SOFT252: Patient Management System Report

1. Introduction

This document will explain the design on the Patient Management System for SOFT252 Java coursework; alongside with relevant diagrams to display how all the classes are related. The relevant design patterns that have been used within this project will be described, and why they have been chosen over any other design patterns. It will then state what can be done to improve the programme, and end with a conclusion of the system.

In this project, we were assigned to create a system that a hospital might use in order to manage its patients and members of staff, which includes: admins, doctors and secretaries. Each user would have their own unique ID and password that would be used to log into the system. As each user's ID was unique to their position, this means that it would allow them to only perform specified tasks whilst logged in. A different form would need to be generated when they are logged in. Each user's functional requirements are displayed below.

Functionality	User
Log in to the system using unique ID and password, and perform user-specific functionalities.	All users
Request to create account – this requires approval from a secretary.	Patient
Rate doctors and provide feedback messages.	Patient
View doctors' ratings.	Patient
Request appointment – there should be an avenue to ask for a specific doctor and a range of potential dates.	Patient
View his or her own history.	Patient
View appointment.	Patient
View prescription.	Patient
Request account termination.	Patient
Create own account.	Administrator
Add or remove doctor and secretary accounts.	Administrator
View the ratings of the doctors.	Administrator
Provide feedback to each doctor based on ratings and comments from patients.	Administrator
Approve patient accounts.	Secretary
Receive requests for appointments.	Secretary
Create an appointment between a patient and a free doctor.	Secretary
Give medicines to patients if available.	Secretary
Order and stock medicines if necessary.	Secretary
Remove patients.	Secretary
Approve account removal request from patients.	Secretary
View appointments.	Doctor
Make notes during a consultation.	Doctor
Inspect patient history.	Doctor
Propose and create future appointments for a specific patient.	Doctor
Prescribe medicines and dosages.	Doctor
Create new medicines and request secretaries to order these.	Doctor

2. Design Implementation

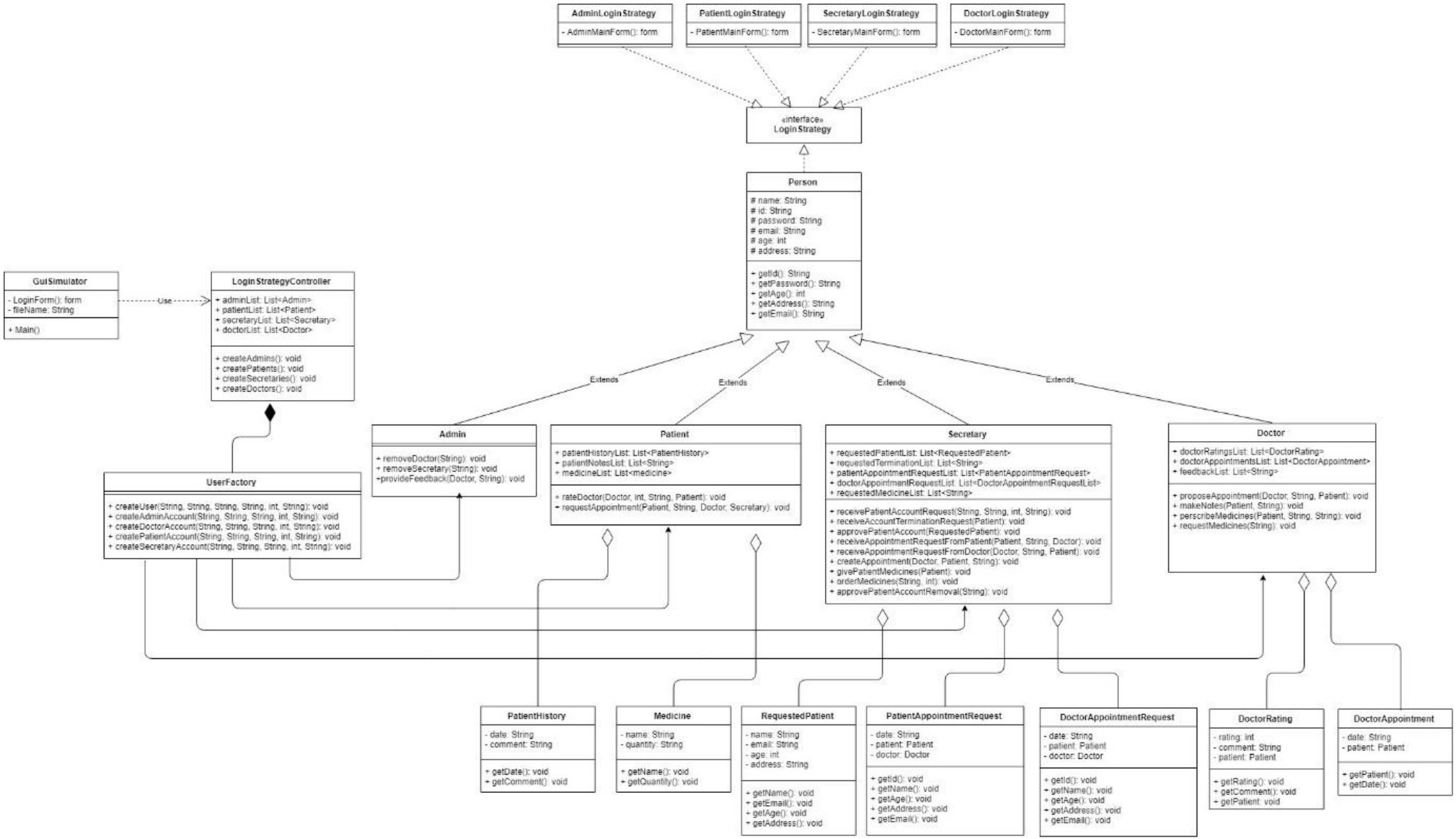
MVC: Following the initial design of the system, it was decided that the project's data will not be implemented into the GUIs and be located in the individual classes or main methods. This would allow for a more reliable system, and thus allowing the code to be reusable without modification, and open for extension. This would therefore allow the Patient Management System to be developed upon in the future. Currently, there are multiple user asset lists that have been implemented from '*GUISimulator*' main to populate the system.

Strategy Pattern: On run time, the user would need to be logged into a GUI that would suite their position within the system. This meant that the programme would require some form of way to distinguish which user was being logged in, and the appropriate GUI would be provided. As a result, the Strategy Pattern was implemented; the programme checks for the details of the ID and produces the appropriate strategy. These could be one of the following: '*AdminLoginStrategy*', '*DoctorLoginStrategy*', '*PatientLoginStrategy*' or '*SecretaryLoginStrategy*' which all implement the Interface '*LoginStrategy*'. The code for each strategy will create the relevant form for the user; for example an Admin would generate the '*AdminMainForm*'. Each user has their dedicated array list which allows the programme to login; it would cycle through the list to find the user with the exact user ID as the one inputted in the text field on the login form. It would also make it convenient for when users needed to be added or removed from the list.

Template Method Pattern: This design pattern has been used to structure the user classes for the system. As a person they would have values assigned to them, such as their name or age. However; in order to ensure that each user had methods specific to their role and responsibility, it was decided to inherit its functionality from an abstract '*Person*' class. As it is abstract, it would not be instantiated, but allows the other user classes to use its methods and variables.

Factory Pattern: In order to instantiate the users from within the system, it was necessary to implement a factory pattern. This therefore offered a simpler way to create each user as the '*UserFactory*' would be called instead of the appropriate user arrayList. This design pattern was implemented specifically for the users; however, it would also be beneficial for any class located in the '*user_assets*' package.

3. UML


AdminLoginStrategy
- AdminMainForm(): form
PatientLoginStrategy
- PatientMainForm(): form
SecretaryLoginStrategy
- SecretaryMainForm(): form
DoctorLoginStrategy
- DoctorMainForm(): form
«interface»
LoginStrategy
Person
name: String
id: String
password: String
email: String
age: int
address: String
+ getId(): String
+ getPassword(): String
+ getAge(): int
+ getAddress(): String
+ getEmail(): String
GuiSimulator
- LoginForm(): form
- fileName: String
+ Main()
Use
LoginStrategyController
+ adminList: List<Admin>
+ patientList: List<Patient>
+ secretaryList: List<Secretary>
+ doctorList: List<Doctor>
+ createAdmins(): void
+ createPatients(): void
+ createSecretaries(): void
+ createDoctors(): void
Extends
Extends
Extends
Extends
Admin
+ removeDoctor(String): void
+ removeSecretary(String): void
+provideFeedback(Doctor, String): void
Patient
+ patientHistoryList: List<PatientHistory>
+ patientNotesList: List<String>
+ medicineList: List<medicine>
+ rateDoctor(Doctor, int, String, Patient): void
+ requestAppointment(Patient, String, Doctor, Secretary): void
Secretary
+ requestedPatientList: List<RequestedPatient>
+ requestedTerminationList: List<String>
+ patientAppointmentRequestList: List<PatientAppointmentRequest>
+ doctorAppointmentRequestList: List<DoctorAppointmentRequestList>
+ requestedMedicineList: List<String>
+ receivePatientAccountRequest(String, String, int, String): void
+ receiveAccountTerminationRequest(Patient): void
+ approvePatientAccount(RequestedPatient): void
+ receiveAppointmentRequestFromPatient(Patient, String, Doctor): void
+ receiveAppointmentRequestFromDoctor(Doctor, String, Patient): void
+ createAppointment(Doctor, Patient, String): void
+ givePatientMedicines(Patient): void
+ orderMedicines(String, int): void
+ approvePatientAccountRemoval(String): void
Doctor
+ doctorRatingsList: List<DoctorRating>
+ doctorAppointmentsList: List<DoctorAppointment>
+ feedbackList: List<String>
+ proposeAppointment(Doctor, String, Patient): void
+ makeNotes(Patient, String): void
+ perscribeMedicines(Patient, String, String): void
+ requestMedicines(String): void
UserFactory
+ createUser(String, String, String, String, int, String): void
+ createAdminAccount(String, String, String, int, String): void
+ createDoctorAccount(String, String, String, int, String): void
+ createPatientAccount(String, String, String, int, String): void
+ createSecretaryAccount(String, String, String, int, String): void
PatientHistory
- date: String
- comment: String
+ getDate(): void
+ getComment(): void
Medicine
- name: String
- quantity: String
+ getName(): void
+ getQuantity(): void
RequestedPatient
- name: String
- email: String
- age: int
- address: String
+ getName(): void
+ getEmail(): void
+ getAge(): void
+ getAddress(): void
PatientAppointmentRequest
- date: String
- patient: Patient
- doctor: Doctor
+ getId(): void
+ getName(): void
+ getAge(): void
+ getAddress(): void
+ getEmail(): void
DoctorAppointmentRequest
- date: String
- patient: Patient
- doctor: Doctor
+ getId(): void
+ getName(): void
+ getAge(): void
+ getAddress(): void
+ getEmail(): void
DoctorRating
- rating: int
- comment: String
- patient: Patient
+ getRating(): void
+ getComment(): void
+ getPatient: void
DoctorAppointment
- date: String
- patient: Patient
+ getPatient(): void
+ getDate(): void

4. Areas for Development

The system has been created with time in consideration; it is a very large project which would take multiple months by a few programmers. As a result, there are certain aspects of the programme that can be improved upon; some of which have been added to a degree, whilst others have not been implemented into the project at all. Firstly; any dates that have been used in the project have not been implemented in such a manor that the user cannot select anything other than a date. This is due to the use of Text Fields, thus giving the user the ability to type anything they want.

Currently, serialization has been implemented with the users; however, it has not been used in any of the user assets for the project. This would need to be implemented if changes were to be saved when exiting the programme. Some users are initialised on start-up to prevent a lockout if the data.bin file were to become corrupted; which acts as a safety net, particularly for new users with lack of experience. In some instances, Netbeans proved to be troublesome; for instance my project could not generate a JavaDoc file.

Thirdly; medicines have been added to the system, however they have not been implemented with any design pattern. As a result, this part of the system is not well organised. In order to improve this, the medicines would need a design pattern, such as factory or observer. A factory pattern could be used for creation of the different medicines and dosages, whilst an observer pattern could maintain stock inventory, whilst updating the observers in the system.

5. Conclusion

Overall, the system that has been created is successful in completing all of the functional requirements, as they have all be incorporated into the programme. Certain aspects can be developed; however, every system is bound to have a few problems. If there had been more time, these minor inconveniences would be tweaked in order to make the programme more stable and possibly, more efficient. One thing that would be done differently, if given the chance, would be to spend more time researching the benefits of each design pattern and plan the system fully prior to development. More attention to detail would be spent with the user assets, as they define the system's functionalities.