patient), we thought of how we could refactor our design without needing separate Doctor and Patient classes.

- **Solution:** Focusing on scalability, we decided in order to have Singleton Instances of Patient and Doctor, we can seperate them into separate classes to implement the design pattern. This helped us to maintain our code and separate the two roles that we have in our system. We realized that in order to ensure a scalable system where unique features may be added to each role, having separate classes inheriting from User was necessary.