
UTM
UNIVERSITI TEKNOLOGI MALAYSIA

Faculty of Computing

**PROJECT REPORT**

**Subject:** SECJ2154-01 PENGATURCARAAN BERORENTASIKAN OBJEK (OBJECT ORIENTED PROGRAMMING)

**Session:** 2024/2025 Semester 2

| | |
|---|---|
| **Group Name** | WEBUG |
| **Group Members** | 1. MUHAMMAD AMIRUN IRFAN BIN SAMSUL SHAH (A23CS0121)<br><br>2. MUHAMMAD HAFIZ BIN MOHD SHAHARUDDIN (A23CS0130)<br><br>3. RAVINESH A/L MARAN (A23CS0175)<br><br>4. WELSON WOONG LU BIN (A23CS0196) |
| **Section** | 01 |
| **Lecturer's Name** | DR. NORSHAM BINTI IDRIS |

# Table of Content


**1.0 Introduction** **3**
**2.0 Use Case Diagram** **4**
2.1 Use Case Diagram Explanation 5
**3.0 Class Diagram** **6**
3.1 Class Diagram Explanation 6
3.1.1 Patient 6
3.1.2 Appointment 7
3.1.3 Medicine (Abstract Class) 7
3.1.4 Prescription 7
3.1.5 MedicationSchedule 7
3.1.6 PharmacyInventory 8
3.1.7 Dispensable (Interface) 8
**4.0 Complete Java Program** **9**
**5.0 Task Allocations** **21**
**6.0 Applied Object-Oriented Concepts Explanations** **22**
6.1 Encapsulation 22
6.2 Inheritance 23
6.3 Polymorphism 24
6.4 Abstraction 24
6.5 Association, Aggregation, and Composition 25
6.6 Interface and Implementation 27
6.7 Exception Handling 27
**7.0 Conclusion** **28**

# 1.0 Introduction

The Efficient Medicine System is designed to streamline and enhance the accuracy of medicine distribution within healthcare settings, especially in hospitals. Currently, many patients experience delays and inefficiencies due to separate processes for medical consultation and prescription dispensing. This system addresses these issues by allowing doctors to directly record and manage patient medication through a centralized system. It helps minimize prescription errors, speeds up the dispensing process, and ensures that each patient receives the correct dosage and instructions — whether the medicine requires a prescription or not. Additionally, it keeps track of each patient's medication history and schedules, making it easier to manage chronic conditions such as diabetes or hypertension. By improving accessibility and efficiency, this system supports Sustainable Development Goal 3 (SDG 3), which aims to ensure healthy lives and promote well-being for all. It empowers healthcare professionals and patients alike by reducing waiting times, enhancing prescription accuracy, and improving overall patient care.

# 2.0 Use Case Diagram

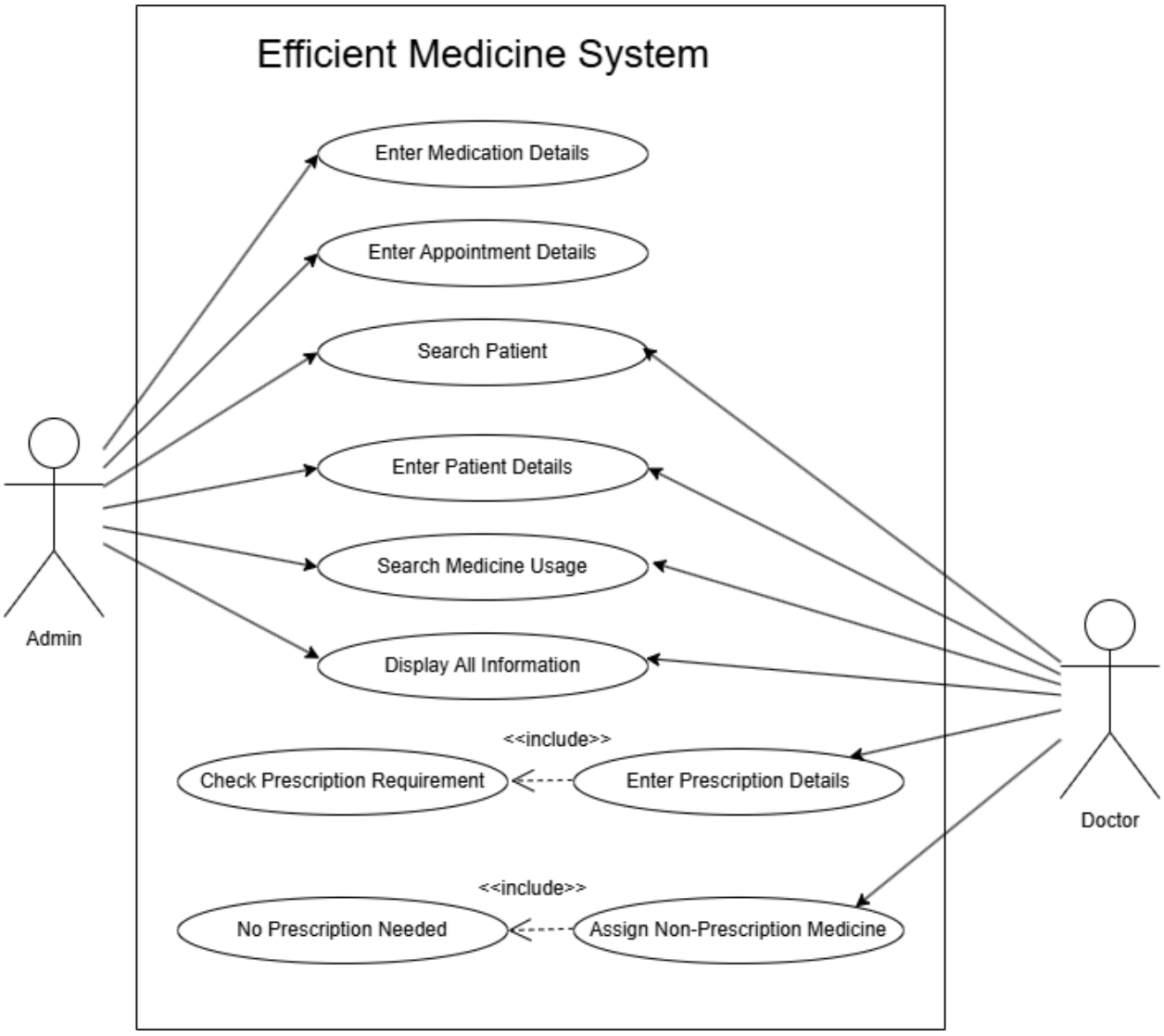


*Figure 1.0: Use Case Diagram of Efficient Medicine System*

## 2.1 Use Case Diagram Explanation

The system involves the following actors:

1. **Admin**
- Responsible for managing administrative functions such as entering patient details, appointment scheduling, and medicine inventory.
2. **Doctor**
- Responsible for assigning medicines (with or without prescription) and entering prescription instructions.

The system contains several use cases categorized by actor responsibilities:

| | |
|---|---|
| Enter Patient Details | Adds new patient information including name, age, and illness. |
| Enter Appointment Details | Schedules appointments for patients on available days. |
| Enter Medication Details | Adds new medicines to the pharmacy inventory along with dosage and prescription requirements. |
| Search Patient | Finds patient records based on name. |
| Search Medicine Usage | Checks which patients are using a specific medicine. |
| Display All Information | Displays inventory, patient details, prescriptions, and appointments. |
| Enter Prescription Details | Enters specific instructions for a medicine that requires a prescription.<br>- **<<include>> Check Prescription Requirement**<br>- The system checks whether the selected medicine requires a prescription. |
| Assign Non-Prescription Medicine | Assigns medicines that do not need prescriptions, such as over-the-counter drugs.<br>- **<<include>> No Prescription Needed**<br>- The system confirms that the medicine can be assigned without prescription. |

# 3.0 Class Diagram

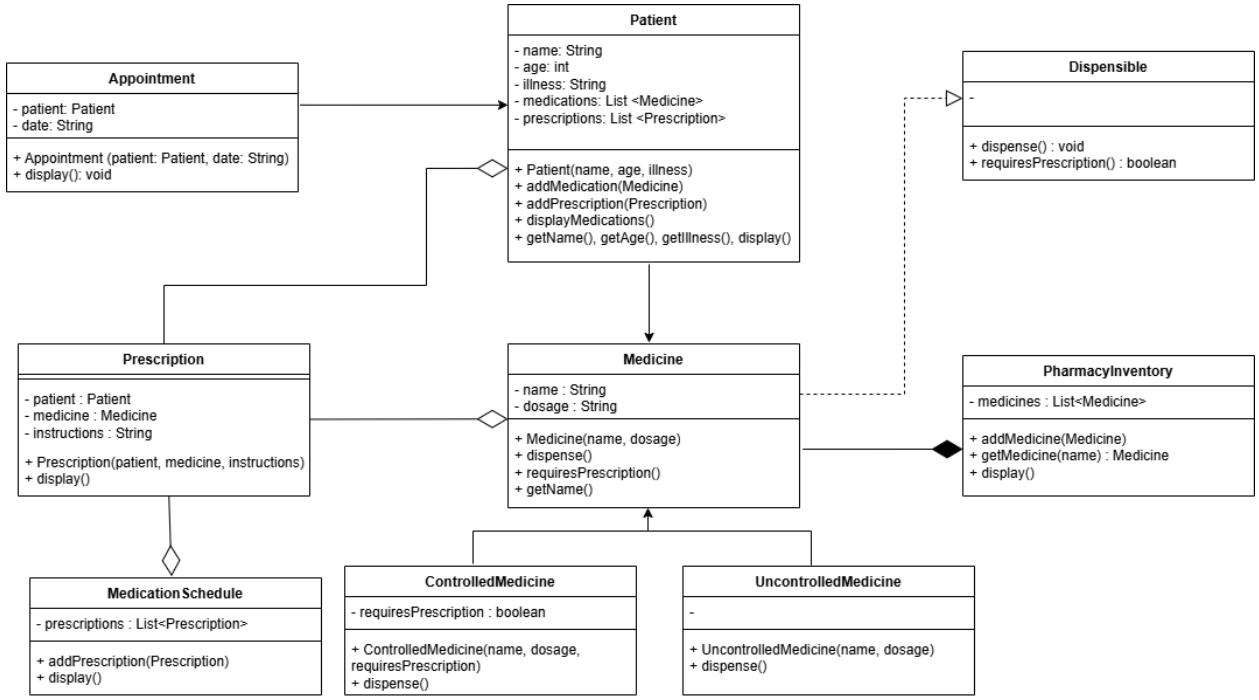


*Figure 2.0: Class Diagram of Efficient Medicine System*

# 3.1 Class Diagram Explanation

## 3.1.1 Patient

| | |
|---|---|
| Attributes | Name, age, illness, list of medications, list of prescriptions. |
| Responsibilities | Can add medications and prescriptions, display their medication list, and retrieve personal details. |
| Relationships | - Has a composition relationship with Medication and Prescription (patient owns and manages these objects).<br>- Associated with Appointment and Prescription. |

### 3.1.2 Appointment

This class represents an appointment for a patient on a specific date. It also contains a reference to the patient and the appointment date.

### 3.1.3 Medicine (Abstract Class)

| | |
|---|---|
| Attributes | Name and dosage. |
| Methods | dispense(), requiresPrescription(), getName(). |
| ControlledMedicine | Includes a flag (requiresPrescription) and overrides dispense(). |
| UncontrolledMedicine | No prescription required; has its own implementation of dispense(). |

### 3.1.4 Prescription

The Prescription class serves the purpose of linking a specific patient to a particular medicine, along with the required instructions for how the medicine should be taken. For instance, it might specify that the medication should be taken "2 times a day". This class plays a critical role in cases where the medicine is a controlled substance, requiring proper documentation and oversight before it can be dispensed. The prescription provides both traceability and ensures that patients follow the correct dosage guidelines.

### 3.1.5 MedicationSchedule

The MedicationSchedule class is designed to manage a list of prescriptions associated with patients. It provides functionalities to add new prescriptions to the schedule and to display them when needed. This class uses a composition relationship with the Prescription class, meaning that the schedule owns and controls the lifecycle of the prescriptions it contains. The medication schedule ensures that each patient has an organized list of their prescribed medicines along with the instructions, supporting proper administration and medication tracking.

### 3.1.6 PharmacyInventory

The PharmacyInventory class maintains a complete list of all medicines available in the pharmacy. It provides methods to add medicines to the system, retrieve medicine information by name, and display the current inventory. This class also uses a composition relationship with the Medicine class, indicating that the inventory owns the medicine instances and is responsible for managing their existence. PharmacyInventory acts as a central hub that supports medication tracking and ensures that the right medicine is available for prescription and dispensing.

### 3.1.7 Dispensable (Interface)

The Dispensable interface defines a standard set of methods that all medicines must implement: dispense() and requiresPrescription(). This interface ensures that each type of medicine, whether controlled or uncontrolled, adheres to a consistent contract for how it is dispensed and whether it needs a prescription. The Medicine class implements this interface using a realization relationship, typically represented by a dotted arrow with a triangle head in UML. This design encourages flexibility and scalability, allowing the system to handle various types of medicines while ensuring uniform behavior across the system.

# 4.0 Complete Java Program

```
import java.util.*;

interface Dispensible {
    void dispense();
    boolean requiresPrescription();
}

class Medicine implements Dispensible {
    protected String name;
    protected String dosage;

    public Medicine(String name, String dosage) {
        this.name = name;
        this.dosage = dosage;
    }

    public void dispense() {
            System.out.println("Dispensing " + name + " with dosage " +
dosage);
    }

    public String getName() {
        return name;
    }

    public boolean requiresPrescription() {
        return false; // default for base medicine
    }
}

class ControlledMedicine extends Medicine {
    private boolean requiresPrescription;

        public ControlledMedicine(String name, String dosage, boolean
requiresPrescription) {
        super(name, dosage);
        this.requiresPrescription = requiresPrescription;
    }
```

```
    @Override
    public void dispense() {
         System.out.print("Dispensing controlled medication (" + name + ")
with dosage = " + dosage);
        if (requiresPrescription) {
            System.out.println(" (requires prescription)");
        } else {
            System.out.println();
        }
    }

    @Override
    public boolean requiresPrescription() {
        return requiresPrescription;
    }
}

class UncontrolledMedicine extends Medicine {
    public UncontrolledMedicine(String name, String dosage) {
        super(name, dosage);
    }

    @Override
    public void dispense() {
        System.out.println("Dispensing uncontrolled medication (" + name +
") with dosage = " + dosage);
    }

    @Override
    public boolean requiresPrescription() {
        return false;
    }
}

class Patient {
    private String name;
    private int age;
    private String illness;
    private List<Medicine> medications = new ArrayList<>();
```

```
    private List<Prescription> prescriptions = new ArrayList<>();

    public Patient(String name, int age, String illness) {
        this.name = name;
        this.age = age;
        this.illness = illness;
    }

    public void addMedication(Medicine medication) {
        if (medications.size() < 10) {
            medications.add(medication);
        } else {
            System.out.println("Cannot add more medications. Maximum limit
reached.");
        }
    }

    public void addPrescription(Prescription prescription) {
        if (prescriptions.size() < 10) {
            prescriptions.add(prescription);
        }
    }

    public void displayMedications() {
        System.out.println("Medications for " + name + ":");
        for (int i = 0; i < medications.size(); i++) {
            Medicine m = medications.get(i);
            m.dispense();
            if (i < prescriptions.size()) {
                                    System.out.println("Instructions:  "  +
prescriptions.get(i).getInstructions());
            }
        }
    }

    public String getName() {
        return name;
    }

    public int getAge() {
```

```
        return age;
    }

    public String getIllness() {
        return illness;
    }

    public void display() {
           System.out.println("Patient: " + name + ", Age: " + age + ",
Illness: " + illness);
    }

    public List<Medicine> getMedications() {
        return medications;
    }
}

class Prescription {
    private Patient patient;
    private Medicine medicine;
    private String instructions;

        public  Prescription(Patient  patient,  Medicine  medicine,  String
instructions) {
        this.patient = patient;
        this.medicine = medicine;
        this.instructions = instructions;
    }

    public void display() {
            System.out.print("Prescription  for  " + patient.getName() + "
(Illness: " + patient.getIllness() + "): ");
        medicine.dispense();
        System.out.println("Instructions: " + instructions + "\n");
    }

    public Medicine getMedicine() {
        return medicine;
    }
```

```
    public String getInstructions() {
        return instructions;
    }
}

class PharmacyInventory {
    private List<Medicine> medicines = new ArrayList<>();

    public void addMedicine(Medicine medicine) {
        for (Medicine m : medicines) {
            if (m.getName().equalsIgnoreCase(medicine.getName())) {
                System.out.println("Medicine already exists.");
                return;
            }
        }
        if (medicines.size() < 10) {
            medicines.add(medicine);
        } else {
                    System.out.println("Inventory full. Cannot add more
medicines.");
        }
    }

    public Medicine getMedicine(String name) {
        for (Medicine m : medicines) {
            if (m.getName().equalsIgnoreCase(name))
                return m;
        }
        return null;
    }

    public void display() {
        System.out.println("\nPharmacy Inventory:");
        for (Medicine m : medicines) {
            m.dispense();
        }
    }

    public List<Medicine> getMedicines() {
        return medicines;
```

```
    }
}

class Appointment {
    private Patient patient;
    private String date;

    public Appointment(Patient patient, String date) {
        this.patient = patient;
        this.date = date;
    }

    public void display() {
        System.out.println("Appointment for " + patient.getName() + " on "
+ date);
    }
}

class MedicationSchedule {
    private List<Prescription> prescriptions = new ArrayList<>();

    public void addPrescription(Prescription prescription) {
        if (prescriptions.size() < 10) {
            prescriptions.add(prescription);
        } else {
             System.out.println("Medication schedule full. Cannot add more
prescriptions.");
        }
    }

    public void display() {
        System.out.println("\nMedication Schedule:");
        for (Prescription p : prescriptions) {
            p.display();
        }
    }
}

public class MedicineSystem {
    public static void main(String[] args) {
```

```
        Scanner scanner = new Scanner(System.in);
        PharmacyInventory inventory = new PharmacyInventory();
        MedicationSchedule schedule = new MedicationSchedule();
        List<Patient> patients = new ArrayList<>();
        boolean[] appointments = new boolean[30];

        while (true) {
            try {
                        System.out.println("\nWelcome to Efficient Medicine
system\n----------------------------");
                System.out.println("1. Enter Patient Details");
                System.out.println("2. Enter Medication Details");
                System.out.println("3. Enter Prescription Details");
                System.out.println("4. Enter Appointment Details");
                System.out.println("5. Display All Information");
                System.out.println("6. Search Patient");
                System.out.println("7. Search Medicine Usage");
                System.out.println("8. Exit");
                System.out.print("Choose an option: ");

                int choice = Integer.parseInt(scanner.nextLine());
                if (choice == 8)
                    break;

                switch (choice) {
                    case 1: {
                        String name;
                        do {
                            System.out.print("\nEnter patient name: ");
                            name = scanner.nextLine().trim();
                        } while (name.isEmpty());

                        boolean exists = false;
                        for (Patient p : patients) {
                            if (p.getName().equalsIgnoreCase(name)) {
                                exists = true;
                                break;
                            }
                        }
                        if (exists) {
```

```
                            System.out.println("Patient already exists.");
                            break;
                        }

                        int age;
                        do {
                               System.out.print("Enter patient age (0-130):
");
                            try {
                                                                    age  =
Integer.parseInt(scanner.nextLine());
                            } catch (NumberFormatException e) {
                                age = -1;
                            }
                        } while (age < 0 || age > 130);

                        System.out.print("Enter patient illness: ");
                        String illness = scanner.nextLine();

                        Patient p = new Patient(name, age, illness);
                        p.display();
                        patients.add(p);
                        break;
                    }
                    case 2: {
                        System.out.print("\nEnter medicine name: ");
                        String name = scanner.nextLine();
                        System.out.print("Enter dosage: ");
                        String dosage = scanner.nextLine();
                             System.out.print("Does the medicine require a
prescription? (true/false): ");
                                            boolean  requiresPrescription  =
Boolean.parseBoolean(scanner.nextLine());

                        Medicine med = requiresPrescription
                                    ? new ControlledMedicine(name, dosage,
true)
                                : new UncontrolledMedicine(name, dosage);

                        inventory.addMedicine(med);
```

```
                        inventory.display();
                        break;
                    }
                    case 3: {
                        System.out.println("\nAvailable patients:");
                           patients.forEach(p -> System.out.println("- " +
p.getName()));

                                System.out.print("\nEnter patient name for
prescription: ");
                        String patientName = scanner.nextLine();

                        Patient foundPatient = patients.stream()
                                                                .filter(p  ->
p.getName().equalsIgnoreCase(patientName))
                                .findFirst().orElse(null);

                        if (foundPatient == null) {
                            System.out.println("Patient not found.");
                            break;
                        }

                        System.out.println("Available medicines:");
                                    inventory.getMedicines().forEach(m  ->
System.out.println("- " + m.getName()));

                                System.out.print("Enter medicine name for
prescription: ");
                        String medName = scanner.nextLine();

                                                    Medicine  foundMed  =
inventory.getMedicine(medName);
                        if (foundMed == null) {
                                 System.out.println("Medicine not found in
inventory.");
                            break;
                        }

                        String instructions;
                        if (!foundMed.requiresPrescription()) {
```

```
                                instructions = "No instructions required
(non-prescription)";
                                System.out.println("Note: No instructions
needed for non-prescription medicine.");
                        } else {
                                    System.out.print("Enter prescription
instructions: ");
                            instructions = scanner.nextLine();
                        }

                        Prescription pres = new Prescription(foundPatient,
foundMed, instructions);
                        foundPatient.addMedication(foundMed);
                        foundPatient.addPrescription(pres);
                        schedule.addPrescription(pres);
                        schedule.display();
                        break;
                    }
                    case 4: {
                                System.out.println("\nCurrent Appointment
Schedule:");
                        for (int i = 0; i < appointments.length; i++) {
                              System.out.println("Day " + (i + 1) + ": " +
(appointments[i] ? "Occupied" : "Free"));
                        }
                               System.out.print("Enter day for appointment
(1-30): ");
                        int day = Integer.parseInt(scanner.nextLine());
                         if (day < 1 || day > 30 || appointments[day - 1])
{
                                  System.out.println("Invalid or occupied
day.");
                            break;
                        }
                                System.out.print("Enter patient name for
appointment: ");
                        String patientName = scanner.nextLine();
                        Patient foundPatient = patients.stream()
                                                                .filter(p  ->
p.getName().equalsIgnoreCase(patientName))
```

```
                                .findFirst().orElse(null);
                        if (foundPatient != null) {
                            appointments[day - 1] = true;
                                        new Appointment(foundPatient,
String.valueOf(day)).display();
                        } else {
                            System.out.println("Patient not found.");
                        }
                        break;
                    }
                    case 5: {
                        inventory.display();
                        schedule.display();
                                 System.out.println("\nCurrent Appointment
Schedule:");
                        for (int i = 0; i < appointments.length; i++) {
                              System.out.println("Day " + (i + 1) + ": " +
(appointments[i] ? "Occupied" : "Free"));
                        }
                        break;
                    }
                    case 6: {
                         System.out.print("\nEnter patient name to search:
");
                        String searchName = scanner.nextLine();
                        Patient pat = patients.stream()
                                                                .filter(p ->
p.getName().equalsIgnoreCase(searchName))
                                .findFirst().orElse(null);
                        if (pat != null) {
                            pat.display();
                            pat.displayMedications();
                        } else {
                            System.out.println("Patient not found.");
                        }
                        break;
                    }
                    case 7: {
                        System.out.print("\nEnter medicine name to search:
");
```

```
                        String searchMed = scanner.nextLine();
                        boolean found = false;
                        for (Patient p : patients) {
                            for (Medicine med : p.getMedications()) {
                                                                    if
(med.getName().equalsIgnoreCase(searchMed)) {
                                        System.out.println(p.getName() + "
uses " + med.getName());
                                    found = true;
                                }
                            }
                        }
                        if (!found) {
                            System.out.println("No patients are using this
medicine.");
                        }
                        break;
                    }
                    default:
                            System.out.println("Invalid option. Please try
again.");
                }
            } catch (Exception e) {
                System.out.println("Error: " + e.getMessage());
            }
        }

        scanner.close();
    }
}
```

# 5.0 Task Allocations

| NAME | TASK |
|---|---|
| MUHAMMAD AMIRUN IRFAN BIN SAMSUL SHAH | 1. Implemented Patient class<br>2. Handled Patient registration and search logic<br>3. Managed Appointment scheduling functionality<br>4. Wrote report section on Patient class, Appointment class, and Association relationships |
| MUHAMMAD HAFIZ BIN MOHD SHAHARUDDIN | 1. Implemented Medicine, ControlledMedicine, UncontrolledMedicine classes<br>2. Designed PharmacyInventory and medicine search<br>3. Wrote report section on Inheritance, Polymorphism, and Inventory clas |
| RAVINESH A/L MARAN | 1. Implemented Prescription and MedicationSchedule classes<br>2. Developed Prescription logic with validation and instruction handling<br>3. Wrote report section on Aggregation, Prescription module, and SDG 3 relevance |
| WELSON WOONG LU BIN | 1. Developed Dispensible interface<br>2. Integrated interface implementation and exception handling across the system<br>3. Coordinated main program logic and menu navigation<br>4. Wrote report section on Interface and Implementation, Exception Handling, and Conclusion |

# 6.0 Applied Object-Oriented Concepts Explanations

## 6.1 Encapsulation

```
lass Patient {
    private String name;
    private int age;
    private String illness;
    private List<Medicine> medications = new ArrayList<>();
    private List<Prescription> prescriptions = new ArrayList<>();

    public Patient(String name, int age, String illness) {
        this.name = name;
        this.age = age;
        this.illness = illness;
    }

    public void addMedication(Medicine medication) {
        if (medications.size() < 10) {
            medications.add(medication);
        } else {
            System.out.println("Cannot add more medications. Maximum limit
reached.");
        }
    }

    public void addPrescription(Prescription prescription) {
        if (prescriptions.size() < 10) {
            prescriptions.add(prescription);
        }
    }

    public void displayMedications() {
        System.out.println("Medications for " + name + ":");
        for (int i = 0; i < medications.size(); i++) {
            Medicine m = medications.get(i);
            m.dispense();
            if (i < prescriptions.size()) {
```

```
                                System.out.println("Instructions:  "  +
prescriptions.get(i).getInstructions());
            }
        }
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getIllness() {
        return illness;
    }

    public void display() {
           System.out.println("Patient: " + name + ", Age: " + age + ",
Illness: " + illness);
    }

    public List<Medicine> getMedications() {
        return medications;
    }
}
```

Encapsulation is the concept of hiding internal data and exposing only necessary functionalities through methods. In this system, all classes such as Patient, Medicine, and Prescription have private attributes, and access is provided via public getter and setter methods. This restricts direct access to internal data and ensures proper data control.

## 6.2 Inheritance

```
class UncontrolledMedicine extends Medicine {
    public UncontrolledMedicine(String name, String dosage) {
        super(name, dosage);
```

```
    }

    @Override
    public void dispense() {
        System.out.println("Dispensing uncontrolled medication (" + name +
") with dosage = " + dosage);
    }

    @Override
    public boolean requiresPrescription() {
        return false;
    }
}
```

Inheritance enables the reuse of common functionality. In this system, the base class Medicine is extended by subclasses such as ControlledMedicine and UncontrolledMedicine. These subclasses inherit common attributes and methods but may override some behaviors.

## 6.3 Polymorphism

```
Medicine med = new ControlledMedicine("Para", "500mg", true);
med.dispense();  // Will use the overridden method in ControlledMedicine
```

Polymorphism allows the use of a single interface to represent different types. The dispense() method behaves differently depending on whether it is called on a ControlledMedicine or an UncontrolledMedicine object.

## 6.4 Abstraction

```
interface Dispensible {
    void dispense();
    boolean requiresPrescription();
}
```

Abstraction hides complex logic and exposes only essential functionalities. In the system, an interface named Dispensable is used to define a general contract for medicines that can be dispensed. Any medicine, regardless of type, will implement this interface and define its own logic.

## 6.5 Association, Aggregation, and Composition

```
class Prescription {
    private Patient patient;
    private Medicine medicine;
    private String instructions;

        public Prescription(Patient patient, Medicine medicine, String
instructions) {
        this.patient = patient;
        this.medicine = medicine;
        this.instructions = instructions;
    }

    public void display() {
            System.out.print("Prescription for " + patient.getName() + "
(Illness: " + patient.getIllness() + "): ");
        medicine.dispense();
        System.out.println("Instructions: " + instructions + "\n");
    }

    public Medicine getMedicine() {
        return medicine;
    }

    public String getInstructions() {
        return instructions;
    }
}
```

Association represents a general relationship between two classes. For example, a Prescription is associated with both a Patient and a Medicine. These objects can exist independently.

```
class Patient {
    private String name;
    private int age;
    private String illness;
    private List<Medicine> medications = new ArrayList<>();
    private List<Prescription> prescriptions = new ArrayList<>();

    public Patient(String name, int age, String illness) {
        this.name = name;
        this.age = age;
        this.illness = illness;
    }

    public void addMedication(Medicine medication) {
        if (medications.size() < 10) {
            medications.add(medication);
        } else {
            System.out.println("Cannot add more medications. Maximum limit
reached.");
        }
    }
```

Aggregation is a stronger association where one class “has” another class, but both can still exist independently. The Patient class has a list of Medicine objects.

```
class PharmacyInventory {
    private List<Medicine> medicines = new ArrayList<>();

    public void addMedicine(Medicine medicine) {
        for (Medicine m : medicines) {
            if (m.getName().equalsIgnoreCase(medicine.getName())) {
                System.out.println("Medicine already exists.");
                return;
            }
        }
        if (medicines.size() < 10) {
            medicines.add(medicine);
        } else {
```

```
                    System.out.println("Inventory full. Cannot add more
medicines.");
        }
    }
```

Composition is a form of aggregation where the contained object cannot exist without the container. In this system, PharmacyInventory owns and manages Medicine objects completely.

## 6.6 Interface and Implementation

```
interface Dispensible {
    void dispense();
    boolean requiresPrescription();
}

class Medicine implements Dispensible {
    protected String name;
    protected String dosage;
```

Interfaces provide a blueprint for what methods a class must implement. The Dispensable interface ensures that any class that implements it will have dispense() and requiresPrescription() methods.

## 6.7 Exception Handling

```
try {
    Medicine med = inventory.getMedicine("unknown");
    if (med == null) {
        throw new IllegalArgumentException("Medicine not found!");
    }
    med.dispense();
} catch (IllegalArgumentException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Exception handling is used to gracefully manage runtime errors such as null references or invalid operations. In this system, exceptions are caught and appropriate messages are shown to users, preventing the program from crashing.

# 7.0 Conclusion

In conclusion, the Efficient Medicine System developed in this project demonstrates a complete and functional object-oriented approach to managing patients, medications, prescriptions, and pharmacy inventory. By applying the core principles of Object-Oriented Programming (OOP)—namely encapsulation, inheritance, polymorphism, and abstraction—the system achieves modularity, code reusability, and scalability, making it easier to maintain and extend in the future.

The integration of interface implementation and proper exception handling further improves the system's robustness and reliability, ensuring that users are notified of invalid inputs and system errors gracefully. Relationships such as association, aggregation, and composition were effectively modeled through the interactions between Patient, Medicine, Prescription, and Inventory classes, reflecting real-world healthcare structures.

Additionally, this system supports Sustainable Development Goal 3 (Good Health and Well-Being) by promoting more organized and accessible healthcare information management. It enhances the accuracy of medication dispensing and ensures that prescriptions are properly handled for controlled medications, contributing to safer healthcare delivery.
Overall, this project provides a practical, educational implementation of an object-oriented healthcare application that can be extended with additional features such as user authentication, database integration, and graphical interfaces for real-world deployment.