# Team 2 Project

**Project Scenario: Health Care Application Development - Patient Management System**

For the purpose of this Team 2 project, we have selected the task of developing a Python-based desktop software solution, namely, a patient management system for using in the healthcare sector aimed at providing functionality to manage patient records, schedule appointments, and offer basic diagnostic tools. The system is required to allow patients to book appointments and provide healthcare providers with an easy-to-use interface to manage their schedules efficiently.

**Patient Management Software Solutions Benefits**

* Improves efficiency by reducing manual paperwork.
* Ensures better accuracy in handling patient data.
* Enhances patient care through accessible records and streamlined processes.
* Saves time for staff by automating routine tasks.

## Development Process

Our Team 2 has decided on the following development process based on our selected scenario, covering all recommended steps of development:

### Requirement Gathering and Analysis

* **Purpose**: Identify the needs of stakeholders (patients, healthcare providers, administrators).
* Features to include:
  + Patient record management.
  + Appointment booking and scheduling.
  + Authentication for users.
  + Diagnostic tools (e.g., BMI calculator).
  + Notifications (email/SMS).
* Techniques:
  + Conduct interviews/surveys to understand user expectations.
  + Research existing systems to identify pain points and areas for improvement.
  + Define functional and non-functional requirements.
* **Outcome**: Detailed Software Requirements Specification (SRS) document.

### Design Phase: Use Case Diagram

A use case diagram outlines interactions between actors (e.g., patients, providers, system) and the system’s features.

Here we will draw a diagram, which will include representations of the following information:

* Actors:
  + Patient: Manages personal data, books appointments, views diagnostic results.
  + Healthcare Provider: Manages appointments, views patient records.
  + System Admin: Oversees database and system functionality.
* Use Cases:
  + Register/Login.
  + Manage patient records (CRUD: Create, Read, Update, Delete).
  + Schedule appointments.
  + Send notifications (email/SMS).

### Technology Stack Selection

Programming Language: Python.

**Frontend**: Tkinter for desktop GUI.

**Backend**: SQLite for local data storage.

Libraries:

* Werkzeug: For password hashing (authentication).
* tkcalendar: For calendar widgets.
* Matplotlib: For health data visualisations.
* smtplib/Twilio: For notifications.

**Packaging**: PyInstaller for standalone executables.

**Testing**: Pytest for unit tests, Unittest for integration.

### Development

Patient Record Management:

* Design an SQLite database schema with fields: name, age, gender, contact, medical history.
* Use Tkinter forms to add, view, edit, and delete records.

Appointment Scheduling:

* Create a calendar-based UI using tkcalendar.
* Prevent overlapping appointments by checking the database for conflicts.

Authentication:

* Hash passwords using Werkzeug.
* Create login and registration forms.

Notifications:

* Use Python’s smtplib to send email reminders or Twilio API for SMS.

Diagnostic Tools:

* Implement BMI calculators and other trackers using formulas.
* Use Matplotlib for visual health insights.

### Testing

* Unit Testing:
  + Validate individual modules (e.g., database operations, notification system).
* Integration Testing:
  + Test workflows like booking appointments and notification triggers.
* UI Testing:
  + Simulate user interactions in the GUI to identify usability issues.

### Deployment

* Use **PyInstaller** to package the Python app into an executable for distribution.
* Test the packaged application on target platforms (Windows, macOS, Linux).

### Configure Monitoring Systems

* Add logging for errors and usage statistics.
* Use libraries like logging to capture data into log files.
* Set up alert mechanisms (e.g., emails) for critical failures.

### Training and User Adoption

* Create a user manual explaining the app's functionality.
* Offer workshops or tutorials for healthcare providers and admins.
* Collect user feedback to refine the interface.

### Maintenance and Updates

* Monitor the application for bugs and patch them promptly.
* Release updates to improve performance, add features, and ensure compatibility with new systems.

### Customisation and Expansion

* Customisation:
  + Allow users to personalise the interface (e.g., themes, provider-specific features).
* Expansion:
  + Add networking for multi-user access across locations.
  + Integrate advanced diagnostic tools or AI-based health insights.

## Software Requirements Specification (SRS)

1. Introduction

* **Purpose:** The purpose of this desktop application is to enhance healthcare service delivery by providing a Python-based solution for patient record management, appointment scheduling, and basic diagnostic tools. The application aims to streamline healthcare operations for patients and providers.
* **Scope:** This desktop application will allow patients to book appointments and access diagnostic tools, while healthcare providers will manage patient data and schedules on their desktop computers. The application will function offline for core features and sync data when online.
* Definitions, Acronyms, and Abbreviations:
  + BMI: Body Mass Index
  + UI: User Interface
  + PyQt/Tkinter: Python libraries for creating graphical user interfaces
* **References:** References to Python libraries, healthcare data standards, and development tools will be included as needed.

2. Overall Description

* **Product Perspective:** The application is a standalone desktop software developed in Python, running locally on user machines.
* Product Features:
  + Patient record management
  + Appointment scheduling and booking
  + Diagnostic tools (e.g., BMI calculator, blood pressure tracker)
  + Notifications system via email or local desktop alerts
* User Classes and Characteristics:
  + **Patients:** Expect a simple user interface to book appointments and use diagnostic tools.
  + **Healthcare Providers:** Require robust tools to manage records and schedules efficiently.
* Operating Environment:
  + Desktop application for Windows, macOS, or Linux.
  + Requires Python runtime and relevant dependencies.
* Design Constraints:
  + Develop with PyQt or Tkinter for a responsive, user-friendly interface.
  + Optionally, data should sync with an online database when the system is connected to the internet.
* Assumptions and Dependencies:
  + Users will have basic desktop knowledge and necessary system requirements.

3. Specific Requirements

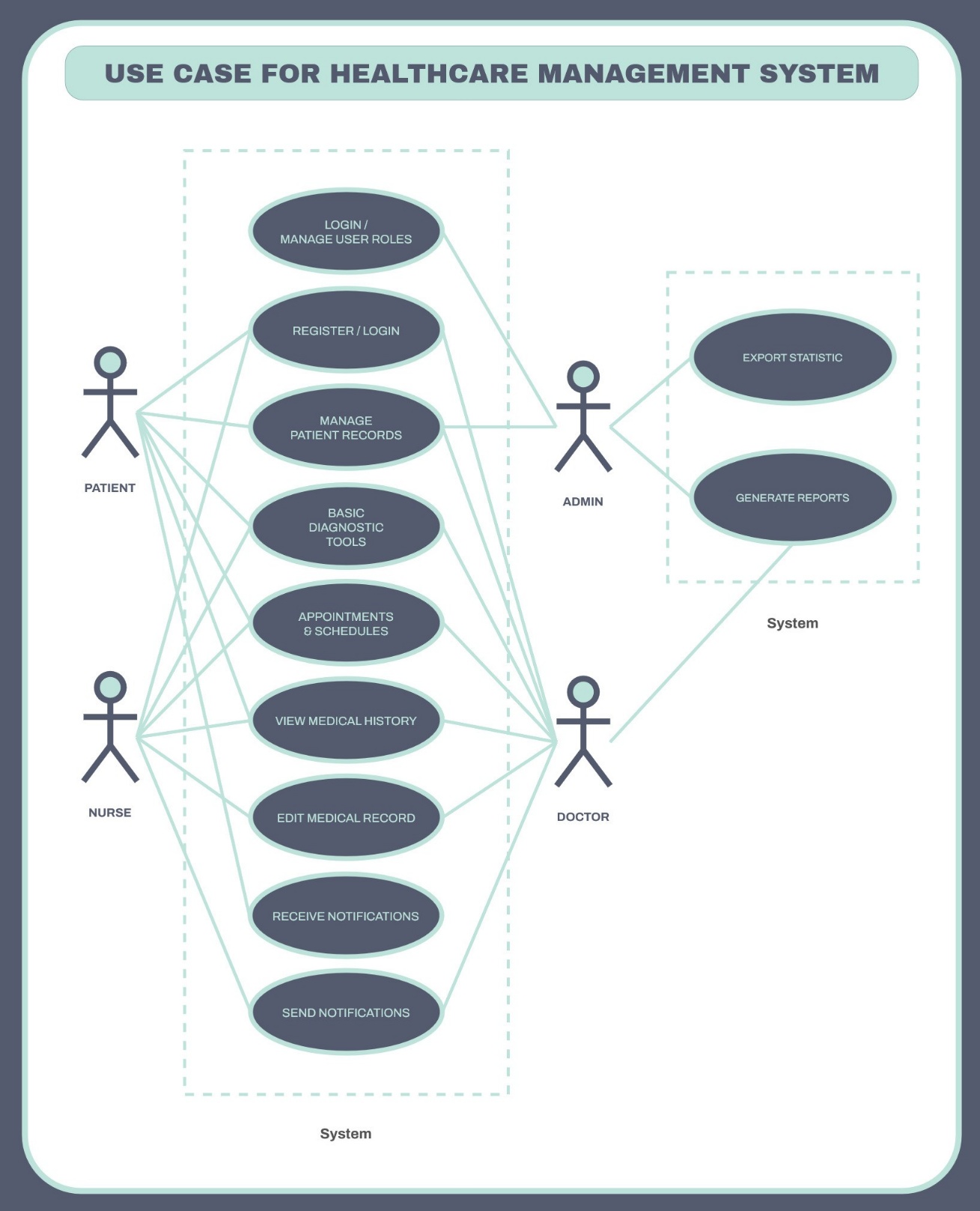
* Functional Requirements:
  + Patient Management:
    - Add, edit, view, and delete patient records.
    - Each record will include fields for name, age, gender, contact details, and medical history.
  + User Registration and Authentication:
    - Provide secure login for both patients and healthcare providers.
    - Implement encryption for passwords and sensitive data storage.
  + Appointment Management:
    - Patients can book appointments via the app.
    - Healthcare providers can manage their schedules through a calendar interface.
  + Diagnostic Tools:
    - Include tools such as BMI calculator, blood pressure tracker, and calorie counter for offline functionality.
  + Offline and Sync Features:
    - Allow users to perform most tasks offline.
    - Sync data to a central online database when the system connects to the internet.
* Non-Functional Requirements:
  + **Performance:** The system should operate smoothly on devices meeting minimum hardware specifications.
  + **Usability:** Focus on an intuitive and visually appealing design.
  + **Security:** Ensure local data encryption and secure syncing processes.
* External Interfaces:
  + **Email API:** Enable email notifications for appointments and other reminders.
  + **Calendar Integration:** Provide a built-in calendar view for appointment management.

4. Appendices

* Desktop application architecture diagrams
* Sample user interface mockups
* Patient data schema and synchronization logic

## Design Phase: Use Case Diagram

Use Case Diagram helps to visually represent the interactions between users (actors) and the system. We have developed the following Use Case Diagram for our scenario of the **Patient Management System**.



Use Case Diagram

Use Case Diagram Conventions

* Actors: Patient, Healthcare Provider, and System Administrator placed outside the system boundary.
* Use Cases: Use cases/functionalities are represented as ovals within the system boundary.
* Connections:
  + **Patient** → Use cases such as "Appointments," "Diagnostic Tools."
  + **Healthcare Provider** → Use cases such as "Manage Patient Records," "Schedules."
  + **System Administrator** → Use cases such as "Manage User Roles."

Diagram Overview

Actors

1. **Patient** – Interacts with the system to provide details or view information. Registers, logs in, manages appointments, and uses diagnostic tools.
2. **Healthcare Provider** – Manages patient records, schedules appointments, views schedules, and processes updates. Also views/records medical history.
3. **System Administrator** – Manages user roles and system maintenance (optional).

Use Cases

1. Manage User Roles
   * System Administrators manage overall user roles, provide support such as onboarding, password reminders, security, etc.
2. User Registration and Authentication
   * Secure login for patients and providers.
3. Manage Patient Records
   * Add, edit, view, and delete patient records. Different users have different levels of access.
4. Basic Diagnostic Tools
   * Patients access tools like BMI calculator and blood pressure tracker.
5. Appointment Booking
   * Patients book appointments and view schedules.
6. Appointment Scheduling
   * Providers schedule and manage appointments.
7. View Medical History
   * Patients view simplified history, whilst providers may view detailed history.
8. Edit Medical Record
   * Providers may edit medical records.
9. Receive Notifications
   * Patients receive email or SMS reminders for appointments.
10. Send Notifications
    * Providers send email or SMS reminders for appointments.
11. Statistics
    * System administrators manage stats and share with relevant health authorities (if applicable).
12. Reports
    * System administrators and providers may generate individual patient reports.

## Detailed Technology Stack Selection & Functionality Roadmap

A detailed roadmap for Team 2’s Health Care Application Development specifically tailored for a **desktop application** version using **Python, SQLite** and **Tkinter**:

1. Setup and Environment

* **Language**: Python
* **Framework**: Tkinter for the graphical user interface (GUI).
* **Database**: SQLite for local storage.
* Libraries:
  + **Werkzeug**: For password hashing.
  + **Pandas/Numpy**: For basic diagnostic tools and calculations.
  + **SMTP** or **Twilio**: For notifications.
  + **Matplotlib**: For data visualisation (e.g., health trends).

2. Key Features

Patient Management

* **Database Schema**: Create a table in SQLite for patient records with fields such as name, age, gender, contact details, and medical history.
* CRUD Operations:
  + Implement forms for adding/editing patient details using Tkinter’s Entry widget.
  + Display patient records in a Tkinter Treeview widget for easy viewing and navigation.

User Registration and Authentication

* Local Authentication:
  + Create login and registration forms with Tkinter.
  + Hash passwords using **Werkzeug** or Python’s bcrypt library.
  + Separate login portals for patients and healthcare providers.

Appointment Scheduling

* **Logic**: Use timestamps and a scheduling algorithm to prevent overlapping appointments.
* Calendar Widget:
  + Use Tkinter’s Calendar or third-party libraries like **tkcalendar** to show available slots.

Appointment Booking

* **Patient View**: Create a form for patients to browse available time slots and book appointments.
* **Provider View**: Build a dashboard for providers to view and manage bookings.

Calendar Integration

* **Calendar Feature**: Use **tkcalendar** or integrate APIs like Google Calendar for syncing schedules.

Notifications

* Implement email notifications using Python’s smtplib library.
* Integrate SMS reminders via Twilio.

Basic Diagnostic Tools

* BMI Calculator:
  + Input: Height and weight.
  + Output: BMI value and recommendations.
* Blood Pressure Tracker:
  + Allow users to input readings manually and store values.
  + Plot trends using **Matplotlib**.
* Calorie Counter:
  + Create a simple form for users to input meals and calculate total calorie intake.

3. User Interface

* Use **Tkinter** to design the interface:
  + **Menu Bar**: Add menus for navigation (e.g., File, Edit, Help).
  + **Tabs**: Implement a tabbed interface using ttk.Notebook for sections like Patient Management, Appointment Scheduling, and Diagnostic Tools.
  + **Buttons and Forms**: Make intuitive forms for data entry and buttons for actions like “Save” or “Delete.”

4. Testing and Validation

* Write **unit tests** for individual components (e.g., patient record management, appointment logic).
* Perform **integration tests** to ensure end-to-end workflows function correctly.
* Validate inputs (e.g., prevent invalid appointment times, ensure secure passwords).

5. Deployment

* Package the application as a standalone executable using **PyInstaller** or **cx\_Freeze**.
* Test the application on target operating systems (Windows, macOS, Linux).

6. Documentation

* Create a user manual detailing how patients and healthcare providers can use the application.
* Document the codebase with comments and explanations for easier maintenance.

## Development

Patient Management System functionality using Python, Tkinter and SQLite.

### Patient Record Management

#### Initial Idea & Attempt

**SQLite Database Schema**

First attempt at creating the SQLite database schema for **Patient Record Management** with fields: name, age, gender, contact, medical history:

|  |  |  |
| --- | --- | --- |
| SQLite database schema idea |  | Comments |
|  |  |  |
| CREATE TABLE IF NOT EXISTS patients ( |  |  |
| ID INTEGER PRIMARY KEY AUTOINCREMENT, |  | -- Unique ID for each patient |
| name TEXT NOT NULL, |  | -- Patient's full name |
| age INTEGER NOT NULL, |  | -- Patient's age |
| gender TEXT, |  | -- Gender |
| contact TEXT NOT NULL UNIQUE, |  | -- Contact details (unique) |
| medical\_history TEXT |  | -- Medical history as a text field |
| ); |  |  |

1. Setting Up SQLite Database

Python function to create the database and table:

import sqlite3

def setup\_database():

    # Connect to SQLite database (or create if it doesn't exist)

    conn = sqlite3.connect("healthcare.db")

    # Create a cursor object to execute SQL commands

    cursor = conn.cursor()

    # Create a table for patients

    cursor.execute("""CREATE TABLE IF NOT EXISTS patients (

        ID INTEGER PRIMARY KEY AUTOINCREMENT,

        name TEXT NOT NULL,

        age INTEGER NOT NULL,

        gender TEXT,

        contact TEXT NOT NULL UNIQUE,

        medical\_history TEXT

    )""")

    # Commit the changes and close the connection

    conn.commit()

    conn.close()

2. Tkinter GUI Application

Creating Tkinter forms for **add, view, edit, and delete** functionalities.

a. Import Required Modules

import tkinter as tk

from tkinter import ttk, messagebox

import sqlite3

b. Main GUI Window

root = tk.Tk()

root.title("Patient Management System")

root.geometry("1200x400")

3. Adding Patient Records

def add\_patient():

    conn = sqlite3.connect("healthcare.db")

    cursor = conn.cursor()

    cursor.execute("INSERT INTO patients (name, age, gender, contact, medical\_history) VALUES (?, ?, ?, ?, ?)",

                   (name\_var.get(), age\_var.get(), gender\_var.get(), contact\_var.get(), history\_var.get()))

    conn.commit()

    conn.close()

    messagebox.showinfo("Success", "Patient added successfully!")

4. Viewing Patient Records

# Treeview to display patients

tree = ttk.Treeview(root, columns=("ID", "Name", "Age", "Gender", "Contact", "Medical History"), show="headings")

tree.heading("ID", text="ID")

tree.heading("Name", text="Name")

tree.heading("Age", text="Age")

tree.heading("Gender", text="Gender")

tree.heading("Contact", text="Contact")

tree.heading("Medical History", text="Medical History")

tree.grid(row=6, column=0, columnspan=4)

load\_patients()

def load\_patients():

    conn = sqlite3.connect("healthcare.db")

    cursor = conn.cursor()

    cursor.execute("SELECT \* FROM patients")

    rows = cursor.fetchall()

    conn.close()

    # Clear existing rows

    for row in tree.get\_children():

        tree.delete(row)

    # Populate Treeview with patient records

    for row in rows:

        tree.insert("", tk.END, values=row)

5. Editing Patient Records

def edit\_patient():

    selected = tree.selection()

    if not selected:

        messagebox.showwarning("Warning", "Please select a patient to edit.")

        return

    conn = sqlite3.connect("healthcare.db")

    cursor = conn.cursor()

    cursor.execute("""

        UPDATE patients

        SET name=?, age=?, gender=?, contact=?, medical\_history=?

        WHERE id=?

    """, (name\_var.get(), age\_var.get(), gender\_var.get(), contact\_var.get(), history\_var.get(), tree.item(selected[0])["values"][0]))

    conn.commit()

    conn.close()

    messagebox.showinfo("Success", "Patient updated successfully!")

6. Deleting Patient Records

def delete\_patient():

    selected = tree.selection()

    if not selected:

        messagebox.showwarning("Warning", "Please select a patient to delete.")

        return

    conn = sqlite3.connect("healthcare.db")

    cursor = conn.cursor()

    cursor.execute("DELETE FROM patients WHERE id=?", (tree.item(selected[0])["values"][0],))

    conn.commit()

    conn.close()

    messagebox.showinfo("Success", "Patient deleted successfully!")

7. Clear Fields

def clear\_fields():

    name\_var.set("")

    age\_var.set("")

    gender\_var.set("")

    contact\_var.set("")

    history\_var.set("")

8. Form Fields

tk.Label(root, text="Name").grid(row=0, column=0)

tk.Entry(root, textvariable=name\_var).grid(row=0, column=1)

tk.Label(root, text="Age").grid(row=1, column=0)

tk.Entry(root, textvariable=age\_var).grid(row=1, column=1)

tk.Label(root, text="Gender").grid(row=2, column=0)

tk.Entry(root, textvariable=gender\_var).grid(row=2, column=1)

tk.Label(root, text="Contact").grid(row=3, column=0)

tk.Entry(root, textvariable=contact\_var).grid(row=3, column=1)

tk.Label(root, text="Medical History").grid(row=4, column=0)

tk.Entry(root, textvariable=history\_var).grid(row=4, column=1)

8. Action Buttons

tk.Button(root, text="Add Patient", command=add\_patient).grid(row=5, column=0)

tk.Button(root, text="Edit Patient", command=edit\_patient).grid(row=5, column=1)

tk.Button(root, text="Delete Patient", command=delete\_patient).grid(row=5, column=2)

tk.Button(root, text="Clear Fields", command=clear\_fields).grid(row=5, column=3)

A screenshot of a computer error

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Outcome of first attempt at creating the Patient Records Management form.

A screenshot of a computer error

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Data got deleted without warning when trying to use the Edit Patient button.

A screenshot of a computer error

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Delete function works but deletes record without warning.

This initial interface creates and/or connects with the database for patient data handling. So it works, but not robust enough as it allows blank additions, accidental deletions and confusing to edit as it lacks visual cues.

#### Improving the Initial Idea

**Enhance Validation:** Add checks to ensure fields are filled correctly (e.g., valid age, contact format).

**Search Feature:** Implement a search bar to filter patient records dynamically.

**Export Data:** Include functionality to export patient records to CSV or Excel.

**Advanced UI:** Use a Python GUI library like PyQt for a more modern and sleek interface.

1. Enhanced Validation

We'll validate the input fields to ensure users provide correct and meaningful data, this will ensure data integrity.

# Function to validate fields before adding or editing a patient

# This function checks if the input fields are filled correctly and shows error messages if not.

def validate\_fields():

    if not name\_var.get().strip():

        messagebox.showerror("Error", "Name is required!")

        return False

    if not age\_var.get().isdigit() or int(age\_var.get()) <= 0:

        messagebox.showerror("Error", "Age must be a positive number!")

        return False

    if gender\_var.get().strip().lower() not in ("male", "female", "other"):

        messagebox.showerror("Error", "Gender must be 'Male', 'Female', or 'Other'!")

        return False

    if not contact\_var.get().strip():

        messagebox.showerror("Error", "Contact is required!")

        return False

    return True

Modifying add\_patient() and edit\_patient() with following logic to validate inputs:

    if not validate\_fields():

        return

Adding Patient works well. Editing requires visual cue. Deletion requires visual cue and confirmation.

Adding Visual Cue

When selecting a record in the Tree View rows the data from the selected record should populate the form to provide visual cue as to what is being edited or deleted.

Functionality to Populate the Form for visual cue:

def populate\_fields(event):

    # Get the selected row

    selected = tree.selection()

    if selected:

        # Get the values from the selected row

        values = tree.item(selected[0], "values")

        # Populate the form fields with the selected row's data

        name\_var.set(values[1])

        age\_var.set(values[2])

        gender\_var.set(values[3])

        contact\_var.set(values[4])

        history\_var.set(values[5])

# Binding the Treeview selection event to the populate\_fields() function

tree.bind("<<TreeviewSelect>>", populate\_fields)

Implementing a confirmation dialog in delete\_patient() to display the details of the record being deleted and asking the user for confirmation:

# Get the selected row's values

values = tree.item(selected[0], "values")

record\_details = f"ID: {values[0]}\nName: {values[1]}\nAge: {values[2]}\nGender: {values[3]}\nContact: {values[4]}\nMedical History: {values[5]}"

# Display confirmation dialog

confirm = messagebox.askyesno("Confirm Deletion", f"Are you sure you want to delete the following record?\n\n{record\_details}")

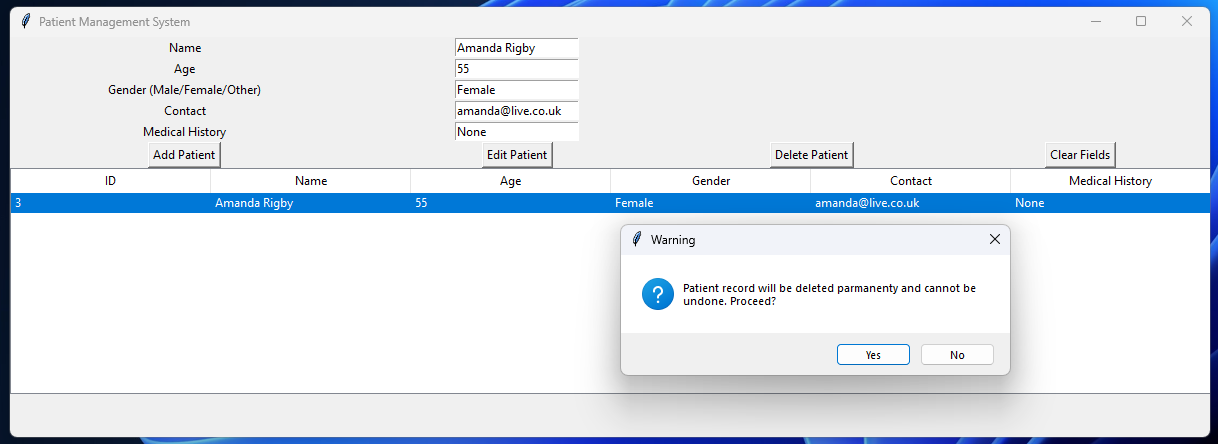
if not confirm:

    return

A screenshot of a computer

AI-generated content may be incorrect.

Confirmation dialogue ensures the user is deleting the intended record, if not user can cancel the action.



If user accidentally selected yes, the first time for record deletion, this warning will remind that the deletion is permanent and cannot be undone, providing second chance of minimising errors.

All actions are working as intended except the **Add Patient** function when a record is selected in the Tree View and the form is populated but remains unchanged.

Disabling <Add Patient> button when an existing record is selected

To prevent the **Add Patient** button from being clicked with the same data when a record is selected and unchanged, we will compare the form data with the selected record's original data. If the data in the form matches the selected record's data, the **Add Patient** button will be disabled. Only if the data changes, the button will be re-enabled.

This requires implementing quite an advanced comparison and validation process. For the sake of user experience and data consistency we will attempt to implement it.

a. Track Selected Record

Need a variable to store the original data of the selected record:

original\_record = None  # Global variable to track selected record data

Modifying the populate\_fields() function to store the selected record’s data:

b. Compare Data

A new function to compare the current form data with the original record:

def check\_data\_changed(\*args):

    global original\_record

    if not original\_record:

        add\_button.config(state="normal")  # Enable Add button if no record is selected

        return

    # Compare the form data with the original record

    if (name\_var.get() == original\_record[1] and

        age\_var.get() == original\_record[2] and

        gender\_var.get() == original\_record[3] and

        contact\_var.get() == original\_record[4] and

        history\_var.get() == original\_record[5]):

        add\_button.config(state="disabled")  # Disable Add button if data hasn't changed

    else:

        add\_button.config(state="normal")  # Enable Add button if data changes

Add Patient button is referenced with a variable ‘add\_button’

add\_button = tk.Button(root, text="Add Patient", command=add\_patient).grid(row=5, column=0)

c. Bind Validation

Using the trace method to bind changes in form fields to the check\_data\_changed() function:

name\_var.trace("w", check\_data\_changed)

age\_var.trace("w", check\_data\_changed)

gender\_var.trace("w", check\_data\_changed)

contact\_var.trace("w", check\_data\_changed)

history\_var.trace("w", check\_data\_changed)

A screenshot of a computer

AI-generated content may be incorrect.

Also need to restrict addition of New Patient if contact details are not unique, so adding a duplicate check in the add\_patient() function. This is required for unavoidable situations when Add Patient button becomes or remains enabled. Checking the database for duplicates is resource intensive compared to disabling the button.

    # Check for duplicate Contact value

    cursor.execute("SELECT COUNT(\*) FROM patients WHERE contact = ?", (contact\_var.get(),))

    duplicate\_count = cursor.fetchone()[0]

    if duplicate\_count > 0:

        messagebox.showerror("Error", "A patient with this contact already exists!")

        conn.close()

        return

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

To improve user experience Gender Entry box will be replaced with a dropdown menu with available options:

# Options for the dropdown

gender\_options = ["Male", "Female", "Other"]

gender\_var.set("Select Gender")  # Set the default value

# Create a dropdown menu for gender

gender\_dropdown = tk.OptionMenu(root, gender\_var, \*gender\_options)

gender\_dropdown.grid(row=2, column=1)

Validate function validate\_fields() updated with:

    if gender\_var.get() not in ("Male", "Female", "Other"):

        messagebox.showerror("Error", "Please select a valid gender from the dropdown!")

        return False

A screenshot of a computer

AI-generated content may be incorrect.

For additional UX functionality a search bar is added.

Search Functionality:

def search\_patients():

query = search\_var.get().strip().lower()

for row in tree.get\_children():

tree.delete(row)

conn = sqlite3.connect("healthcare.db")

cursor = conn.cursor()

cursor.execute("SELECT \* FROM patients WHERE LOWER(name) LIKE ?", (f"%{query}%",))

rows = cursor.fetchall()

conn.close()

for row in rows:

tree.insert("", tk.END, values=row)

----

a search bar at the top of the UI:

search\_var = tk.StringVar()

tk.Label(root, text="Search:").grid(row=0, column=4)

tk.Entry(root, textvariable=search\_var).grid(row=0, column=5)

tk.Button(root, text="Search", command=search\_patients).grid(row=0, column=6)

A screenshot of a computer

AI-generated content may be incorrect.

3. Export Data

Allow users to export patient records to a CSV file.

import csv

def export\_data():

conn = sqlite3.connect("healthcare.db")

cursor = conn.cursor()

cursor.execute("SELECT \* FROM patients")

rows = cursor.fetchall()

conn.close()

with open("patients\_export.csv", "w", newline="", encoding="utf-8") as file:

writer = csv.writer(file)

writer.writerow(["ID", "Name", "Age", "Gender", "Contact", "Medical History"])

writer.writerows(rows)

messagebox.showinfo("Success", "Data exported to 'patients\_export.csv'!")

Add an Export Data Button:

tk.Button(root, text="Export Data", command=export\_data).grid(row=5, column=4)

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

4. Advanced UI

For a modern and responsive design, Tkinter interface can be improved by organising components with frames and adding styles.

Using Frames for Layout:

Refactoring the input and button sections into separate frames:

input\_frame = tk.Frame(root, padx=10, pady=10)

input\_frame.grid(row=0, column=0, columnspan=4)

button\_frame = tk.Frame(root, padx=10, pady=10)

button\_frame.grid(row=1, column=0, columnspan=4)

Placing the components into their respective frames:

Appointment Scheduling:

* Create a calendar-based UI using tkcalendar.
* Prevent overlapping appointments by checking the database for conflicts.

Authentication:

* Hash passwords using Werkzeug.
* Create login and registration forms.

Notifications:

* Use Python’s smtplib to send email reminders or Twilio API for SMS.

Diagnostic Tools:

* Implement BMI calculators and other trackers using formulas.
* Use Matplotlib for visual health insights.

## Testing

## Deployment

## Configure monitoring systems to ensure continuous operation.

## Training and User Adoption

## Maintenance and Updates

## Customization and Expansion

3.3 Explain how you would respond to feedback received during a code review, including asking questions for clarification and confirming changes made to the code.

Screenshots or demonstrations of using Git for version control and participating in planning, reflection, and code review processes.