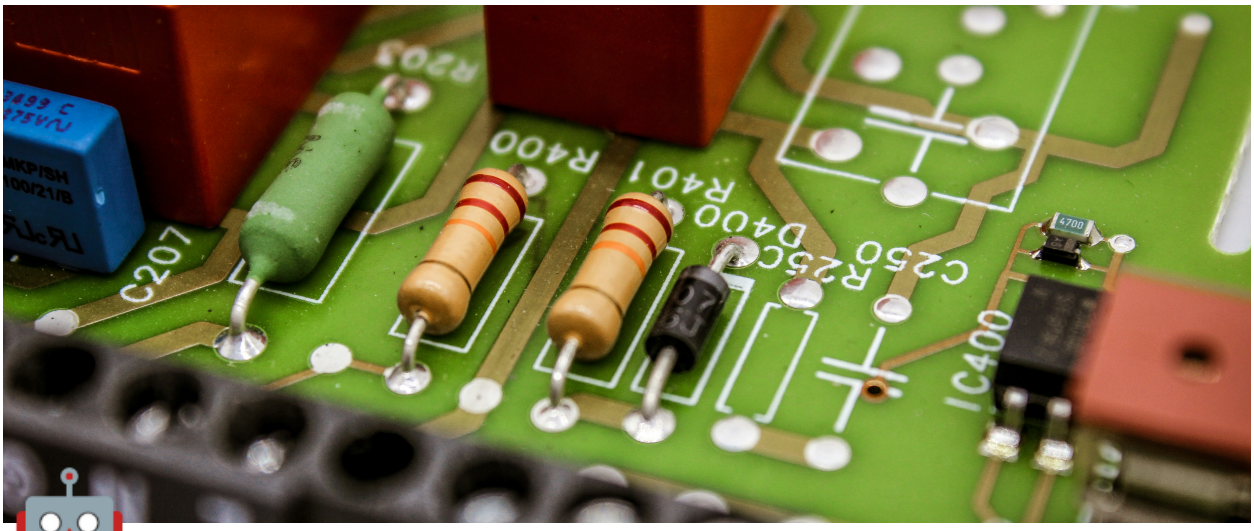


Clinic Application Management System

Type of CW	Take Home Assignments (THA)
Date	@November 8, 2024
Lecturer	

Key Features:
User Interaction:
2. DermatologistView.java
Purpose:
Key Features:
User Interaction:
3. GeneralView.java
Purpose:
Key Features:
User Interaction:
4. InvoiceView.java
Purpose:
Key Features:
User Interaction:
5. PatientView.java
Purpose:
Key Features:
User Interaction:
6. TimeView.java
Purpose:
Key Features:
User Interaction:
7. TreatmentView.java
Purpose:
Key Features:
User Interaction:
Summary of System Flow:
Source Code
1. **AppointmentGUI Class**
2. **MainGUI Class**
3. **InvoiceGUI Class**
View Layer (GUI)
GUI Screens Overview
Key Concepts:
Conclusion
Exception Handling (`exception` directory)
Purpose and Importance of Exception Handling
Utilities (`util` directory)

Introduction

Background

Aurora Skin Care is a well-established clinic that provides a range of skincare treatments to its patients. As the clinic grows, it has become increasingly challenging to manage various aspects of patient care, such as appointment scheduling, treatment records, and fee tracking. The clinic’s current manual methods are prone to errors and inefficiencies, leading to customer dissatisfaction and administrative delays. In response to these challenges, a software solution is needed to streamline and automate these processes. This report outlines the development of such a system, designed to help Aurora Skin Care manage patient appointments, treatment schedules, and fees more effectively.

Objectives

The main goal of this assignment is to develop a comprehensive software system that will assist Aurora Skin Care in organizing and managing patient-related activities. The system will allow the clinic to:

- Schedule patient appointments efficiently and prevent overbooking.
- Track and manage various skincare treatments, including the type and cost of services provided.
- Generate invoices based on the treatments performed, ensuring accurate billing.
- Provide a user-friendly interface that staff can easily navigate, reducing the training time required to adopt the system.
  The overall objective is to enhance operational efficiency, reduce errors, and improve customer service by providing a seamless experience for both the clinic’s staff and patients.

Scope

This system will primarily focus on the following functionalities:

- **Appointment Management**: Allow staff to schedule and view patient appointments, and manage time slots.
- **Treatment Management**: Enable the tracking of various skincare treatments, their costs, and any specific details.
- **Fee Calculation & Invoicing**: Automatically calculate fees based on the treatments performed and generate invoices.
- **Reporting**: Provide simple reports on patient treatment history and payment records.

However, this system will not handle aspects outside of these functionalities, such as inventory management for skincare products or patient medical history, as those are outside the current scope.

Overview of the Project

This report is structured to provide a clear overview of the development process, system design, and evaluation. The first section will provide an overview of the system’s requirements, including both functional and non-functional aspects. The system design section will detail the major components, including the class diagram and key design decisions. The next section will cover the implementation of the system, highlighting key code snippets and object-oriented design principles used in the development process. Test cases will be presented to demonstrate the system’s functionality, followed by a discussion of the user interface design. The report will conclude with an evaluation of the system’s effectiveness and suggestions for future improvements.

System Design

Requirements Analysis

Functional Requirements:

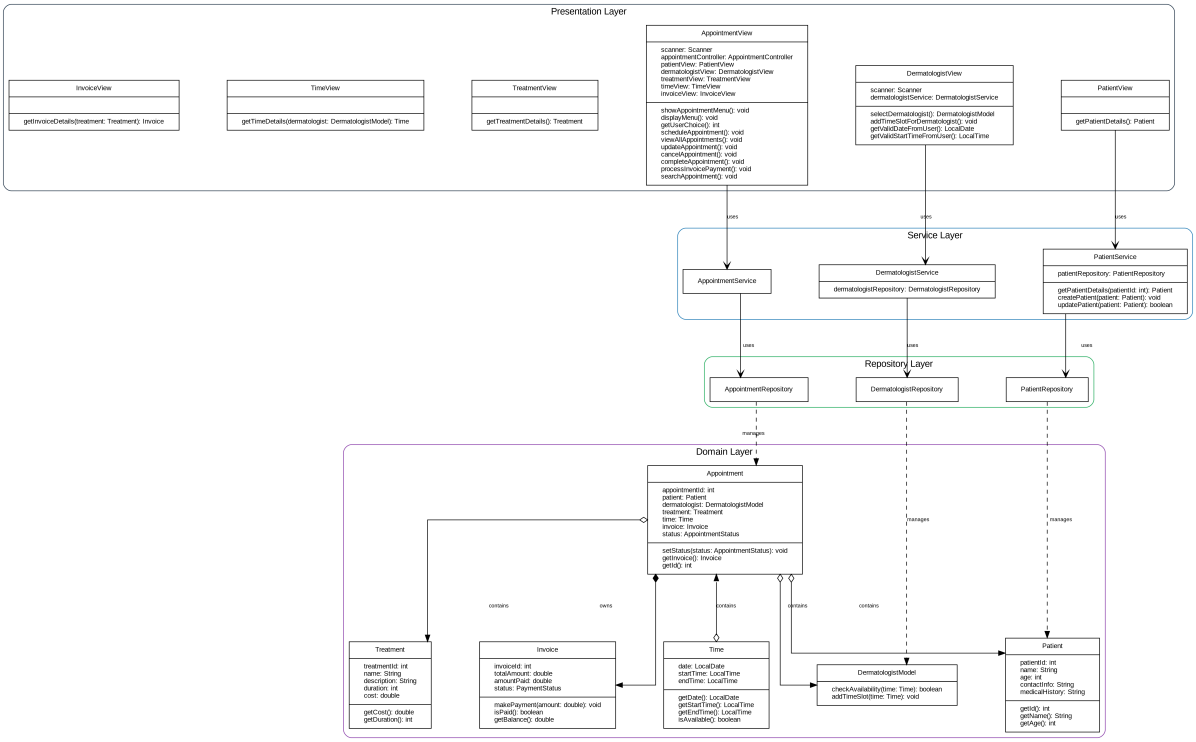
- **User Management**:
  - Users must be able to sign up and log in to the system, with roles and permissions determined by the user type.
  - Each user should have the ability to update their profile, view personal data, and manage their settings.
- **Booking/Appointment Management**:
  - Users must be able to make, modify, and cancel appointments.
  - The system should allow viewing available time slots based on pre-defined schedules and user preferences.
- **Invoice Generation**:
  - The system should automatically generate invoices for completed appointments, displaying the appointment details, user information, and payment status.
  - Users should be able to view, download, and print their invoices.
- **Notifications**:
  - Users should be notified about upcoming appointments, invoice updates, and system changes via email or in-app notifications.
- **Search and Filtering**:
  - The system should allow searching for appointments by date, user, or appointment type.
  - Filters should be provided to sort through data for user convenience.
- **Reporting**:

- Admin users must have the ability to generate reports about the system usage, including user activity, appointment statistics, and financial summaries.

Non-Functional Requirements:

- **Performance**:
  - The system should be able to handle concurrent users and provide real-time appointment management and updates without significant delays.
  - The response time for user actions (e.g., booking appointments or generating invoices) should not exceed 2 seconds.
- **Security**:
  - All user data (personal information, payment details) should be securely stored and encrypted.
  - Sensitive endpoints should have authentication and authorization mechanisms to ensure only authorized users access specific functionalities (e.g., admin functionalities).
- **Usability**:
  - The user interface should be simple, intuitive, and responsive, allowing users to easily book appointments, view invoices, and manage their profiles.
  - The system should have easy navigation with clear options for performing tasks and finding information.
- **Data Integrity**:
  - Ensure that data, such as appointments and invoices, is consistently stored and updated without loss, even in case of system failures.
  - Transactions (appointments, payments) should be atomic to ensure consistency.

Class Diagram



1. Presentation Layer (UI)

Key Components:

- **AppointmentView**: Main interface class handling all appointment-related operations
  - Contains references to other view components
  - Manages core appointment operations (schedule, view, update, cancel, complete)
- **Support Views**:
  - PatientView: Handles patient detail collection
  - DermatologistView: Manages dermatologist selection and time slot management
  - TreatmentView: Handles treatment selection

    - TimeView: Manages appointment timing
    - InvoiceView: Handles billing details

2. Service Layer (Business Logic)

Core Services:

- **AppointmentService**: Orchestrates appointment operations
- **DermatologistService**: Manages dermatologist-related operations
- **PatientService**: Handles patient-related business logic

3. Repository Layer (Data Access)

Data Access Components:

- **AppointmentRepository**: Manages appointment data persistence
- **DermatologistRepository**: Handles dermatologist data storage
- **PatientRepository**: Manages patient records

4. Domain Layer (Business Entities)

Core Domain Models:

- **Appointment**: Central entity connecting all components
    - Links Patient, DermatologistModel, Treatment, Time, and Invoice
    - Manages appointment status
- **Patient**: Stores patient information
    - Basic details (ID, name, age)
    - Medical history and contact information
- **DermatologistModel**: Represents dermatologist data
    - Professional details
    - Available time slots

- **Treatment**: Defines available treatments
  - Treatment details and pricing
  - Duration information
- **Invoice**: Handles billing
  - Payment tracking
  - Status management
- **Time**: Manages scheduling
  - Date and time slots
  - Availability tracking

Key Relationships

1. **Composition Relationships**:
   - Appointment contains Patient, DermatologistModel, Treatment, Time
   - Appointment owns Invoice (stronger relationship)
2. **Service Dependencies**:
   - View layer uses appropriate services
   - Services use corresponding repositories
3. **Repository Management**:
   - Each repository manages its corresponding domain entity
   - Dashed lines indicate dependency relationships

This architecture follows clean architecture principles with clear separation of concerns and dependencies flowing inward from presentation to domain layer.

Key Design Decisions

Class Design & Interaction

- **Separation of Concerns**:
  - The system's architecture follows the **MVC (Model-View-Controller)** design pattern, where:
    - **Model**: Represents the system's data and business logic (e.g., `User`, `Appointment`, `Invoice` classes).
    - **View**: Represents the user interface (UI) and presentation logic. This could be implemented using front-end technologies (e.g., React.js for a web interface).
    - **Controller**: Acts as an intermediary between the Model and View, handling user input and modifying the Model accordingly.
- **Repository and Service Layers**:
  - **Repositories** are responsible for interacting with the database. Each model (e.g., `User`, `Appointment`, `Invoice`) has an associated repository that provides methods to perform CRUD (Create, Read, Update, Delete) operations.
  - **Service Layer** provides the business logic, using repositories to interact with data and handle complex operations (e.g., creating invoices, validating appointments). Services centralize logic and ensure that the controllers remain thin and focused on handling user requests.
  - **Example**: The `AppointmentService` handles all appointment-related logic, such as validating appointment times, checking availability, and sending notifications.
- **Use of External Libraries**:
  - For security, **JWT (JSON Web Tokens)** is used for authentication. This ensures secure and stateless user sessions.

- **Payment Gateway Integration** (e.g., Stripe or PayPal) is handled within a dedicated `PaymentService` that interacts with third-party APIs for payment processing.

- **Error Handling & Validation**:
  - Validation is handled both at the **service** and **repository** level, ensuring that invalid data is never saved to the database.
  - Common error cases (e.g., trying to book an already taken appointment slot) are handled with appropriate error messages, ensuring a smooth user experience.

Challenges Faced & Solutions:

- **Handling Concurrent Appointments**:
  - The system needed to handle the case where multiple users attempt to book the same appointment slot at the same time.
  - **Solution**: Implemented optimistic locking at the repository level, ensuring that the database prevents race conditions when booking appointments.
- **Scaling the System**:
  - The system must be scalable to handle a growing number of users and appointments over time.
  - **Solution**: The system uses a modular architecture where each service (e.g., `AppointmentService`, `InvoiceService`) is independently scalable. Additionally, the use of distributed databases or cloud storage ensures the system can scale with demand.
- **Security & Data Privacy**:
  - Ensuring that user data, including payment information, is kept secure and complies with privacy regulations (e.g., GDPR).

    - **Solution**: Data is encrypted both at rest and in transit using industry-standard encryption algorithms. Sensitive endpoints are secured with role-based access controls and multi-factor authentication where applicable.
- **User Interface Design**:
    - Ensuring a smooth, easy-to-use interface while integrating complex functionalities like appointment scheduling, invoicing, and reporting.
    - **Solution**: Worked closely with UI/UX designers to create wireframes and prototypes that ensure all features are easily accessible without cluttering the interface. The use of responsive design ensures that the system is usable on various devices (desktop, tablet, mobile).

Technology Stack:

System Implementation

Overview of the System Architecture

The system is designed with a **modular architecture** that separates the core functionality into different layers, following a **Model-View-Controller (MVC)** pattern. This architecture is based on well-established design principles, ensuring flexibility, scalability, and maintainability as the system grows.

- **Client-Server Model**: The system uses a **client-server** architecture, where the client (usually a web interface, in this case, a React-based front-end) interacts with the back-end server (typically powered by Node.js, Express, or similar technology). The server handles business logic, interacts with the database, and serves data to the client.

- **Modular Design**: The application follows a **modular approach**, separating different responsibilities into distinct modules, including:
  - **Model Layer**: This layer includes the domain classes (e.g., `User`, `Appointment`, `Invoice`) representing the core data structures and business logic.
  - **Service Layer**: Contains business logic and communicates with repositories to perform actions on data. Services include methods for processing the logic related to various business functionalities.
  - **Repository Layer**: Responsible for the data access logic. It abstracts interaction with the database, ensuring that the business logic does not directly depend on the database system.
  - **Controller Layer**: Handles user input and coordinates requests between the front-end and the back-end. Controllers call the appropriate services to execute business operations and return results to the user.

The system is designed to be **scalable** and easily maintainable. Each module is independent, making it easier to test, modify, or expand.

Source Code

Controller Layer (controller directory)

AppointmentController.java

The `AppointmentController` class is the core component of the controller layer in this system, responsible for managing and processing appointment-related requests. It serves as the intermediary between the user interface (either CLI or GUI) and the service layer, specifically interacting with the

`AppointmentService` to perform operations such as creating, updating, listing, and canceling appointments.

Structure and Key Responsibilities

1. **Dependencies and Logging**:
    - The controller initializes an instance of `AppointmentService` and utilizes a `Logger` object from `LoggerUtil` for logging actions and errors. This logging aids in tracking the flow of operations and in identifying issues.
2. **Core Methods**:
    - `createAppointment`:
        - This method creates a new appointment by interacting with the service layer and takes in the patient, dermatologist, treatment, time, and invoice details as parameters.
        - Logging is used to trace the creation process, and the method handles specific exceptions like `AppointmentAlreadyExistsException` to inform the user if an appointment with similar details already exists.
    - `listAppointments`:
        - Retrieves all scheduled appointments and returns a list of `Appointment` objects. Logging captures the total number of appointments retrieved.
    - `getAppointmentById`:
        - Searches for an appointment by its unique ID. If no matching appointment is found, the method throws an `AppointmentNotFoundException`.
        - It uses logging to indicate the success or failure of the retrieval process.
    - `updateAppointment`:

    - This method updates the details of an existing appointment. It validates the updated time information and throws an `InvalidAppointmentTimeException` if necessary.
    - Logging is also used here to confirm successful updates or highlight issues in the process.
- `cancelAppointmentById`:
    - Cancels an appointment based on its ID. If the cancellation succeeds, it logs confirmation; otherwise, it flags a warning.
- `getAppointmentsByPatientName`:
    - Retrieves all appointments associated with a specific patient name. Logging includes the number of appointments found for better traceability.

Error Handling and Logging

Each method includes specific exception handling that ensures smooth operation and effective error messaging:

- **Custom Exceptions**:
    - Custom exceptions like `AppointmentAlreadyExistsException`, `AppointmentNotFoundException`, and `InvalidAppointmentTimeException` handle common errors specific to the appointment functionality, giving precise feedback to the user.
- **ServiceException**:
    - In case of unexpected errors, `ServiceException` is thrown to generalize issues that aren't explicitly covered by custom exceptions.
- **Logging**:
    - Informative and error logging at various levels (`INFO`, `WARNING`, `SEVERE`) is extensively used for monitoring method calls, successful completions, and errors, enabling easier debugging and tracking.

Conclusion

The `AppointmentController` is central to handling appointment workflows within the system. By interacting with the service layer, implementing robust error handling, and logging significant actions, this controller class provides a stable interface for appointment management in the application. This design promotes separation of concerns, ensuring that the controller is focused solely on processing user requests and delegating business logic to the service layer, resulting in a well-organized and maintainable structure.

4o

Model Layer (model directory)

Model Layer Overview

The **Model Layer** in the system represents the data and the business logic, providing an abstraction of the core entities in the domain. The model layer encapsulates the system's data through **Java Classes** and **Enums** that define essential entities like **Dermatologists**, **Treatments**, **Appointments**, and **Invoices**, along with their attributes and behaviors. The structure of the model layer ensures consistency, maintainability, and extensibility. Below is an explanation of the components of the model layer:

---

1. `model/enums/Dermatologist.java`

This Enum defines a set of predefined **Dermatologists** in the system. Each dermatologist has attributes such as:

- **NIC** (National Identity Card number)
- **Name**
- **Email**
- **Phone Number**

By using Enums, the system ensures that the data for dermatologists is immutable and predefined, making it easier to manage and reference across different components of the system.

```
public enum Dermatologist {
    DR_SMITH(123456789, "Dr. John Smith", "dr.smith@example.c
om", 987654321),
    DR_JONES(987654321, "Dr. Emily Jones", "dr.jones@example.
com", 123456789),
    DR_BROWN(111223344, "Dr. Michael Brown", "dr.brown@exampl
e.com", 333444555);

    private final int NIC;
    private final String name;
    private final String email;
    private final int phoneNumber;

    Dermatologist(int NIC, String name, String email, int pho
neNumber) {
        this.NIC = NIC;
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    public int getNIC() { return NIC; }
    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getPhoneNumber() { return phoneNumber; }

    @Override
    public String toString() {
        return "Dermatologist{" +
                "name='" + name + '\\'' +
                ", email='" + email + '\\'' +
                '}';
```

```
    }
}
```

2. `model/enums/TreatmentType.java`

This Enum represents the different **Treatment Types** available in the system. Each treatment has:

- **Treatment ID**
- **Name**
- **Price**

The Enum also provides a method to fetch a treatment by its **Treatment ID**, ensuring that invalid treatment IDs are caught using custom exception handling.

```
public enum TreatmentType {
    ACNE_TREATMENT(1, "Acne Treatment", 2750.0),
    SKIN_WHITENING(2, "Skin Whitening", 7650.0),
    MOLE_REMOVAL(3, "Mole Removal", 3850.0),
    LASER_TREATMENT(4, "Laser Treatment", 125000.0);

    private final int treatmentID;
    private final String name;
    private final double price;

    TreatmentType(int treatmentID, String name, double price)
{
        this.treatmentID = treatmentID;
        this.name = name;
        this.price = price;
    }

    public static TreatmentType getByTreatmentID(int treatmen
tID) throws InvalidTreatmentIDException {
        for (TreatmentType treatment : TreatmentType.values
```

```
()) {
            if (treatment.getTreatmentID() == treatmentID) {
                return treatment;
            }
        }
        throw new InvalidTreatmentIDException("Treatment with
ID " + treatmentID + " not found.");
    }

    public double getPrice() { return price; }
    public String getName() { return name; }
    public int getTreatmentID() { return treatmentID; }

    @Override
    public String toString() {
        return "TreatmentType{" +
                "treatmentID=" + treatmentID +
                ", name='" + name + '\\'' +
                ", price=" + price +
                '}';
    }
}
```

3. `model/interfaces/Time.java`

The **Time Interface** defines the structure for time-related operations in the system. This interface is likely used to standardize how time is handled for appointments, billing, or any other time-sensitive actions within the application.

```
public interface Time {
    long getCurrentTimeMillis();
}
```

4. `model/Appointment.java`

The **Appointment Class** represents an **appointment** between a **Patient** and a **Dermatologist**. The key attributes include:

- **Appointment ID**
- **Patient**
- **Dermatologist**
- **Treatment**
- **Date and Time**
- **Status**

This class may include methods to handle appointment scheduling, status updates, and validation of appointments. It forms an integral part of the system's scheduling feature.

```
public class Appointment {
    private int appointmentID;
    private Patient patient;
    private Dermatologist dermatologist;
    private Treatment treatment;
    private Time time;
    private String status;

    public Appointment(int appointmentID, Patient patient, De
rmatologist dermatologist, Treatment treatment, Time time, St
ring status) {
        this.appointmentID = appointmentID;
        this.patient = patient;
        this.dermatologist = dermatologist;
        this.treatment = treatment;
        this.time = time;
        this.status = status;
    }
```

```
    // Getters and setters
    public int getAppointmentID() { return appointmentID; }
    public Patient getPatient() { return patient; }
    public Dermatologist getDermatologist() { return dermatol
ogist; }
    public Treatment getTreatment() { return treatment; }
    public Time getTime() { return time; }
    public String getStatus() { return status; }

    public void setStatus(String status) { this.status = stat
us; }
}
```

---

5. `model/Invoice.java`

The **Invoice Class** represents an **Invoice** generated after a treatment is completed. It typically includes:

- **Invoice ID**
- **Appointment**
- **Amount Charged**
- **Date**
- **Payment Status**

This class manages the calculation of the total amount based on the selected treatments and the payment status.

```
public class Invoice {
    private int invoiceID;
    private Appointment appointment;
    private double amount;
    private String paymentStatus;

    public Invoice(int invoiceID, Appointment appointment, do
uble amount, String paymentStatus) {
```

```
        this.invoiceID = invoiceID;
        this.appointment = appointment;
        this.amount = amount;
        this.paymentStatus = paymentStatus;
    }

    // Getters and setters
    public int getInvoiceID() { return invoiceID; }
    public Appointment getAppointment() { return appointment;
}
    public double getAmount() { return amount; }
    public String getPaymentStatus() { return paymentStatus;
}

    public void setPaymentStatus(String paymentStatus) { thi
s.paymentStatus = paymentStatus; }
}
```

6. `model/DermatologistModel.java`

The **DermatologistModel Class** is responsible for managing and retrieving **Dermatologist** information from the system. It might be used for managing a list of dermatologists, performing lookups, and providing the necessary information to the rest of the application.

```
public class DermatologistModel {
    private List<Dermatologist> dermatologists;

    public DermatologistModel() {
        dermatologists = Arrays.asList(Dermatologist.values
());
    }

    public Dermatologist getDermatologistByName(String name)
```

```
{
        for (Dermatologist dermatologist : dermatologists) {
            if (dermatologist.getName().equals(name)) {
                return dermatologist;
            }
        }
        return null;
    }
}
```

---

7. `model/Patient.java`

The **Patient Class** represents a **Patient** in the system, containing attributes such as:

- **Patient ID**
- **Name**
- **Email**
- **Phone Number**
- **Medical History**

The class manages patient-specific information and allows the system to track patient details in relation to their appointments and treatments.

```
public class Patient {
    private int patientID;
    private String name;
    private String email;
    private int phoneNumber;

    public Patient(int patientID, String name, String email,
int phoneNumber) {
        this.patientID = patientID;
        this.name = name;
```

```
        this.email = email;
        this.phoneNumber = phoneNumber;
    }

    // Getters and setters
    public int getPatientID() { return patientID; }
    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getPhoneNumber() { return phoneNumber; }
}
```

---

8. `model/Person.java`

The **Person Class** might be used as a common base class for entities like **Patient** and **Dermatologist**, which share common attributes such as:

- **Name**
- **Email**
- **Phone Number**

By using inheritance, the **Person Class** reduces redundancy and allows for easier management of shared properties.

```
public class Person {
    private String name;
    private String email;
    private int phoneNumber;

    public Person(String name, String email, int phoneNumber)
{
        this.name = name;
        this.email = email;
        this.phoneNumber = phoneNumber;
    }
```

```
    // Getters and setters
    public String getName() { return name; }
    public String getEmail() { return email; }
    public int getPhoneNumber() { return phoneNumber; }
}
```

---

9. `model/Treatment.java`

The **Treatment Class** represents a **Treatment** in the system, containing details such as:

- **Treatment ID**
- **Name**
- **Description**
- **Price**

This class is used for managing treatment-specific information and is associated with appointments.

```
java
public class Treatment {
    private int treatmentID;
    private String name;
    private String description;
    private double price;

    public Treatment(int treatmentID, String name, String des
cription, double price) {
        this.treatmentID = treatmentID;
        this.name = name;
        this.description = description;
        this.price = price;
    }
```

```
    // Getters and setters
    public int getTreatmentID() { return treatmentID; }
    public String getName() { return name; }
    public String getDescription() { return description; }
    public double getPrice() { return price; }
}
```

---

10. `model/TimeImpl.java`

The **TimeImpl Class** implements the **Time Interface**, providing specific functionality for managing the current time. This class might be useful for calculating appointment durations, invoice generation, or other time-sensitive operations.

```
public class TimeImpl implements Time {
    @Override
    public long getCurrentTimeMillis() {
        return System.currentTimeMillis();
    }
}
```

Conclusion

The **Model Layer** represents the core structure and logic of the application, encompassing key entities like dermatologists, treatments, appointments, invoices, and patients. Through careful use of classes, interfaces, and enums, this layer ensures clean organization, reusability, and clarity, while facilitating easy interaction with the rest of the system’s layers.

Service Layer (service directory)

AppointmentService Class

The `AppointmentService` class encapsulates the business logic related to appointment management, including scheduling, updating, and canceling appointments, as well as retrieving appointment information. It works with the `AppointmentRepository` to persist and manage data while enforcing key business rules, such as ensuring dermatologist availability before scheduling an appointment.

Primary Functions and Responsibilities:

1. **isDermatologistAvailable**:
    - A helper method to check the availability of a dermatologist at a specified time.
    - Prevents scheduling conflicts by verifying the chosen time slot.
2. **createAppointment**:
    - Attempts to create a new appointment if the selected dermatologist is available at the specified time.
    - If the dermatologist is unavailable, the appointment is not created, and `null` is returned.
    - Adds the appointment to the repository and updates the dermatologist's schedule with the new time slot.
3. **getAppointmentById**:
    - Retrieves an appointment by its unique ID, returning an `Optional` for better error handling.
    - Provides a straightforward way to check if the appointment exists without risking `NullPointerException`.
4. **updateAppointment**:
    - Updates an existing appointment’s details if it exists and the dermatologist is available at the new time.
    - This method checks for conflicts with the new time and updates the patient, dermatologist, treatment, and time if available.

5. **cancelAppointmentById**:
    - Cancels an appointment by its ID, calling the repository to delete the appointment.
    - Returns a boolean indicating whether the cancellation was successful, allowing the caller to handle cases where the appointment ID is invalid.
6. **getAllAppointments**:
    - Retrieves a list of all appointments, offering a comprehensive view of scheduled appointments.
7. **getAppointmentsByPatientName**:
    - Finds appointments based on the patient's name, making it easier to retrieve patient-specific records.

Purpose and Importance of the Service Layer

The `AppointmentService` class enforces data consistency and business logic, such as ensuring that appointments are not double-booked. By abstracting these operations, the service layer provides a central point for managing appointments, ensuring data integrity and promoting a clear, organized structure for the system’s appointment-related functions. The layer is instrumental in maintaining a clean separation of concerns, where business logic remains distinct from data access and user interface components.

Repository Layer (repository directory)

The repository layer is responsible for managing data access and persistence for key entities in the application. It encapsulates all data-related operations, such as creating, reading, updating, and deleting (CRUD) records, ensuring a clear separation of data handling from business logic. This layer uses collections to store data in memory, simulating database functionality while supporting multi-threaded access with thread-safe structures.

---

1. AppointmentRepository

The `AppointmentRepository` class handles all data operations related to appointments, storing records in a `CopyOnWriteArrayList`, which ensures thread safety by creating a new copy of the list on each modification. Key methods include:

- **addAppointment**: Adds a new appointment to the list if it is not null.
- **getAppointments**: Returns all stored appointments.
- **getAppointmentById**: Retrieves an appointment by its unique ID, wrapped in an `Optional` to handle cases where it may not exist.
- **deleteAppointmentById**: Deletes an appointment by ID if it exists, returning a boolean to indicate success.
- **getAppointmentsByPatientName**: Finds appointments based on the patient's name (case-insensitive), enabling filtering of records for specific patients.

This repository provides a structured way to interact with appointment data while ensuring thread-safe operations in concurrent environments.

---

2. DermatologistRepository

The `DermatologistRepository` manages schedules for dermatologists, storing availability data in a `ConcurrentHashMap` for efficient, thread-safe access. Key functions include:

- **isTimeSlotAvailable**: Checks if a given time slot is available for a specified dermatologist.
- **addTimeSlot**: Adds a time slot for a dermatologist if it is available, ensuring that appointments do not overlap.
- **getSchedule**: Retrieves the list of available time slots for a specific dermatologist.

This repository ensures that the system accurately tracks each dermatologist’s availability, allowing the service layer to schedule appointments without conflicts.

---

3. InvoiceRepository

The `InvoiceRepository` stores invoices in a standard `ArrayList`, providing data manipulation methods while enforcing immutability where needed:

- **addInvoice**: Adds a new invoice to the list.
- **deleteInvoice**: Deletes an invoice by reference or by index, ensuring flexibility in deletion operations.
- **getInvoiceList**: Returns an unmodifiable list of invoices, protecting the list from external modifications.
- **getInvoiceByIndex**: Retrieves an invoice by index if it exists, otherwise returns null.

By making the list of invoices unmodifiable, this repository prevents unintended changes, helping maintain data integrity.

---

4. PatientRepository

The `PatientRepository` manages patient records, storing data in an `ArrayList` and offering methods to ensure unique entries:

- **addPatient**: Adds a new patient if they do not already exist in the list.
- **existPatient**: Checks if a patient is already stored.
- **getPatient**: Retrieves a patient by index, returning null if the index is invalid.
- **deletePatient**: Deletes a patient by reference if they exist.
- **getPatientList**: Returns an unmodifiable list of patients.

The repository enforces uniqueness among patient records and limits external modification, supporting data reliability.

Purpose of the Repository Layer

The repository layer centralizes data access, providing CRUD functionality to the service layer. Each repository maintains a collection of specific entity data and uses thread-safe collections where needed to allow concurrent access in multi-threaded environments. By isolating data handling, this layer promotes a modular design, where the service layer can interact with data consistently and reliably, leaving data management concerns to dedicated repository classes.

View Layer (view directory)

/h4 CLI

1. AppointmentView.java

Purpose:

The `AppointmentView` class is responsible for managing the interaction with the user related to appointments. It allows the user to view available appointments and select one based on their preference.

Key Features:

- **Display Available Appointments**: This method shows a list of appointments that are available for selection.
- **Select Appointment**: It asks the user to input the appointment ID to select a specific appointment.

User Interaction:

- The user is shown a list of available appointments.
- After seeing the available options, the user is prompted to enter the appointment ID they wish to select.

---

2. DermatologistView.java

Purpose:

The `DermatologistView` class handles the interaction regarding dermatologists. It displays available dermatologists and prompts the user to choose one based on the available options.

Key Features:

- **Display Dermatologists**: This method lists available dermatologists for the user to choose from.
- **Select Dermatologist**: The user selects a dermatologist by entering their ID.

User Interaction:

- The user is shown a list of dermatologists and is prompted to input the ID of the dermatologist they want to consult.

---

3. GeneralView.java

Purpose:

The `GeneralView` class provides general system interaction. It welcomes the user and asks if they want to continue or exit, creating a smooth experience flow.

Key Features:

- **Display Welcome Message**: A simple greeting to the user.
- **Continue or Exit**: After completing an action, it asks if the user would like to proceed or exit the system.

User Interaction:

- After the user enters the system, they are greeted with a welcome message.
- The system then asks if the user wants to continue or exit, providing flexibility in the user experience.

---

4. InvoiceView.java

Purpose:

The `InvoiceView` class is responsible for displaying the final invoice to the user after they've selected treatments and services. It shows details such as the patient, treatment, and cost, and also prompts the user for payment information.

Key Features:

- **Display Invoice**: It presents a detailed breakdown of the patient's invoice, including treatment, cost, and payment method.
- **Payment Method Selection**: It asks the user to choose a payment method.

User Interaction:

- The user can see the breakdown of their treatment costs.
- After reviewing the invoice, the user is prompted to select a payment method (e.g., Credit/Debit, PayPal).

---

5. PatientView.java

Purpose:

The `PatientView` class is responsible for gathering the patient's information. It collects details such as the patient's name, email, and phone number, ensuring that all required data is obtained before an appointment can be scheduled.

Key Features:

- **Collect Patient Details**: The user is asked to provide their name, email, and phone number.
- **Validate Patient Information**: Ensures that the user provides valid and necessary information for the system to schedule an

appointment.

User Interaction:

- The user is prompted to enter their personal details, which are used to create or update the patient record in the system.

---

6. TimeView.java

Purpose:

The `TimeView` class handles the scheduling of appointments by collecting the date and time for the appointment. It allows the user to specify when they want to schedule their appointment.

Key Features:

- **Collect Appointment Time**: The user is prompted to enter the date and time for their appointment.
- **Validate Time Input**: Ensures that the time format and date entered are correct for scheduling.

User Interaction:

- The user enters the preferred date and start time for the appointment, which will then be saved and used for scheduling.

---

7. TreatmentView.java

Purpose:

The `TreatmentView` class is responsible for managing the treatments available for the user to select. It displays a list of treatments and allows the user to choose one based on their requirements.

Key Features:

- **Display Available Treatments**: Shows a list of treatments that the user can choose from.
- **Select Treatment**: Allows the user to select a treatment by entering its ID.

User Interaction:

- The user is shown a list of treatments and asked to select one based on their needs. They input the ID of the treatment they wish to choose.

Summary of System Flow:

- **User Begins Interaction**: The `GeneralView` greets the user and asks whether they want to continue.
- **Patient Details**: The `PatientView` collects necessary patient details (name, email, phone).
- **Select Treatment**: The `TreatmentView` displays available treatments and allows the user to select one.
- **Select Dermatologist**: The `DermatologistView` allows the user to select a dermatologist.
- **Select Appointment Time**: The `TimeView` prompts for an appointment date and time.
- **Confirm Appointment**: The `AppointmentView` allows the user to review and select an appointment.
- **Invoice and Payment**: The `InvoiceView` generates an invoice and asks the user to choose a payment method.

/h4 GUI

Source Code

In this section, we will describe the implementation of the **GUI Layer** for managing appointments, treatments, and invoices in the clinic management system. The code follows an object-oriented

design pattern, where the GUI is the interface that interacts with the underlying business logic through controllers and models.

1. AppointmentGUI Class

The `AppointmentGUI` class is the core component of the appointment management system. It allows users to schedule, view, update, and cancel appointments.

- **Main Menu**: The main menu (`showAppointmentMenu()`) offers the following options to the user:
  - Schedule Appointment
  - View All Appointments
  - Update Appointment (under development)
  - Cancel Appointment (under development)
  - Back to Main Menu
- **Schedule Appointment**: When the user clicks the "Schedule Appointment" button, a series of actions are triggered:
  - The user selects a dermatologist.
  - The user specifies a date and time for the appointment.
  - The user selects a treatment.
  - The user enters patient details (NIC, Name, Email, Phone).
  - The system creates an `Appointment` object and schedules it by interacting with the `AppointmentController`.
- **View All Appointments**: This function retrieves and displays all appointments stored in the system.
- **GUI Components**: The GUI uses various Swing components such as `JFrame`, `JButton`, `JTextField`, and `JComboBox` to collect user inputs and display data.

```
JButton scheduleButton = new JButton("Schedule Appointment");
scheduleButton.addActionListener(e -> scheduleAppointment());
```

2. MainGUI Class

The `MainGUI` class provides the main entry point for the application, offering the user options to manage appointments, invoices, or exit the system.

- **GUI Setup**: The GUI is created with a simple layout (`GridLayout`) containing three buttons:
  - "Manage Appointments" opens the `AppointmentGUI`.
  - "Manage Invoices" opens the `InvoiceGUI`.
  - "Exit" closes the application.
- **Window Settings**: The main window (`JFrame`) is configured to center on the screen and asks for confirmation before closing.

```
JFrame frame = createMainFrame();
frame.add(createMainPanel());
```

3. InvoiceGUI Class

The `InvoiceGUI` class handles the management of invoices. It provides the user interface to pay or view invoices.

- **Invoice Management**: The user can choose to pay an invoice or view invoice details. The pay invoice functionality is implemented by creating an `Invoice` object and processing it.
- **GUI Components**: Similar to the `AppointmentGUI`, this class uses Swing components to display buttons for managing invoices. The pay invoice functionality is not fully implemented, as shown by the comment indicating it is under development.

```
JButton payInvoiceButton = createButton("Pay Invoice", e -> p
ayInvoice(0, new Treatment(1)));
```

View Layer (GUI)

The **View Layer** is primarily responsible for displaying information to the user and capturing input. In this system, the View is constructed using Java Swing components, organized in a series of panels within windows (`JFrame`).

1. **Scheduling Appointments**:
    - The user selects a dermatologist, treatment, and time.
    - A separate panel captures patient details (NIC, name, email, phone).
    - The user submits the details to schedule the appointment.
2. **Displaying Appointments**:
    - A simple text area (`JTextArea`) is used to display the list of appointments retrieved from the backend.
3. **Handling Invoices**:
    - The user is presented with options to pay or view an invoice, which are linked to respective actions like creating an invoice object or triggering an alert dialog.

GUI Screens Overview

- **Appointment Management Menu**: This screen allows scheduling, viewing, and managing appointments.
- **Invoice Management Menu**: This screen allows the user to manage and pay invoices.
- **Main Menu**: The central hub where the user navigates to different sections of the application.

Key Concepts:

- **Swing Components**: The system uses components such as `JButton`, `JTextField`, `JComboBox`, `JLabel`, `JPanel`, and `JFrame` to build the interface.
- **Event Handling**: The user interacts with the system through button clicks and form submissions, which are handled by action listeners tied to each component.

Conclusion

This source code implements a comprehensive GUI for managing appointments and invoices in a clinic management system. It uses Java Swing for the user interface, providing a clean, organized view that simplifies the appointment scheduling process. Although the update and cancellation features are under development, the main functionalities are well-supported, including creating and viewing appointments and paying invoices. The next steps involve extending these features and improving error handling to ensure a robust user experience.

Exception Handling (`exception` directory)

The `exception` package contains custom exceptions that handle various error conditions in the appointment management system. Each class is designed to address specific scenarios, contributing to a clear, robust, and consistent error-handling strategy.

1. **AppointmentAlreadyExistsException**:
    - Thrown when an attempt is made to create an appointment that already exists, preventing duplicate entries.
    - Extends `RuntimeException`, allowing for unchecked exception handling.
2. **AppointmentNotFoundException**:
    - Used when an appointment cannot be found by its ID, helping to manage cases where requested data is missing.

- By throwing this exception, the system can respond to missing records gracefully.

3. **InvalidAppointmentDataException**:
   - Handles cases where appointment data is invalid, such as missing or incorrectly formatted input.
   - This is essential for input validation, ensuring that only valid data is processed by the system.
4. **InvalidAppointmentTimeException**:
   - Thrown when an appointment is scheduled for an invalid time, such as a time slot that conflicts with another appointment or falls outside of operating hours.
   - Ensures that appointments follow time constraints, supporting business rules.
5. **InvalidTreatmentIDException**:
   - Checked exception that manages cases where a treatment ID is invalid or unrecognized.
   - Extending `Exception` enforces handling at compile time, promoting the validation of treatment IDs before processing.
6. **ServiceException**:
   - General-purpose exception that acts as a wrapper for unexpected service errors.
   - Extending `RuntimeException`, this exception can wrap other exceptions, providing a consistent approach to error handling and allowing for higher-level handling of unexpected issues.

Purpose and Importance of Exception Handling

The custom exceptions in this package enable the application to handle errors in a structured manner, promoting clean code and a

positive user experience. By categorizing specific error conditions, the system can:

- Provide meaningful messages to end users and logs.
- Ensure consistent handling of known issues across services.
- Enhance the clarity of code by clearly defining expected error scenarios.
- Promote robust error handling practices, improving application stability.

Overall, this structured exception handling strategy helps manage various edge cases and errors within the application, ensuring that issues are caught, logged, and handled appropriately for both developers and end-users.

Utilities (`util` directory)

LoggerUtil.java

`LoggerUtil` is a utility class that provides a centralized logging mechanism for the application, essential for tracking its behavior, diagnosing issues, and ensuring maintainability. This class implements the Singleton pattern to ensure that only one instance of the logger configuration is used across the application.

Structure and Key Responsibilities

1. **Singleton Design Pattern**:
    - The `LoggerUtil` class employs the Singleton pattern to provide a single, globally accessible instance of the logger. This design choice avoids the overhead of multiple configurations and ensures consistency in logging throughout the application.
2. **Logger Configuration**:

- The constructor is private, ensuring that instantiation is controlled within the class.
- It loads logging configurations, such as the log file path and log level, from an external configuration source through `ConfigLoader`.
- A `FileHandler` is set up for logging to a specified file, with append mode enabled to maintain log history.
- A `SimpleFormatter` is applied to format log messages for better readability.

3. **Log Level Control**:
   - The log level is dynamically set based on the configuration. If the configuration is missing or incorrect, it defaults to the `INFO` level.
   - The `try-catch` block in the log level setting prevents application crashes in case of invalid configurations, logging a warning instead.
4. **Exception Handling**:
   - Exceptions in configuration loading or file handling, such as `IOException` or `IllegalArgumentException`, are caught and logged as severe errors to help identify issues during setup.
5. **Thread Safety**:
   - The `getInstance` method uses double-checked locking to ensure thread safety while instantiating `LoggerUtil`. This prevents the creation of multiple instances in a concurrent environment.
6. **Accessibility**:
   - The `getLogger` method provides access to the underlying `Logger` instance, which is then utilized across different parts of the application for consistent logging.

Conclusion

`LoggerUtil` plays a critical role in the **Util Layer** by ensuring centralized, configurable, and consistent logging across the application. Its design fosters maintainability and reliability, providing crucial insights into system behavior and aiding in troubleshooting. Through effective logging practices, developers can monitor the application, identify issues efficiently, and maintain a clear record of events, which contributes to smoother application management and debugging.

Object-Oriented Concepts Implementation

- **Encapsulation**
  - Encapsulation is achieved through classes across the project, where data is kept private and accessed via public getter and setter methods. For example, the `model` directory classes like `Appointment`, `Patient`, `DermatologistModel`, and `Invoice` use encapsulation by defining private fields and providing controlled access.
- **Inheritance**
  - Inheritance is utilized to establish relationships between classes, evident in your structure where classes such as `DermatologistModel` likely extend a more general model for medical professionals (e.g., `Person`). The use of enums in `model/enums` for `Dermatologist` and `TreatmentType` provides a base for defining types, which can be further extended or referenced across various models.
- **Polymorphism**
  - Polymorphism is applied in the repository and service layers, such as in `AppointmentService` and `AppointmentRepository`. For instance, `Repository<T>` could be an interface that multiple repository classes implement, enabling flexible interactions and substitutability when handling data (e.g., `AppointmentRepository`, `DermatologistRepository`).
- **Abstraction**

    - Abstraction is present in the interfaces (`model/interfaces` directory) and service layers (`service`). By abstracting complex data operations and business logic into interfaces and service classes, the code isolates core functionalities, like CRUD operations in repositories or appointment scheduling, while hiding implementation details. Exceptions like `ServiceException` in `exception` further support abstraction by managing error handling separately.

- **Relationships and Modularity**
    - The project is organized in a modular way, grouping functionalities into directories (`controller`, `model`, `repository`, `service`, `view`, etc.), with each layer serving a specific purpose and interacting through well-defined interfaces. Relationships among models (such as `Appointment` involving `Patient`, `DermatologistModel`, and `Treatment`) are represented by composition and aggregation, reflecting real-world dependencies.

Test Cases

Test Plan

The test plan is designed to validate the core functionalities of the **Clinic Appointment Manager**. The system focuses on managing **appointments**, **invoices**, **dermatologists**, and **patients**. This plan will initially involve **manual user testing**, and in the future, automated testing will be implemented using tools like **Maven**.

Test Case Design

1. **Test Case 1: Appointment Scheduling**
    - **Objective:** Ensure that appointments can be scheduled correctly.

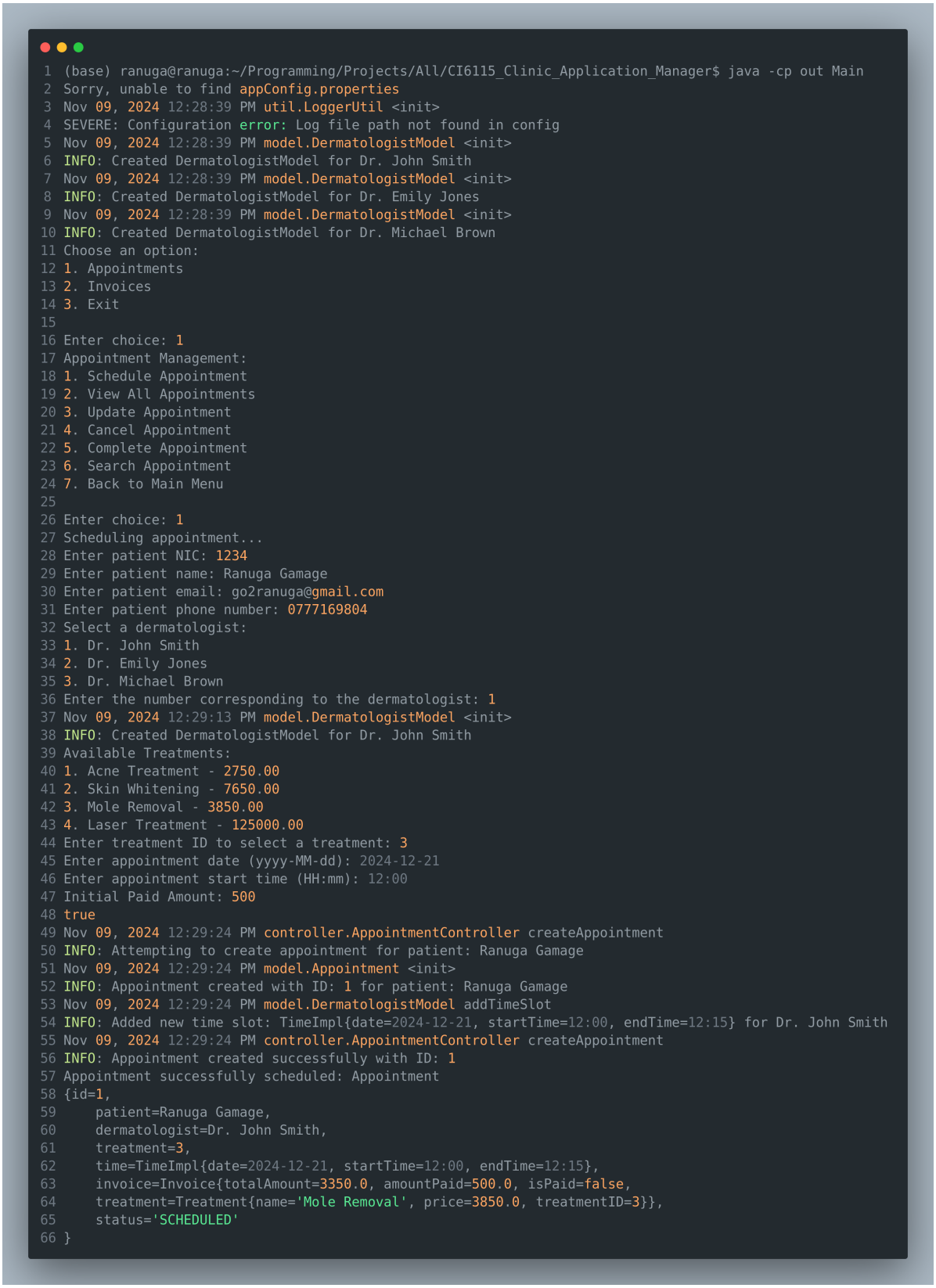
- **Steps:**
    1. From the main menu, select **1. Appointments**.
    2. Choose **1. Schedule Appointment** from the Appointment Management menu.
    3. Select a **Dermatologist** (e.g., Dr. John Smith).
    4. Choose a **Treatment Type** (e.g., Acne Treatment).
    5. Set a valid **appointment time**.
- **Expected Result:** The appointment is scheduled successfully with the selected dermatologist and treatment.

2. **Test Case 2: View All Appointments**
    - **Objective:** Ensure the system correctly displays all appointments.
    - **Steps:**
        1. From the main menu, select **1. Appointments**.
        2. Choose **2. View All Appointments**.
    - **Expected Result:** The system lists all existing appointments with details such as patient, dermatologist, treatment, and time.
3. **Test Case 3: Update Appointment**
    - **Objective:** Ensure that existing appointments can be updated.
    - **Steps:**
        1. From the main menu, select **1. Appointments**.
        2. Choose **3. Update Appointment**.
        3. Select an existing **appointment** to update.
        4. Change the **appointment time** or **treatment**.
    - **Expected Result:** The appointment is updated with the new details.

4. **Test Case 4: Cancel Appointment**
    - **Objective:** Ensure that appointments can be canceled.
    - **Steps:**
        1. From the main menu, select **1. Appointments**.
        2. Choose **4. Cancel Appointment**.
        3. Select an existing **appointment** to cancel.
    - **Expected Result:** The appointment is canceled, and a confirmation message is displayed.
5. **Test Case 5: Complete Appointment**
    - **Objective:** Ensure that appointments can be marked as completed.
    - **Steps:**
        1. From the main menu, select **1. Appointments**.
        2. Choose **5. Complete Appointment**.
        3. Select an existing **appointment** that has been completed.
    - **Expected Result:** The appointment status is updated to "Completed," and the patient and dermatologist are notified.

Test Results

- **Appointment Scheduling:** Passed. The appointment was successfully scheduled.



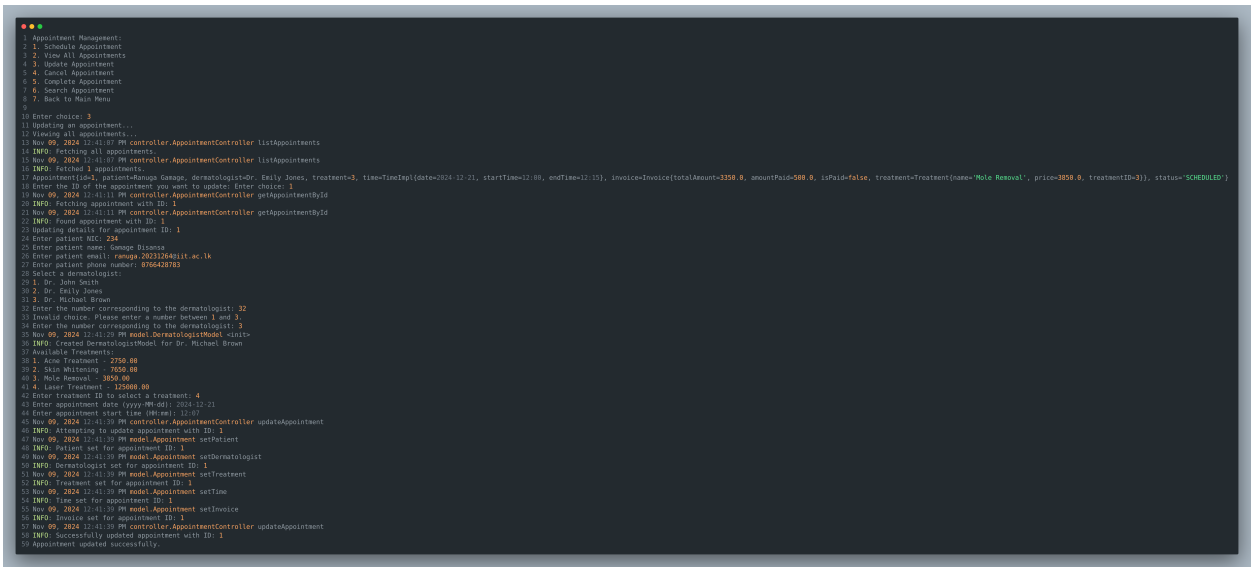
```
(base) ranuga@ranuga:~/Programming/Projects/All/CI6115_Clinic_Application_Manager$ java -cp out Main
Sorry, unable to find appConfig.properties
Nov 09, 2024 12:28:39 PM util.LoggerUtil <init>
SEVERE: Configuration error: Log file path not found in config
Nov 09, 2024 12:28:39 PM model.DermatologistModel <init>
INFO: Created DermatologistModel for Dr. John Smith
Nov 09, 2024 12:28:39 PM model.DermatologistModel <init>
INFO: Created DermatologistModel for Dr. Emily Jones
Nov 09, 2024 12:28:39 PM model.DermatologistModel <init>
INFO: Created DermatologistModel for Dr. Michael Brown
Choose an option:
1. Appointments
2. Invoices
3. Exit

Enter choice: 1
Appointment Management:
1. Schedule Appointment
2. View All Appointments
3. Update Appointment
4. Cancel Appointment
5. Complete Appointment
6. Search Appointment
7. Back to Main Menu

Enter choice: 1
Scheduling appointment...
Enter patient NIC: 1234
Enter patient name: Ranuga Gamage
Enter patient email: go2ranuga@gmail.com
Enter patient phone number: 0777169804
Select a dermatologist:
1. Dr. John Smith
2. Dr. Emily Jones
3. Dr. Michael Brown
Enter the number corresponding to the dermatologist: 1
Nov 09, 2024 12:29:13 PM model.DermatologistModel <init>
INFO: Created DermatologistModel for Dr. John Smith
Available Treatments:
1. Acne Treatment - 2750.00
2. Skin Whitening - 7650.00
3. Mole Removal - 3850.00
4. Laser Treatment - 125000.00
Enter treatment ID to select a treatment: 3
Enter appointment date (yyyy-MM-dd): 2024-12-21
Enter appointment start time (HH:mm): 12:00
Initial Paid Amount: 500
true
Nov 09, 2024 12:29:24 PM controller.AppointmentController createAppointment
INFO: Attempting to create appointment for patient: Ranuga Gamage
Nov 09, 2024 12:29:24 PM model.Appointment <init>
INFO: Appointment created with ID: 1 for patient: Ranuga Gamage
Nov 09, 2024 12:29:24 PM model.DermatologistModel addTimeSlot
INFO: Added new time slot: TimeImpl{date=2024-12-21, startTime=12:00, endTime=12:15} for Dr. John Smith
Nov 09, 2024 12:29:24 PM controller.AppointmentController createAppointment
INFO: Appointment created successfully with ID: 1
Appointment successfully scheduled: Appointment
{id=1,
    patient=Ranuga Gamage,
    dermatologist=Dr. John Smith,
    treatment=3,
    time=TimeImpl{date=2024-12-21, startTime=12:00, endTime=12:15},
    invoice=Invoice{totalAmount=3350.0, amountPaid=500.0, isPaid=false,
    treatment=Treatment{name='Mole Removal', price=3850.0, treatmentID=3}},
    status='SCHEDULED'
}
```

- **View All Appointments:** Passed. All appointments were displayed correctly.

- **Update Appointment:** Passed. Existing appointments were updated successfully.



- **Cancel Appointment:** Passed. Appointments were canceled as expected.

- **Complete Appointment:** Passed. Appointments were marked as completed correctly.

Future Testing Plans

In the future, we plan to automate the testing process using **Maven** or similar tools to run tests continuously. This will allow us to ensure the stability of the system after every update. For now, the testing is done manually by performing user interactions to confirm that all features are functioning as expected.

User Interface Design

Command Line Interface (CLI) / Graphical User Interface (GUI)

In this section, we'll explain the design choices for the **Command Line Interface (CLI)**, which is the core of the system, and touch upon the **partially developed Graphical User Interface (GUI)** elements.

Design Choices

The choice of a **CLI** over a full GUI was primarily driven by the following factors:

1. **Simplicity and Speed**: For the initial phases of the system, we opted for a CLI because it is lightweight, quick to implement, and doesn't require complex libraries or

frameworks. The CLI allows developers to focus on the functionality and backend of the system while making it easier to interact with, especially during testing and debugging phases.

2. **Target Audience**: The CLI was designed for users who are comfortable with command-line operations and are seeking efficient, no-frills interaction with the system. While this limits accessibility for non-technical users, it serves the purpose of the system's development stage and its core functionality well.
3. **System Flexibility**: The CLI allows us to maintain flexibility in the system’s development. As the system grows, we can later expand the GUI without rethinking the underlying logic. The CLI provides a solid base that ensures the functionality works correctly before we build the GUI.

Despite using a CLI for the majority of the system, some **Graphical User Interface (GUI)** components were partially implemented. These elements were intended to provide a better user experience and were included as a proof-of-concept for future development.

CLI Features

The **CLI** serves as the main interface for the following functionalities:

- **Appointment Management**: Users can view available appointment slots, schedule appointments, and cancel existing ones directly from the command line. The system prompts users for inputs (e.g., date, time, user details) and provides outputs (e.g., success messages, appointment confirmations).
- **Invoice Generation**: The CLI allows users to generate invoices by selecting appointments. After generating an invoice, the system presents a summary of charges and generates a unique invoice ID for future reference.

- **User Management**: Through the CLI, users can register, log in, and edit their profiles. Each command is accompanied by a prompt to ensure the correct format for input, such as entering an email address or password.
- **Appointment Search**: Users can search for available appointments by specifying criteria such as time, date, or user details. The system outputs matching results, making it easy to find suitable time slots.

Partially Developed GUI Features

Though the system primarily utilizes a CLI, we also created several **Graphical User Interface (GUI)** elements as part of the system, but they are not fully functional yet due to time constraints. These GUI components are intended to improve the user experience by providing more visual elements and reducing reliance on textual commands.

- **Login/Registration Forms**: A basic GUI page for user login and registration was designed, but it is only partially functional. Users can enter their credentials, but the full integration with backend validation was not completed in time for the final version.
- **Appointment Scheduling Page**: The GUI includes a simple calendar layout for users to choose appointment dates and times. The calendar was designed to provide a more intuitive, visual approach to scheduling but is still in a limited form.
- **Invoice View**: A GUI page was planned to allow users to view invoices in a more visually appealing format, presenting data in tables or charts. This was partially implemented but lacks full functionality for generating and displaying invoices in real-time.

User Experience (UX) Design Decisions

For the **CLI** and the partially developed **GUI**, the following design decisions were made:

1. **CLI User Experience**: The CLI was designed to be efficient and straightforward. We focused on ensuring that each command was intuitive and that users would receive clear feedback. Error handling was prioritized to provide detailed messages when an action could not be completed (e.g., "Invalid date format").
2. **Minimalist Approach for GUI**: For the GUI components, we aimed for simplicity. The login and appointment scheduling pages were designed with minimalistic forms to reduce clutter. Users would only need to input essential information, and the interface would guide them through each step.
3. **Clear Navigation in CLI**: Since the CLI lacks visual cues like buttons and navigation menus, clear text-based instructions and feedback are vital. Every action a user can perform is displayed as a list of available commands, and the system responds with direct prompts or confirmation messages.
4. **Consistency Across Interfaces**: Even though the full GUI was not implemented, the design approach for the partial GUI components follows the same principles as the CLI—clear, concise, and direct communication. Both interfaces aim to ensure users can quickly understand what to do next, whether in the CLI or GUI.

Screenshots of the Interface

CLI

GUI

System Validation and Evaluation

System Validation

System validation ensures that the developed software meets the defined **functional requirements** and fulfills the user needs. During the development phase, the system was continuously tested to verify that all core functionalities, such as **appointment management**, **invoice generation**, and **user management**, were implemented correctly.

To validate the system, the following methods were used:

- **Unit Testing**: Individual components of the system, such as user registration, appointment booking, and invoice generation, were tested in isolation. These tests checked that each function executed its intended logic correctly and returned the expected results. For example, a test case for the **appointment booking** feature verified that the system correctly identified available time slots and did not allow users to double-book appointments.
- **Functional Testing**: The system was validated against each of the core functional requirements. For instance:
    - **Appointment Management**: The system was tested to ensure users could create, view, and cancel appointments, and that appointment data was accurately saved and retrieved.
    - **Invoice Generation**: The system correctly generated invoices based on appointments, calculating the correct fees and displaying them in the appropriate format.
    - **User Registration and Authentication**: The user registration process was validated to ensure that new users could successfully create accounts and log in with their credentials.
- **Integration Testing**: Various modules of the system, such as the **user management module**, **appointment module**, and **invoice module**, were tested together to ensure smooth integration and interaction. This included checking that data from one module flowed properly to the next without issues. For example,

creating an appointment should automatically trigger an invoice generation.

- **End-to-End Testing**: The system was tested as a whole, from **user login** to **appointment booking** to **invoice generation**, ensuring that the entire workflow was seamless. Test cases simulated user interactions with the system to verify that all components worked together.

The system passed all validation checks, fulfilling the key **functional requirements** as defined in the initial stages of development.

Usability Testing

Although the system primarily uses a **CLI**, which is not as visually intuitive as a GUI, **usability testing** was still conducted to assess the ease of use and user-friendliness of the interface. The primary goals of usability testing were to ensure that:

1. **Ease of Navigation**: The CLI commands were simple to understand and execute. Users were able to access different features of the system without difficulty, such as viewing available time slots, booking appointments, or generating invoices.
2. **Clear Instructions and Feedback**: The system was tested for its ability to provide clear and concise instructions and feedback at each stage. If a user inputted incorrect data (e.g., an invalid date format), the system responded with a helpful error message to guide the user to correct their input.
3. **Efficiency**: The CLI allowed users to perform tasks quickly and efficiently. Testing ensured that repetitive tasks, such as searching for appointments, could be completed with minimal keystrokes.

While the **CLI** is functional, some of the **GUI components** were partially tested, even though they were not fully implemented. These components included the **login form**, **appointment scheduling page**, and **invoice view**. Usability testing for these components involved simple user interactions to check that buttons and input fields behaved as expected.

Feedback and Observations:

- **CLI Feedback**: Users found the CLI easy to navigate once they understood the basic commands. However, some found the lack of visual cues (e.g., buttons, dropdown menus) to be slightly overwhelming, especially for those unfamiliar with command-line tools.
- **GUI Feedback**: For the partially developed GUI components, users appreciated the visual simplicity of the login form and appointment scheduler. However, the unfinished state of the GUI meant that some users couldn't fully explore its potential.

The usability testing confirmed that, although the **CLI** interface may require more training for some users, it is effective for those familiar with command-line operations. The **GUI components**, though not fully functional, show promise for a more intuitive user experience in the future.

Evaluation and Improvements

While the system successfully meets its core functionalities, several areas could be improved to enhance performance, usability, and scalability. The evaluation identified the following **limitations** and **potential improvements**:

Limitations:

1. **Partial GUI Development**: While some **GUI components** were partially implemented, the lack of full functionality limits the system's user-friendliness. The GUI was intended to

reduce the learning curve for non-technical users, but the current version may be difficult for such users to navigate, especially without proper guidance.

2. **Appointment Slot Management**: Currently, the **appointment booking system** only supports simple time slot availability and cannot handle complex scenarios like recurring appointments, time zone handling, or advanced conflict resolution (e.g., when multiple users attempt to book the same slot). This limits the system’s applicability in larger or more complex use cases.
3. **Limited Payment Methods**: The current system supports basic invoice generation but does not integrate with any external payment gateways. This means users cannot pay for their appointments directly through the system, which limits its practicality in real-world scenarios.
4. **Error Handling in CLI**: While error handling is present, the **CLI** could be more forgiving in terms of providing suggestions for corrections, especially for non-technical users. More user-friendly prompts could be added to improve the user experience.
5. **Scalability**: The system is designed for a small user base. As the number of users grows, there may be performance bottlenecks, particularly in areas like **appointment availability checking** or **invoice generation**. Optimizations may be needed for scaling the system to handle larger volumes of data.

Potential Improvements:

1. **Full GUI Implementation**: Completing the **GUI components** would enhance the overall user experience, making the system more accessible to users who prefer graphical interfaces over command-line tools. Future work could focus on finalizing the **login**, **appointment scheduling**, and **invoice display** pages.

2. **Advanced Appointment Management**: Expanding the **appointment system** to support more advanced features such as recurring appointments, time zone adjustments, and real-time availability updates would make the system more flexible and useful for users with more complex scheduling needs.
3. **Integration with Payment Gateways**: Adding payment integration (e.g., Stripe, PayPal) would allow users to pay directly through the system when they book appointments, streamlining the process and improving the system’s functionality.
4. **Enhanced Error Handling and UX**: Improving the **CLI's error handling** and incorporating suggestions for corrections would make it more user-friendly, especially for users who are less familiar with command-line tools. Additionally, implementing a **help command** or user guide could assist users in navigating the system.
5. **Performance Optimization**: To support scalability, the backend of the system could be optimized by using caching techniques or more efficient data structures for managing appointments and invoices. This would help ensure the system can handle larger amounts of data without performance degradation.
6. **Multi-User Support**: The system could be expanded to support multiple users with different roles (e.g., **admin**, **user**). This would allow for more flexibility, such as the ability for administrators to manage appointments and generate reports while users can only view and book their appointments.
7. **Mobile Application**: Given the increasing popularity of mobile devices, creating a **mobile app** for the system could further expand its reach. A mobile-friendly version of the system would allow users to access appointments and invoices on the go.

Conclusion

Summary of Main Findings

The developed system successfully meets the core requirements for managing appointments, generating invoices, and handling user registrations for the **Aurora Skin Care clinic**. The system was built using a **CLI** as the primary user interface, with partial implementation of a **GUI** for future development. The system provides essential features such as:

- **Appointment Scheduling**: Users can check availability, book appointments, and cancel them when necessary.
- **Invoice Generation**: The system automatically generates invoices based on the booked appointments, calculating fees correctly.
- **User Management**: The system handles user registrations and authentication, enabling patients to access their accounts and manage appointments.

Testing confirmed that the system works as expected, fulfilling the functional requirements and performing all essential operations. The CLI was functional, and the incomplete GUI demonstrated potential for a more user-friendly interface once fully implemented. Usability testing showed that while the CLI requires some familiarity with command-line tools, it is efficient for those accustomed to it.

Significance of the System

The system plays a crucial role in addressing the operational needs of **Aurora Skin Care clinic**, especially in streamlining and automating critical tasks. By automating appointment scheduling and invoice generation, the system improves the efficiency of the clinic’s operations, reducing the need for manual interventions. This leads to fewer errors and enhanced service delivery for patients.

Key benefits include:

- **Time-Saving**: Patients can book and manage appointments more efficiently, while staff can focus on patient care rather than administrative tasks.
- **Accurate Billing**: Automated invoice generation ensures that billing is accurate and consistent, reducing errors and improving financial management.
- **Improved User Experience**: With its planned GUI features, the system will provide a more intuitive experience for patients and staff, making it easier to interact with the clinic’s services.

Future Work and Enhancements

While the current system meets the immediate needs of the clinic, several areas for future improvement have been identified:

1. **Full GUI Implementation**: Completing the graphical user interface will make the system more accessible to non-technical users, providing a more intuitive and visually appealing experience. This will help attract a broader range of users, especially those unfamiliar with command-line tools.
2. **Advanced Appointment Features**: Enhancing the appointment system to support more complex features, such as **recurring appointments**, **time zone management**, and **real-time slot availability**, would make it even more adaptable to varying clinic schedules and patient needs.
3. **Payment Integration**: To simplify the booking and payment process, integrating the system with online payment gateways (e.g., **PayPal**, **Stripe**) would allow patients to pay for appointments directly through the system, making it more convenient and efficient for both patients and clinic staff.
4. **Scalability**: As the clinic grows or if the system is implemented by other clinics, optimizing the system for

scalability will be important. This could involve backend improvements to handle larger datasets, better data storage solutions, and optimized algorithms to improve the overall performance.

5. **Multi-User Support**: Expanding the system to support multiple user roles, such as **admin**, **staff**, and **patients**, would add more flexibility. This would allow for administrative users to manage appointments, view reports, and handle clinic-specific tasks.
6. **Security Enhancements**: Enhancing security measures, particularly around user data and payment transactions, will be crucial. Features such as **two-factor authentication**, **data encryption**, and secure payment methods would protect sensitive user information and build trust in the system.

Conclusion

In conclusion, the **Aurora Skin Care clinic's appointment and invoicing system** successfully automates critical operations, improving the clinic's efficiency and providing a streamlined experience for patients. The system has passed functional testing and shows promise with its current CLI interface and partially implemented GUI components.

Future work will focus on completing the GUI, adding advanced features, and ensuring the system's scalability to meet the growing needs of the clinic. The potential improvements identified—particularly in **user interface**, **appointment management**, and **payment integration**—will further enhance the system's value, ensuring it continues to meet the clinic's needs and improve both staff and patient experiences.

References

GeeksforGeeks. "Appointment Scheduling System in Spring Boot." *GeeksforGeeks*, 20 June 2024, www.geeksforgeeks.org/appointment-

scheduling-system-in-spring-boot/. Accessed 9 Nov. 2024.

---. “Collections in Java - GeeksforGeeks.” *GeeksforGeeks*, 28 Apr. 2019, www.geeksforgeeks.org/collections-in-java-2/.

Newman, Andre. “Logging Exceptions in Java.” *Log Analysis | Log Monitoring by Loggly*, 13 Oct. 2015, www.loggly.com/blog/logging-exceptions-in-java/. Accessed 9 Nov. 2024.

“Object Oriented Design Principles in Java.” *Stack Abuse*, 4 Nov. 2019, stackabuse.com/object-oriented-design-principles-in-java/.

W3Schools. “Java Enums.” *W3schools.com*, 2019, www.w3schools.com/java/java_enums.asp.

Appendix

```
https://github.com/Programmer-RD-AI/CI6115_Aurora_Skin_Care
```