Week 1 Programming Assignment: Patient Visit Manager

Course Module: Week 1
Assignment Title: Patient Visit Manager Console Application

Objective

Design and implement a console-based C# application that simulates a basic clinic patient visit management system. The application should support adding, updating, deleting, and searching patient visit records with persistent data storage and basic reporting capabilities.

Project Scope & Description

The Patient Visit Manager allows users to manage clinic visits through the console. Students will build core functionality that covers:
- Managing patient visit records (add/update/delete/search)
- Storing records in a file (CSV/text-based)
- Supporting different types of visits (e.g., Consultation, Follow-up, Emergency)
- Generating reports and statistics
- Simulating user notifications
- Enabling undo/redo of operations

Technical Requirements

1. Core Functionalities

- Add new patient visit record
- Update existing patient visit information
- Delete patient visit record
- Search visits by patient name, doctor name, date, or type

2. Visit Management

- Categorize by visit type
- Each visit includes:
  - Patient Name
  - Visit Date
  - Visit Type
  - Description/Notes
  - Doctor Name (optional)

3. File-Based Data Storage

- Use .csv or .txt files
- Save data between runs
- Handle file I/O exceptions gracefully

4. Reporting Features

- Individual visit summaries
- Total visit count per type
- Weekly visit summary

5. System Features

- Simulated notification system (e.g., “Visit added!”)
- Undo/Redo Functionality:
  - Maintain a command history for Add, Update, and Delete actions
  - Support undoing the last 10 actions
  - Allow redoing undone actions if no new action has replaced them
  - Use appropriate data structures (e.g., stacks) to manage history
- Generate 300–500 mock entries for testing/reporting

Learning Objectives

- Practice C# syntax
- Apply OOP principles
- Work with file I/O
- Use collections (List, Dictionary)
- Implement structured logic
- Include error handling and validation

Deliverables

- Console Application

Submission Guidelines

- Upload to Azure DevOps Repository
- Branch Name: patient-visit-manager
- Submission Date: 31st July, 2025 (Day End)
- Please use **.gitignore** file to exclude unnecessary file from getting uploaded in the repo
- Late submission policy applies

🔧 Extension Tasks

The following tasks are designed to make the Patient Visit Manager more realistic and challenging.

1. Time-Slot Conflict Detection

- Prevent overlapping appointments for the same patient within a 30-minute window. Display a warning and optionally allow override.
- **Implementation Guide:**
  - When adding a new visit, manually loop through the existing visit list.
  - For each visit, check:
  - If the `PatientName` matches
  - If the `VisitDateTime` is within ±30 minutes of the new visit
  - If a conflict is detected, print a warning:
    - "Warning: This patient has another visit within 30 minutes. Proceed? (Y/N)"
    - Read user input and proceed or cancel accordingly.

2. Filtering and Sorting

- Allow filtering and sorting of visit records via menus.
- **Implementation Guide:**
  - Prompt users with filter options:
    - Filter by doctor name, type, or date range

3. Role-Based Console Access

- Create roles: Admin (full access) and Receptionist (limited to add/search/report). Prompt for login with hardcoded credentials.
- **Implementation Guide:**
  - Create an enum for `UserRole { Admin, Receptionist }`
  - At program start, ask for role selection (hardcoded username/password).
  - Based on role, show or restrict options:
    - Admin sees full menu
    - Receptionist sees limited menu (e.g., only Add/Search/Report)

4. User Activity Logging

- Maintain a log file that records all user actions with timestamps and outcome status (success/failure).
- **Implementation Guide:**
  - Create or append to `activity_log` file.

5. Visit Duration and Fee Calculation

- Add visit duration and calculate fees based on type. Load rates from a JSON config file for easy updates.
- **Implementation Guide:**
  - Add `DurationInMinutes` property to `Visit` class.
  - Store fee rules in a `fees.json` file:

```
{
  "Consultation": 500,
  "Follow-up": 300,
  "Emergency": 1000
}
```

Advanced Design Extension Tasks (SOLID Principles)

1. Separate Core Responsibilities into Focused Classes

Enhance the maintainability and scalability of your application by organizing your code into specialized classes, each responsible for a single area of functionality. For example:

- Create a separate class Visit Manager that handles business logic related to visits (add, update, delete, validations, etc.).
- Create a separate class for I/O operations that deals only with saving, reading, and updating visits in the file system (CSV or text file).
- Create a separate class for notifications that handles all user messages such as "Visit added!" or conflict warnings, etc.
- Create a separate class to generate various reports and summaries.

Goal: Avoid putting too much logic into one class (especially Program.cs), which makes the system rigid and harder to test or extend in the future.

2. Introduce Interfaces for Pluggable Components

To improve flexibility and future-proof your system, define interfaces for the major components that your application relies on. Some examples include:

- IVisitRepository: Abstracts how visit data is stored or retrieved.
- ILogger: Represents any logging mechanism (e.g., file logger, console logger).
- INotificationService: Defines methods to inform the user (e.g., through console messages or a potential future UI).

Task: Update your main logic classes to rely on these interfaces rather than directly on their implementations. Then, use constructor injection (just like we studied dependency inversion) to plug in the appropriate classes.

Benefit: You can later replace a text file with a database, or replace console output with UI alerts, without changing the rest of the application logic.

3. Implement Visit Types as Specialized Classes

Rather than handling visit types (Consultation, Emergency, Follow-up) using if/else conditions or strings, model them as distinct classes:

- Create an abstract Visit base class or IVisit interface.
- Implement specific visit types such as ConsultationVisit, FollowUpVisit, and EmergencyVisit.

Each specialized class can encapsulate logic specific to that visit type, such as:

- Custom validation rules
- Fee calculation logic
- Behavior under specific conditions (e.g., urgency level for emergencies)

Advantage: This design encourages extensibility. If a new visit type needs to be added in the future, it can be done with minimal changes to existing code.

4. Make Role-Based Access Explicit via Services

Currently, role-based access may be managed through conditional checks. Instead, structure this behavior using dedicated service classes:

- Create AdminService to expose all functionalities: add, update, delete, search, and reporting.
- Create ReceptionistService that limits functionality to only add, search, and generate reports.

Each service can internally use shared logic (e.g., from VisitManager) but restrict what is accessible to the user based on their role.

Why this matters: Keeping role logic separated enhances security, reduces the risk of exposing unauthorized operations, and simplifies permission management.

*Note: It is not expected for you to complete all of the tasks mentioned above. However, implementation of SOLID principles is mandatory and should be clearly demonstrated in your submissions.