

**WACHEMO UNIVERSITY**

**SCHOOL OF COMPUTING AND INFORMATICS**

**DEPARTMENT OF SOFTWARE ENGINEERING**

**PROJECT DOCUMENTATION**:

**PROJECT TITLE: CLINIC MANAGEMENT SYSTEM (CMS)**

**GROUP MEMBER’S**

**Name ID Group**

1. **Amanuel birhane 1402769 7**
2. **Ashenafi abebe 1402864 7**
3. **Makida nebyu 1404267 7**
4. **Kalkidan addis 1404086 7**

**Submission date: May 03, 2024**

TABLE OF CONTENTS

**Contents** page

Contents

[Chapter One 1](#_Toc167079908)

[1.1 requirements analysis (use case) 1](#_Toc167079909)

[1.2 Use case diagram components 2](#_Toc167079910)

[1.3 Example of use case model 5](#_Toc167079911)

[1.4 Use case Description/template 5](#_Toc167079912)

[1.5 Tools and steps to draw Use Case 16](#_Toc167079913)

[Chapter Two 18](#_Toc167079914)

[2.1 High-Level Sequence Diagram 18](#_Toc167079915)

[2.2 Component of High-level Sequence Diagram 18](#_Toc167079916)

[2.3 Example of High-Level Sequence 20](#_Toc167079917)

[2.4 Tools and Steps to Draw High-Level Sequence Diagram 23](#_Toc167079918)

[Chapter Three 26](#_Toc167079919)

[3.1 Low-level (Detail) Design (class design) 26](#_Toc167079920)

[3.2 Components of Class Diagram 26](#_Toc167079921)

[3.3 Example of Class Diagram 28](#_Toc167079922)

[3.4 Tools and steps to draw Low-level Class Diagram 29](#_Toc167079923)

[Chapter Four 31](#_Toc167079924)

[4.1 Implementation (export class diagram into code and update code and diagram) 31](#_Toc167079925)

[4.2 Steps to generate code from class diagram 33](#_Toc167079926)

[Chapter Five 34](#_Toc167079927)

[5.1 Change Management (version control using Git) 34](#_Toc167079928)

[5.2 Steps and tools that we use in our project to implement git 35](#_Toc167079929)

[Chapter Six 38](#_Toc167079930)

[6.1 Unit Test 38](#_Toc167079931)

[6.2 Steps and Tools Used in Unit Test 41](#_Toc167079932)

[Chapter seven 42](#_Toc167079933)

[7.1 Build (prepare build script for compilation, unit test, jar file creation) 42](#_Toc167079934)

[7.2 Steps and tools that we use in our project to implement build 45](#_Toc167079935)

# Chapter One

## 1.1 requirements analysis (use case)

To identify and analyze the requirements of a clinic management system, we need to consider various aspects of the system and its functionalities. Here are some key requirements to consider:

1. Patient Management:

- Patient registration.

- Appointment scheduling and management.

- Patient search and retrieval of patient information.

2. Doctors Management:

- Doctors registration and profile management.

- Availability management for appointments and consultations.

- Medical record access and update permissions.

3. Appointment Management:

- Ability to schedule appointments based on healthcare provider availability.

- Real-time visibility of available time slots.

- Appointment rescheduling and cancellation.

4. Medical Records Management:

- Electronic Medical Records (EMR) for each patient.

- Recording and tracking of patient medical history.

- Secure access and appropriate permissions for viewing and updating medical records.

8. System Security and Privacy:

- User authentication and access control mechanisms.

- Audit logs for tracking system access and changes.

**Use Cases:**

Based on the identified requirements, the clinic management system should support the following use cases:

Register new Patients, Register new doctors, See medical reports, See admin's profile, Accept appointments, Send appointments to doctor, Schedule appointment, Edit appointment, See medical reports, See appointments, See medical reports, Generate medical reports, Edit medical reports, See doctor's profile

## 1.2 Use case diagram components

A use case diagram for a clinic management system typically consists of the following components:

**1. Actors:** Actors represent the roles or entities that interact with the system. In a clinic management system, the common actors may include:

- *Patient*: Represents individuals who receive healthcare services from the clinic.

- *Doctor*: they are one the clinical staff who provide medical treatment for patients.

- *Admin*: Handles administrative tasks such as scheduling appointments, managing doctors and patients, and handling inquiries.

**2. Use Cases:** Use cases represent specific tasks or functionalities provided by the system. They describe the interactions between actors and the system. Some common use cases in a clinic management system include:

- *Register new* *Patients*: Involves adding (registering) patients within the system.

- Register new *doctors*: Involves adding (registering) doctors to the system.

- *see medical reports*: this use case gives the privilege to access (see) medical reports of a patient that are generated by the doctors.

- *see admin's profile*: this use case allows to access the admin’s profile.

- *accept Appointment*: Allows to schedule appointments by accepting from patients.

- *send appointments to doctor*: Enables to send the appointments to the doctor by receiving from the patient.

- *Schedule appointments*: this use case allows booking an appointments.

- *edit appointments*: this use case allow editing the existed appointment like its time, date… and so on.

- *see appointments*: this give the privilege of accessing (seeing) the available appointments of the patients.

- *generate medical reports*: enables to generate medical reports and make it visible to the authorized persons.

- *edit medical reports*: this use case includes editing or adding necessary information to the medical report.

- *see doctor's profile*: this task is related to having the privilege of accessing doctor’s profile.

**3. Relationships:**

- *Associations*:

- An association represents a relationship between two elements, typically an actor and a use case.

- Associations are depicted as lines connecting the elements, without any arrowheads.

- *Example*: admin has association with manage patients, manage doctors…

- *Include*:

- The include relationship represents a dependency between two use cases.

- Include relationships are depicted with a directed arrowhead pointing from the including use case to the included use case.

- *Example*: all the use case of actors includes the use case login, login includes sign up, and so on.

- *Extend*:

- The extend relationship indicates an optional or alternative behavior that can be added to a base use case.

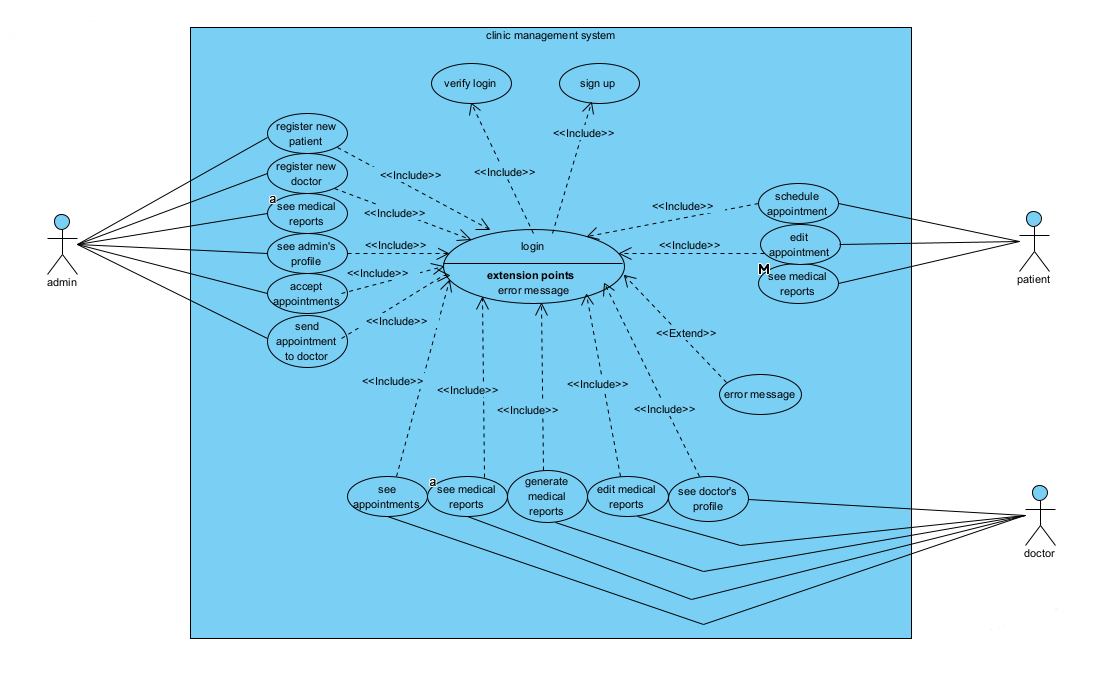
- Extend relationships are depicted with a directed arrowhead pointing from the extending use case to the extended use case.

- *Example*: login use case extends error message use case.

**4. System Boundary:** The system boundary defines the scope of the clinic management system. It encapsulates all the actors and use cases directly related to the system.

Typically, actors (representing users, external systems, or entities) are positioned outside the system boundary, while the use cases (representing the system's functionalities or behaviors) are placed inside the boundary.

## 1.3 Example of use case model

Here is an example use case that we have drawn for the clinic management system:

## 1.4 Use case Description/template

1. Use Case: Register New Patient

Description:

This use case allows the admin to register a new patient into the clinic management system.

Actors:

- Admin

Preconditions:

- The receptionist has access to the clinic management system or (logged in).

Main Flow:

1. The admin selects the option to register a new patient.

2. The system presents a form for entering the patient's details, such as name, contact information, and medical history.

3. The admin fills in the patient's information.

4. The system validates and saves the patient's details.

5. The system generates a unique patient ID and assigns it to the new patient.

6. The system confirms the successful registration of the patient.

Post conditions:

- The new patient is registered in the system with a unique patient ID.

2. Use Case: Register New Doctor

Description:

This use case allows the admin to register a new doctor in the clinic management system.

Actors:

- Admin

Preconditions:

- The admin has access to the clinic management system.

Main Flow:

1. The admin selects the option to register a new doctor.

2. The system presents a form for entering the doctor's details, such as name, contact information, and specialization.

3. The admin fills in the doctor's information.

4. The system validates and saves the doctor's details.

5. The system generates a unique doctor ID and assigns it to the new doctor.

6. The system confirms the successful registration of the doctor.

Post conditions:

- The new doctor is registered in the system with a unique doctor ID.

3. Use Case: See Medical Reports

Description:

This use case allows authorized actors in this case the admin, doctors and patients to view medical reports of patients.

Actors:

- Admin

- Patient

- Doctor

Preconditions:

- The actor has access to the clinic management system.

Main Flow:

1. The actor selects the option to view medical reports.

2. The system presents a list of patients.

3. The actor selects a patient from the list.

4. The system retrieves and displays the medical reports associated with the selected patient.

Post conditions:

- The actor can view the medical reports of the selected patient.

Exceptions:

- If the patient is not registered in the system, the admin may need to create the patient's record before viewing medical reports.

4. Use Case: See Admin's Profile

Description:

This use case allows the administrator to view their own profile information.

Actors:

- Administrator

Preconditions:

- The administrator has successfully logged into the clinic management system.

Main Flow:

1. The administrator selects the option to view their profile.

2. The system retrieves and displays the profile information of the administrator, including their name, contact details, and role.

Post conditions:

- The administrator can view their own profile information.

5. Use Case: Accept Appointments

Description:

This use case allows the admin to accept appointment requests made by patients.

Actors:

- Admin

Preconditions:

- The admin has access to the clinic management system.

- A patient has made an appointment request.

Main Flow:

1. The admin selects the option to view appointment requests.

2. The system presents a list of pending appointment requests.

3. The admin selects a request from the list.

4. The system displays the details of the appointment request, including the patient's information and preferred appointment time.

5. The admin accepts the appointment request.

6. The system confirms the acceptance of the appointment and updates the appointment status.

Post conditions:

- The appointment request is accepted and the appointment status is updated in the system.

Alternative Flows:

1. The admin selects the option to view appointment requests.

2. The system presents a list of pending appointment requests.

3. The admin selects a request from the list.

4. The system displays the details of the appointment request, including the patient's information and preferred appointment time.

5. The admin rejects the appointment request.

6. The system confirms the rejection of the appointment and updates the appointment status.

6. Use Case: Send Appointment to Doctor

Description:

This use case allows the admin to send an accepted appointment to the assigned doctor.

Actors:

- Admin

Preconditions:

- The admin has accepted an appointment request.

- The appointment request has been assigned to a doctor.

Main Flow:

1. The admin selects the option to send appointments to doctors.

2. The system presents a list of accepted appointments.

3. The receptionist selects an appointment from the list.

4. The system displays the details of the appointment, including the patient's information and scheduled time.

5. The receptionist assigns the appointment to the corresponding doctor.

6. The system notifies the assigned doctor about the new appointment.

Post conditions:

- The appointment is assigned to the doctor and they receive a notification.

7. Use Case: Schedule Appointment

Description:

This use case allows the patient to schedule an appointment.

Actors:

- Patient

Preconditions:

- The patient has access to the clinic management system.

- The patient's details are already registered in the system.

Main Flow:

1. The patient selects the option to schedule an appointment.

2. The system presents a form for entering the patient's details and preferred appointment time.

3. The patient selects him/herself from the system or enters their information manually.

4. The patient enters the preferred appointment date and time.

5. The system validates the availability of the chosen time slot.

6. The system confirms the successful scheduling of the appointment.

Post conditions:

- The appointment is scheduled and added to the system.

Alternative Flows:

- If the requested date and time are not available, the system informs the patient and prompts them to select an alternative time or date.

- If there are any conflicts or issues with the patient's information or eligibility, the system may display appropriate error messages or alerts to the patient, guiding them to resolve the issues.

8. Use Case: Edit Appointment

Description:

This use case allows the patient to modify the details of an existing and its own appointments only.

Actors:

- Patient

Preconditions:

- The patient has access to the clinic management system.

- And that particular patient’s appointment exists in the system.

Main Flow:

1. The patient selects the option to view appointments.

2. The system presents a list of its own appointments.

3. The patient selects the appointment to be edited.

4. The system displays the details of the selected appointment.

5. The patient makes the necessary changes to the appointment details, such as the date, time, or other things.

6. The system validates the modified appointment details.

7. The system updates the appointment with the new information.

8. The system confirms the successful modification of the appointment.

Post conditions:

- The appointment is updated with the modified details in the system.

2. Use Case: See Appointments

Description:

This use case allows authorized actors like admins and doctors to view a list of appointments.

Actors:

- Admin

- Doctor

Preconditions:

- The actor has access to the clinic management system.

- There is an existing appointment in the system.

Main Flow:

1. The actor selects the option to view appointments.

2. The system retrieves and displays a list of appointments.

3. The actor can filter or sort the list based on parameters like date, patient name, or status.

4. The system presents the filtered/sorted list of appointments to the actor.

Post conditions:

- The actor can view the list of appointments based on their preferences.

3. Use Case: Generate Medical Reports

Description:

This use case allow doctors to generate medical reports for patients.

Actors:

- Doctor

Preconditions:

- The doctor has access to the clinic management system.

- An appointment has been conducted and medical data is available.

Main Flow:

1. The doctor selects the option to generate a medical report.

2. The system presents a list of completed appointments.

3. The doctor selects the appointment for which the report needs to be generated.

4. The system retrieves the relevant medical data and presents it in a report format.

5. The doctor reviews and verifies the medical report.

6. The system generates the final medical report.

Post conditions:

- The medical report is generated and available for further actions or viewing.

4. Use Case: Edit Medical Reports

Description:

This use case allow doctors to edit medical reports for patients.

Actors:

- Doctor

Preconditions:

- The doctor has access to the clinic management system.

- A medical report exists for a patient.

Main Flow:

1. The doctor selects the option to edit a medical report.

2. The system presents a list of medical reports.

3. The doctor selects the medical report to be edited.

4. The system retrieves the medical report and displays it for editing.

5. The doctor makes the necessary changes to the medical report.

6. The system validates the modified medical report.

7. The system updates the medical report with the new information.

8. The system confirms the successful modification of the medical report.

Post conditions:

- The medical report is updated with the modified details in the system.

5. Use Case: See Doctor's Profile

Description:

This use case allows admins and doctors to view the profile information of doctors.

Actors:

- Admin

- Doctors

Preconditions:

- The actor has access to the clinic management system.

- A doctor is registered in the system.

Main Flow:

1. The actor selects the option to view the doctor's profile.

2. The system presents a list of registered doctors.

3. The actor selects the desired doctor from the list.

4. The system retrieves and displays the profile information of the selected doctor, including their name, contact details, and specialization.

Post conditions:

- The actor can view the profile information of the selected doctor.

## 1.5 Tools and steps to draw Use Case

Here are the tools and steps we used to create a use case diagram:

1. Selecting Diagramming Tool:

There are various tools we can use to draw a use case diagram, but we used Visual Paradigm to draw our use case diagram.

- Visual Paradigm: is a comprehensive UML modeling tool with advanced features.

2. Identifying Actors:

Identified actors which are involved in the clinic management system, are patients, doctors, and administrators.

3. Identifying Use Cases:

Involves Identifying the main tasks or functionalities provided by the system. For a clinic management system, common use cases may include registering patients, scheduling appointments, generating medical reports, and so on.

4. Creating Actors and Use Cases:

By using the selected diagramming tool we create actor symbols (stick figures) and use case symbols (ellipses or rectangles) on the canvas.

5. Connecting Actors and Use Cases:

By Using lines or arrows to establish associations between actors and use cases. These associations represent the interactions between actors and the system. For example, draw a line connecting the "Patient" actor to the "Schedule Appointment" use case to represent the patient's interaction with the scheduling functionality.

6. Adding Relationships:

We use include and extend connectors to represent relationships between use cases. For example, the login use case includes sign in and login use case again extends error message.

7. Organizing the Diagram:

And then we arrange the actors and use cases on the diagram canvas in a logical and visually appealing manner. We use alignment and spacing tools provided by the diagramming tool to ensure clarity and readability.

9. Reviewing and Refining:

We have reviewed the use case diagram for accuracy, completeness, and consistency. Making any necessary revisions or adjustments to improve the diagram's clarity and effectiveness.

# Chapter Two

## 2.1 High-Level Sequence Diagram

A high-level sequence diagram for a clinic management system provides an overview of the interactions and message exchanges between different actors and system components.

## 2.2 Component of High-level Sequence Diagram

In a high-level sequence diagram for a clinic management system, the components typically include the following:

1. Actors:

- Patient: Represents the individual seeking healthcare services from the clinic.

- Admin: Handles administrative tasks such as registering patients/doctors, managing patient records, and handling inquiries.

- Doctor: represent physicians who provide medical care to patients.

2. Lifelines:

- Lifelines represent the individual instances of the actors or system components involved in the sequence of interactions.

- Each actor or system component is represented by a vertical line (lifeline) with the name of the component written on top.

Example: clinic management system, schedule appointment UI, register patients UI, and so on.

3. Messages:

- Messages represent the communication or interaction between different actors or system components.

- They are represented by arrows pointing from the sender to the receiver along the lifelines.

- Messages may include the name of the message, parameters or data being passed, and return values.

Example: fill details (), confirm registration (), update appointments (), etc.

4. Activation Boxes:

- Activation boxes represent the duration of time during which a component is actively performing a task or processing a message.

- They are drawn as rectangles or boxes on the lifeline, usually aligned with the message being processed.

5. Return Messages:

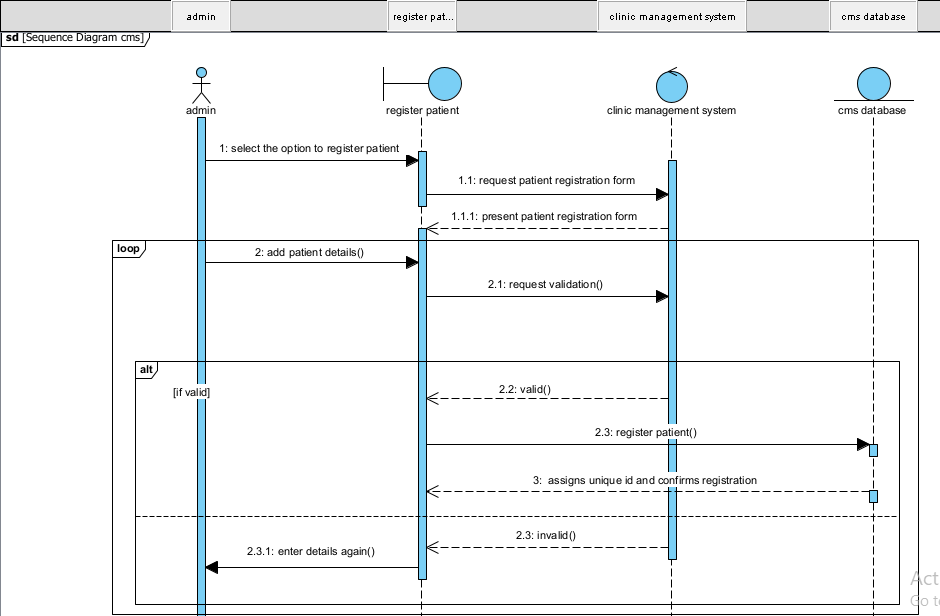
- Return messages indicate the response or result of a previously sent message.

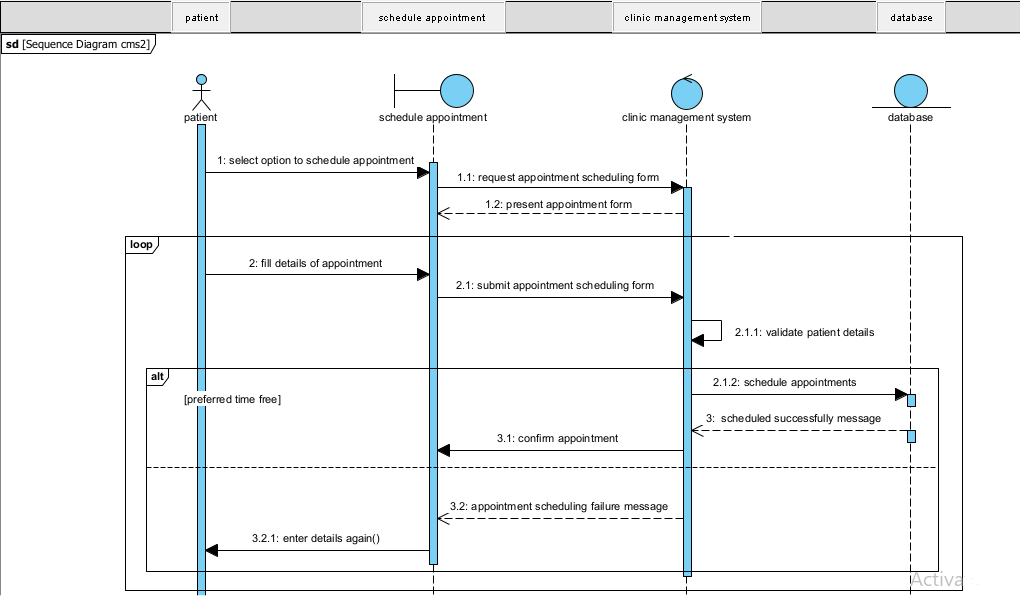
- They are represented by dashed arrows pointing back from the receiver to the sender.

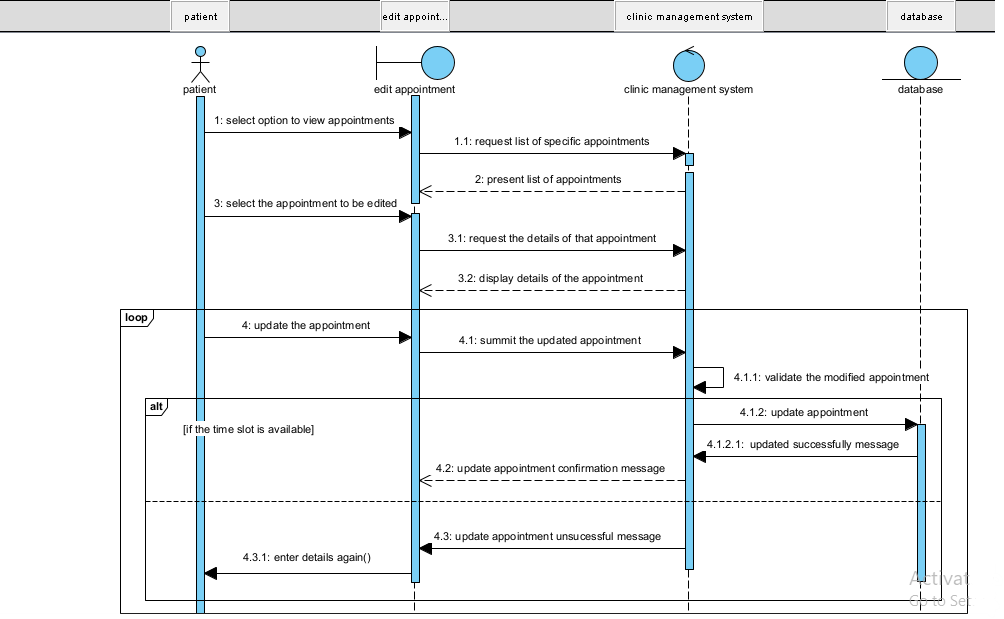
6. Optional Notations:

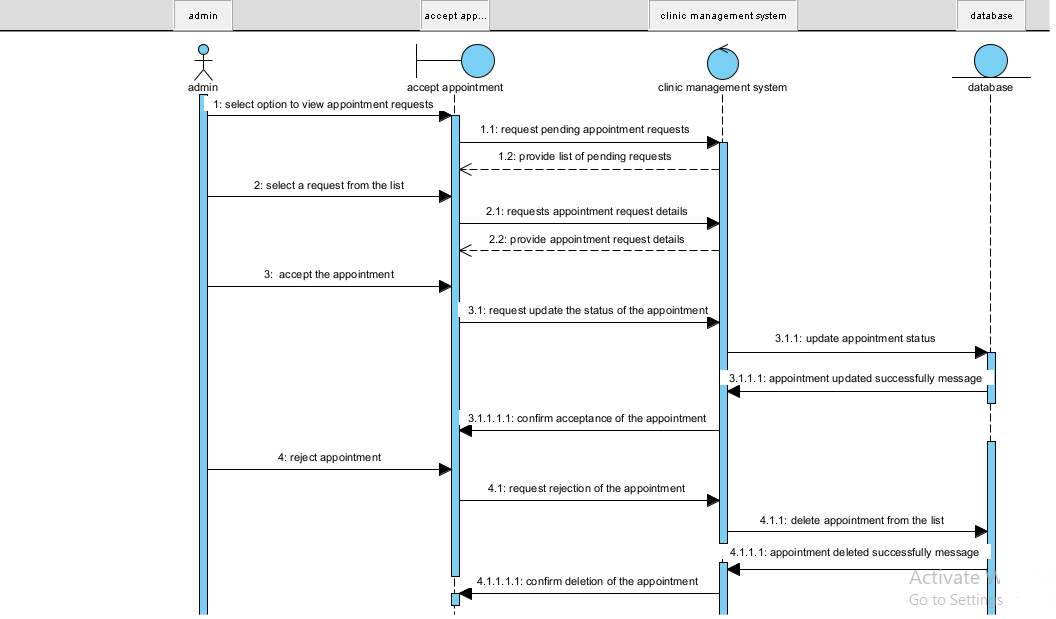
- A high-level sequence diagram may include additional notations, such as conditions or loops, to represent decision points or repetitions in the sequence of interactions.

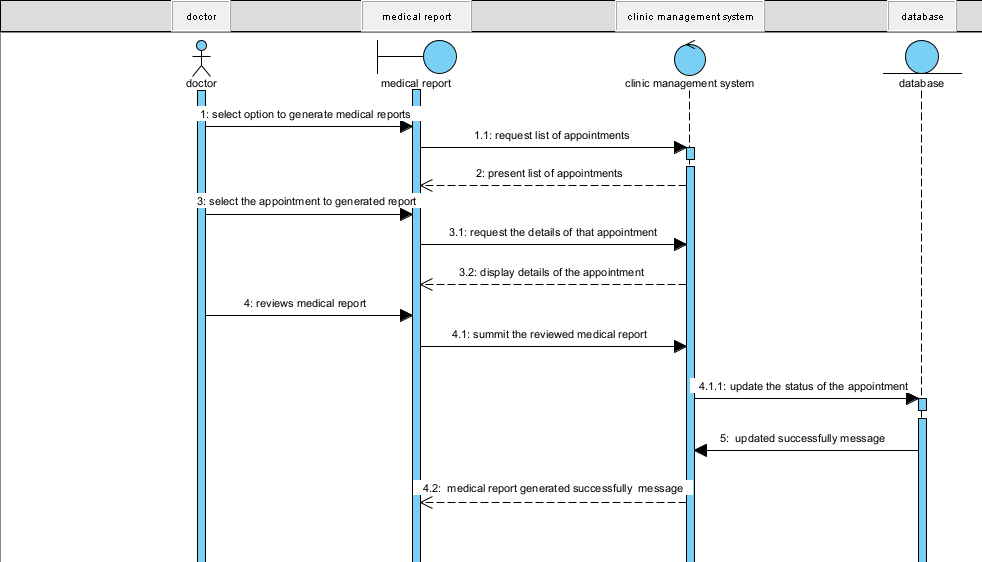
## 2.3 Example of High-Level Sequence

****

****

****

****

****

## 2.4 Tools and Steps to Draw High-Level Sequence Diagram

To draw a high-level sequence diagram for a clinic management system, there are various diagramming tools that support UML (Unified Modeling Language) modeling.

1. Selecting a Diagramming Tool: we choose visual paradigm to model our sequence diagram.

- Visual Paradigm: is a comprehensive UML modeling tool with advanced features.

2. Identifying the Actors:

- Identify the actors involved in the clinic management system, such as patients, doctors, and administrators.

3. Identifying the Key Interactions:

- Determining the main tasks or actions performed by the actors and system components in the clinic management system.

- Identifying the key interactions or scenarios that we want to depict in the high-level sequence diagram.

4. Create Lifelines:

- Using the selected diagramming tool we create lifelines for each actor and system component involved in the sequence of interactions.

- Lifelines are represented as vertical lines, and each line represents an instance of an actor or system component.

5. Define Messages:

- Determining the messages exchanged between the actors and system components.

- Specify the content of each message, including the name of the message, any parameters or data being passed, and return values if applicable.

6. Connecting Lifelines and Messages:

- By using arrows we connect the lifelines and represent the flow of messages between the actors and system components.

- Arrows show the direction of message exchange, starting from the sender and pointing to the receiver.

7. Adding Activation Boxes:

- We add activation boxes on the lifelines to indicate the duration of time during which a component is actively processing a message.

- Activation boxes are represented as rectangles or boxes aligned with the lifeline and are placed over the relevant section of the lifeline.

8. Include Return Messages:

- If necessary, we can include return messages to represent the response or result of a previously sent message.

- Return messages are represented as arrows pointing back from the receiver to the sender.

9. Organize the Diagram:

- Arranging the lifelines, messages, and activation boxes on the diagram canvas in a logical and visually appealing manner.

- We Use alignment and spacing tools provided by the diagramming tool to ensure clarity and readability.

11. Review and Refine:

- We have reviewed the high-level sequence diagram for accuracy, completeness, and consistency.

- Making any necessary revisions or adjustments to improve the diagram's clarity and effectiveness.

12. Save and Share:

- Saving the completed high-level sequence diagram in a suitable format (e.g., we save in PNG format).

- Share the diagram with stakeholders or team members involved in the clinic management system development process.

# Chapter Three

## 3.1 Low-level (Detail) Design (class design)

A low-level (detailed) design for a clinic management system involves designing the classes and relationships between them.

# 3.2 Components of Class Diagram

In a class diagram for a clinic management system, the components typically include classes, associations, attributes, and methods. Here's an explanation of each component:

1. Class:

- Represents a blueprint for creating objects with similar properties and behaviors.

- It is represented as a rectangle with three compartments: the top compartment contains the class name, the middle compartment contains the class attributes, and the bottom compartment contains the class methods.

In this low level class diagram we have classes: patient, doctor, admin, appointment, and medical reports

2. Association:

- Represents a relationship between classes, indicating that objects of one class are connected or interact with objects of another class.

- Associations are typically depicted as lines connecting the participating classes, with optional arrows indicating the direction of the relationship.

- Multiplicity indicators (such as 1, \*, or 0...1) can be included near the association line to show the number of instances involved in the relationship.

We have a relations like:

* 1 admin can manage many patients and doctors
* 1 patient can book many appointments
* 1 doctor can manage many medical reports

3. Attributes:

- Represent the characteristics or properties of a class.

- Attributes describe the data associated with an object and are typically listed in the middle compartment of the class rectangle.

- Each attribute has a name and a data type that define its structure.

4. Methods:

- Represent the behaviors or operations that can be performed by objects of a class.

- Methods define the actions that objects can take or the operations they can perform.

- Methods are listed in the bottom compartment of the class rectangle and include their names, parameters, return types, and any exceptions they may throw.

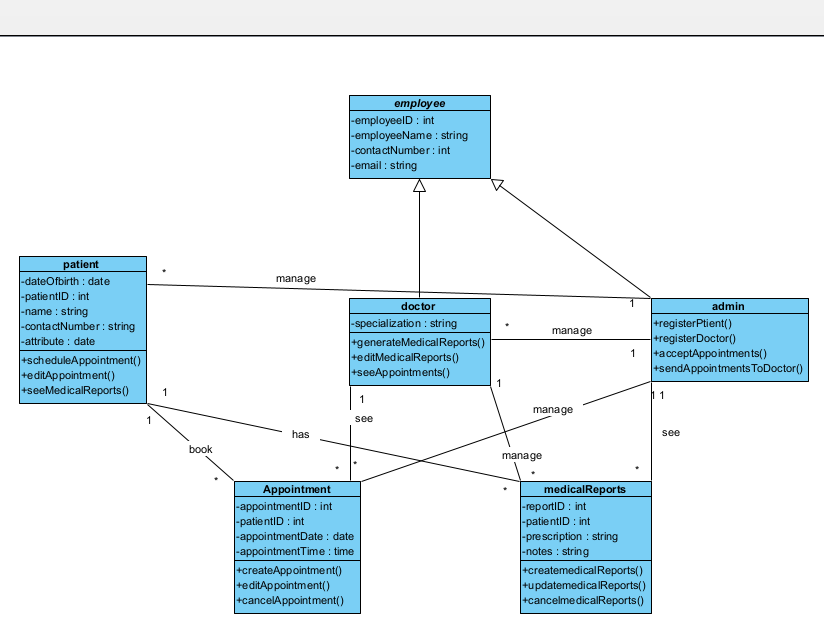
5. Inheritance:

- Represents an "is-a" relationship between classes, indicating that one class inherits properties and behaviors from another class.

- Inheritance is depicted using an arrow with an open triangle pointing to the superclass (the class being inherited from) from the subclass (the class inheriting the properties and behaviors).

In this class diagram the classes doctor and admin inherit from the class employee.

## 3.3 Example of Class Diagram

****

## 3.4 Tools and steps to draw Low-level Class Diagram

To draw a low-level class diagram for a clinic management system, there are various diagramming tools that support UML (Unified Modeling Language) modeling.

1. Selecting a Diagramming Tool: we choose visual paradigm to model our sequence diagram.

- Visual Paradigm: is a comprehensive UML modeling tool with advanced features.

2. Identify the Classes:

- Identify the classes that represent the main entities and concepts in the clinic management system.

- Some examples of classes in a clinic management system include Patient, admin, Appointment, Medical reports, appointment, etc.

3. Create Class Boxes:

- Use the selected diagramming tool to create a box for each identified class.

- Each class box should have three compartments: the top compartment contains the class name, the middle compartment contains the class attributes, and the bottom compartment contains the class methods.

4. Define Class Attributes:

- Identify the attributes (properties) that belong to each class.

- Add the attributes to the appropriate class boxes, listing their names and data types.

- Consider the visibility (public, private, protected) and any constraints or validation rules associated with each attribute.

5. Define Class Methods:

- Identify the behaviors or operations that each class can perform.

- Add the methods to the appropriate class boxes, listing their names, parameters, return types, and any exceptions they may throw.

- Consider the visibility and access modifiers (public, private, protected) for each method.

6. Establish Class Relationships:

- Determine the relationships between the classes, such as associations, aggregations, compositions, and inheritances.

- Use appropriate symbols and connectors provided by the diagramming tool to represent these relationships.

- Connect the related classes with lines, and add multiplicity indicators (such as 1, \*, or 0...1) to show the cardinality of the relationships.

7. Refine and Organize the Diagram:

- Review the class diagram for consistency, completeness, and clarity.

- Organize the class boxes and relationships in a logical and visually appealing manner.

- Use alignment and spacing tools provided by the diagramming tool to ensure readability.

10. Save and Share:

- Save the completed low-level class diagram in a suitable format (e.g., PDF, PNG, or JPEG).

- Share the diagram with stakeholders or team members involved in the clinic management system development process.

# Chapter Four

## 4.1 Implementation (export class diagram into code and update code and diagram)

To translate the class diagram of the clinic management system into executable Java code and maintain consistency between the code and the diagram in Visual Paradigm, we can follow these steps:

1. Identify Classes: Identify the classes in the class diagram. Each class represents a concept or entity in the clinic management system.

2. Define Class Attributes: For each class, define the attributes (properties or data) that are associated with the class. These attributes will be represented as member variables in the Java code.

3. Define Class Methods: Determine the methods (functions or behavior) that the classes will have. These methods will be implemented as member functions in the Java code.

4. Implement Associations and Relationships: Identify the associations and relationships between the classes. Implement these associations using appropriate Java constructs such as member variables and methods. For example, if there is an association between two classes, we can create member variables in the respective classes to represent the association.

5. Implement Inheritance and Interfaces: If the class diagram includes inheritance or interfaces, implement them in the Java code accordingly. Use Java's inheritance mechanism to extend classes or implement interfaces as specified in the class diagram.

6. Generate Java Code: Use Visual Paradigm's code generation feature to generate the Java code from the class diagram. Visual Paradigm provides options to generate code in various programming languages, including Java. Configure the code generation settings to match your requirements and preferences.

7. Update Code and Diagram Consistently: After generating the initial code, we can update the code and the class diagram in Visual Paradigm in a synchronized manner. Any changes made in the code should be reflected in the class diagram, and vice versa. Visual Paradigm provides reverse engineering capabilities to update the class diagram based on changes made in the code.

8. Maintain Consistency: Continuously review and update both the code and the class diagram as needed during the development process. Ensure that any modifications or additions made in either the code or the diagram are reflected in the other to maintain consistency.

public abstract class employee {

    private int employeeID;

    private string employeeName;

    private int contactNumber;

    private string email;

}

public class doctor extends employee {

    private string specialization;

    public void generateMedicalReports() {

        // TODO - implement doctor.generateMedicalReports

        throw new UnsupportedOperationException();

    }

    public void editMedicalReports() {

        // TODO - implement doctor.editMedicalReports

        throw new UnsupportedOperationException();

    }

    public void seeAppointments() {

        // TODO - implement doctor.seeAppointments

        throw new UnsupportedOperationException();

    }

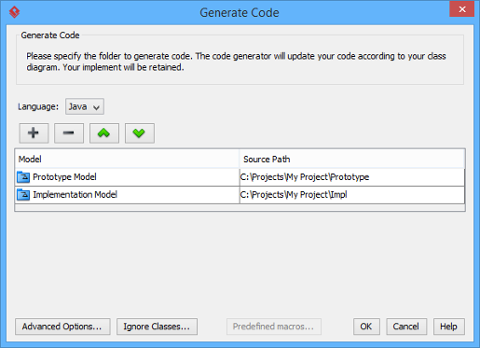
}

## 4.2 Steps to generate code from class diagram

We can generate Java code from all classes in current project. To generate code from project:

1. Select **Tools > Code > Generate Java Code...** from the toolbar.

2. In the **Generate Code** dialog box, specify the mapping between model and source path. Model is a UML element that acts as a container of other elements. Classes and packages under a model will be generated to the mapped source path. You can add multiple model-to-source-path mapping by pressing the **+** button. If you are not using model to structure your

Project, or if you want to generate all classes in project to the same folder, regardless of their parent model, keep model to be **<root>**.

|  |
| --- |
|  |
| The mappings between models and source paths are defined |

1. Optionally, configure the advanced code generation options by clicking **Advanced Options...**. Read the section *Advanced Options* in this chapter for details about the options.
2. Click **OK**to proceed with generation.

# Chapter Five

## 5.1 Change Management (version control using Git)

Version control, especially when implemented with Git, plays a crucial role in the development of a clinic management system. Here are the key reasons why version control is important:

1. Tracking and managing changes: The clinic management system is a complex software project that undergoes continuous development and updates. Version control allows you to track and manage changes made to the system's source code, configuration files, documentation, and other project assets. Git, in particular, provides a detailed history of all changes, enabling you to view previous versions, understand what modifications were made, and revert to earlier states if necessary.

2. Collaboration and team coordination: Developing a clinic management system often involves multiple developers, designers, and other team members working together. Git provides a collaborative environment where team members can work concurrently on different features or modules without interfering with each other's work. The ability to branch, merge, and resolve conflicts in Git ensures smooth collaboration and helps maintain a coherent codebase.

3. Branching and feature development: Git's branching capabilities are especially valuable for the development of a clinic management system. With Git, you can create branches to work on new features, bug fixes, or experiments without affecting the main codebase. This allows developers to isolate their work, test changes independently, and merge them back into the main branch (e.g., `master` or `main`) when they are ready. Branching facilitates parallel development and enables teams to work on multiple features simultaneously.

4. Code review and quality control: Git's pull request feature, commonly used in conjunction with platforms like GitHub or Bitbucket, provides a mechanism for code review and quality control. When a developer completes a feature or bug fix in a branch, they can submit a pull request to have their code reviewed by other team members. This allows for feedback, suggestions, and identification of potential issues or improvements before the code is merged into the main branch. Code review enhances code quality, ensures adherence to coding standards, and helps catch bugs or vulnerabilities early in the development process.

5. Reproducibility and bug tracking: Version control with Git enables you to reproduce specific versions of your clinic management system. Each commit represents a stable state of the project, making it easier to identify and track down bugs or issues that may arise. By examining the changes made in each commit, you can isolate the introduction of bugs and identify the responsible code. This greatly aids in debugging, troubleshooting, and maintaining the stability and reliability of the system.

6. Backup and disaster recovery: Git serves as an effective backup mechanism for your clinic management system's source code and project assets. By pushing your changes to a remote repository (e.g., GitHub, Bitbucket), you create an off-site copy of your code, reducing the risk of data loss due to hardware failures, accidental deletions, or other disasters. In case of any issues with your local repository, you can easily clone the remote repository and continue your work.

## 5.2 Steps and tools that we use in our project to implement git

To implement Git for version control in the clinic management system using GitHub as the repository hosting service, you can follow these steps:

Step 1: Set Up Git

1. Install Git: Download and install Git from the official Git website. Follow the installation instructions for your operating system.

2. Configure Git: Setting up our name and email address in Git using the following commands:

git config --global user.name "Your Name"

git config --global user.email "your@email.com"

Step 2: Create a New Repository on GitHub

1. Sign up for GitHub: Go to the GitHub website (https://github.com) and sign up for an account if you don't have one already.

2. Create a New Repository: Click on the "New" button on the GitHub homepage to create a new repository. Provide a name for the repository (e.g., "clinic-management-system") and optionally add a description.

3. Initialize the Repository: Choose the option to initialize the repository with a README file. This will create an initial commit in the repository.

Step 3: Clone the Repository

1. Clone the Repository: On your local machine, open Git CMD or Git Bash and navigate to the directory where you want to store the clinic management system's code. Use the following command to clone the repository:

git clone <repository-url>

Replace `<repository-url>` with the URL of the repository you created on GitHub.

2. Change Directory: Enter the newly created directory using the following command:

cd <repository-name>

Replace `<repository-name>` with the name of the repository.

Step 4: Collaborate with Git

1. Create a Branch: Before making any changes, create a new branch for your work using the following command:

git branch <branch-name>

Replace `<branch-name>` with a descriptive name for your branch.

2. Switch to the New Branch: Switch to the newly created branch using the following command:

git checkout <branch-name>

3. Make Changes: Make the necessary changes to the clinic management system's code using your preferred code editor or IDE.

4. Stage and Commit Changes: Use the following commands to stage and commit our changes:

git add <file1> <file2> ... # Stage specific files

git add . # Stage all changes

git commit -m "Commit message"

Replace `<file1>`, `<file2>`, etc., with the names of the files you want to stage.

5. Push Changes: Push our changes to the remote repository on GitHub using the following command:

git push origin <branch-name>

This will make your changes available to other collaborators.

6. Create a Pull Request: If we want to merge our changes into the main codebase, go to the GitHub repository page and click on the "New pull request" button. Select our branch and provide a description of the changes. Once created, the pull request can be reviewed and approved by other collaborators.

Step 5: Review and Approve Pull Requests (Collaborator Workflow)

1. Review Pull Requests: As a collaborator, we can review pull requests created by others. Review the changes made and provide feedback or approval.

2. Approve and Merge: Once the changes in the pull request have been reviewed and approved, a collaborator with merge permissions can merge the changes into the main codebase on GitHub.

# Chapter Six

## 6.1 Unit Test

Unit testing plays a significant role in the development process of a clinic management system. It involves writing and executing tests on individual units of code, typically at the function or class level. Here are the key reasons why unit testing is important:

1. Detecting and preventing bugs: Unit tests help identify bugs and errors in code early in the development process. By writing tests that cover different code paths and scenarios, developers can catch issues before they propagate to other parts of the system. Detecting and fixing bugs early reduces the time and effort required for debugging and prevents the accumulation of technical debt.

2. Ensuring code correctness: Unit tests provide a means to verify the correctness of code implementations. By defining test cases that encompass various scenarios and expected outcomes, developers can validate that their code behaves as intended. This is particularly important in the clinic management system, where accuracy and reliability are crucial for patient data management, scheduling, billing, and other critical functionality.

3. Facilitating refactoring and code maintenance: Unit tests act as a safety net when making changes to the codebase. When refactoring or modifying existing code, running the associated unit tests ensures that the changes don't introduce unexpected behavior or regressions. If a test fails after a modification, it alerts developers to potential issues and guides them in fixing the problem promptly. This allows for more confident refactoring and code maintenance, promoting system stability and maintainability.

4. Enabling code reusability and modularity: Unit testing encourages developers to write modular, decoupled code. When code is independent and testable in isolation, it becomes easier to reuse in different parts of the clinic management system. Unit tests act as documentation for how the code should be used and provide reassurance that it functions correctly regardless of its context. This promotes code reusability, reduces redundancy, and improves overall development efficiency.

5. Supporting collaboration and team productivity: Unit tests facilitate collaboration among team members. When different developers work on different units of the clinic management system, unit tests serve as a shared understanding of how the code should behave. New developers joining the project can refer to existing tests to understand the expected behavior and make modifications with confidence. Additionally, running unit tests automatically provides quick feedback on the status of the codebase, allowing developers to identify and address issues promptly.

6. Regression testing and continuous integration: Unit tests form a critical part of regression testing. By running the tests regularly, either manually or as part of a continuous integration (CI) process, developers can ensure that changes made to the codebase don't break existing functionality. This helps maintain system stability and prevents unintended consequences when new features or bug fixes are introduced.

7. Improving code design and architecture: Writing unit tests often leads to better code design and architecture. To make code testable, developers are encouraged to reduce dependencies, modularize functionality, and adhere to SOLID principles. This, in turn, leads to more maintainable and loosely coupled code, making future enhancements and modifications easier.

Here is an example of the Junit test we have done for the class patient:

import org.junit.Before;

import org.junit.Test;

import static org.junit.Assert.\*;

public class PatientTest {

private Patient patient;

@Before

public void setUp() throws exception {

patient = new Patient();

}

@Test

public void testScheduleAppointment() {

// Test scheduling an appointment

patient.scheduleAppointment();

}

@Test

public void testEditAppointment() {

// Test editing an appointment

patient.editAppointment();

}

@Test

public void testSeeMedicalReports() {

// Test seeing medical reports

patient.seeMedicalReports();

}

}

## 6.2 Steps and Tools Used in Unit Test

Performing unit testing involves several steps to ensure that individual units of code are tested thoroughly. Here is an outline of the typical steps involved in unit testing:

1. Identify units to be tested: Determine the specific units of code that need to be tested. Units can include individual functions, methods, or classes. In the context of the clinic management system, units might encompass modules responsible for patient registration, appointment scheduling, billing calculations, or data validation.

2. Create test cases: Define test cases that cover different scenarios and potential edge cases. Test cases should encompass a range of inputs and expected outputs to validate the behavior of the unit being tested. For example, in the patient registration module, test cases might include verifying that a new patient is correctly added to the system when all required fields are provided, or checking that an error is thrown when invalid data is entered.

3. Choose a unit testing framework: Select a unit testing framework that is suitable for our programming language and development environment. Commonly used unit testing frameworks include:

- JUnit: A popular framework for Java applications.

- PyTest: A testing framework for Python applications.

- NUnit: A framework for testing .NET applications.

- RSpec: A behavior-driven development framework for Ruby applications.

From this we used the Junit to test our code.

4. Write unit tests: Implement the unit tests using the chosen framework. Write test methods that correspond to the identified test cases. Each test method should invoke the unit being tested with the appropriate inputs and assert the expected output or behavior. For example, a test method in the billing calculations module might calculate a bill amount and assert that it matches the expected result.

5. Execute unit tests: Run the unit tests using the testing framework. The framework will execute all the defined test methods and report the results. Typically, the framework provides feedback on the number of tests executed, the number of tests passed, and any tests that failed or encountered errors. This step helps identify any issues in the units under test.

6. Analyze test results: Review the test results to identify any failures or errors. Failed tests indicate that the unit being tested does not behave as expected, while errors may indicate issues with the test setup or environment. Analyzing the results helps pinpoint defects and guide the debugging process.

7. Debug and fix issues: If any tests fail or encounter errors, debug the code to identify and fix the issues. Refactor the code if necessary and rerun the unit tests to ensure that the fixes have resolved the problems. Iteratively debug and fix until all tests pass successfully.

8. Repeat for all units: Repeat the steps above for each unit of code that requires testing. Ensure that all units are thoroughly tested according to the defined test cases.

9. Automate testing: Consider automating the execution of unit tests as part of a continuous integration (CI) or continuous delivery (CD) pipeline. Automating the tests allows for regular execution and immediate feedback on the health of the codebase, ensuring that any regressions or issues are caught early.

# Chapter seven

## 7.1 Build (prepare build script for compilation, unit test, jar file creation)

The process of creating a build for the clinic management system involves several stages, including compilation, unit testing, and JAR (Java Archive) file creation. Here's an overview of each stage:

1. Compilation: Compilation is the process of converting the source code of the clinic management system into executable code that can be understood and executed by the computer. Compilation is language-specific, so the process may vary depending on the programming language used.

In the case of a Java-based clinic management system, the source code is typically written in Java and compiled using a Java compiler (e.g., `javac`). The compiler checks the syntax, resolves dependencies, and generates bytecode files (`.class` files) for each class in the system. The bytecode is a platform-independent representation of the code that can be executed by the Java Virtual Machine (JVM).

2. Unit Testing: Unit testing, as discussed earlier, involves testing individual units of code in isolation to ensure their correctness and functionality. During the unit testing stage, you run the unit tests written specifically for each unit of the clinic management system.

Using a unit testing framework like JUnit, we can execute the unit tests that we've created. The framework provides assertions and test runners that help automate the execution and reporting of test results. The goal is to ensure that each unit of code behaves as expected and passes the defined test cases.

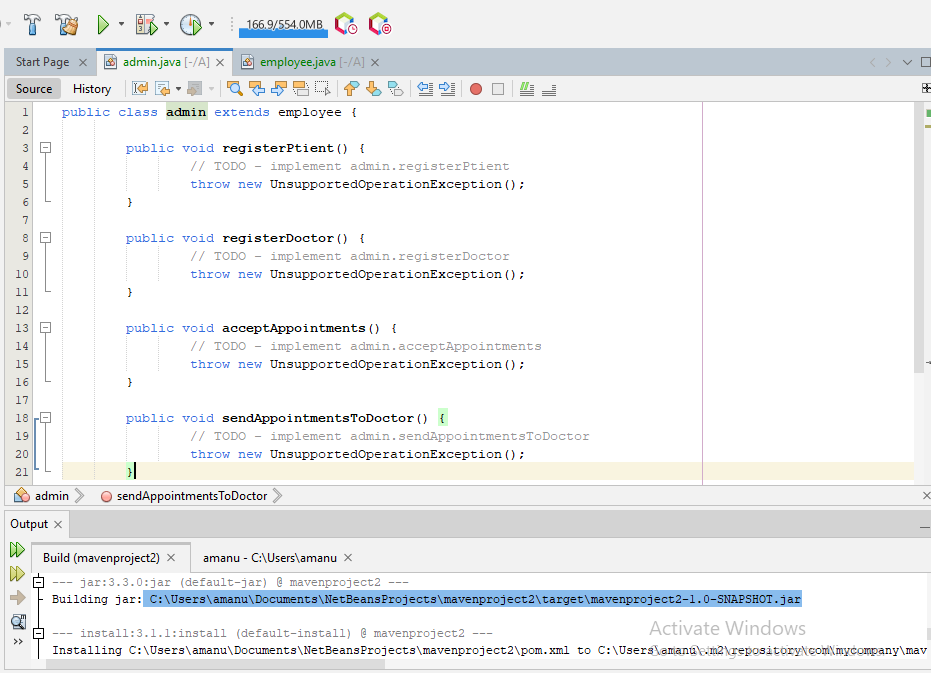
Unit tests are typically executed automatically as part of a build or continuous integration (CI) process to catch any regressions or issues early in the development cycle.

3. JAR File Creation: After successful compilation and unit testing, the next step is to package the compiled code and necessary resources into a JAR file. A JAR file is a compressed archive format used to distribute Java classes and related resources.

To create a JAR file, we use the Java Archive (JAR) tool, which is included in the Java Development Kit (JDK). The JAR tool allows us to specify the compiled bytecode files, along with any required resource files (such as configuration files or static assets), and create a single JAR file that encapsulates all the necessary components of the clinic management system.

The JAR file can be executed directly, or it can be included as a dependency in other Java projects. It provides a convenient way to distribute and deploy the clinic management system as a standalone application or as part of a larger software ecosystem.

During the build process, additional stages may be included based on the specific requirements of the clinic management system, such as integration testing, code analysis, or documentation generation. However, the compilation, unit testing, and JAR file creation stages are fundamental in creating a functional and deployable build for the clinic management system.

Here we have created a jar file by using apache netbeans:

## 7.2 Steps and tools that we use in our project to implement build

Implementing the build process for a clinic management system involves a series of steps to automate the compilation, testing, packaging, and deployment of the software. Here are the general steps involved:

1. Version Control: Set up a version control system, such as Git, to manage the source code of the clinic management system. Version control allows multiple developers to collaborate, track changes, and maintain a history of the codebase.

2. Build Configuration: Create a build configuration file that specifies the necessary build steps, dependencies, and settings for the clinic management system. This file can be written in a build configuration language specific to the chosen build automation tool.

3. Dependency Management: Identify and manage the dependencies required by the clinic management system. Use a dependency management tool or build automation tool that can resolve and fetch the necessary libraries or frameworks automatically. Common dependency management tools include Maven, Gradle, or npm, depending on the programming language and ecosystem used.

4. Build Automation Tool: Choose a build automation tool that suits our development environment. The build automation tool orchestrates the build process, executing the defined build configuration and managing the necessary tasks. Some popular build automation tools are:

-Apache Maven: A widely used build automation and dependency management tool for Java-based projects.

- Ant: A Java-based build tool that allows you to define custom build scripts.

- Apache NetBeans: integrates with two popular build automation tools: Apache Maven and Apache Ant.

In this project we have done the implementation of build by using apache netbeans.

5. Continuous Integration (CI): Integrate the build process into a CI system for automated and regular builds. CI systems automatically trigger the build process whenever changes are pushed to the version control repository. This ensures that the clinic management system is built and tested continuously, reducing the risk of integration issues and catching bugs early. Popular CI tools include Jenkins, Travis CI, CircleCI, or GitLab CI/CD.

6. Automated Testing: Incorporate automated testing into the build process. Write unit tests, integration tests, and any other necessary test suites for the clinic management system. Configure the build automation tool or CI system to execute these tests as part of the build process. This ensures that the tests are run automatically, providing feedback on the health and correctness of the system.

7. Packaging and Deployment: Define the packaging and deployment steps for the clinic management system. Specify how the compiled code, configuration files, and resources should be packaged into executable artifacts. Depending on the technology stack, this might involve creating JAR files, Docker containers, or deploying to cloud platforms like AWS or Azure.

8. Build Trigger: Configure the build automation tool or CI system to trigger the build process automatically on specific events, such as code commits, pull requests, or scheduled intervals. This ensures that the build process is initiated consistently and provides timely feedback on the status of the clinic management system.

9. Build Notifications and Reports: Set up notifications and reports to communicate build results and status. Configure the build automation tool or CI system to send notifications, such as emails or chat messages, to relevant stakeholders, informing them about the build outcomes. Generate and store build reports that include test results, code coverage, and other relevant metrics to track the quality and progress of the clinic management system.