Wachamo University

College of Engineering and Technology
Department of Software Engineering

Software Engineering Tools and practice

Pharmacy management system (PMS)

Prepared by

GROUP MEMBER	ID
1, Ashenafi Ayele	1501716
2, Abdisa teso	1500571
3, Nejwa Million	150119
4, Metekiya zerihun	1501110

Submitted to: Mr Feyisa Kedir

Date: April 25, 2025

Table of contents

Chapter One

Requirement Analysis

1.1 Requirement Analysis (Use case)

Functional Requirements for ' Administrator' Actor Use Cases:

1. Login

- The system must authenticate the administrator using valid credentials.
- On successful login, the admin is redirected to the dashboard.
- If credentials are invalid, the system must display an error message.

2. Register Users

- The administrator must be able to add new Pharmacists.
- The system must collect user details: name, role, username, and password.
- The system must validate and store the new user data.

3. Manage Medicines

- The administrator must be able to add, edit, and delete medicine records.
- Each medicine record must include: name, type, price, quantity, and expiry date.

4. Track Inventory

- The system must allow the admin to view current stock levels of all medicines.
- The system should notify medicines that are low in stock.
- Filtering and sorting by name, type, or quantity must be available.

5. Monitor Expired Medicines

- The system must display a list of all expired medicines.
- Admin must be able removed.
- Expired items should be flagged automatically based on expiry date.

6. View Sales Reports

- The system must generate and display reports summarizing sales over a selected time period.

- Each report must include medicine name, quantity sold, total sales value, and pharmacist details.
- Reports must be viewable by day, week, month, or custom range.

7. View Weekly Reports

- The system must automatically generate weekly inventory movement reports.
- Reports must include: items sold, current stock, and expired items.

8. View Activity Logs

- The system must record and display logs of pharmacist activities (e.g., login, sales).
- Logs must include: user email, action performed.
- Admin must be able to filter logs by user email.

9. Logout

- Admin must be able to securely log out of the system.
- The session must be terminated to prevent unauthorized access.

Functional Requirements for ' pharmacist' Actor Use Cases:

1. Login

- The system must authenticate the pharmacist using a valid username and password.
- On successful login, the pharmacist is redirected to their dashboard with limited privileges.
- Invalid credentials must prompt an error message.

2. View Medicines

- The system must display a list of all available medicines.
- Each medicine entry must show: name, type, quantity in stock, price, and expiry date.
- Pharmacist can search or filter the medicine list by name, type.

3. Manage Medicines

- The pharmacist must be able to add, edit, and delete medicine records.
- Each medicine record must include: name, type, price, quantity, and expiry date

4. Sell Medicine

- Pharmacist must be able to select medicines and enter the quantity to sell.

- The system must validate available stock before completing the sale.
- Upon successful transaction:
    - The stock must be reduced accordingly.
    - A sales record must be created with medicine name, quantity, date/time, and pharmacist ID.

4. Receive Stock Alerts

- The system must display real-time alerts for:
    - Medicines with low stock
    - Medicines expiry.

5. Generate weekly report

- Report must be generated from day to day pharmacist sell, update, delete of the stock
- Reports must include: items sold, current stock, and expired items.

6. Logout

- Pharmacist must be able to securely log out of the system.
- The session must be terminated to prevent unauthorized access.

1.2 Use case diagram components

Actors Identification

No	Actor	Description
1	**Administrator**	Manages system operations and overall pharmacy management.
2	**Pharmacist**	The person interacts with end user, Focuses on daily operations (selling medicines, stock updates).

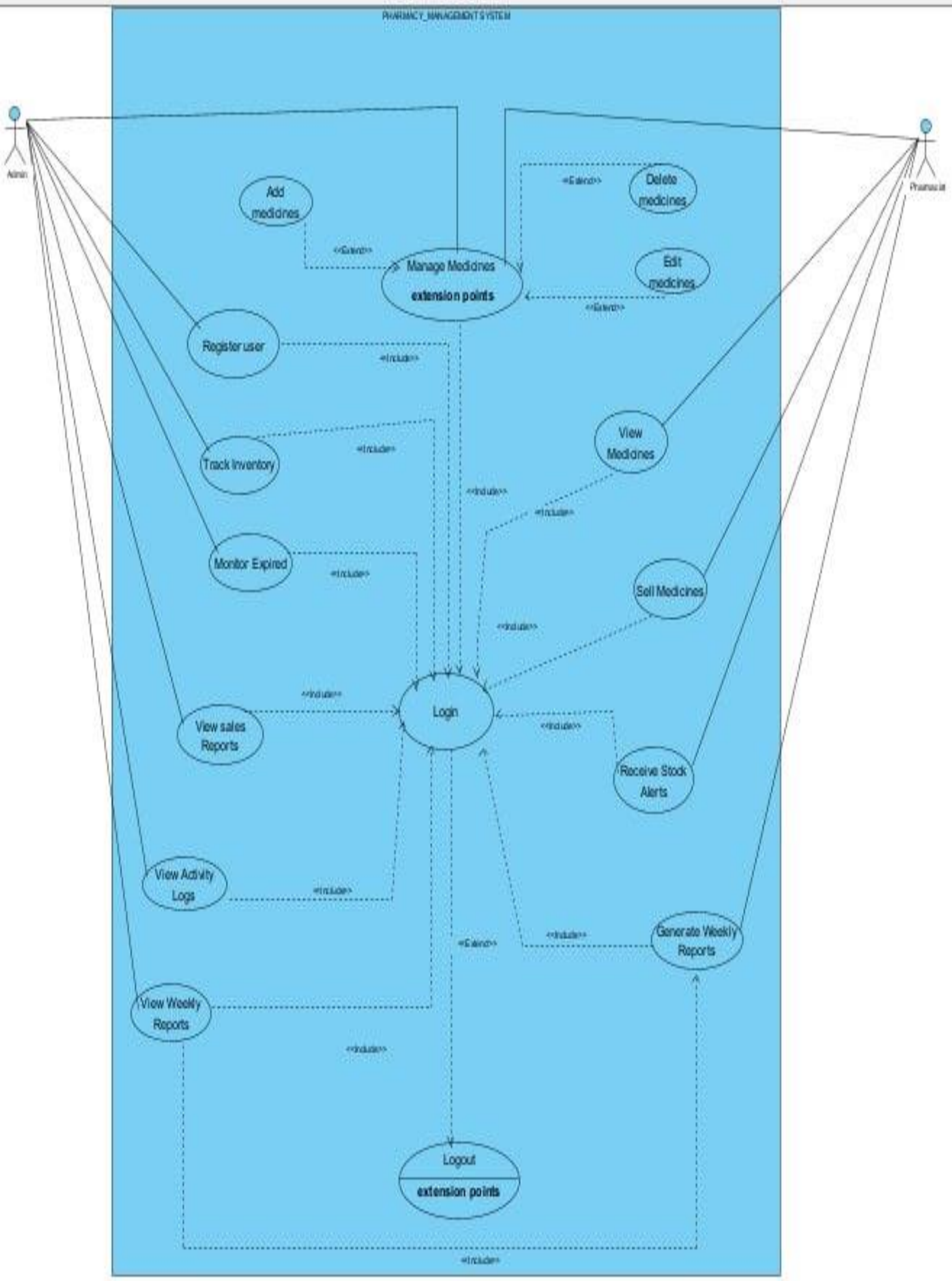
Use Cases

Use cases describe the specific functionalities or services provided by the system. They represent goals that actors want to achieve. In your system, the use cases for each actor are:

Category	ID	**Use Case Name**	**Use Case Description**
Common use Case	UC-1	login	Login in the system using secure credentials
	UC-2	Manage medicines	Add, update and delete medicine
	UC 3	Logout	Log out from the system securely

Admin use case	UC-4	Register user	Add pharmacist into the system
	UC-5	Track Inventory	view current stock levels of all medicines
	UC-6	Monitor Expired Medicines	Generate a list of medicines nearing expiry and remove expired items.
	UC-7	View Sales Reports	a summary of sold medicine
	UC-8	View Weekly Reports	View weekly sold, expired ,and stock level of medicine
	UC- 9	View Activity Logs	Check logs of user actions (e.g., who modified stock or who sold medicine).
Pharmacist	UC-10	View Medicines	Search medicines by name, category, or availability.
	UC-11	Sell Medicine	Process a sale: Select medicine, sell for customer and deduct stock automatically.
	UC-12	Receive Stock Alerts	Get notified when stock is low or nearing expiry.
	UC-13	Generate weekly report	Generate weekly sold, expired ,and stock level of medicine

1.3 Example of use case model


PHARMACY_MANAGEMENT SYSTEM
Admin
Pharmacist
Add medicines
<<Extend>>
Manage Medicines
extension points
Delete medicines
Edit medicines
Register user
<<Include>>
Track Inventory
View Medicines
Monitor Expired
Sell Medicines
View sales Reports
Login
Receive Stock Alerts
View Activity Logs
Generate Weekly Reports
View Weekly Reports
Logout
extension points

1.4 use Case Description Template

Use Case ID		UC-1
Use Case Name		login
Actor		Administrator &pharmacist
Summary		Authenticate and access system based on role
Precondition		User is registered
Basic Scenario	**Actor Action**	**System response**
. 1. User submits credentials. 2. System checks them. 3. System routes to dashboard	Enters username & password	Verifies credentials; on success, redirects to role-specific dashboard.
Alternative Scenario		If credentials are wrong: show error & request re-entry.
Post Condition		User lands on their respective dashboard.

Use Case ID		UC-2
Use Case Name		Manage medicine
Actor		Administrator &pharmacist
Summary		Add, update, or delete medicines
Precondition		User is logged in
Basic Scenario	**Actor Action**	**System response**

1. User enters “Manage Medicines.” 2. Inputs new/edited data. 3. System validates & saves.	Chooses to add/edit/delete a medicine; fills in details.	Validates inputs; on success, updates database and acknowledges..
Alternative Scenario		If validation fails: prompt user to correct errors.
Post Condition		Medicine records updated per user’s action.

Use Case ID		UC-3
Use Case Name		Log out
Actor		Administrator &pharmacist
Summary		Securely end user session and return to login
Precondition		User is logged in
Basic Scenario	**Actor Action**	**System response**
1. User clicks “Logout.” 2. System terminates session. 3. Redirects to login page.	Clicks logout button.	Destroys session tokens; redirects to login screen.
Alternative Scenario		-
Post Condition		User session ended

Use Case ID	UC-4
Use Case Name	Register user
Actor	Administrator

Summary		Add new users pharmacists
Precondition		Admin is logged in
Basic Scenario	**Actor Action**	**System response**
1. Admin opens “Register User.” 2. Inputs user data & role. 3. Submits form. 4. System saves record & confirms..	Navigates to user registration; fills in details & assigns role.	Persists new-user info; displays confirmation message.
Alternative Scenario		-
Post Condition		New user created with credentials & assigned role.

Use Case ID		UC-5
Use Case Name		Track Inventory
Actor		Administrator
Summary		Overview and update of current stock levels
Precondition		Admin is logged in
Basic Scenario	**Actor Action**	**System response**
1.admin opens inventory. 2. System lists each stock quantity. 3. User modifies counts if needed. 4. System saves updates.	Accesses “Track Inventory”; reviews and/or edits quantities.	Shows up-to-date stock levels; persists any changes upon submission..
Alternative Scenario		-
Post Condition		Inventory view shows latest

	data; any edits are stored.

Use Case ID		UC-6
Use Case Name		Monitor Expired Medicines
Actor		Administrator
Summary		Identify expired or near-expiry medicines
Precondition		Admin is logged in
Basic Scenario	**Actor Action**	**System response**
1. System scans expiry dates. 2. Generates list of expired/soon-to-expire. 3. User takes action (remove/replace).	Reviews flagged items.	Reviews flagged items.
Alternative Scenario		-
Post Condition		Expired/near-expiry medicines are identified for action.

Use Case ID		UC-7
Use Case Name		View Sales Reports
Actor		Administrator
Summary		
Precondition		Admin is logged in
Basic Scenario	**Actor Action**	**System response**

1. Admin goes to “Sales Reports.” 2. Picks time frame. 3. System compiles & shows report. 4. Option to export/download.	Selects report section; enters date range.	Queries database; displays report.
Alternative Scenario		-
Post Condition		Sales report is available on screen

Use Case ID		UC-8
Use Case Name		View Weekly Reports
Actor		Administrator
Summary		Provide weekly summaries of sales, inventory, activities
Precondition		Weekly data accumulated
Basic Scenario	**Actor Action**	**System response**
1. User selects week. 2. System aggregates sales, inventory, actions. 3. Displays summary report with charts/tables.	Chooses reporting module; picks week.	Collates relevant data; renders weekly dashboard and allows export.
Alternative Scenario		-
Post Condition		Weekly summary is generated and available

Use Case ID	UC-9
Use Case Name	View Activity Logs

Actor		Administrator
Summary		View system activity logs for audit
Precondition		Admin is logged in
Basic Scenario	**Actor Action**	**System response**
1. Admin visits “Activity Logs.” 2. Applies optional filters. 3. System retrieves & displays entries with timestamps and user IDs.	Opens logs section; sets any date/user filters.	Persists new-user info; displays confirmation message.
Alternative Scenario		-
Post Condition		Admin can review or export filtered log data..

Use Case ID		UC-10
Use Case Name		View Medicines
Actor		pharmacist
Summary		View list of available medicines and details
Precondition		pharmacist is logged in
Basic Scenario	**Actor Action**	**System response**
1. User opens medicine list. 2. System shows names, stock, expiry, price. 3. User filters/searches as needed.	Selects “View Medicines	Displays table of all medicines with key fields; supports filter/search.
Alternative Scenario		-
Post Condition		Medicine list displayed; user may

	interact with i

Use Case ID		UC-11
Use Case Name		sell medicine
Actor		Pharmacist
Summary		Sell medicines and record transactions
Precondition		Pharmacist is logged in Medicine is in stock
Basic Scenario	**Actor Action**	**System response**
1. Pharmacist selects item & quantity. 2. System checks availability. 3. Pharmacist confirms sale. 4. System updates stock & issues receipt.	Chooses medicine & enters quantity; confirms sale details.	Validates stock; deducts quantity; generates printable receipt.
Alternative Scenario		If insufficient stock: notify user & block sale.
Post Condition		Sale recorded; stock levels updated; receipt created.

Use Case ID	UC-12
Use Case Name	Receive Stock Alerts
Actor	Pharmacist

Summary		Alert when stock falls below threshold
Precondition		Admin is logged in Continuous inventory monitoring
Basic Scenario	**Actor Action**	**System response**
1. System checks levels periodically or on update. 2. When threshold breached, generates alert. 3. Alert appears in dashboard or notify.	-	Monitors stock; triggers alert event when any item < threshold.
Alternative Scenario		-
Post Condition		Pharmacist receives visual/notification alert.

Use Case ID		UC-13
Use Case Name		Generate weekly report
Actor		Pharmacist
Summary		Provide weekly summaries of sales, inventory, activities
Precondition		Pharmacist logged in Weekly data accumulated from daily pharmacist work
Basic Scenario	**Actor Action**	**System response**
1. User selects week. 2. System aggregates sales, inventory, actions. 3. Displays summary report with charts/tables	-	System Collates relevant data; renders weekly dashboard

Alternative Scenario	-
Post Condition	Weekly summary is generated and available

1.5 Tools and steps to draw Use Case

- **Diagramming Interface**: Visual Paradigm provides an intuitive drag-and-drop interface for creating and organizing diagram elements.
- **UML Toolset**: It includes comprehensive UML support for various diagram types, including use case diagrams.
- **Actor and Use Case Elements**: Predefined shapes for actors, use cases, and system boundaries.
- **Relationship Tools**: Tools for drawing associations, generalizations, includes, and extends relationships.
- **Documentation and Notes**: Features for adding detailed descriptions and notes to diagram elements

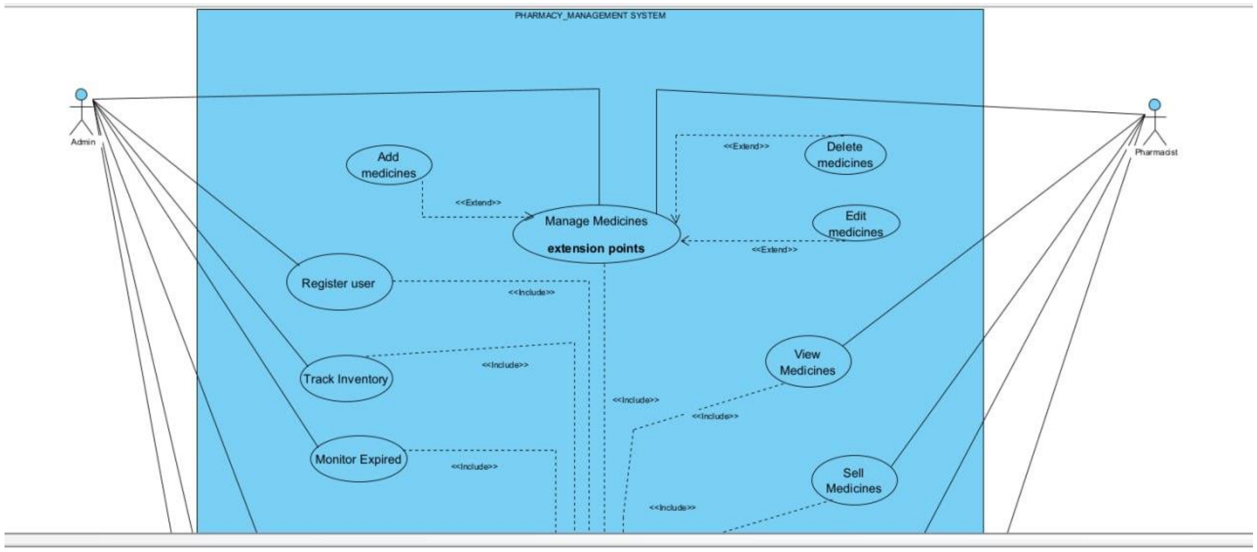
Methodologies

Unified Modeling Language (UML):

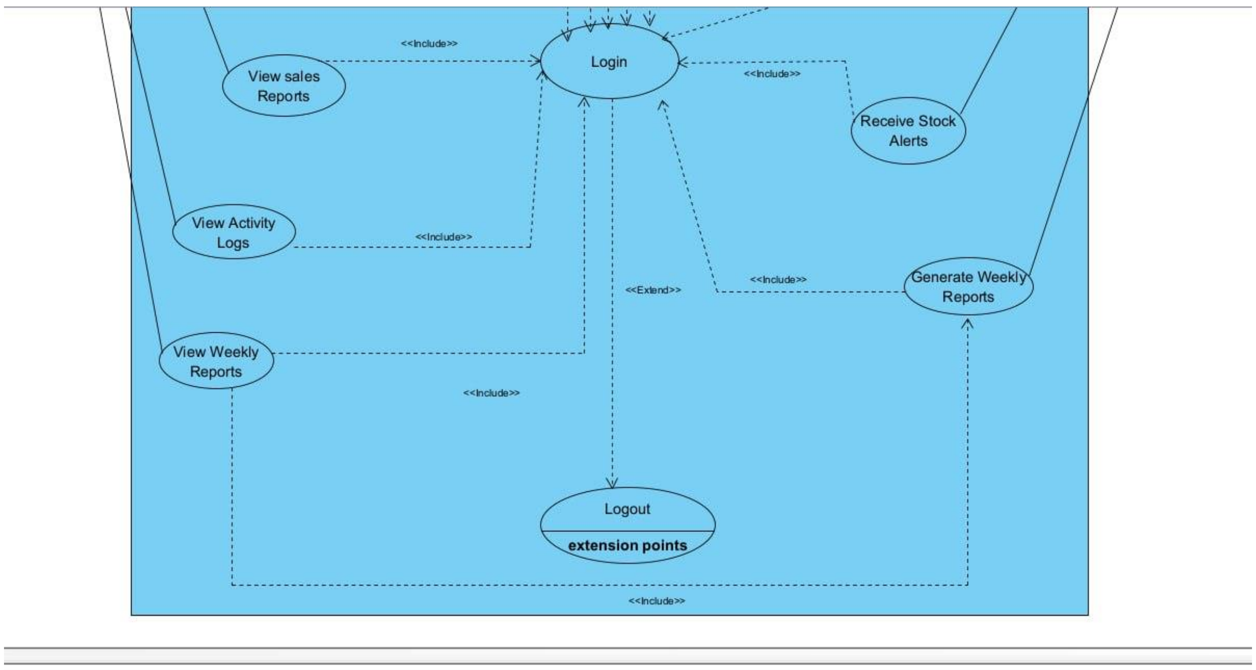
- **Actors**: Represent external entities that interact with the system.
- **Use Cases**: Represent the functional requirements or interactions between actors and the system.
- **System Boundary**: Defines the scope of the system and encapsulates all use cases.
- **Relationships**: Includes associations, generalizations, and dependencies between actors and use cases.

Step-by-Step Guide to Drawing a Use case Diagram

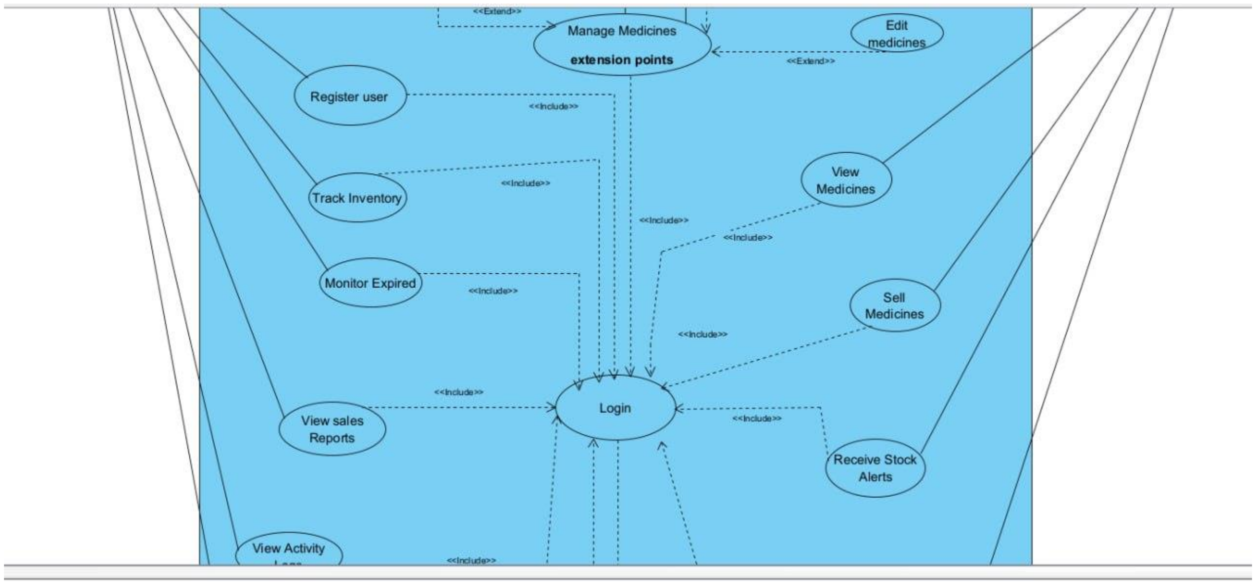
STEP-1


PHARMACY_MANAGEMENT SYSTEM
Admin
Pharmacist
Add medicines
<<Extend>>
Manage Medicines
extension points
<<Extend>>
Delete medicines
Edit medicines
<<Extend>>
Register user
<<Include>>
Track Inventory
<<Include>>
Monitor Expired
<<Include>>
<<Include>>
View Medicines
<<Include>>
Sell Medicines
<<Include>>

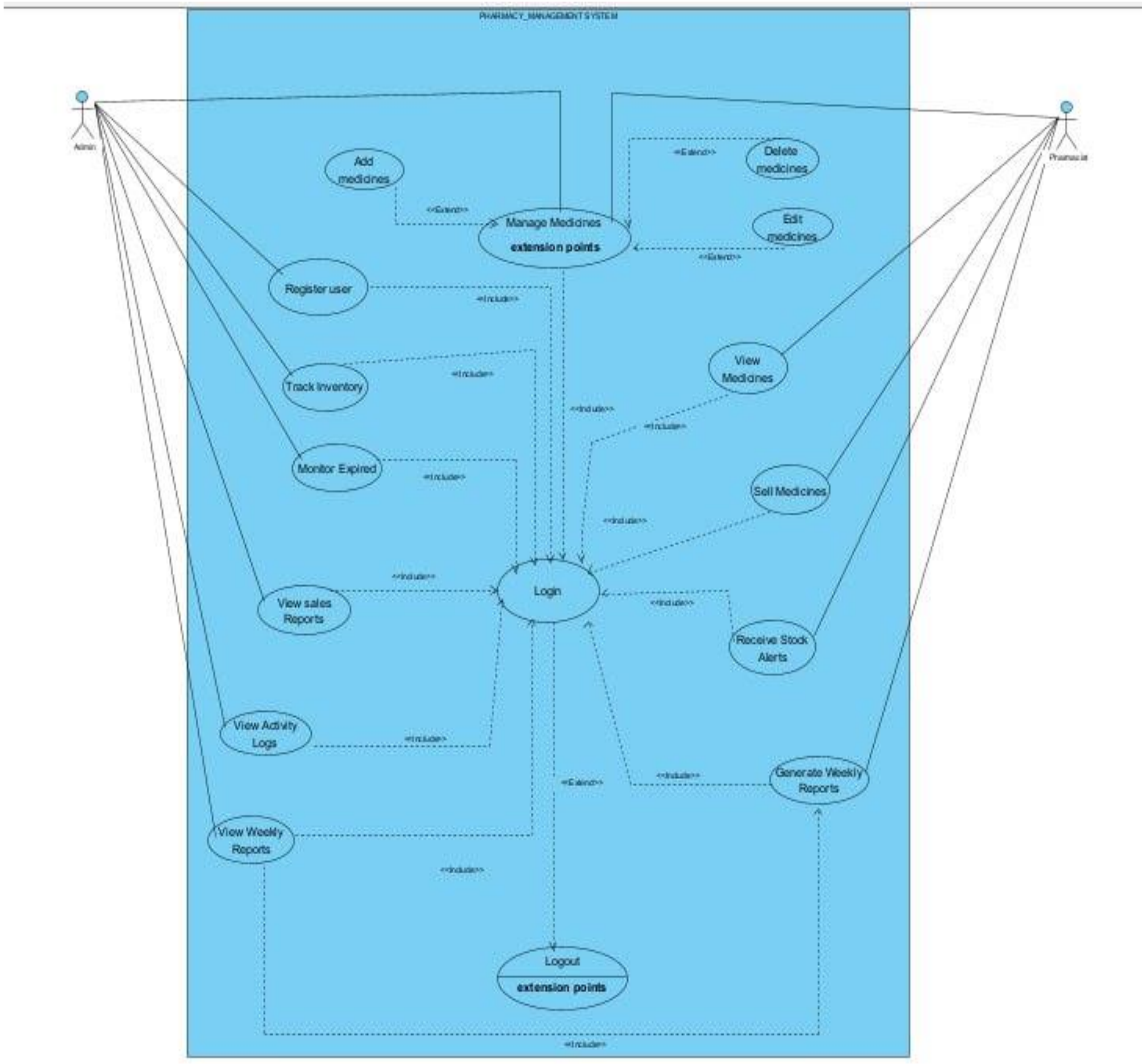
STEP- 2


View sales Reports
Login
Receive Stock Alerts
View Activity Logs
Generate Weekly Reports
View Weekly Reports
Logout
extension points
<<Include>>
<<Extend>>

Step-3


Manage Medicines
extension points
Edit medicines
Register user
Track Inventory
Monitor Expired
View Medicines
Sell Medicines
View sales Reports
Login
Receive Stock Alerts
View Activity Logs
<<Include>>
<<Extend>>

STEP-4


PHARMACY_MANAGEMENT SYSTEM
Admin
Pharmacist
Add medicines
Manage Medicines
extension points
Delete medicines
Edit medicines
Register user
Track Inventory
Monitor Expired
View Medicines
Sell Medicines
Login
View sales Reports
Receive Stock Alerts
View Activity Logs
Generate Weekly Reports
View Weekly Reports
Logout
extension points
<<Extend>>
<<Include>>

Chapter Two

2.1 High Level Sequence Diagram

A high-level sequence diagram is a critical tool in system design and architecture. It visually represents the interactions between different entities or components in a system. For a pharmacy Management System, this diagram serves as a blueprint illustrating how administrator and pharmacist interact and communicate with each other.

Importance of Sequence Diagrams

Sequence diagrams offer several benefits:
Clarity: They provide a clear visualization of the system's behavior and flow of operations.

Communication: They facilitate better communication among stakeholders by presenting complex interactions in an easily understandable format.

Analysis helps spot slowdowns, inefficient areas, or parts that need improvement in the system design.

2.2 Components of High-level Sequence Diagram

Sequence diagram consists

Actors: Entities that interact with the system, such as, admin and Pharmacist.

Objects: Instances of classes or components within the system, like, medicine management system, sell system registration system.

Lifelines: Vertical lines representing the timeline or existence of an actor or object during the interaction.

Messages: Arrows indicating the flow of communication or method calls between actors and objects.

Activations: Rectangular boxes showing the duration of a method call or process

Actors in the pharmacy management system

1. Administrator

- **Role**: Oversees the entire Pharmacy Management System, ensuring it functions smoothly.

- **Responsibilities**:
    - **System Management**: Configures and maintains system settings and updates.
    - **User Management**: Adds, removes, and assigns roles to pharmacists.
    - **Data Management**: Ensures data accuracy, security, and handles backup procedures.
- **Interactions**:
    - Adds or removes pharmacists and assigns roles.
    - Monitors system logs and user activity (e.g., who sold medicine, who updated stock).
    - Views weekly reports (sold, expired, stock levels).
    - Reviews stock alerts and ensures critical inventory is managed.
    - Manages expired medicines and ensures proper disposal or removal.

2. Pharmacist

Role: Frontline staff responsible for daily pharmacy operations.

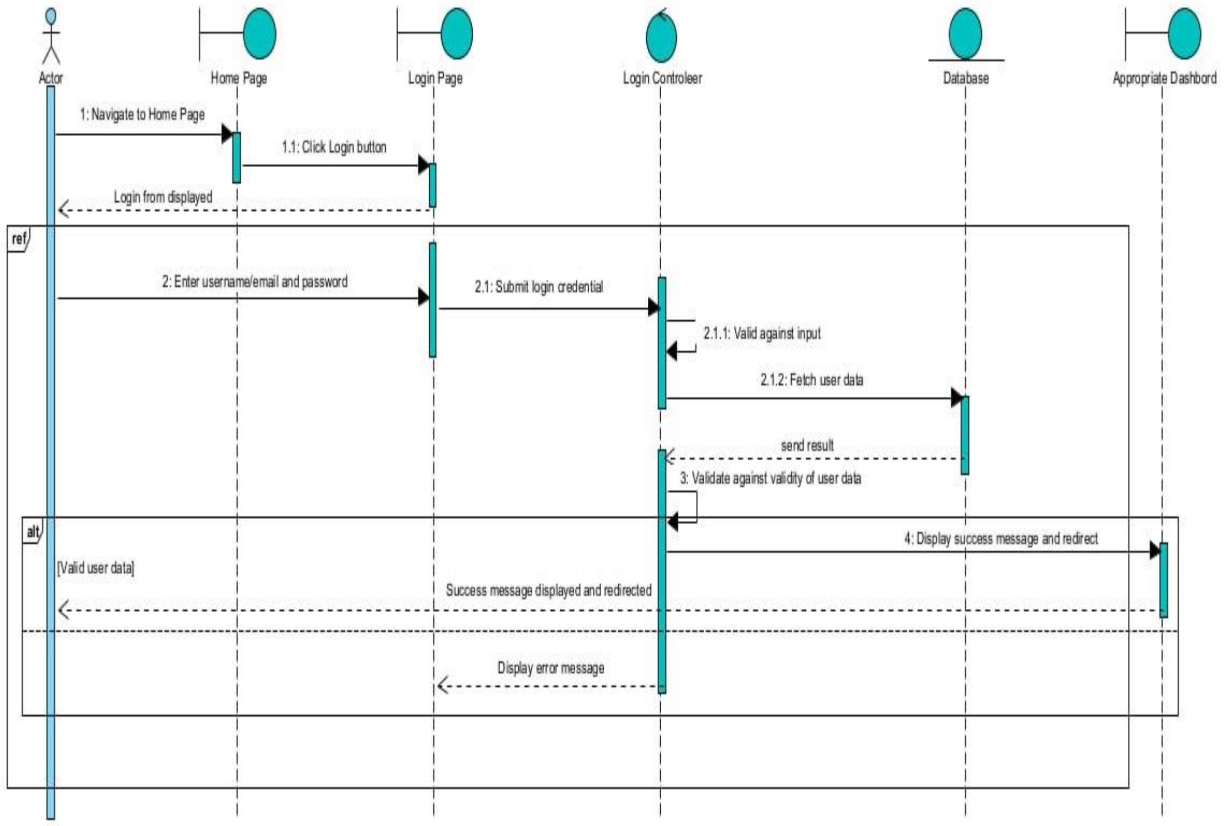
Responsibilities:

- **Medicine Management**: Add, update, and delete medicines from inventory.
- **Inventory Tracking**: Monitor current stock levels and receive stock alerts.
- **Sales Processing**: Sell medicine, ensuring accurate deduction from stock.
- **Reporting**: Generate and view sales reports and weekly summaries.

Interactions:

- Registers new medicines into the system.
- Processes sales: selects medicine, confirms quantity, and completes sale.
- Views medicine details by name, category, or availability.
- Monitors inventory and updates.
- Generates reports (weekly sold, expired, stock level).
- Receives automated alerts for low stock or nearing expiry.

2.3 Example of High Level Sequence



1. Login Sequence Diagram Description

This sequence diagram describes the step-by-step flow of how a user logs into the Pharmacy Management System. It demonstrates how user credentials are validated and how users are redirected based on role after successful authentication.

Actors Involved:

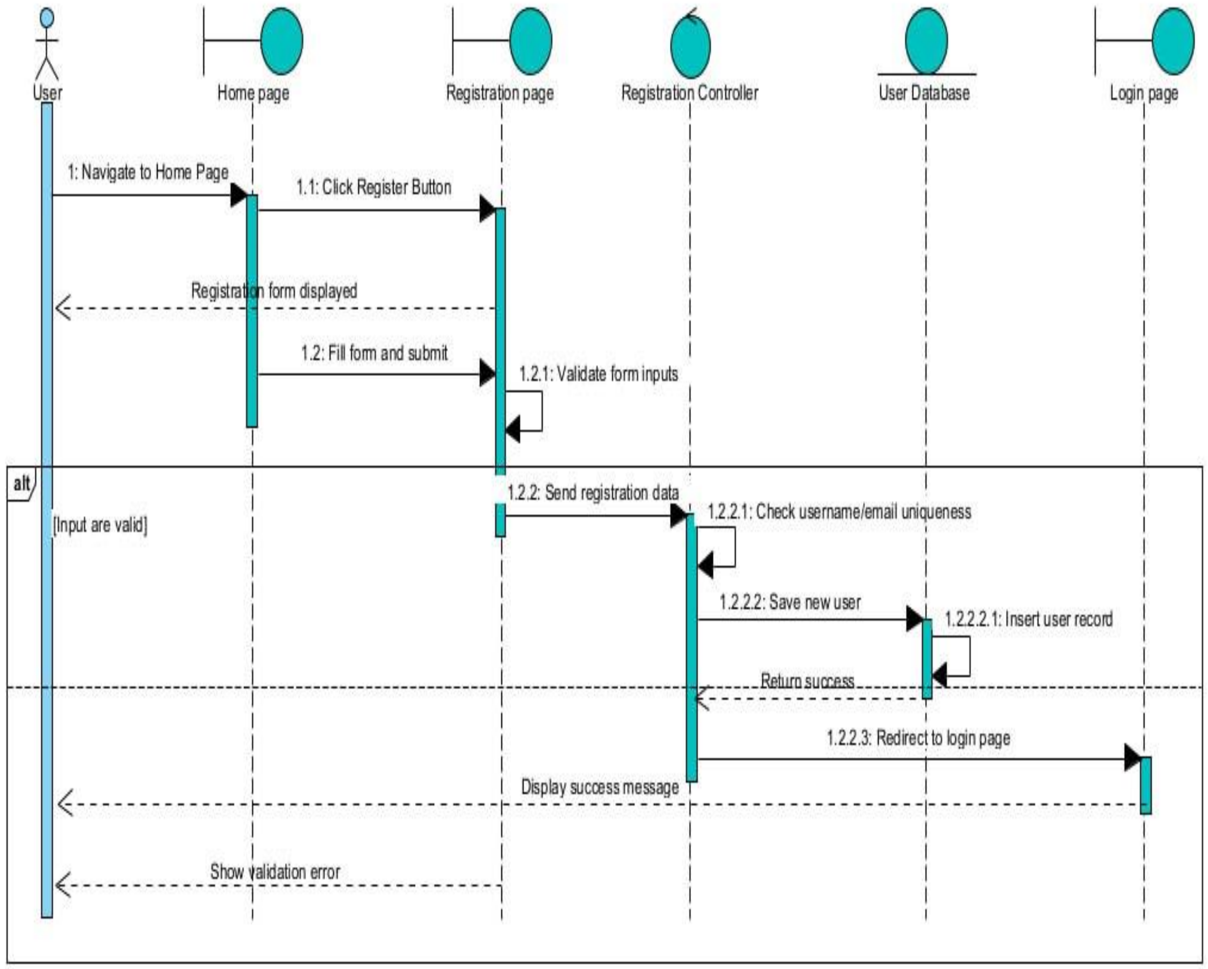
- **Actor (User)**: Initiates the login process.
- **Home Page**: Interface to navigate to the login page.
- **Login Page**: Accepts user credentials.
- **Login Controller**: Handles validation and data verification.
- **Database**: Stores user information.
- **Appropriate Dashboard**: Redirects to user-specific dashboard (Admin, Pharmacist, etc.).

Sequence Flow:

1. The user navigates to the home page.
2. The user clicks on the login button, and the login form is displayed.
3. The user enters their username/email and password.
4. The login page submits the credentials to the login controller.
5. The controller performs:
    - Input validation.
    - Fetches user data from the database.
    - Validates credentials against database records.
6. Based on the result:
    - If valid, a success message is shown and the user is redirected to their appropriate dashboard.
    - If invalid, an error message is displayed.

Alternative Flow:

- If the credentials are invalid, the system informs the user with an error message instead of proceeding to the dashboard.


User
Home page
Registration page
Registration Controller
User Database
Login page
1: Navigate to Home Page
1.1: Click Register Button
Registration form displayed
1.2: Fill form and submit
1.2.1: Validate form inputs
alt
[Input are valid]
1.2.2: Send registration data
1.2.2.1: Check username/email uniqueness
1.2.2.2: Save new user
1.2.2.2.1: Insert user record
Return success
1.2.2.3: Redirect to login page
Display success message
Show validation error

Pharmacist Registration Sequence Diagram

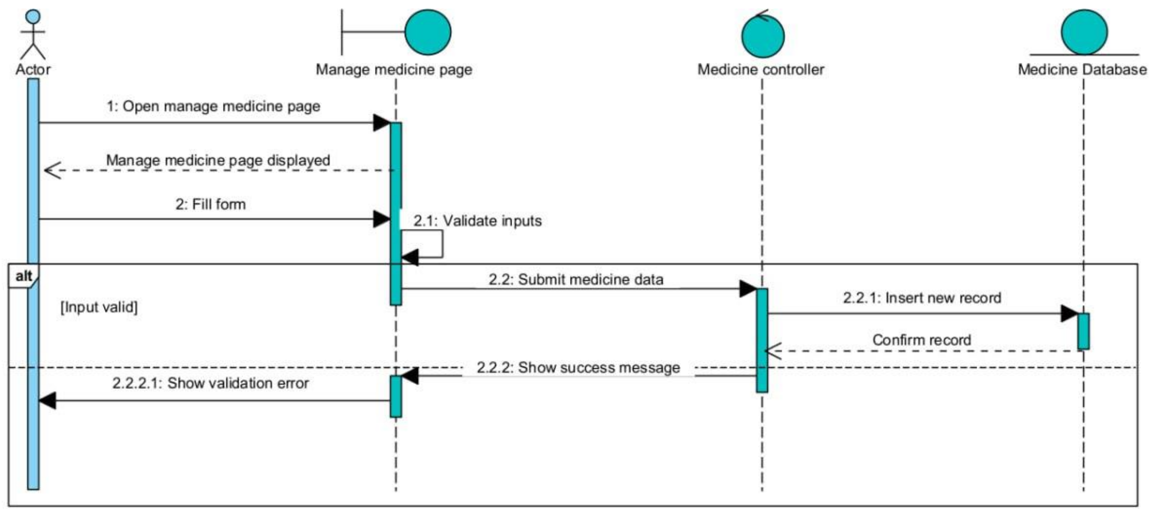
This diagram outlines the process for a new Pharmacist to register in the system. The actors and systems involved are:

- **Admin:** The Administration attempting to create a new Pharmacist account.
- **Home page:** The initial page of the application.
- **Registration page:** The page containing the registration form.
- **Registration Controller:** A component responsible for handling registration logic.
- **User Database:** The database storing user account information.

- **Login page:** The page where users can log in after registration.

The steps are as follows:

1. The Admin navigates to the Home page.
2. The Admin clicks the "Register" button on the Home page, which displays the Registration page.
3. The Admin fills in the registration form and submits it.
4. The Registration page sends the form inputs to the Registration Controller.
5. The Registration Controller validates the form inputs.
6. [Alternative path: Input are valid]
   - The Registration Controller sends the registration data to the User Database.
   - The User Database checks if the username/email is unique.
   - The User Database saves the new user record by inserting a record.
   - The User Database returns a success message to the Registration Controller.
7. **[Alternative path: Input are invalid]**
   - The Registration Controller sends a validation error message back to the Registration page, which is then shown to the Admin



n.

Manage Medicines Sequence Diagram

This sequence diagram illustrates the process of managing medicines in the Pharmacy Management System. It covers how a user accesses the manage medicine page, enters medicine details, and how the system validates and stores the data.

Actors Involved:

- Actor (User): Initiates the medicine management process by interacting with the system.
- Manage Medicine Page: Interface where the user inputs medicine data.
- Medicine Controller: Handles form submission and communicates with the database.
- Medicine Database: Stores the medicine records.

Sequence Flow:

1. The Actor opens the Manage Medicine Page.
2. The system displays the Manage Medicine Page to the Actor.
3. The Actor fills the medicine form.
4. The Manage Medicine Page validates the inputs.

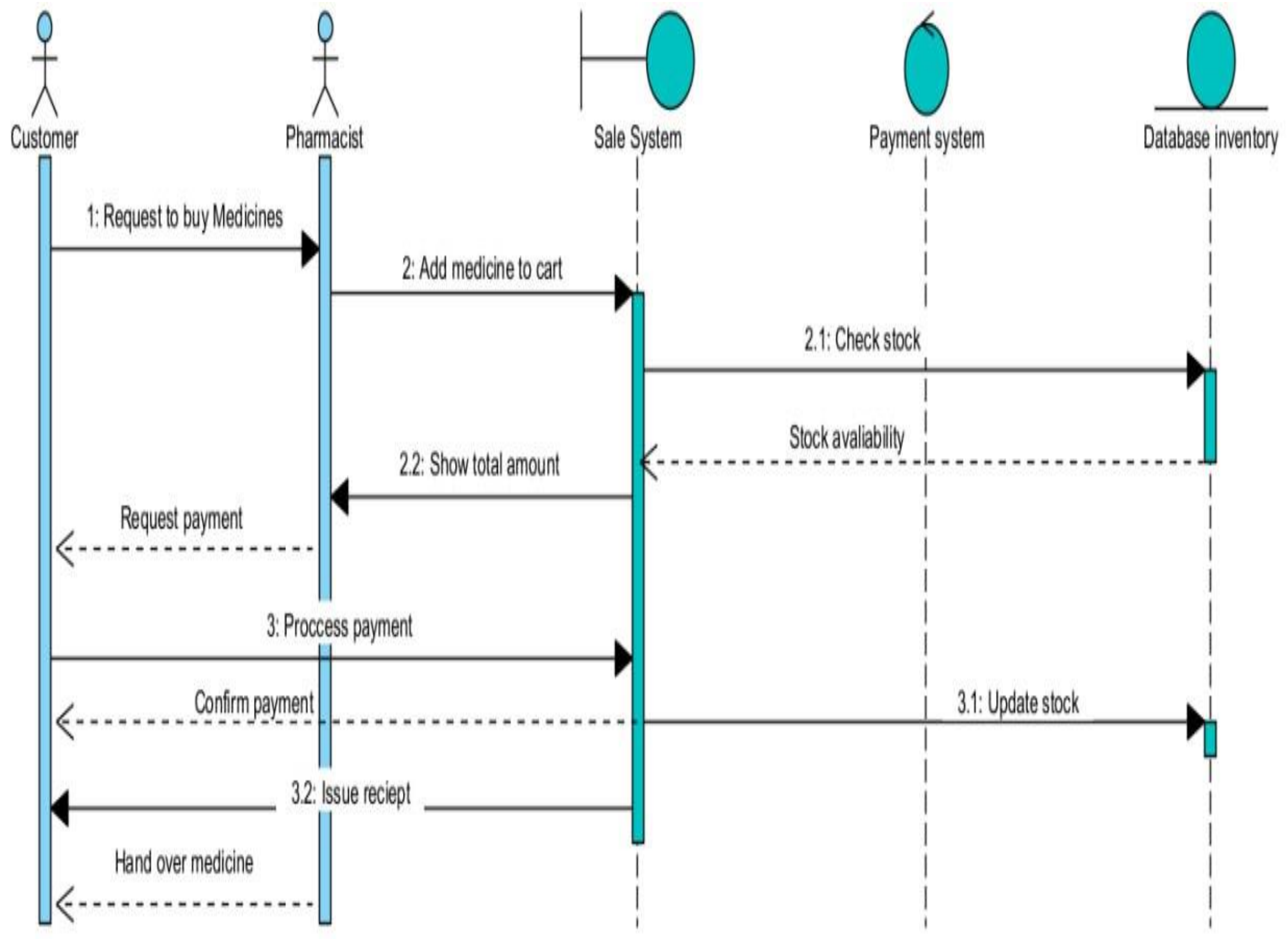
Alternative Flow:

- If the input is valid:

5. The Manage Medicine Page submits the medicine data to the Medicine Controller.
6. . The Medicine Controller inserts the new record into the Medicine Database
7. The Medicine Database confirms the insertion.
8. The Medicine Controller shows a success message to the Actor.

If the input is invalid:

The system displays a validation error to the Actor.



Sell Medicine Sequence Diagram

This sequence diagram illustrates the step-by-step process of managing a medicine purchase in the Pharmacy Management System. It demonstrates how a customer's purchase request is handled, including stock verification, payment processing, and inventory updates.

Actors Involved:

- **Customer**: Initiates the purchase request and completes payment.
- **Pharmacist**: Manages the cart, calculates totals, issues receipts, and hands over medicines.
- **Sale System**: Verifies stock, processes payments, and updates inventory.

- **Database Inventory**: Stores real-time stock data.

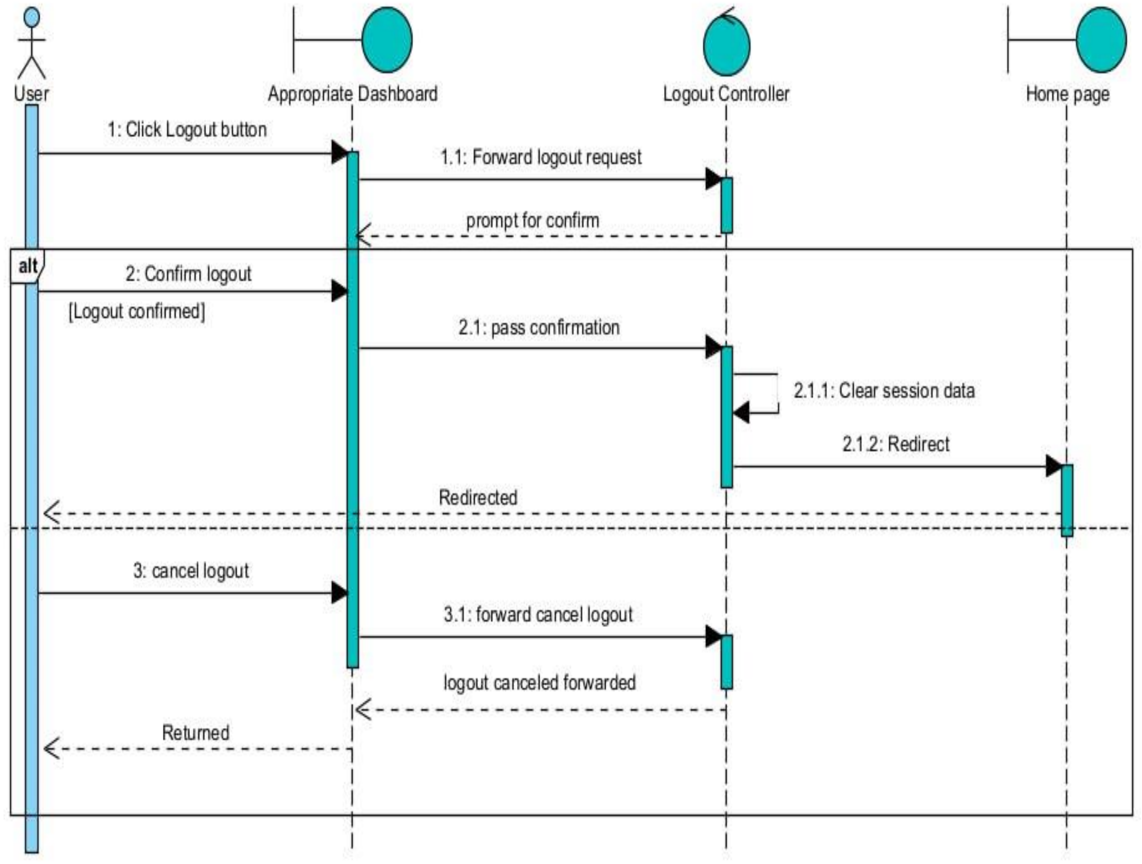
Sequence Flow:

1. **Customer** requests to purchase medicines.
2. **Pharmacist** adds the selected medicines to the cart.
3. **Sale System** checks stock availability in the Database Inventory:
    - If stock is available, proceed.
    - If unavailable, notify the Pharmacist/Customer (see *Alternative Flow*).
4. **Pharmacist** displays the total amount to the Customer.
5. **Customer** initiates payment.
6. **Sale System** processes the payment
7. **Sale System** updates the stock in the Database Inventory post-payment.
8. **Pharmacist** issues a receipt to the Customer.
9. **Pharmacist** confirms payment success and hands over the medicines.

Alternative Flow:

- **Stock Unavailable**
  - The Sale System alerts the Pharmacist about insufficient stock.
  - Pharmacist informs the Customer, and the payments canceled.
- **Payment Failure**
  - The Sale System notifies the Customer to retry or use an alternative payment method.

- If payment retries fail, the transaction is terminated, and stock remains unchanged



Logout Sequence Diagram

This diagram details the process of a user logging out of the system. The involved components are:

- **User:** The logged-in individual wishing to log out.
- **Appropriate Dashboard:** The user's currently viewed dashboard or main application area.
- **Logout Controller:** A component responsible for handling the logout process.
- **Home page:** The page the user is redirected to after logging out.

The sequence of events is as follows:

1. The User clicks the "Logout" button on the Appropriate Dashboard.
2. The Appropriate Dashboard forwards the logout request to the Logout Controller and may prompt the user for confirmation.
3. **[Alternative path: Logout confirmed]**

- The User confirms the logout.
- The Appropriate Dashboard passes the confirmation to the Logout Controller.
- The Logout Controller clears the user's session data.
- The Logout Controller redirects the User to the Home page.

4. **[Alternative path: Cancel logout]**
    - The User cancels the logout.
    - The Appropriate Dashboard forwards the cancel logout request to the Logout Controller.
    - The Logout Controller acknowledges the cancellation, and the user is returned to the Appropriate Dashboard.

2.4 Tools and Steps to Draw High-Level Sequence Diagram using Visual Paradigm

Step 1: Open Visual Paradigm and Create a New Project

- Launch Visual Paradigm.
- Create a new project or open an existing one where you want to create the sequence diagram.

Step 2: Access Sequence Diagram Tool

- In the Project Explorer or Toolbar, locate and click on "Diagram" or "New Diagram".
- Choose "Sequence Diagram" from the list of available diagram types.

Step 3: Define Actors and Objects

Drag and drop "Actor" shapes to represent key users such as Pharmacist; and Admin,.

- Drag and drop "Object" shapes to represent system components like Inventory System, Sales Module, medicine Management, and Stock Alert System.

Step 4: Add Lifelines

- Connect each actor and object with a lifeline, representing the timeline of their interaction.
- Right-click on the actor or object and choose "Add Lifeline", or simply drag a lifeline from the actor/object into the diagram.

Step 5: Add Messages and Interactions

- Use message arrows to show interactions (method calls, data exchanges) between actors and objects.
- Click on the lifeline of an actor/object to add messages such as:
    - "check stock availability “
    - " Receive Stock Alerts "
    - "Generate sale report”

Step 6: Add Activation Bars (Optional)

- To represent the duration of an operation, add activation bars**.**
- Drag and extend a bar from the lifeline to show how long the operation takes during the interaction.

Step 7: Add Comments and Notes (Optional)

- Use comments or notes to clarify interactions, explain system logic, or highlight business rules.
- Right-click on the diagram and choose **"**Note" or **"**Comment" to insert these explanations.

Step 8: Review and Validate

- Carefully review the sequence diagram to ensure it accurately captures all necessary interactions and logic.
- Use Visual Paradigm’s validation features to check for any errors, inconsistencies, or missing elements.

Chapter Three

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

Low-level design (LLD), also known as detail design or class design is the phase in software development where the system's components and modules are described in more detail. This phase follows high-level design and involves specifying the details of each class, including their methods, attributes, relationships, and interactions

3.2 Components of Class Diagram

● **User**:

- Attributes: UserId (string), Username (string), Userpassword (string), Userid (string).
- Methods: Login(), Logout().
- Role: Base class for system users, handling authentication and basic user operations.

Admin (inherits from User):

- Attributes: Admin Name (string), Admin phone (int).
- Methods: registerUser(), trackInventory(), monitorExpired(), viewSalesReport(), viewActivityPage(), viewWeeklyReports(), manageMachines().
- Role: Manages user registrations, monitors inventory, checks expired medicines, views reports, and oversees pharmacy equipment.

Pharmacist (inherits from User):

- Attributes: Phamid (string), pharmname (string), pharmphone (string).
- Methods: viewmedicine(), receiveStockAlert(), sellmedicine(), generateWeeklyReports(), manageMedicine().
- Role: Handles medicine sales, receives stock alerts, manages medicine details, and generates reports.

Inventory:

- Attributes: inventoryId (string), medicines (list of Medicine objects).
- Methods: trackStock(), updateStock().
- Role: Tracks real-time stock levels and updates inventory after sales or restocking.

Medicine:

- Attributes: medicineId (string), name (string), type (string), quantity (int), exprimDate (Date), price (double).
- Methods: Expired() (checks if the medicine is expired).
- Role: Represents individual medicines and their properties, including expiration status.

Alert:

- Attributes: alertId (string), message (string), date (Date).
- Methods: sendAlert().
- Role: Generates notifications for low stock, expired medicines, or payment failures.

Report:

- Attributes: reportId (string), reportType (string), generatedDate (Date), content (string).
- Methods: generateReport().
- Role: Creates structured reports (e.g., weekly sales, inventory summaries) for analysis.

Key Relationships:

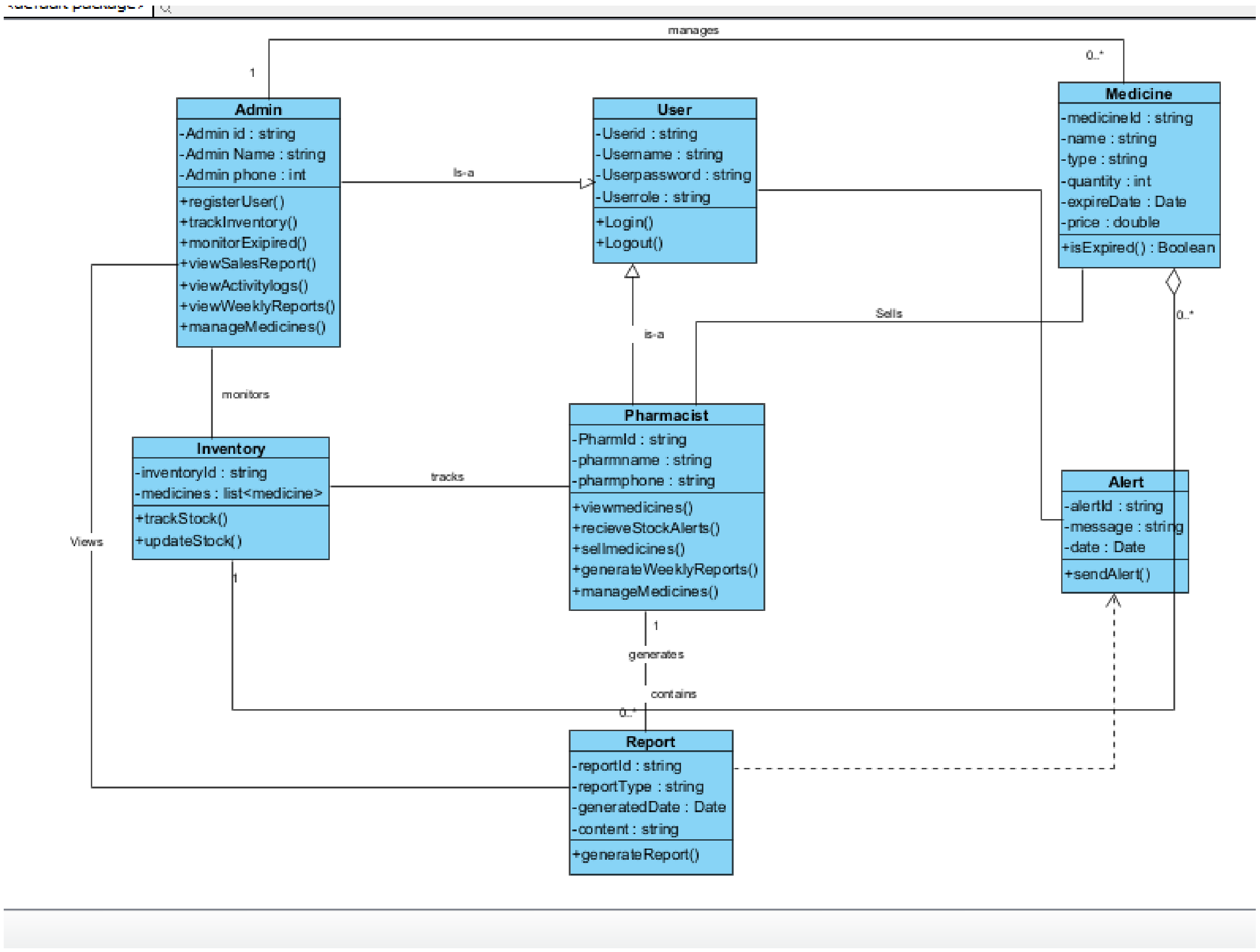
- Admin and Pharmacist inherit from User, enabling role-based access control.
- Inventory contains a list of Medicine objects, updated via trackStock() and updateStock().
- Pharmacist interacts with Report to generate weekly reports and with Alert to receive stock notifications.
- Medicine's Expired() method triggers Alert when expiry dates are near.
- Admin uses trackInventory() and monitorExpired() to manage Inventory and Alert systems.

Key Relationships:

- **Admin** and **Pharmacist** inherit from **User**, enabling role-based access.
- **Inventory** contains a list of **Medicine** objects, updated via sales or restocking.
- **Alert** is triggered by **Inventory** checks (e.g., expiry or low stock) or payment system failures.
- **Pharmacist** interacts with **Report** to generate and view summaries.

3.3 Example of Class Diagram

The following class diagram, created using Visual Paradigm, represents the detailed object-oriented structure of the pharmacy management systems



3.4 Tools and steps to draw Class Diagram

Step 1: Open Visual Paradigm

- Launch Visual Paradigm → Create a **New Project** → Select **UML Diagram** → **Class Diagram**.

Step 2: Add Classes

1. Drag the **Class** icon from the toolbar onto the canvas.
2. Name each class (e.g., user, Admin, pharmacist ,medicine, Inventory, Alert, Report)

Step 3: Define Attributes and Methods

- Double-click a class to add:
  - **Attributes** (e.g. MedicineId: string, quantity :int)
  - **Methods** (e.g., Expired ( ): Boolean, Updatestock( )
- Example for the Medicine class:

Step 4: Add Relationships

Use the **Relationship Toolbar** to connect classes:

1. **Inheritance (Generalization)**:
   - Example: admin and pharmacist inherit from user
   - Use a hollow arrow (▷) pointing to the parent class.
2. **Composition/Aggregation**:
   - Example Inventory "contains" a list of medicine objects.
   - Use a diamond arrow (◆) for composition.
3. **Associations**:
   - Example pharmacists: interacts with report.
   - Use a simple line with an optional label (e.g., "generates").

Step 5: Add Notes or Constraints (Optional)

- Use the **Note Tool** to clarify complex relationships.
  - Example: Add a note to Alert class: *"Triggers when stock < 10 or medicine expires."*

Step 6: Validate and Refine

- Ensure all attributes/methods align with your system's requirements.
- Use **Auto-Layout** to organize classes neatly.

Step 7: Export/Share

- Save the diagram → Export as PNG/PDF via **File > Export**.

Chapter 4

4.1 Implementation Overview

The implementation phase transforms the **Pharmacy Management System's design specifications** into functional code using **Java** within the **Eclipse IDE.** This stage focuses on developing the core classes identified in the class diagram (e.g., User, Admin, Pharmacist, Medicine, Inventory, Alert, and Report), ensuring alignment with the system's requirements for medicine tracking, sales processing, and inventory management. The process involves iterative coding, rigorous testing, and deployment, with **Visual Paradigm** used to generate initial code skeletons from UML diagrams.

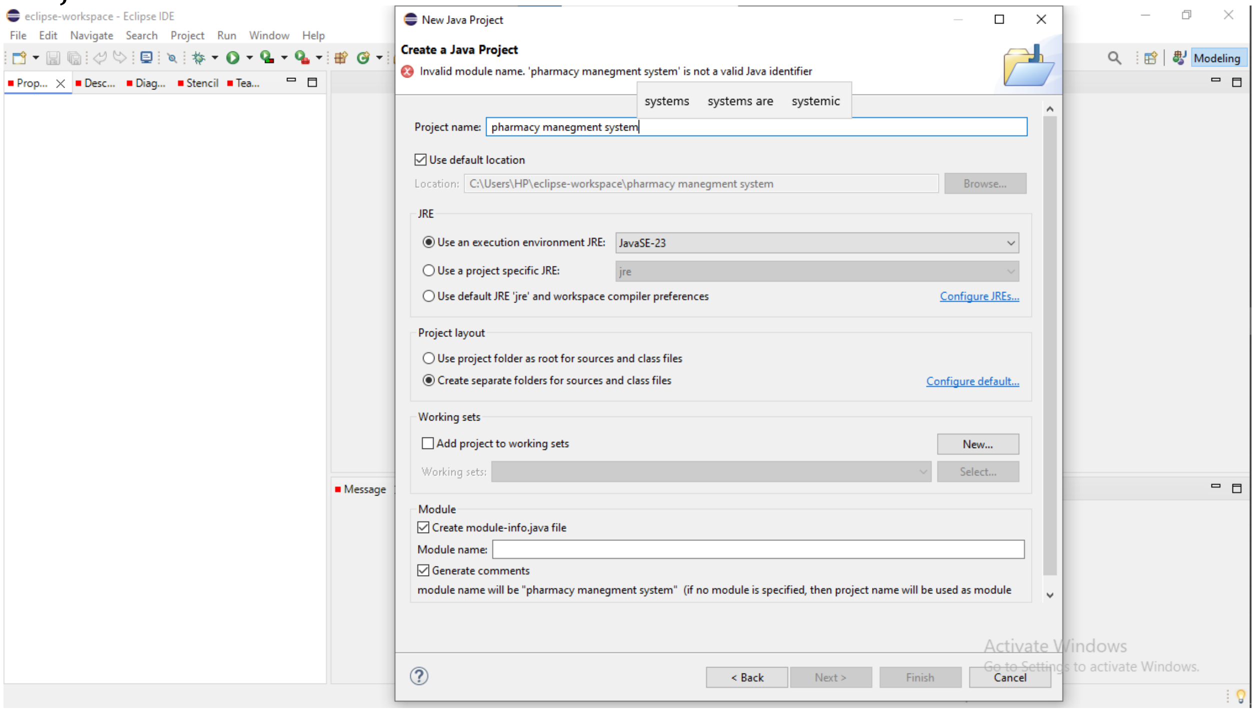
4.2 Sample Class Implementations

Generate Java Code from UML Class

1 Creating a Java Project

2 Start Eclipse.

3 Select File > New > Java Project from the main menu to open the New Java Project window.


eclipse-workspace - Eclipse IDE
File Edit Navigate Search Project Run Window Help
Prop... Desc... Diag... Stencil Tea...
Message
Modeling
Activate Windows
Go to Settings to activate Windows.

New Java Project

Create a Java Project

Invalid module name. 'pharmacy manegment system' is not a valid Java identifier

systems | systems are | systemic

Project name: pharmacy manegment system

☑ Use default location

Location: C:\Users\HP\eclipse-workspace\pharmacy manegment system Browse...

JRE

● Use an execution environment JRE: JavaSE-23

○ Use a project specific JRE: jre

○ Use default JRE 'jre' and workspace compiler preferences Configure JREs...

Project layout

○ Use project folder as root for sources and class files

● Create separate folders for sources and class files Configure default...

Working sets

☐ Add project to working sets New...

Working sets: Select...

Module

☑ Create module-info.java file

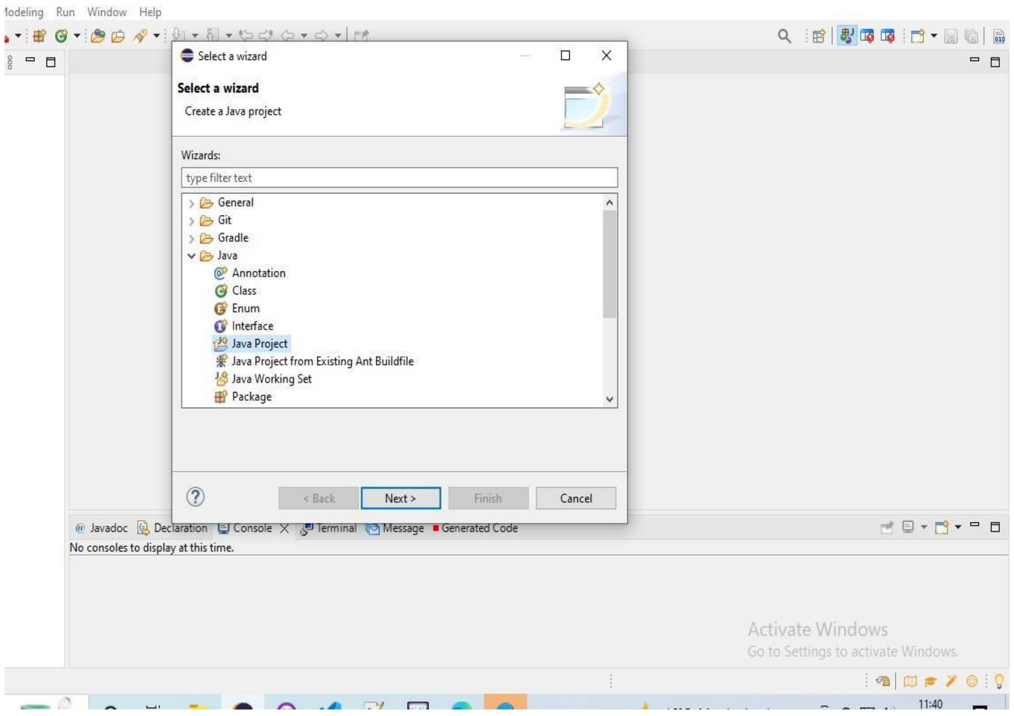
Module name:

☑ Generate comments

module name will be "pharmacy manegment system" (if no module is specified, then project name will be used as module

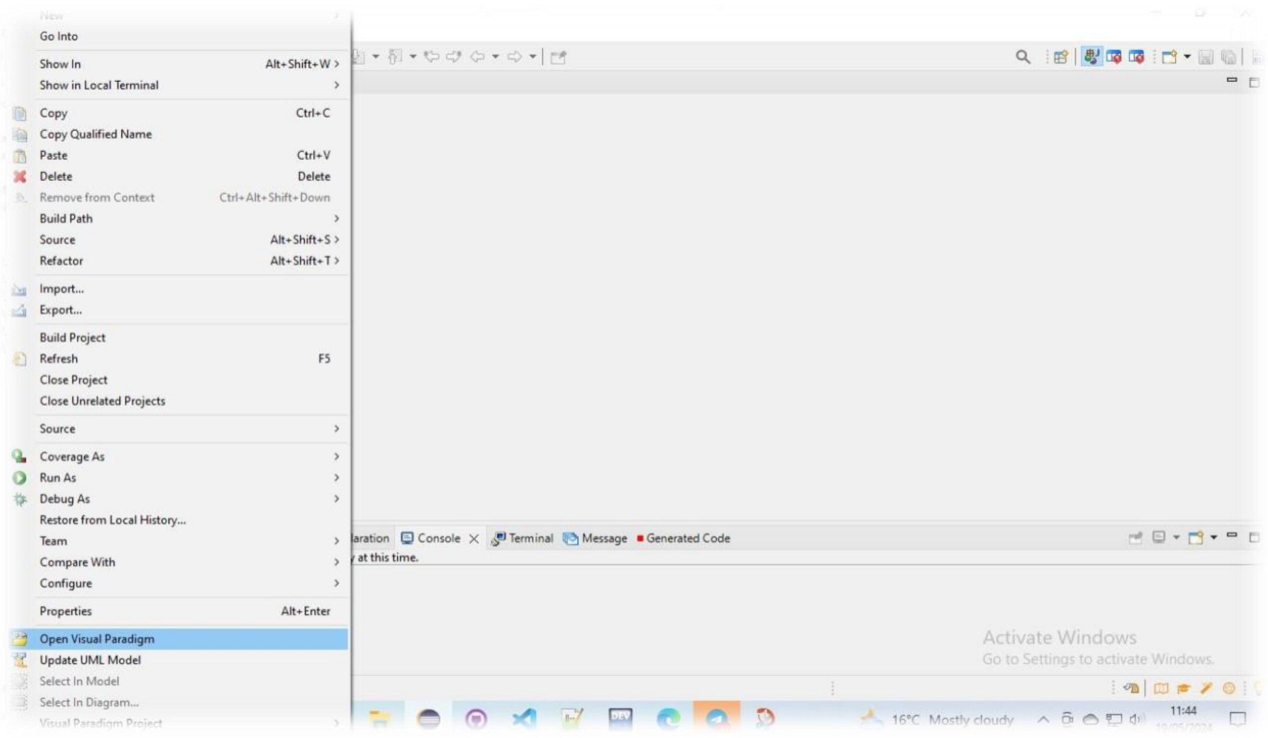
< Back Next > Finish Cancel

4 Clicks FINSH



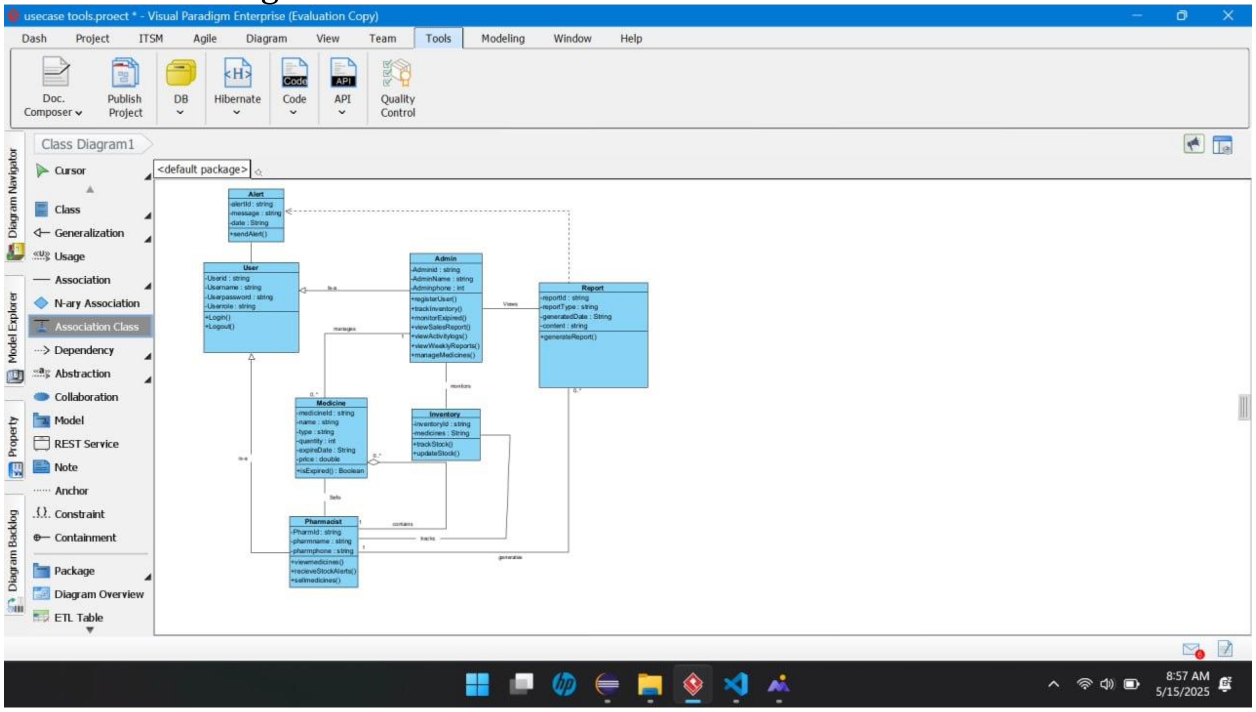
No consoles to display at this time.

Right click on the project node in Package Explorer and select Open Visual Paradigm from the popup menu.

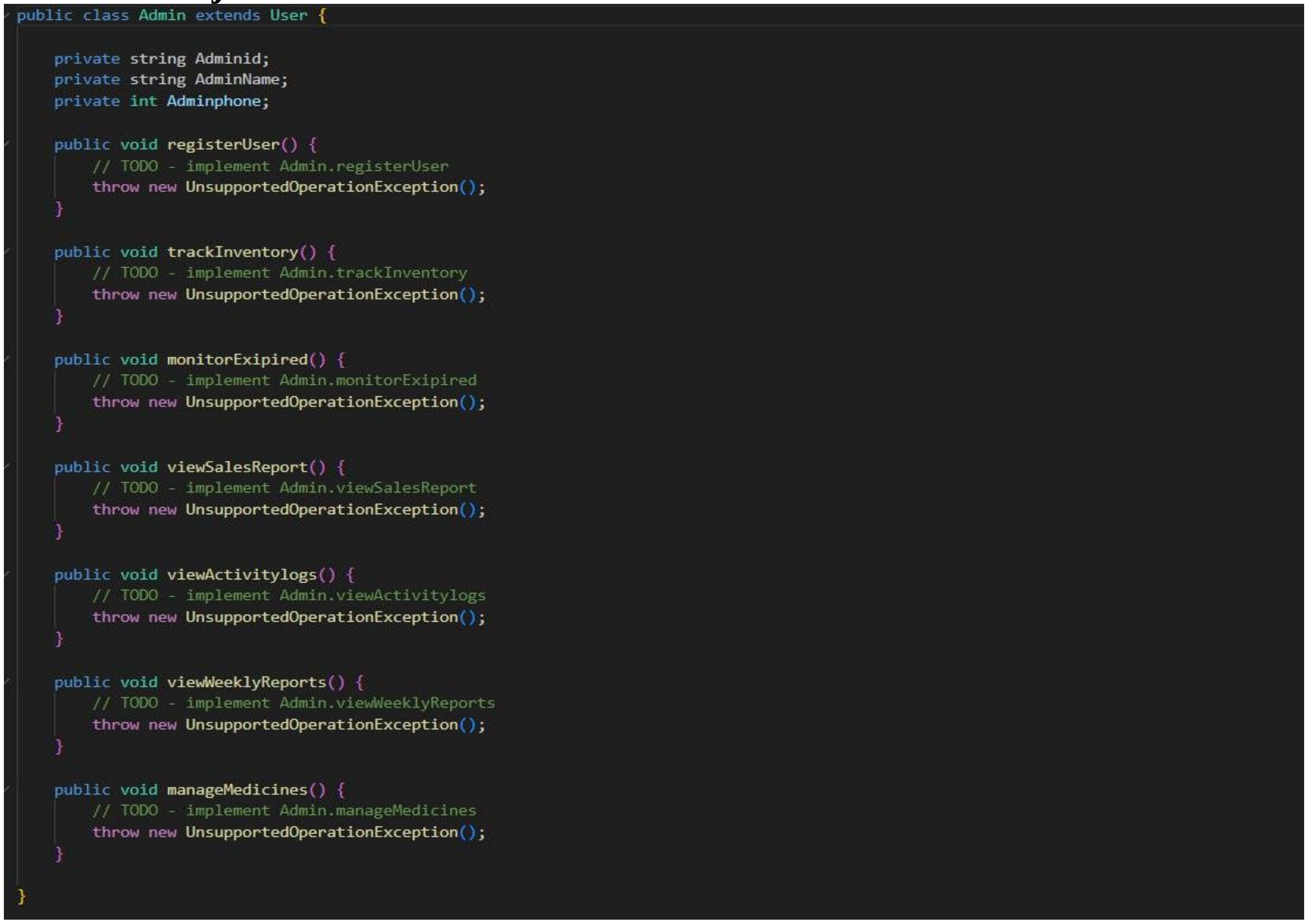


Generate Java Code from UML Class from the class diagram

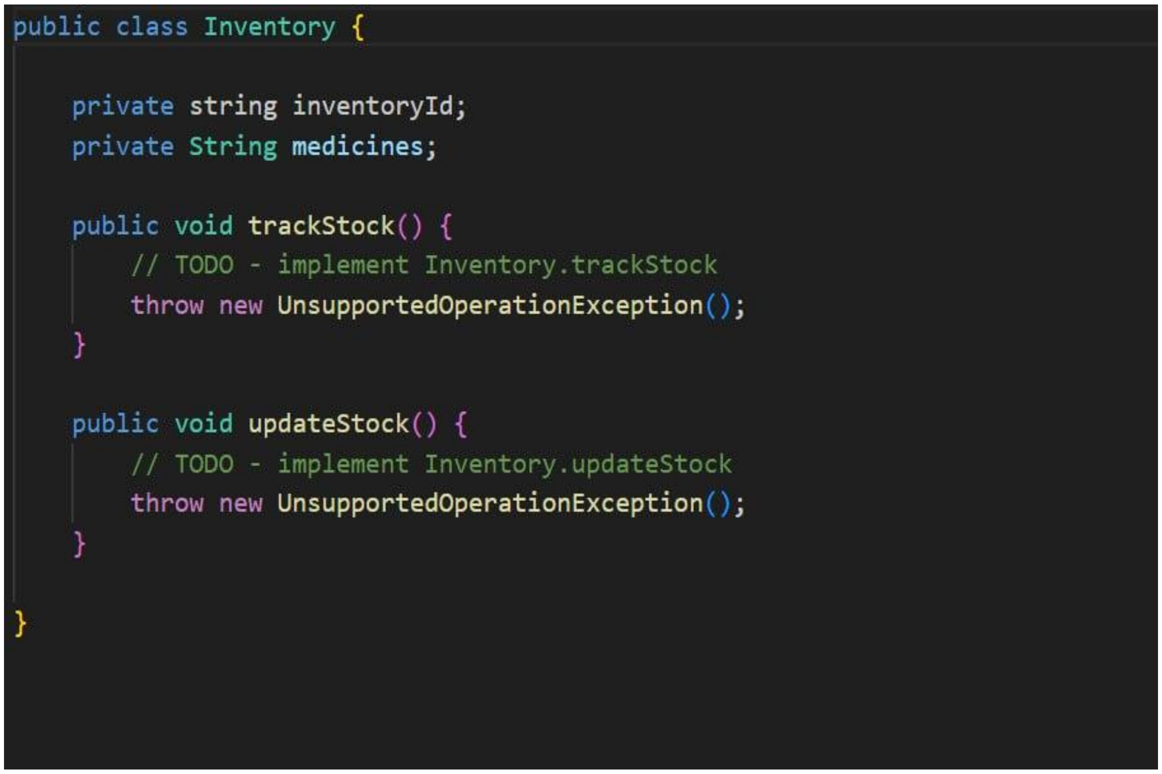
Draw a class diagram



Admin.java



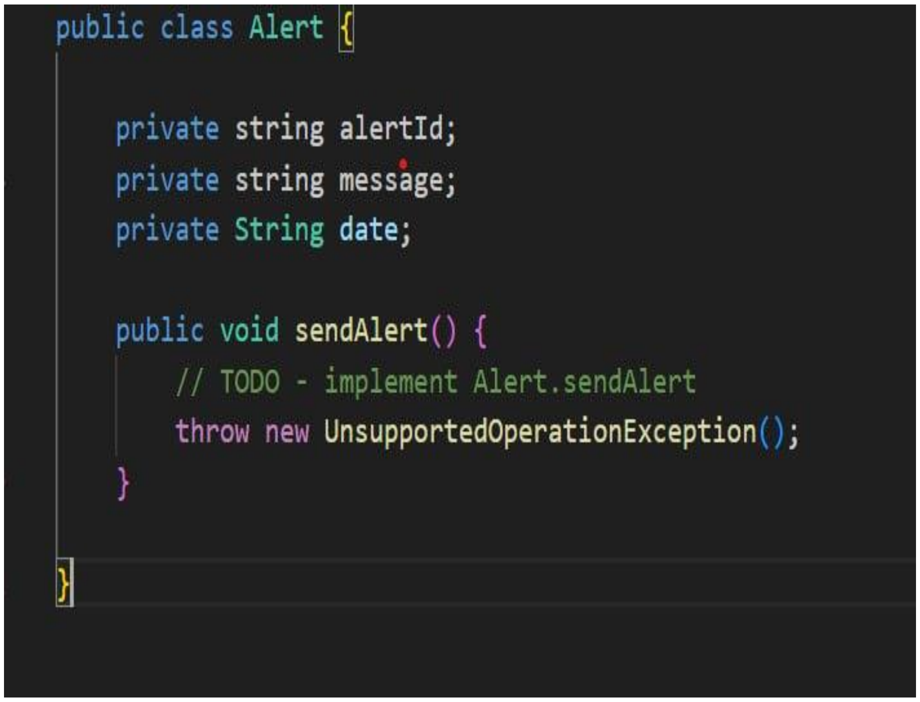
```
public class Admin extends User {

    private string Adminid;
    private string AdminName;
    private int Adminphone;

    public void registerUser() {
        // TODO - implement Admin.registerUser
        throw new UnsupportedOperationException();
    }

    public void trackInventory() {
        // TODO - implement Admin.trackInventory
        throw new UnsupportedOperationException();
    }

    public void monitorExipired() {
        // TODO - implement Admin.monitorExipired
        throw new UnsupportedOperationException();
    }

    public void viewSalesReport() {
        // TODO - implement Admin.viewSalesReport
        throw new UnsupportedOperationException();
    }

    public void viewActivitylogs() {
        // TODO - implement Admin.viewActivitylogs
        throw new UnsupportedOperationException();
    }

    public void viewWeeklyReports() {
        // TODO - implement Admin.viewWeeklyReports
        throw new UnsupportedOperationException();
    }

    public void manageMedicines() {
        // TODO - implement Admin.manageMedicines
        throw new UnsupportedOperationException();
    }

}
```



```
public class Inventory {

    private string inventoryId;
    private String medicines;

    public void trackStock() {
        // TODO - implement Inventory.trackStock
        throw new UnsupportedOperationException();
    }

    public void updateStock() {
        // TODO - implement Inventory.updateStock
        throw new UnsupportedOperationException();
    }

}
```

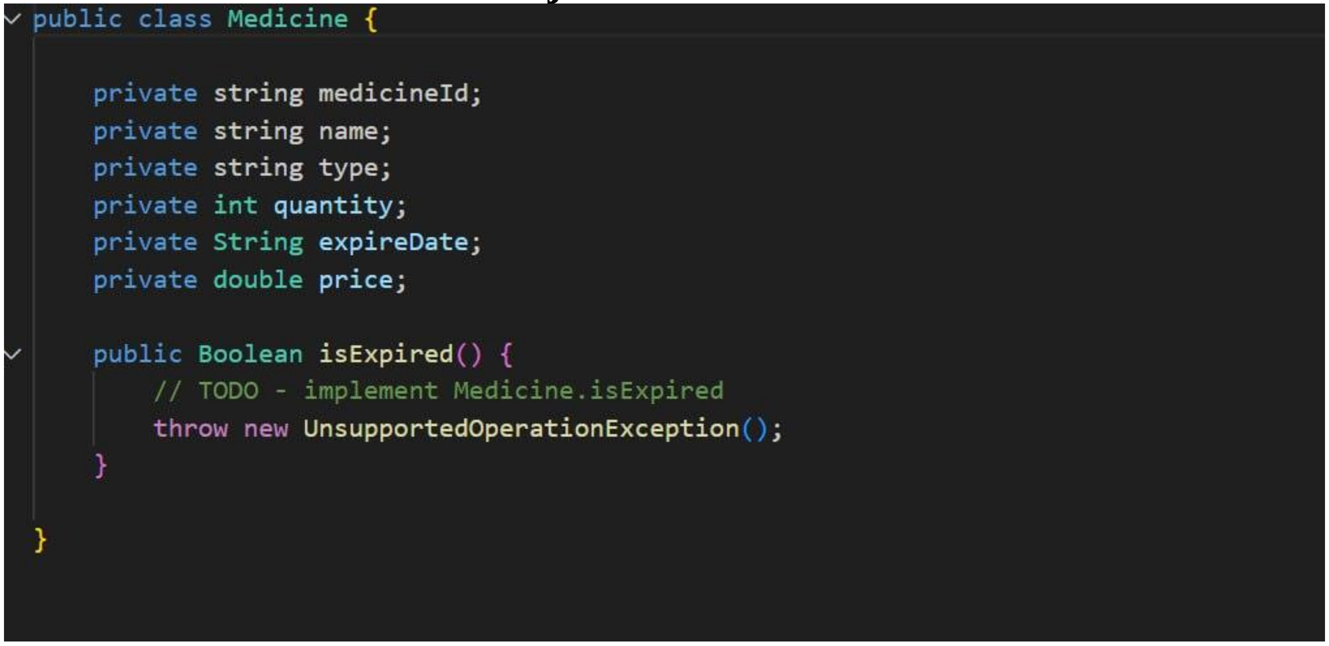
Inventory. java

Alert.java



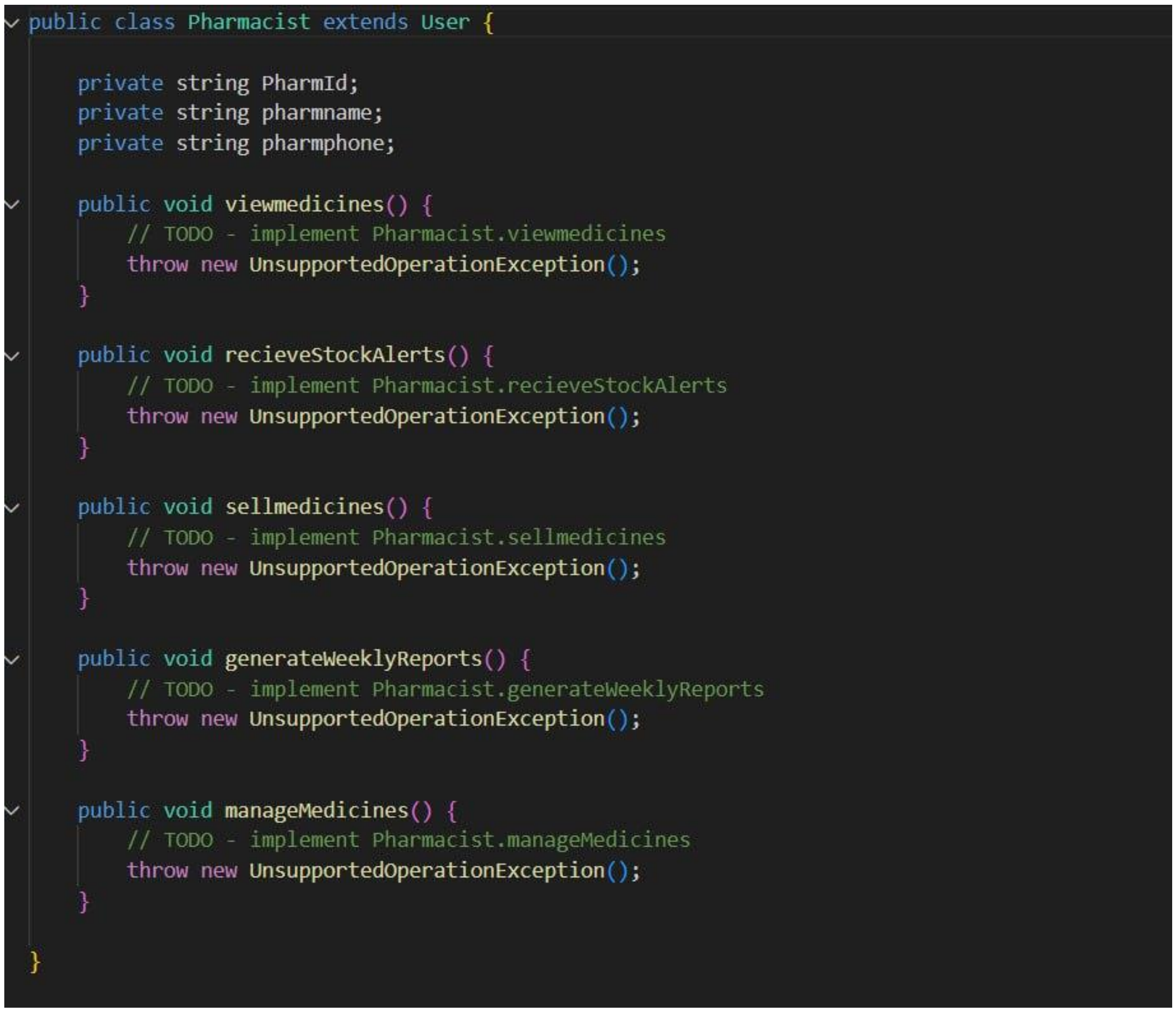
```
public class Alert {

    private string alertId;
    private string message;
    private String date;

    public void sendAlert() {
        // TODO - implement Alert.sendAlert
        throw new UnsupportedOperationException();
    }

}
```

Medicin.java



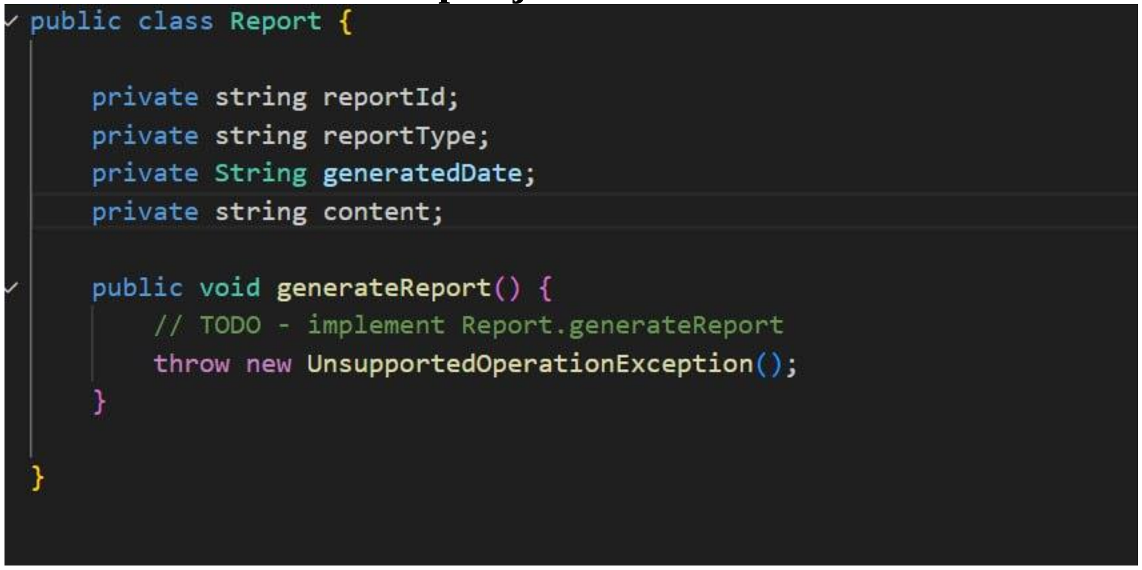
```
public class Medicine {

    private string medicineId;
    private string name;
    private string type;
    private int quantity;
    private String expireDate;
    private double price;

    public Boolean isExpired() {
        // TODO - implement Medicine.isExpired
        throw new UnsupportedOperationException();
    }

}
```

Pharmacist.java



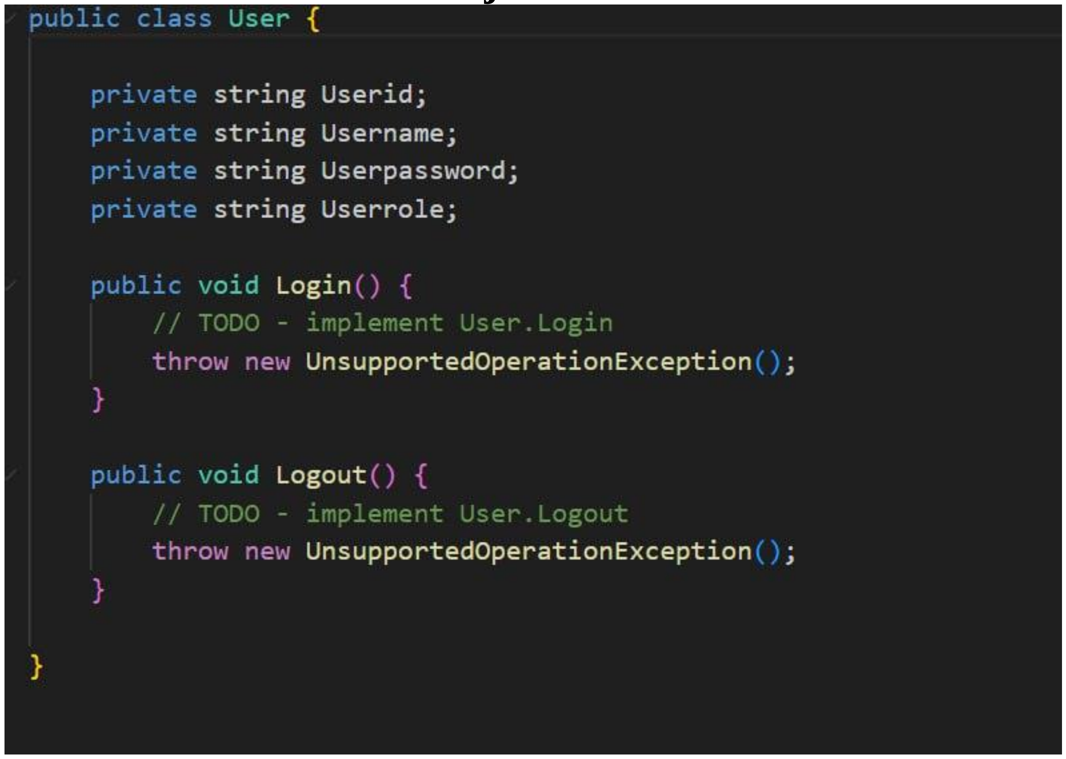
```java
public class Pharmacist extends User {

    private string PharmId;
    private string pharmname;
    private string pharmphone;

    public void viewmedicines() {
        // TODO - implement Pharmacist.viewmedicines
        throw new UnsupportedOperationException();
    }

    public void recieveStockAlerts() {
        // TODO - implement Pharmacist.recieveStockAlerts
        throw new UnsupportedOperationException();
    }

    public void sellmedicines() {
        // TODO - implement Pharmacist.sellmedicines
        throw new UnsupportedOperationException();
    }

    public void generateWeeklyReports() {
        // TODO - implement Pharmacist.generateWeeklyReports
        throw new UnsupportedOperationException();
    }

    public void manageMedicines() {
        // TODO - implement Pharmacist.manageMedicines
        throw new UnsupportedOperationException();
    }

}
```

Repor.java



```
public class Report {

    private string reportId;
    private string reportType;
    private String generatedDate;
    private string content;

    public void generateReport() {
        // TODO - implement Report.generateReport
        throw new UnsupportedOperationException();
    }

}
```

User.java



```
public class User {

    private string Userid;
    private string Username;
    private string Userpassword;
    private string Userrole;

    public void Login() {
        // TODO - implement User.Login
        throw new UnsupportedOperationException();
    }

    public void Logout() {
        // TODO - implement User.Logout
        throw new UnsupportedOperationException();
    }

}
```

CHAPTER FIVE

5.1 Change Management (version control using Git)

Version control is essential for managing changes to a project over time
Git is a popular distributed version control system that allows team collaboration and tracking changes
Git provides features like branching, merging, and conflict resolution to facilitate efficient code management

5.2 Steps and Tools Used to Implement Git

To effectively manage version control and collaboration during the development of our Pharmacy Management System, we used **Git** along with **GitHub**. Below are the enhanced steps and tools applied in our project:

1. Install Git

- **Tool Used**: Git
- Downloaded and installed Git on all development machines to enable version control from the command line.

2. Initialize a Git Repository

- **Step 1**: Created a new directory for the project:

```
Mkdir pharmacy-management
Cd pharmacy-management
```

- **Step 2**: Initialized Git in the project directory:

```
Git init
```

- **Purpose**: This sets up the .git folder where Git stores all version control data.

3. Configure Git Settings (Optional but Recommended)

- Configured user information to associate commits:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

- Checked configuration:

```
git config --list
```

4. Add Project Files to the Repository

- Added existing files to the Git repository:

```
git add .
```

or to add specific files:

```
git add index.html style.css script.js
```

5. Commit Changes

- Committed the staged files with a meaningful message:

```
git commit -m "Initial commit: Added main structure of Pharmacy Management System"
```

6. Connect to a Remote Repository (GitHub)

- **Tool Used**: GitHub (https://github.com/)
- Created a new repository on GitHub.
- Linked the local repo to GitHub:

```
git remote add origin https://github.com/username/pharmacy-management.git
git push -u origin master
```

7. Ongoing Version Control

- After each change, we used:

```
git add .
git commit -m "Describe the change"
git push
```

- Used `git status` to check current status and `git log` to view commit history.

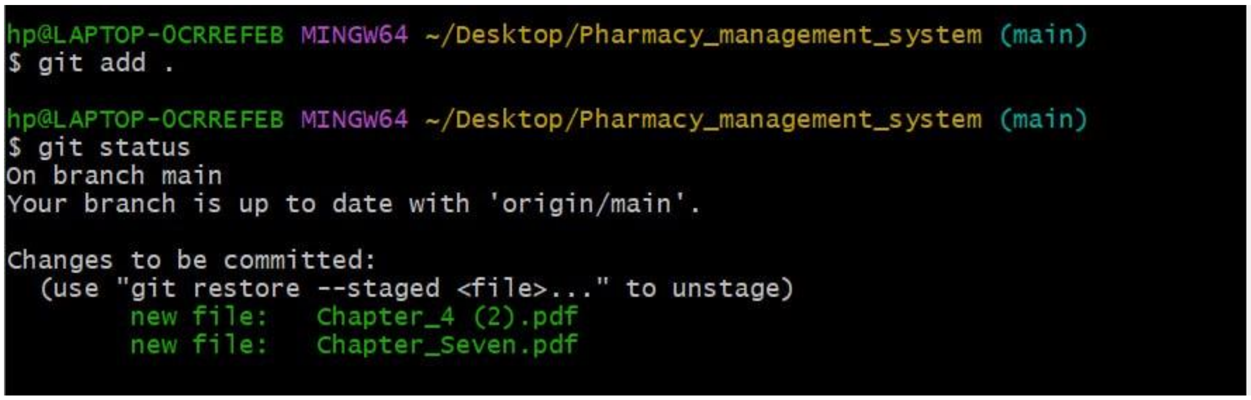
8. Collaboration and Branching

- Created feature branches for team members:

  git checkout -b feature-branch-name

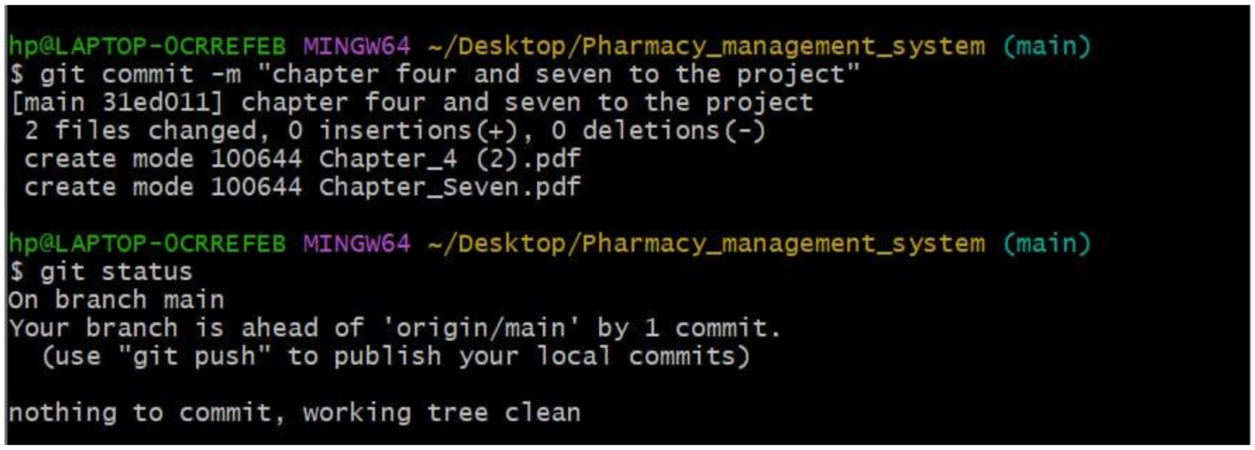
- Merged features into main branch after review:
- git merge feature-branch-name

  Summary
  Using Git and GitHub allowed us to track changes, collaborate effectively, revert to earlier versions, and manage our project in a clean and organized way.



```
hp@LAPTOP-OCRREFEB MINGW64 ~/Desktop/Pharmacy_management_system (main)
$ git add .

hp@LAPTOP-OCRREFEB MINGW64 ~/Desktop/Pharmacy_management_system (main)
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   Chapter_4 (2).pdf
        new file:   Chapter_Seven.pdf
```

Git add



```
hp@LAPTOP-OCRREFEB MINGW64 ~/Desktop/Pharmacy_management_system (main)
$ git commit -m "chapter four and seven to the project"
[main 31ed011] chapter four and seven to the project
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Chapter_4 (2).pdf
 create mode 100644 Chapter_Seven.pdf

hp@LAPTOP-OCRREFEB MINGW64 ~/Desktop/Pharmacy_management_system (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Git commit

Chapter six

6.1 Unit Test

Unit Testing: A software testing method where individual code units (e.g., functions, methods, classes) are validated to ensure they behave as intended.

Key Principles (Short Form):

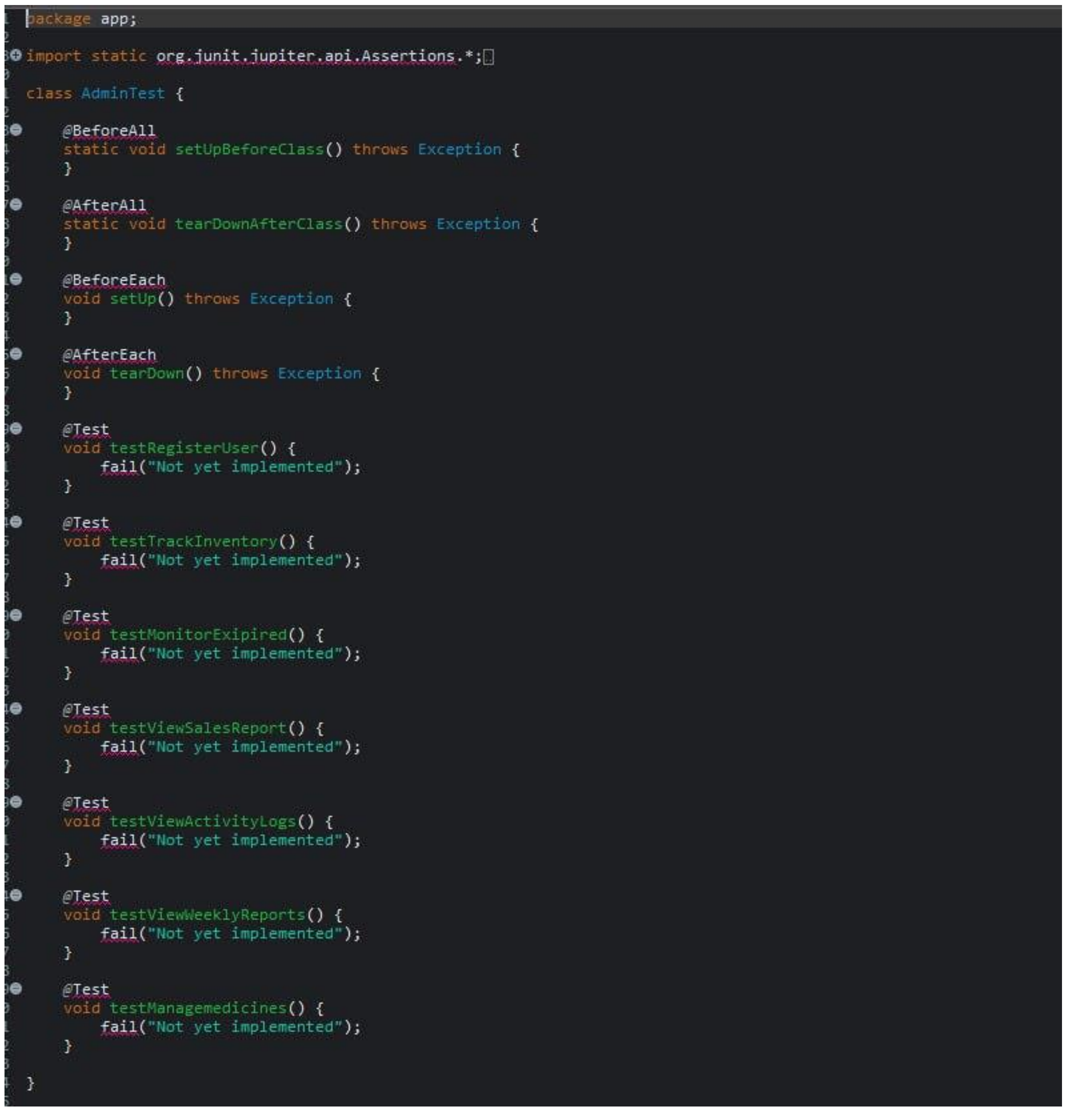
1. **Isolation**: Test one unit at a time; mock external dependencies (e.g., databases).
2. **Automation**: Tests should run automatically without manual intervention.
3. **Fast**: Execute quickly to enable frequent runs during development.
4. **Repeatable**: Produce the same result regardless of environment or execution order.
5. **Self-Validating**: Tests pass/fail without human judgment (e.g., assertions).
6. **FIRST Principle**:
   - **F**ast
   - **I**ndependent (no test depends on another)
   - **R**epeatable
   - **S**elf-Validating
   - **T**imely (written alongside code, not after).
7. **Readability**: Clear names and structure to explain what is being tested.
8. **Coverage**: Aim to test all logical paths (branches, edge cases).

Goal: Catch bugs early, improve code quality, and simplify refactoring.

6.2 Steps and Tools Used in Unit Test

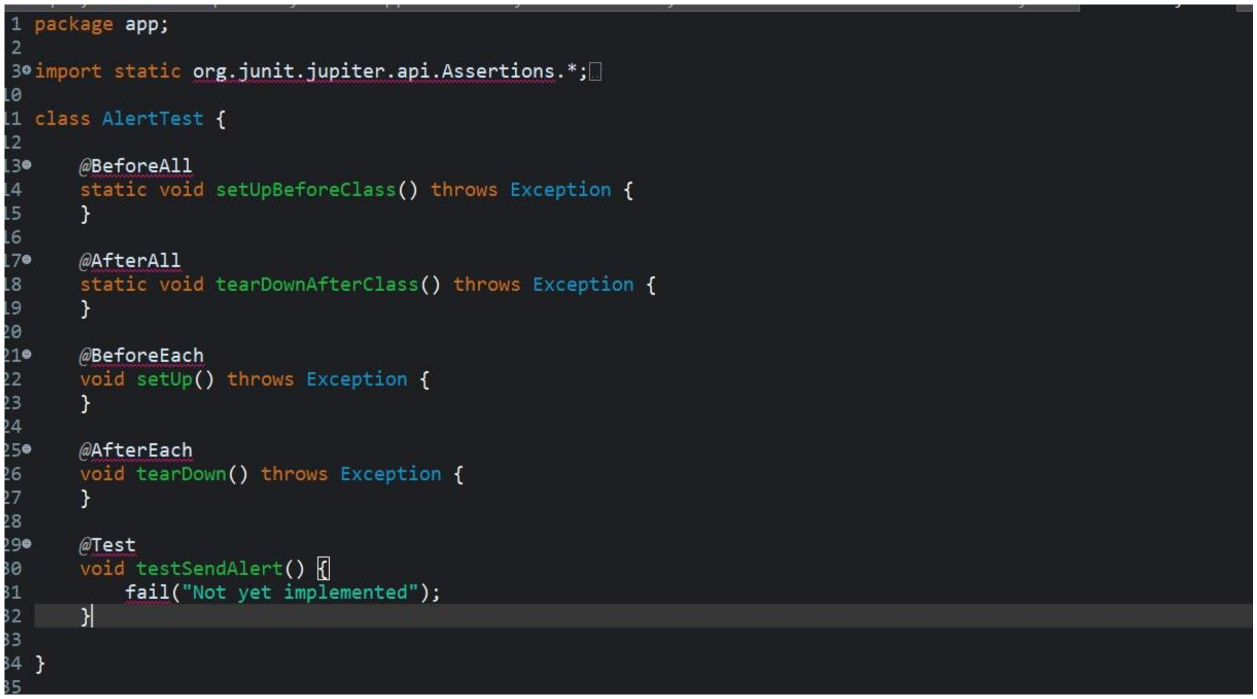
This project utilizes **JUnit**, a widely adopted testing framework for Java. JUnit provides a set of annotations and assertions that enable developers to create and validate test cases effectively, ensuring code correctness and reliability throughout the development process. Additionally, the project is developed using the **Eclipse IDE**, which offers robust features for code writing, debugging, and seamless integration with JUnit.

Admin JU test



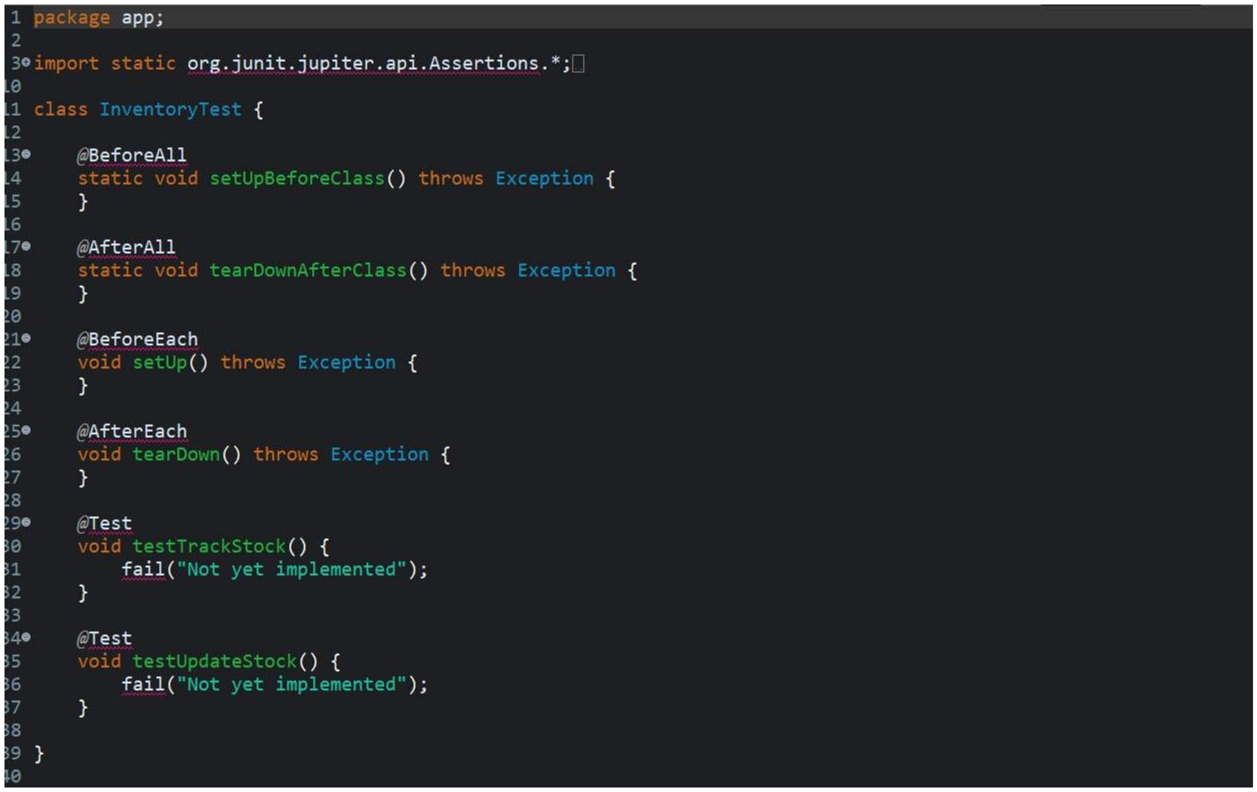
```
package app;

import static org.junit.jupiter.api.Assertions.*;

class AdminTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testRegisterUser() {
        fail("Not yet implemented");
    }

    @Test
    void testTrackInventory() {
        fail("Not yet implemented");
    }

    @Test
    void testMonitorExipired() {
        fail("Not yet implemented");
    }

    @Test
    void testViewSalesReport() {
        fail("Not yet implemented");
    }

    @Test
    void testViewActivityLogs() {
        fail("Not yet implemented");
    }

    @Test
    void testViewWeeklyReports() {
        fail("Not yet implemented");
    }

    @Test
    void testManagemedicines() {
        fail("Not yet implemented");
    }

}
```

Alert JU test



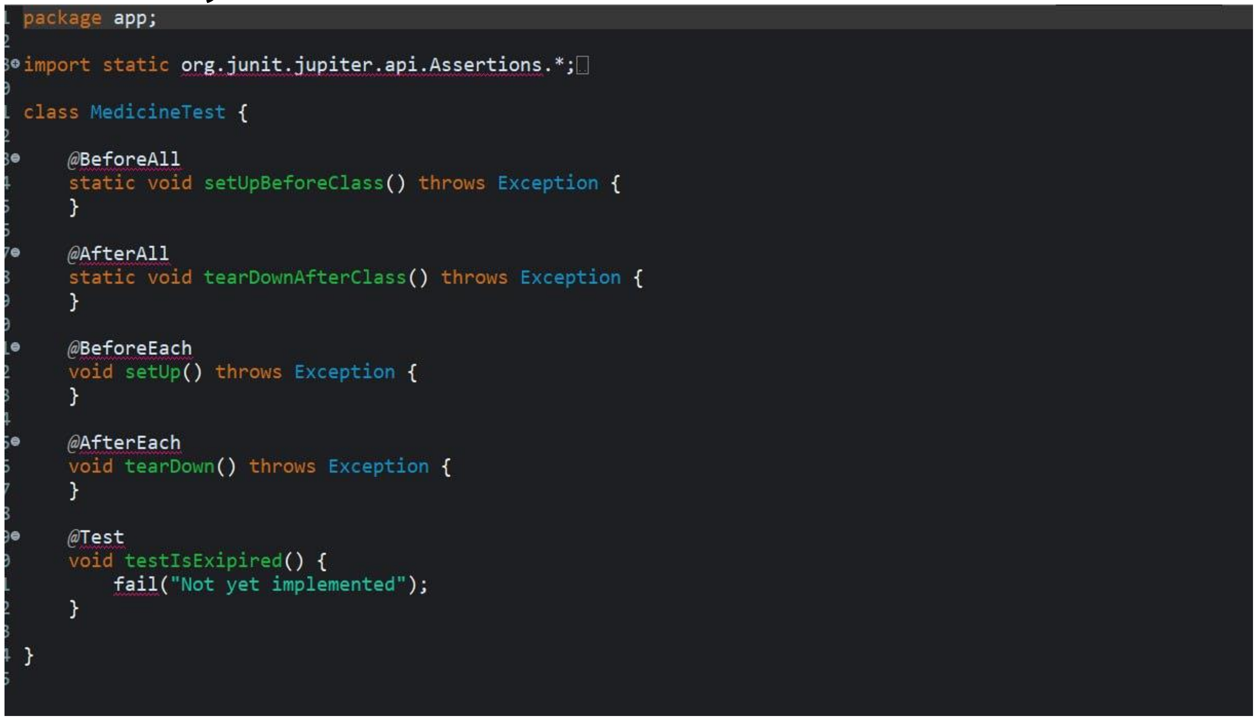
```
package app;

import static org.junit.jupiter.api.Assertions.*;

class AlertTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testSendAlert() {
        fail("Not yet implemented");
    }

}
```

Inventory JU test



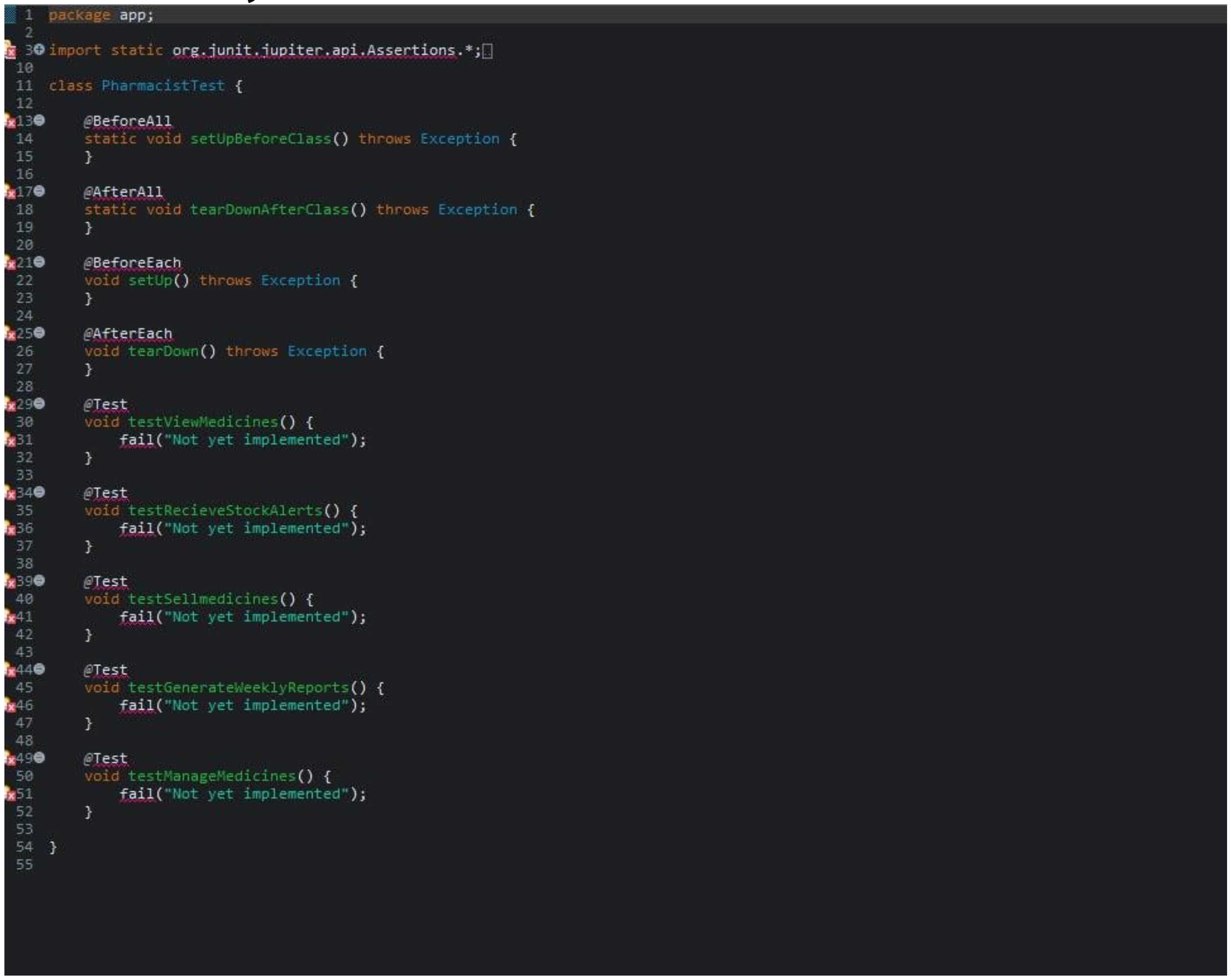
```
package app;

import static org.junit.jupiter.api.Assertions.*;

class InventoryTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testTrackStock() {
        fail("Not yet implemented");
    }

    @Test
    void testUpdateStock() {
        fail("Not yet implemented");
    }

}
```

Medicine JU test



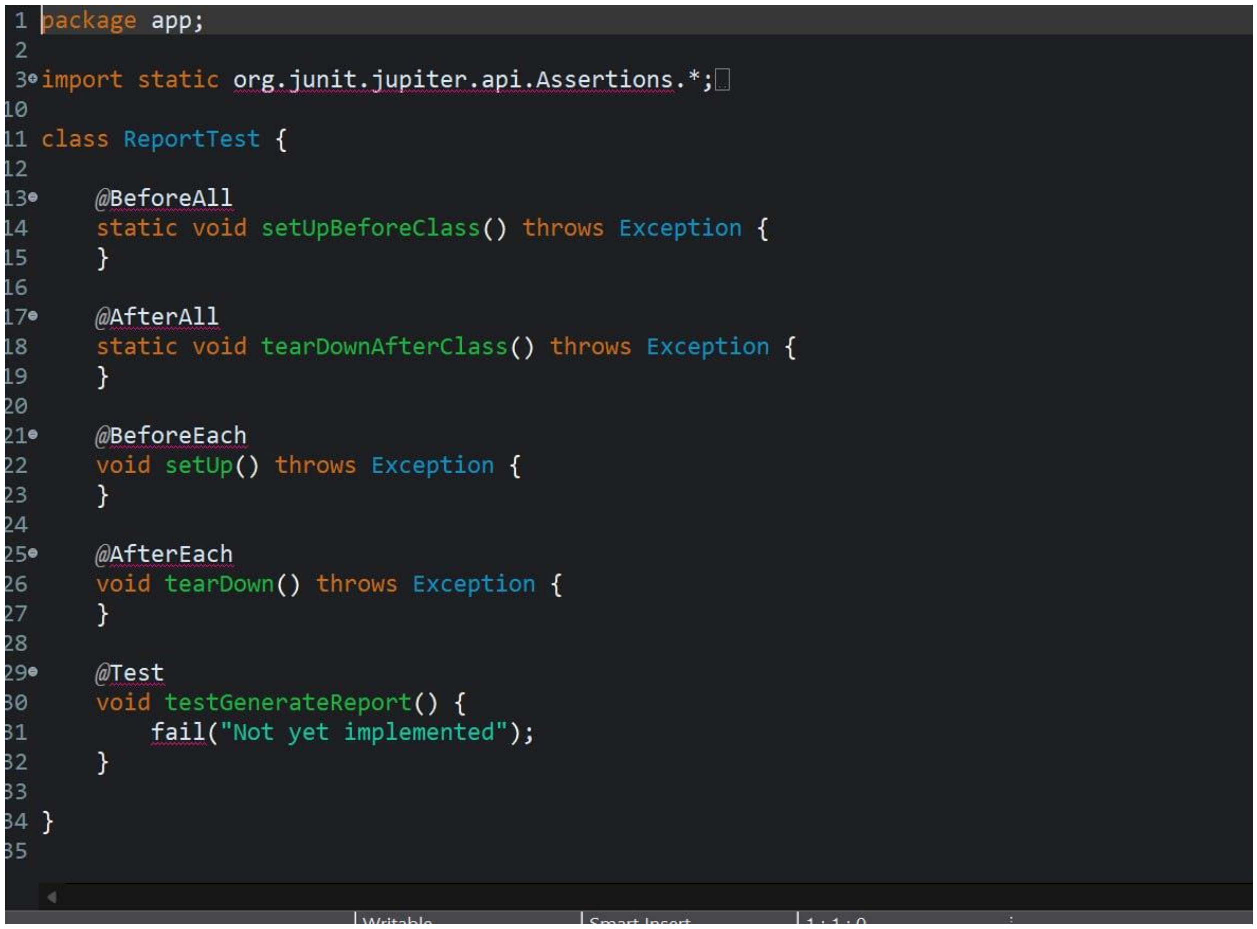
```
package app;

import static org.junit.jupiter.api.Assertions.*;

class MedicineTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testIsExipired() {
        fail("Not yet implemented");
    }

}
```

Pharmacist JU test



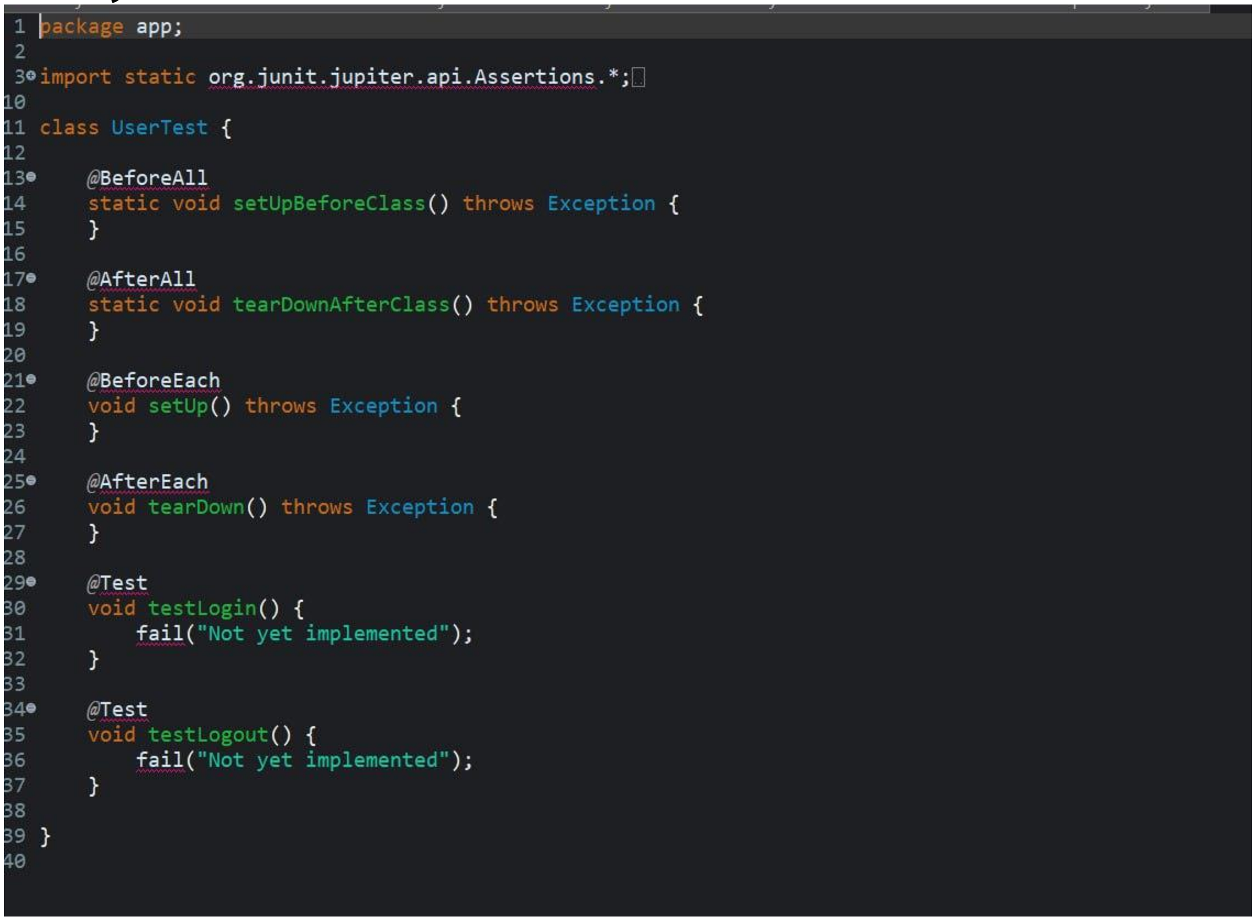
```
package app;

import static org.junit.jupiter.api.Assertions.*;

class PharmacistTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testViewMedicines() {
        fail("Not yet implemented");
    }

    @Test
    void testRecieveStockAlerts() {
        fail("Not yet implemented");
    }

    @Test
    void testSellmedicines() {
        fail("Not yet implemented");
    }

    @Test
    void testGenerateWeeklyReports() {
        fail("Not yet implemented");
    }

    @Test
    void testManageMedicines() {
        fail("Not yet implemented");
    }

}
```

Report JU test



```java
package app;

import static org.junit.jupiter.api.Assertions.*;

class ReportTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testGenerateReport() {
        fail("Not yet implemented");
    }

}
```

User JU Test



```
package app;

import static org.junit.jupiter.api.Assertions.*;

class UserTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
    }

    @BeforeEach
    void setUp() throws Exception {
    }

    @AfterEach
    void tearDown() throws Exception {
    }

    @Test
    void testLogin() {
        fail("Not yet implemented");
    }

    @Test
    void testLogout() {
        fail("Not yet implemented");
    }

}
```

Chapter Seven

7.1 Build Process Overview

The build process turns your Java code into a working application. It checks for errors, runs tests, and creates a file you can run or deploy. This phase compiles the Java source code into executable bytecode, executes unit tests to validate functionality, and generates the application output. It ensures the system is ready for deployment and use, providing a final verification step before release.

7.2 Tools & Simple Steps

- **Eclipse IDE** (easy built-in tools for compiling and testing).
- **Command Line (CLI)** (for manual builds if needed).

Steps:

1. **Compile the Code**:
   - In Eclipse: Right-click project → **Build Project** (converts code to runnable format).
   - CLI: `javac -d ./target src/main/java/com/pharmacy/*.java`
2. **Run Tests**:
   - In Eclipse: Right-click → **Run As → JUnit Test** (checks if everything works).
   - CLI: `mvn test` (if using Maven).
3. **Create Executable File**:
   - In Eclipse: Export → **Runnable JAR** → Pick the main class (creates a `.jar` file).
4. **Run the Application**:
   - Double-click the JAR file or use CLI: `java -jar pharmacy-system.jar`.
5. **Check for Errors**:
   - Look at the console/GUI for issues (fix if needed).

Why It Matters:

- Ensures your app works before users see it.
- Catches bugs early.

- ## Appendix
- ● **File is available in Repository**: Available on GitHub (optional link or clone instructions).
- ● **Screenshots**: Includes all class diagrams, sequence diagrams, code snippets, JUnit test results, Git activity, and build outputs for comprehensive documentation