# CLINIC MANAGEMENT SYSTEM PROJECT REPORT

## 1. Introduction

**Student Name:** Cem Adıgüzel
**Student Number:** *20220305014*
**Course Name:** Object Oriented Programming
**Project Title:** Clinic Management System
**Programming Language:** Java

https://github.com/cemdgzl/nesne2

This project is a simple **Clinic Management System** developed using the Java programming language. The aim of the project is to design and implement a desktop application that demonstrates fundamental **Object-Oriented Programming (OOP)** concepts such as inheritance, interfaces, polymorphism, generics, generic collections, and lambda expressions.

The system allows clinic staff to register patients and doctors, create appointments, and display stored information through both a **console-based interface** and a **graphical user interface (GUI)** implemented using Java Swing. The project follows a **multi-tier architecture** by separating user interface, business logic, and data storage layers.

## 2. Business Description of the Project

Small clinics often need a basic system to manage patient and doctor information without complex infrastructure. This project simulates such a system. The **Clinic Management System** is designed for a receptionist or clinic staff member who performs daily administrative tasks.

The main business functions of the system are:

- Registering patients with personal and medical information,

- Registering doctors along with their medical branch,
- Creating appointments between patients and doctors,
- Listing all registered patients, doctors, and appointments,
- Displaying the list of available doctor branches.

All data is stored temporarily in memory while the program is running. This design is sufficient for educational purposes and for demonstrating software design principles.

## 3. Overall System Design

### 3.1 Use Case Diagram

The system has one primary actor:

- **Receptionist**

The receptionist can perform the following use cases:

- Register Patient
- Register Doctor
- Create Appointment
- List Patients
- List Doctors
- List Appointments
- View Doctor Branches

All use cases are contained within the system boundary named **Clinic Management System**.
(The Use Case Diagram is shown in Figure 1.)

### 3.2 Class Diagram

The class diagram of the system consists of several core classes organized into layers:

- **Model Layer:** `Person`, `Patient`, `Doctor`, `Appointment`
- **Data Layer:** `Repository<T, ID>`, `InMemoryRepository<T, ID>`
- **Business Layer:** `ClinicService`, `ClinicServiceImpl`
- **UI Layer:** `ClinicApp` (Console), `ClinicGuiApp` (Swing GUI)

- **Utility Class:** `PrintUtil`

Key relationships:

- `Person` is an abstract superclass.
- `Patient` and `Doctor` inherit from `Person`.
- `Appointment` has associations with `Patient` and `Doctor`.
- `ClinicServiceImpl` implements the `ClinicService` interface.
- `InMemoryRepository<T, ID>` implements the `Repository<T, ID>` interface.

(The Class Diagram is shown in Figure 2.)

## 4. System Architecture

The project follows a **multi-tier (layered) architecture**:

### 4.1 UI Layer

The system provides two types of user interfaces:

1. **Console Interface (`ClinicApp`)**
   a. Uses `Scanner` for user input.
   b. Menu-driven text-based interface.
   c. Useful for testing and basic interaction.
2. **Graphical User Interface (`ClinicGuiApp`)**
   a. Implemented using **Java Swing**.
   b. Uses buttons, text fields, labels, tabs, and dialogs.
   c. Provides a more user-friendly experience.
   d. Uses `JFrame`, `JPanel`, `JTabbedPane`, and `JTextArea`.

Both interfaces interact only with the business layer, not directly with the data layer.

### 4.2 Business Layer

The business logic is defined by the `ClinicService` interface and implemented by `ClinicServiceImpl`.

Responsibilities of this layer include:

- Creating and managing patients, doctors, and appointments,
- Generating unique IDs for each entity,
- Validating relationships (e.g., ensuring a patient and doctor exist before creating an appointment),
- Managing the set of doctor branches.

This layer ensures that the UI is independent of implementation details.

### 4.3 Data Layer

The data layer uses a **generic repository pattern**:

- `Repository<T, ID>` defines common data operations.
- `InMemoryRepository<T, ID>` is a generic class that stores data using a `Map<ID, T>`.

Each entity type (Patient, Doctor, Appointment) uses the same repository implementation with different generic parameters.

## 5. Object-Oriented Programming Concepts Used

### 5.1 Inheritance

Inheritance is achieved using the abstract class `Person`.

- `Person` contains common fields such as `id`, `name`, and `phone`.
- `Patient` and `Doctor` extend `Person` and reuse its functionality.
- Each subclass overrides the `toString()` method.

### 5.2 Interface

Two main interfaces are used:

- `ClinicService`: Defines business operations.
- `Repository<T, ID>`: Defines generic data access operations.

Concrete classes implement these interfaces, which allows loose coupling and flexibility.

### 5.3 Polymorphism

Polymorphism is applied in the following ways:

- The UI layer works with the `ClinicService` interface, while the actual object is `ClinicServiceImpl`.
- The data layer works with the `Repository<T, ID>` interface, implemented by `InMemoryRepository<T, ID>`.

Thus, different implementations could be introduced without changing existing code.

### 5.4 Generic Class

The `InMemoryRepository<T, ID>` class is a generic class. It can store any entity type by specifying the type parameter. This avoids code duplication and improves reusability.

### 5.5 Generic Collections

The project uses multiple generic collections:

- `Map<ID, T>` for storing entities,
- `List<Patient>`, `List<Doctor>`, `List<Appointment>` for listing data,
- `Set<String>` for storing unique doctor branches.

This satisfies the requirement of using more than one generic collection type.

### 5.6 Lambda Expressions

Lambda expressions are used in several parts of the project:

- Printing lists using `forEach(...)`,
- Iterating over collections in the GUI output area,

- Improving code readability and reducing boilerplate.

Example:

```
list.forEach(item -> System.out.println(item));
```

## 6. Graphical User Interface Design

The graphical user interface is implemented using Java Swing. It consists of:

- A main window (`JFrame`),
- Tab-based navigation (`JTabbedPane`) for Patients, Doctors, and Appointments,
- Forms with text fields and buttons for data entry,
- An output area to display results and system messages.

Error handling is performed using dialog boxes (`JOptionPane`).

## 7. Conclusion

In this project, a simple yet functional Clinic Management System was developed using Java. The application demonstrates fundamental object-oriented programming concepts and software design principles such as inheritance, interfaces, polymorphism, generics, and layered architecture.

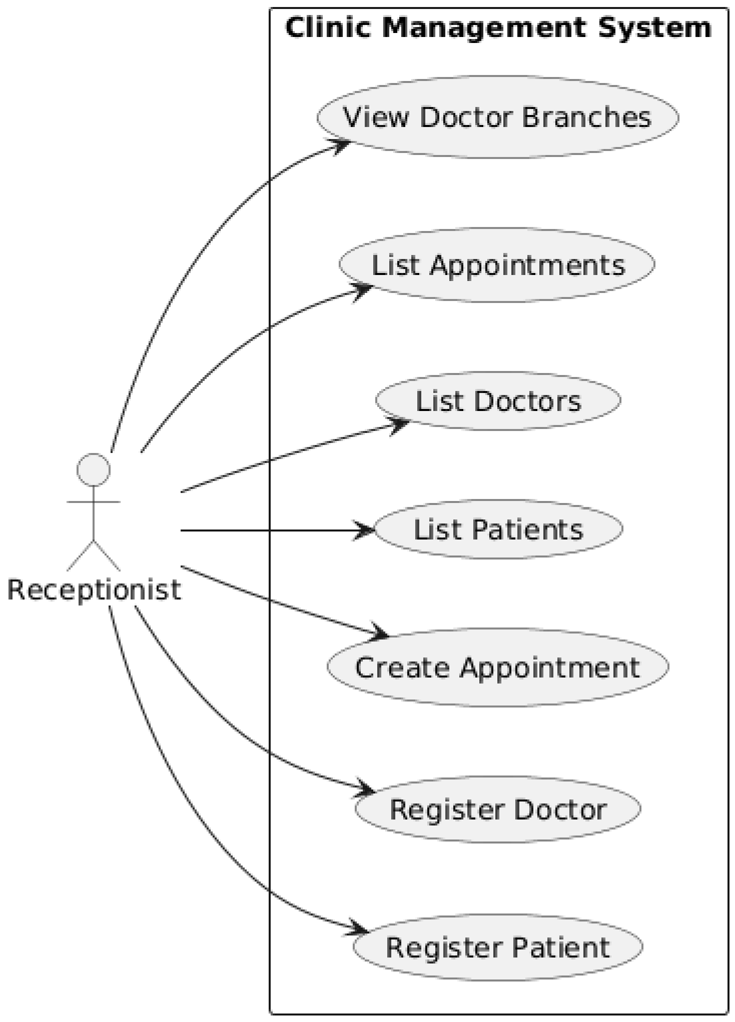
The system provides both console-based and graphical user interfaces, making it flexible and user-friendly. Although the data is stored only in memory, the architecture is suitable for future enhancements such as database integration or advanced GUI designs.

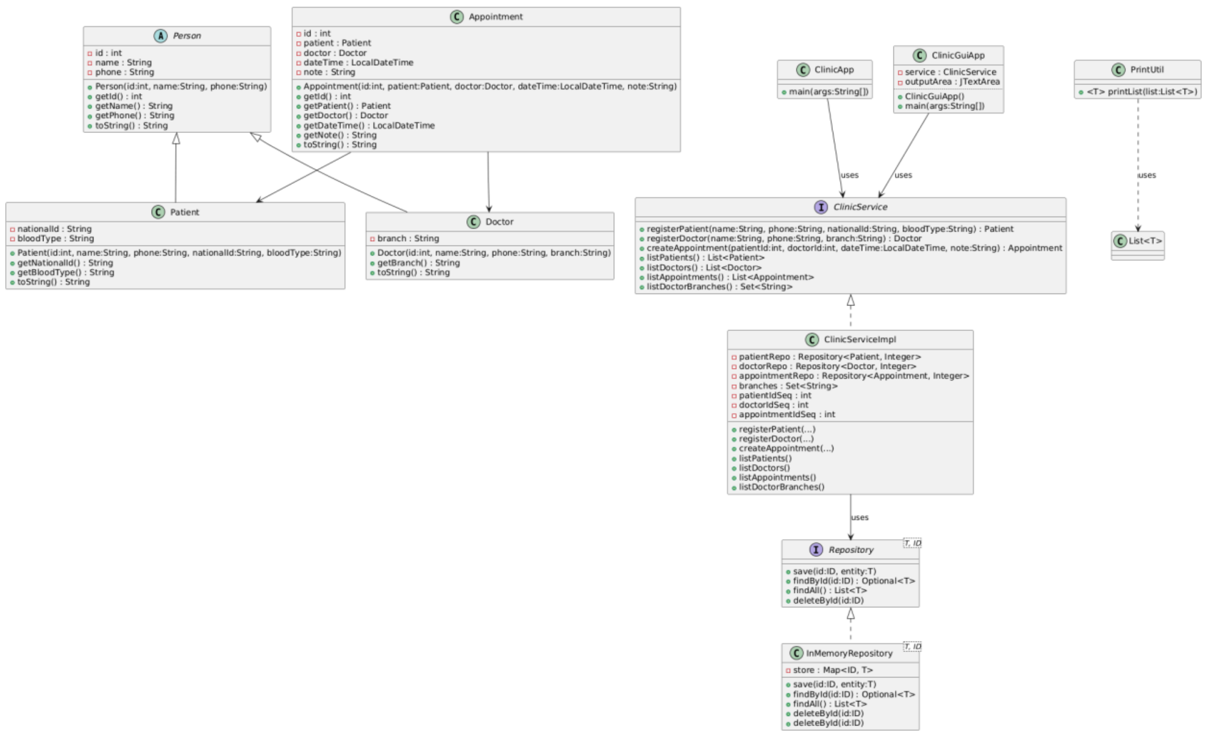
This project successfully fulfills the requirements of the course and provides a solid example of object-oriented software development in Java.

## 8. References

1. Oracle Java Documentation – https://docs.oracle.com/javase
2. Schildt, H. *Java: The Complete Reference*, McGraw-Hill

3. Oracle Swing Tutorial – https://docs.oracle.com/javase/tutorial/uiswing




Person
id : int
name : String
phone : String
Person(id:int, name:String, phone:String)
getId() : int
getName() : String
getPhone() : String
toString() : String
Appointment
id : int
patient : Patient
doctor : Doctor
dateTime : LocalDateTime
note : String
Appointment(id:int, patient:Patient, doctor:Doctor, dateTime:LocalDateTime, note:String)
getId() : int
getPatient() : Patient
getDoctor() : Doctor
getDateTime() : LocalDateTime
getNote() : String
toString() : String
Patient
nationalId : String
bloodType : String
Patient(id:int, name:String, phone:String, nationalId:String, bloodType:String)
getNationalId() : String
getBloodType() : String
toString() : String
Doctor
branch : String
Doctor(id:int, name:String, phone:String, branch:String)
getBranch() : String
toString() : String
ClinicApp
main(args:String[])
ClinicGuiApp
service : ClinicService
outputArea : JTextArea
ClinicGuiApp()
main(args:String[])
PrintUtil
<T> printList(list:List<T>)
uses
uses
uses
ClinicService
registerPatient(name:String, phone:String, nationalId:String, bloodType:String) : Patient
registerDoctor(name:String, phone:String, branch:String) : Doctor
createAppointment(patientId:int, doctorId:int, dateTime:LocalDateTime, note:String) : Appointment
listPatients() : List<Patient>
listDoctors() : List<Doctor>
listAppointments() : List<Appointment>
listDoctorBranches() : Set<String>
List<T>
ClinicServiceImpl
patientRepo : Repository<Patient, Integer>
doctorRepo : Repository<Doctor, Integer>
appointmentRepo : Repository<Appointment, Integer>
branches : Set<String>
patientIdSeq : int
doctorIdSeq : int
appointmentIdSeq : int
registerPatient(...)
registerDoctor(...)
createAppointment(...)
listPatients()
listDoctors()
listAppointments()
listDoctorBranches()
uses
Repository
T, ID
save(id:ID, entity:T)
findById(id:ID) : Optional<T>
findAll() : List<T>
deleteById(id:ID)
InMemoryRepository
T, ID
store : Map<ID, T>
save(id:ID, entity:T)
findById(id:ID) : Optional<T>
findAll() : List<T>
deleteById(id:ID)
deleteById(id:ID)