

CSC 323: Object-Oriented Design

Project – Report

Submitted by:

Student ID	Name	Major
20238113	Yamen Morcel	Computer Science
20238107	Hadi Khawand	Computer Science
20228114	Majd Kansa	Computer Science

Project Description (around 500 words)

The Healthcare Management System (HMS) is designed to streamline the administrative processes within a healthcare facility, such as a hospital. This system is intended to be used by hospital staff, including administrative personnel and medical professionals, to efficiently manage and schedule patient appointments. The main objective of the HMS is to ensure accurate, quick, and organized handling of patient information and appointment scheduling, with a focus on adding and viewing appointments only.

Key Features

- Appointment Scheduling** The primary function of the HMS is to facilitate the scheduling of patient appointments. Users can:
 - Input patient details (name, age, medical history, medications).
 - Assign a doctor to the appointment, specifying the doctor's name and specialization.
 - Set the date and time for the appointment.
 - Specify the type of payment (normal or discounted fee).
 - Record the appointment fee.
- Patient:** The **Patient** class in the Health Management System manages a patient's details including their name, unique identifier (ID), age, medical history, and current medications. It provides constructors to initialize a patient's data with or without a medical history and medications. The class offers methods to add to the patient's medical history, update their medications, retrieve basic patient information, and generate a comprehensive string representation of the patient's details.
 - **Properties:** Name, ID, Age, MedicalHistory, CurrentMedications.
 - **Constructors:** Initializes patient with or without medical history and medications.
 - **Methods:** AddToMedicalHistory, UpdateMedications, GetPatientInfo, ToString.
- The **Doctor** class in the Health Management System represents a doctor, capturing their name, unique identifier (ID), and specialization. It provides constructors for initializing a doctor's data with and without a pre-defined ID. The class also includes an overridden ToString method to generate a string representation of the doctor's details.
 - **Properties:** Name, ID, Specialization.
 - **Constructors:**
 - Doctor(string name, string id, string specialization): Initializes a doctor with a name, ID, and specialization.
 - Doctor(string name, string specialization): Initializes a doctor with a name and specialization, generating a unique ID.
 - **Methods:** ToString (returns a string with the doctor's details).
- Database Management (Conceptual)** Although the system is not connected to a real database, it conceptually simulates database functionalities. All appointments are stored in a collection (like a list) and can be accessed and displayed when needed. No updates are allowed to existing appointments; they can only be viewed.

Class Diagram (UML)

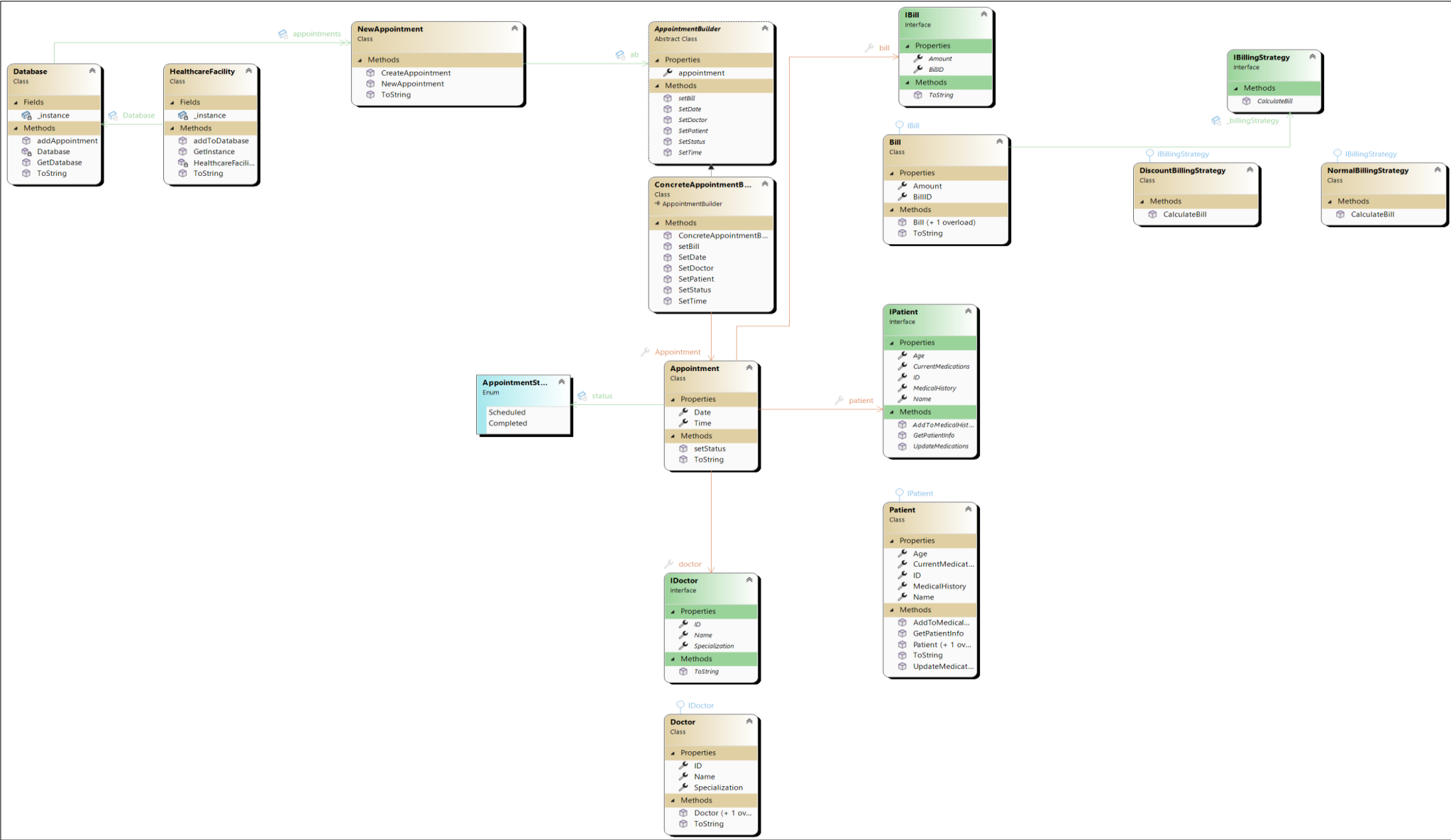


Figure 1. Class Diagram caption ...

Source Code (C#)

```
{  
  
    public class NewAppointment  
    {  
        private AppointmentBuilder ab;  
        public NewAppointment() { ab = new ConcreteAppointmentBuilder(); }  
  
        public void CreateAppointment(IPatient p, IDoctor d, DateTime date, string time, AppointmentStatus  
s, IBill b )  
        {  
            ab.SetPatient( p );  
            ab.SetDoctor( d );  
            ab.SetDate( date );  
            ab.SetTime(time);  
            ab.SetStatus( s );  
            ab.setBill( b );  
        }  
        public override string ToString()  
        {  
            return ab.appointment.ToString();  
        }  
    }  
}
```

Figure 2. Director for the Builder pattern

```
{  
  
    public abstract class AppointmentBuilder  
    {  
        public Appointment appointment { get; set; }  
  
        public abstract void SetPatient(IPatient p);  
        public abstract void SetDoctor(IDoctor d);  
  
        public abstract void SetDate(DateTime date);  
        public abstract void SetTime(string time);  
        public abstract void SetStatus(AppointmentStatus s);  
        public abstract void setBill(IBill b);  
    }  
}
```

Figure 3. Abstract Builder Class

```
{  
  
    public class ConcreteAppointmentBuilder : AppointmentBuilder  
    {  
        public ConcreteAppointmentBuilder() { appointment = new Appointment(); }  
  
        public Appointment Appointment  
        {  
            get => default;  
            set  
            {  
            }  
        }  
    }  
  
    public override void SetPatient(IPatient p) { appointment.patient = p; }  
    public override void SetDoctor(IDoctor d) { appointment.doctor = d; }
```

```

    public override void SetDate(DateTime date) { appointment.Date = date; }
    public override void SetTime(string time) { appointment.Time = time; }
    public override void SetStatus(AppointmentStatus s) { appointment.setStatus(s); }
    public override void setBill(IBill b) { appointment.bill = b; }
}

public class Appointment
{
    private AppointmentStatus status;
    public IPatient patient { get; set; }
    public IDoctor doctor { get; set; }
    public DateTime Date { get; set; }

    public string Time { get; set; }

    public IBill bill { get; set; }

    public void setStatus(AppointmentStatus s)
    {
        status = s;
    }
    public override string ToString()
    {
        StringBuilder sb = new StringBuilder();
        sb.Append($"\\nScheduled Date: {Date.ToString("dddd MMM,dd,yyyy")}\\n");
        sb.AppendLine($"Appointment Time: {Time}");
        sb.AppendLine($"Appointment Status: {status}");
        return $"Appointment Info:\\n{patient.ToString()}\\nDoctor
Information:\\n{doctor.ToString()}\\n\\nSchedule:{sb.ToString()}\\nBill:\\n{bill.ToString()}";
    }
}
}

```

4. Concrete Builder and Product

```

public interface IPatient
{
    string Name { get; set; }
    string ID { get; set; }
    int Age { get; set; }
    List<string> MedicalHistory { get; set; }
    List<string> CurrentMedications { get; set; }

    void AddToMedicalHistory(string medicalHistory);
    void UpdateMedications(string medications);
    string GetPatientInfo();
}

public class Patient: IPatient
{
    public string Name { get; set; }
    public string ID { get; set; }
    public int Age { get; set; }
    public List<string> MedicalHistory { get; set; }
    public List<string> CurrentMedications { get; set; }
    public Patient(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public Patient(string name, int age, List<string> mHistory, List<string> cMedication):
this(name, age)

```

```

{
    ID = Guid.NewGuid().ToString();
    MedicalHistory = mHistory;
    CurrentMedications = cMedicaton;
}

public void AddToMedicalHistory(string medicalHistory)
{
    MedicalHistory.Add(medicalHistory);
}

public void UpdateMedications(string medications)
{
    CurrentMedications.Add(medications);
}

public string GetPatientInfo()
{
    return $"Patient: {Name}, ID: {ID}, Age: {Age}";
}

public override string ToString()
{
    StringBuilder sb = new StringBuilder();
    sb.AppendLine($"Patient: {Name}\nID: {ID}\nAge: {Age}\n");
    sb.AppendLine("Medication(s):");
    foreach (string c in CurrentMedications)
    {
        sb.AppendLine(c);
    }
    sb.AppendLine("\nMedical History:");
    foreach(string c in MedicalHistory)
    {
        sb.AppendLine(c);
    }
    return sb.ToString();
}
}

```

5. Patient Class with interface

```

public interface IDoctor
{
    // Properties
    string Name { get; set; }
    string ID { get; set; }
    string Specialization { get; set; }

    // Methods
    string ToString();
}

public class Doctor: IDoctor
{
    // Properties
    public string Name { get; set; }
    public string ID { get; set; }
    public string Specialization { get; set; }

    public Doctor(string name, string id, string specialization)
    {
        Name = name;
        ID = id;
    }
}

```

```

        Specialization = specialization;
    }

    public override string ToString()
    {
        return $"Doctor: {Name}\nID: {ID}\nSpecialization: {Specialization}";
    }

    public Doctor(string name, string specialization) : this(name, Guid.NewGuid().ToString(),
specialization)
    {
    }
}

```

6. Doctor Class with interface

```

public class HealthcareFacility
{
    private static HealthcareFacility _instance;

    private Database Database;
    private HealthcareFacility() {
        Database = Database.GetDatabase();
    }

    public static HealthcareFacility GetInstance()
    {
        if (_instance == null)
        {
            _instance = new HealthcareFacility();
        }
        return _instance;
    }

    public override string ToString()
    {
        return Database.ToString();
    }

    public void addToDatabase(NewAppointment appointment)
    {
        Database.addAppointment(appointment);
    }
}

```

7. HealthcareFacility (Singleton)

```

internal class Database
{
    private List<NewAppointment> appointments = new List<NewAppointment>();
    static Database _instance;
}

```

```

private Database() { }

public static Database GetDatabase() {
    if (_instance == null)
    {
        _instance = new Database();
    }
    return _instance;
}
public void addAppointment(NewAppointment appointment)
{
    appointments.Add(appointment);
}
public override string ToString()
{
    if (appointments.Count > 0)
    {
        int i = 0;
        StringBuilder sb = new StringBuilder();
        foreach (NewAppointment appointment in appointments)
        {
            sb.AppendLine($"Appointment {++i}\n");
            sb.Append(appointment.ToString());
            sb.AppendLine().AppendLine();
        }
        return sb.ToString();
    }
    return "The Database Is Empty!";
}
}

```

8. Database Class (Singleton)

```

public interface IBill
{
    string BillID { get; }
    decimal Amount { get; set; }
    string ToString();
}

public class Bill: IBill
{
    public string BillID { get; set; }
    public decimal Amount { get; set; }

    private readonly IBillingStrategy _billingStrategy;
    public Bill(IBillingStrategy billingStrategy)
    {
        BillID = Guid.NewGuid().ToString();

        _billingStrategy = billingStrategy;
    }

    public Bill( decimal amount, IBillingStrategy billingStrategy)
    {
        BillID = Guid.NewGuid().ToString();
        _billingStrategy = billingStrategy;
        Amount = _billingStrategy.CalculateBill(amount);
    }
}

```



```

    }
    public override string ToString()
    {
        return $"BillID: {BillID}\nAmount t: {Amount:C}";
    }
}

```

9. Bill class with interface

```

public interface IBillingStrategy
{
    decimal CalculateBill(decimal amount);
}

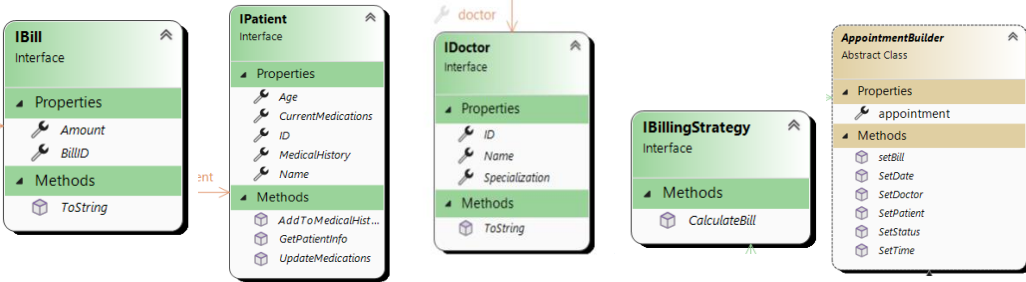
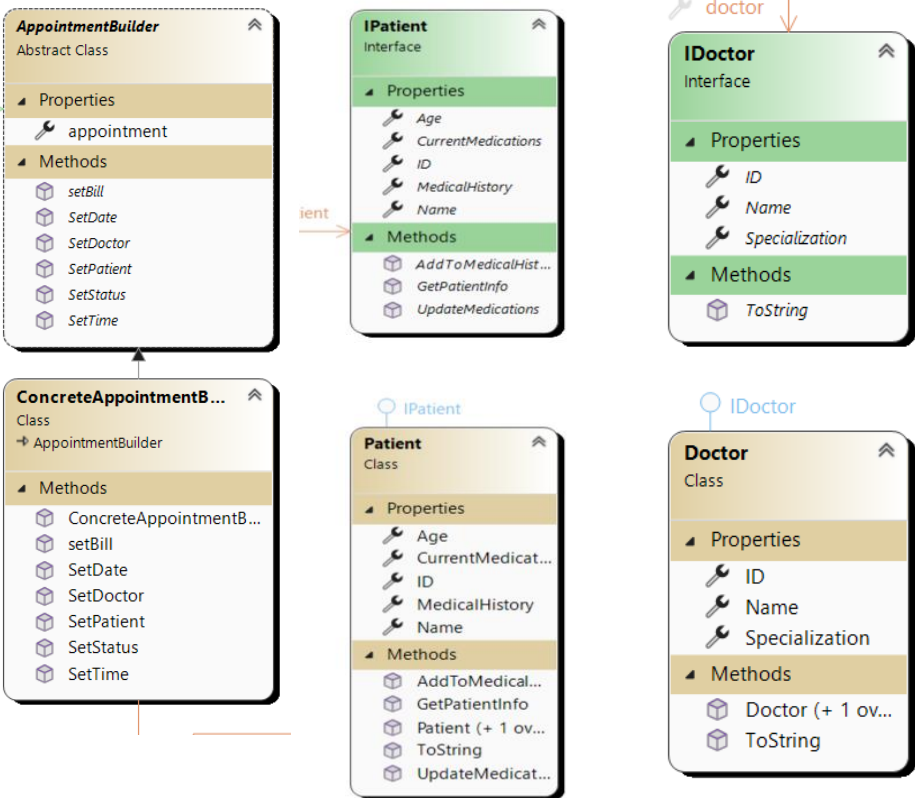
public class NormalBillingStrategy : IBillingStrategy
{
    public decimal CalculateBill(decimal amount)
    {
        return amount;
    }
}

public class DiscountBillingStrategy : IBillingStrategy
{
    public decimal CalculateBill(decimal amount)
    {
        return amount* 0.9M;
    }
}

```

10. Interface with concrete implementations (Strategy)

Descriptions of how and where you have used key concepts

Concept	<p>Description (around 30 words each) refer to the class diagram and source code figures as you see fit</p>
<p>Abstraction (abstract classes and/or interfaces)</p>	 <p>These are all interfaces that are implemented in concrete classes. In Addition to the Abstract Builder class.</p>
<p>Concrete classes</p>	 <p>These concrete classes inherit and implement the methods and properties of the interfaces and abstract classes.</p>
<p>Namespaces</p>	<p>namespace HealthManagementSystem</p>
<p>Methods</p>	<p>Project contains a huge amount of function such as CalculateBill() implemented in all the classes that inherit from the IBillingStrategy interface</p>

Properties	Project contains a lot of properties, for example Patient and Doctor have Name and ID properties. Appointment has many properties: patient, doctor, date, time, bill...
Overriding of methods and/or properties	An example of overriding methods. The CalculateBill() method is overridden in the Discounted and Normal billing strategies that implement the IBillingStrategy interface.
Constructor overloading	<pre> public Patient(string name, int age) { Name = name; Age = age; } public Patient(string name, int age, List<string> mHistory, List<string> cMedication): this(name, age) { ID = Guid.NewGuid().ToString(); MedicalHistory = mHistory; CurrentMedications = cMedication; } </pre>
Enumeration (Enum)	<pre> public enum AppointmentStatus { Scheduled, Completed } </pre>
Collections (e.g., Lists)	The Database stores all Appointment Object in a List of Appointments. <pre> private List<NewAppointment> appointments = new List<NewAppointment>(); </pre>
Loops (e.g., foreach)	Looping through Appointments in Database to Print them. <pre> public override string ToString() { if (appointments.Count > 0) { int i = 0; StringBuilder sb = new StringBuilder(); foreach (NewAppointment appointment in appointments) { sb.AppendLine(\$"Appointment {++i}\n"); sb.Append(appointment.ToString()); sb.AppendLine().AppendLine(); } return sb.ToString(); } return "The Database Is Empty!"; } </pre>

SOLID Principle	Description of where the principle is realized and for what purpose (around 50 Words)
1) Single Responsibility	<p>The Patient class is responsible solely for managing patient-related information. It encapsulates all the properties and methods related to a patient, such as their name, age, medical history, and current medications. It doesn't concern itself with appointments, billing, or any other functionality.</p> <p>The Doctor class manages information related to doctors, including their name, ID, specialization. It doesn't deal with patient details or billing, adhering strictly to SRP.</p> <p>The Appointment class is dedicated to managing appointment-related information. It doesn't handle patient or doctor data beyond what is necessary for the appointment, nor does it deal with billing details directly.</p>
2) Open-Closed	<p>The Appointment class is designed in a way that its existing behavior can be extended through inheritance. We can create a derived class without modifying the original class.</p> <p>The Bill class delegates the billing calculation to an IBillingStrategy interface. This design allows the addition of new billing strategies by implementing the IBillingStrategy interface. The core Bill class remains unchanged</p>
3) Liskov Substitution	<p>The Doctor class defines common properties and methods for all doctors, such as Name, ID, Specialization.</p>
4) Interface Segregation	<p>The IPatient interface includes methods and properties related to patient operations. The Patient class implements this interface, ensuring it only depends on methods relevant to patient functionality.</p> <p>The IDoctor interface includes methods and properties specific to doctor operations. The Doctor class implements this interface, ensuring it only focuses on doctor-related tasks without being concerned with patient or billing details.</p> <p>The IBillingStrategy interface includes methods and properties specific to billing operations. The Bill class implements this interface, ensuring it only handles billing-related tasks without being concerned with patient or doctor details.</p>

5) Dependency Inversion	The IBillingStrategy acts as an abstraction for billing strategies. The Bill class depends on this interface rather than concrete implementations.
-------------------------	--

GOF Design Pattern	Description of where the pattern is realized and for what purpose (around 50 Words)
1) Builder Pattern	The Builder Pattern is used to create Appointment objects with various configurations in a controlled, flexible, and readable manner. This pattern ensures that complex objects that require many steps such as the Appointment objects are constructed, reducing the likelihood of errors and improving the maintainability of the code.
2) Singleton	<p>The HealthcareFacility class ensures that there is only one instance managing appointments. This avoids issues like multiple instances trying to manage the same appointments and ensures that all parts of the application use the same data source for appointments.</p> <p>The Database class ensures that there is only one instance handling database connections and operations, conceptually. This can help manage resources efficiently and avoid multiple connections being opened simultaneously.</p>
3) Strategy	The Bill class uses the IBillingStrategy interface which defines the contract for different billing strategies. This abstraction allows for flexible and interchangeable billing algorithms.

User Interface (Windows Forms)

The image displays the user interface of a Health Management System (HMS) in two parts. The left part is the main menu, and the right part is a form for adding patient information.

Main Menu:

- Title: **- HMS - HealthManagementSystem**
- Buttons: **Add Appointment**, **Display Appointments**, **Exit**
- Icon: A heart with a cross inside.

Patient Information Form:

- Title: **H.M.S.**
- Icons: A doctor, a clipboard, and a document.
- Sections:
 - Patient Information:**
 - Name:
 - Age:
 - Add To Medical History: **Add**
 - Add Medication: **Add**
 - Doctor Information:**
 - Doctor Name:
 - Specialization:
 - Appointment Fee:
 - Extra Information:**
 - Appointment Date:
 - Appointment Time:
 - Billing Type:
- Buttons: **Close**, **Add**

Figure n. User Interface caption

Brief Description of UI (around 50 words)

Simple UI, with a main menu that contains 3 buttons. **Add Appointment** opens a new form to fill appointment info which includes small Add button to add info (Add to lists) and a final large add button to create the appointment, **Display Appointments** opens a new form to display all available appointments, and **Exit** to close the program.