**Object Oriented Programing Project Report**



**Hospital Management System**

**Prepared by**

**at**

**Seneca Polytechnic**

**March 2024**

**Table of Contents**

[Contributions 3](#_Toc162260986)

[I Presentation YouTube link 3](#_Toc162260987)

[II Project Description 4](#_Toc162260988)

[1 Project Overview 4](#_Toc162260989)

[2 The Purpose of the Project 4](#_Toc162260990)

[The User Business or Background of the Project Effort 4](#_Toc162260991)

[3 The Scope of the Work 5](#_Toc162260992)

[III Requirements 5](#_Toc162260993)

[4 Product Use Cases 5](#_Toc162260994)

[5 Product Design Highlights and Descriptions 7](#_Toc162260995)

[IV Design 11](#_Toc162260996)

[6 UML Class Diagram 11](#_Toc162260997)

[7 Data Flow Diagram 12](#_Toc162260998)

[V Testing and Evaluation 12](#_Toc162260999)

[8 Features to be tested. 12](#_Toc162261000)

[9 Pass/Fail Criteria 14](#_Toc162261001)

[VI Project Issues 15](#_Toc162261002)

[10 Lessons Learned 15](#_Toc162261003)

[VII Conclusion 16](#_Toc162261004)

[VIII References 16](#_Toc162261005)

### ****Contributions****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Student ID** | **Name** | **Components** | **Details** | **Development hours** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Presentation YouTube link

# Project Description

## Project Overview

For this project, we have created a Hospital Management System that aims to facilitate efficient management of patient records, appointment scheduling, managing medical staff and procedures within a hospital. It provides a comprehensive solution for organizing patient information, managing appointments, and ensuring seamless communication between medical staff and patients.

## The Purpose of the Project

### The User Business or Background of the Project Effort

Content

In this project, we endeavor to develop a comprehensive hospital management system tailored to the healthcare sector. This system will facilitate the seamless management of patient records, appointment scheduling, medical staff tracking, and medical procedures. By incorporating various STL containers, algorithms, and iterators, we aim to optimize data storage and processing, ensuring efficient operation of the system. The need for such a sophisticated and user-friendly platform arises from the increasing demands and complexities within healthcare administration. Our objective is to deliver a robust solution that empowers healthcare professionals to efficiently manage patient information, streamline administrative tasks, and enhance overall operational efficiency within healthcare facilities.

Motivation

Our motivation for embarking on this project stems from the desire to hone our C++ programming skills while tackling real-world challenges in healthcare management. By undertaking the development of a hospital management system, we seek to gain practical experience in utilizing STL containers, algorithms, and iterators for efficient data handling in a critical domain. Furthermore, we recognize the importance of developing intuitive and reliable software solutions to address the complex needs of healthcare institutions. Through this project, we aim to contribute to the improvement of healthcare services by providing a robust, user-friendly, and scalable hospital management system.

Considerations

It's essential to acknowledge that while our hospital management system aims to address key functionalities such as patient record management, appointment scheduling, staff tracking, and medical procedure handling, it is intentionally simplified and may not encompass all aspects of a comprehensive hospital management system. This simplified approach allows us to focus on core functionalities and optimize the system's performance and usability within the scope of this project. However, it's important to recognize that real-world hospital management systems may involve additional complexities and features, such as billing, inventory management, electronic health records (EHR), and regulatory compliance. Future iterations or expansions of this system could explore incorporating these elements to offer a more comprehensive solution for healthcare institutions.

## The Scope of the Work

The scope of the project encompasses the design, development, and implementation of a Hospital Management System that meets the specified requirements outlined in the project guidelines. This includes the creation of patient class hierarchies, implementation of exception handling mechanisms, integration of STL containers for data storage and manipulation, appointment scheduling functionalities, management of medical staff, and user interface development.

# Requirements

## Product Use Cases

1. **Add Patient**:

* **Actor**: User
* **Description**: Allows users to add a new patient to the system by providing patient details.
* **Preconditions**: User is logged into the system.
* **Postconditions**: New patient is added to the system.
* **Main Flow**:
  1. User selects the "Add patient" option.
  2. User enters patient details such as name, ID, gender, etc.
  3. System validates the input and adds the new patient to the database.
  4. System displays a confirmation message indicating that the patient has been successfully added.

2. **Search Patient**:

* **Actor**: User
* **Description**: Allows users to search for a patient by entering the patient's ID or name.
* **Preconditions**: User is logged into the system.
* **Postconditions**: Patient information is displayed if found; otherwise, an appropriate message is shown.
* **Main Flow**:
  1. User selects the "Search patient" option.
  2. User enters the patient's ID or name.
  3. System retrieves patient information and displays it if found.
  4. If the patient is not found, the system displays a message indicating that the patient does not exist.

3. **Display All Patients**:

* **Actor**: User
* **Description**: Allows users to view a list of all patients currently registered in the system.
* **Preconditions**: User is logged into the system.
* **Postconditions**: List of patients is displayed.
* **Main Flow**:
  1. User selects the "Display all patients" option.
  2. System retrieves the list of all patients from the database.
  3. System displays the list of patients.

4. **Schedule Appointment**:

* **Actor**: User
* **Description**: Allows users to schedule an appointment for a patient with a medical staff member.
* **Preconditions**: User is logged into the system.
* **Postconditions**: Appointment is successfully scheduled.
* **Main Flow**:
  1. User selects the "Schedule appointment" option.
  2. User enters the patient's ID.
  3. User selects the type of medical staff (Doctor/Nurse) and their specialty.
  4. User enters the appointment time.
  5. System validates the input and schedules the appointment.
  6. System displays a confirmation message indicating that the appointment has been scheduled.

5. **Reschedule Appointment**:

* **Actor**: User
* **Description**: Allows users to reschedule an existing appointment for a patient.
* **Preconditions**: User is logged into the system.
* **Postconditions**: Appointment is successfully rescheduled.
* **Main Flow**:
  1. User selects the "Reschedule appointment" option.
  2. User enters the patient's ID.
  3. System displays the patient's existing appointments.
  4. User selects the appointment to be rescheduled.
  5. User enters the new date and time for the appointment.
  6. System validates the input and updates the appointment time.
  7. System displays a confirmation message indicating that the appointment has been rescheduled.

6. **Cancel Appointment**:

* **Actor**: User
* **Description**: Allows users to cancel an existing appointment for a patient.
* **Preconditions**: User is logged into the system.
* **Postconditions**: Appointment is successfully canceled.
* **Main Flow**:
  1. User selects the "Cancel appointment" option.
  2. User enters the patient's ID.
  3. System displays the patient's existing appointments.
  4. User selects the appointment to be canceled.
  5. System cancels the appointment.
  6. System displays a confirmation message indicating that the appointment has been canceled.

7. **Display Booked Appointments**:

* **Actor**: User
* **Description**: Allows users to view all booked appointments.
* **Preconditions**: User is logged into the system.
* **Postconditions**: List of booked appointments is displayed.
* **Main Flow**:
  1. User selects the "Display booked appointments" option.
  2. System retrieves the list of booked appointments.
  3. System displays the list of booked appointments.

8. **Exit**:

* **Actor**: User
* **Description**: Allows users to exit the system.
* **Preconditions**: User is logged into the system.
* **Postconditions**: User exits the system.
* **Main Flow**:
  1. User selects the "Exit" option.
  2. System terminates the application.

## Product Design Highlights and Descriptions

The following highlights the design elements and describes each file and how they work together within the program for the program files: patient.h, patient.cpp, appointment.h, appointment.cpp, medicalstaff.h, medicalstaff.cpp, procedure.h, procedure.cpp, exceptions.h, exceptions.cpp, userinterfaceg1.cpp (our main.cpp file).

Our `patient.h` file serves as the declaration file for the Patient class. It includes necessary header files such as `<iostream>`, `<string>`, and others. Within this file, we declare member variables representing patient information such as name, age, gender, etc. Additionally, member functions such as constructors, destructors, accessors, setters, and other member functions are declared here. We also include a forward declaration of the Appointment class to avoid circular dependencies.

In our `patient.cpp` file, we implement the member functions declared in `patient.h`. Here, we define constructor and destructor implementations, as well as accessors and setters to get and set patient attributes. Member functions like `displayInfo()`, `addAppointment()`, `displayAppointments()`, and `searchById()` are also implemented in this file. Together, these files define the behavior and properties of the Patient class, with `patient.cpp` providing the implementation details for the functions declared in `patient.h`.

Moving on to our `appointment.h` file, it contains the declaration of the Appointment class along with necessary header files such as `<iostream>`, `<string>`, and others. We include a forward declaration of the Patient class to avoid circular dependencies. Within this file, member variables representing appointment information such as patient name, ID, medical staff, appointment time, procedures, and cancellation status are declared. Member functions like constructors, destructors, accessors, and setters are also declared for managing appointment information. Additionally, functions like `appointmentSchedule()`, `cancelAppointment()`, and `appointmentCancel()` are declared for managing appointment scheduling and cancellation.

In our `appointment.cpp` file, we implement the member functions declared in `appointment.h`. Here, we define constructor and destructor implementations, as well as accessors and setters to get and set appointment attributes. Functions like `appointmentSchedule()`, `rescheduleAppointment()`, and `cancelAppointment()` are also implemented in this file to manage appointment scheduling and cancellation. Together, these files define the behavior and properties of the Appointment class, with `appointment.cpp` providing the implementation details for the functions declared in `appointment.h`.

Moving forward to our `medicalstaff.h` file, it contains the declaration of the MedicalStaff, Doctor, Nurse, and MedicalStaffManager classes, along with necessary header files such as `<string>`, `<vector>`, and others. We include a forward declaration of the Appointment class to avoid circular dependencies. Within this file, member variables representing medical staff information such as name, specialization, availability, employee ID, and associated procedures are declared. Member functions like constructors, destructors, display information, and utility functions for managing medical staff are also declared.

In our `medicalstaff.cpp` file, we implement the member functions declared in `medicalstaff.h`. Here, we define constructor and destructor implementations for the MedicalStaff, Doctor, and Nurse classes. Member functions to display information, check availability, get procedures, add procedures, and set appointments for medical staff are implemented here. Additionally, the MedicalStaffManager class implementation includes static methods to assign and find medical staff based on specialization. Together, these files define the behavior and properties of the medical staff classes (MedicalStaff, Doctor, and Nurse) and the manager class (MedicalStaffManager), with `medicalstaff.cpp` providing the implementation details for the functions declared in `medicalstaff.h`.

Moving on to our `procedure.h` file, it contains the declaration of the Procedure class along with necessary header files such as `<string>` and `<vector>`. The Procedure class represents a medical procedure with attributes such as name and description. Constructor, getter methods (`getName()` and `getDescription()`), and setter methods (`setName()` and `setDescription()`) are declared for managing procedure attributes.

In our `procedure.cpp` file, we implement the member functions declared in `procedure.h`. Here, we define constructor and getter/setter method implementations for the Procedure class. Additionally, it includes a function `initializeProcedureList()` that initializes and returns a map of procedures categorized by specialization for doctors and nurses. This function initializes a map of procedures categorized by medical staff specialization, populating the map with predefined procedures for different specializations such as Emergency Physician, Pulmonologist, Gastroenterologist, etc., for both doctors and nurses. Together, these files provide a way to organize and manage medical procedures within the system.

Our `exceptions.h` file defines several exception classes such as `BaseException`, `InvalidAgeException`, `AppointmentConflictException`, `InvalidNameException`, `InvalidIDException`, `GenderException`, `PhoneNumberException`, `FieldNotEmptyException`, `BoolMustBeSetException`, and `InvalidAppointmentTimeFormat`. Each exception class has static methods for handling specific error conditions related to age, appointment conflicts, names, IDs, gender, phone numbers, fields not being empty, boolean flags, and appointment time formats. Additionally, it provides a declaration for the `confirmationException` function, which is used for confirming information with the user.

In our `exceptions.cpp` file, we implement the functionality of the exception classes and utility functions declared in `exceptions.h`. Here, we define the logic for checking the validity of age, names, IDs, gender, phone numbers, and other data types. We handle error reporting by throwing appropriate exceptions and displaying error messages. Additionally, we provide confirmation prompts for validating user inputs. Overall, these files help ensure robust error handling and user input validation within our program, enhancing its reliability and usability.

Our ‘userinterfaceg1.cpp’ serves as our `main.cpp` and is the central hub of our hospital management system, coordinating user interactions and integrating various functionalities provided by other modules. Let's break down its design and highlight its features:

Integration with Other Files:

1. \*\*Header File Inclusion\*\*: We include necessary standard library headers (`iostream`, `vector`, `string`, etc.) and headers for user-defined classes (`exceptions.h`, `medicalStaff.h`, `procedure.h`, `patient.h`, `appointment.h`).

2. \*\*Function Prototypes\*\*: We declare function prototypes for functions defined later in the file and implemented in other files. These prototypes ensure that functions can be called before their definitions.

3. \*\*Function Definitions\*\*: We define functions such as `displayMenu`, `userInput`, and utility functions like `searchPatientById`, `isValidAppointmentTimeFormat`, `and `displayBookedAppointments`. These functions handle user input, display menus, and interact with other modules to perform tasks like adding patients, scheduling appointments, and displaying information.

4. \*\*Main Function\*\*: In our `main` function, we initialize lists for appointments and patients, then call the `userInput` function to start the interactive menu-driven interface. This function acts as the entry point of the program, orchestrating the flow of control and ensuring the smooth execution of the hospital management system.

Design Choices and Highlights:

1. \*\*Modularization\*\*: Our code is well-structured and modular, with functionalities separated into different modules (files) like `exceptions.h`, `medicalStaff.h`, etc. This modular design promotes code organization, reusability, and maintainability.

2. \*\*Exception Handling\*\*: We implement robust exception handling using custom exception classes defined in `exceptions.h`. This ensures that errors and exceptional conditions are properly handled, enhancing the reliability and robustness of our system.

3. \*\*Object-Oriented Design\*\*: We leverage object-oriented principles by defining classes like `Patient`, `Appointment`, and `MedicalStaff`, encapsulating data and behavior within these classes. This promotes code clarity, encapsulation, and code reuse.

4. \*\*Input Validation\*\*: User inputs are validated using exception handling and custom validation functions (`isValidAppointmentTimeFormat`). This ensures data integrity and prevents invalid inputs, enhancing the reliability of our system.

5. \*\*STL Integration\*\*: We utilize Standard Library containers like `list` and `vector` for managing collections of appointments, patients, etc. This leverages the functionality provided by the STL, improving code efficiency and readability.

6. \*\*Memory Management\*\*: Smart pointers (`unique\_ptr`) are used to manage dynamically allocated objects like `Patient`, ensuring proper memory management and reducing the risk of memory leaks.

7. \*\*User Interaction\*\*: The interactive menu-driven interface (`displayMenu`, `userInput`) enables users to interact with the system easily, providing options to perform various tasks like adding patients, scheduling appointments, etc.

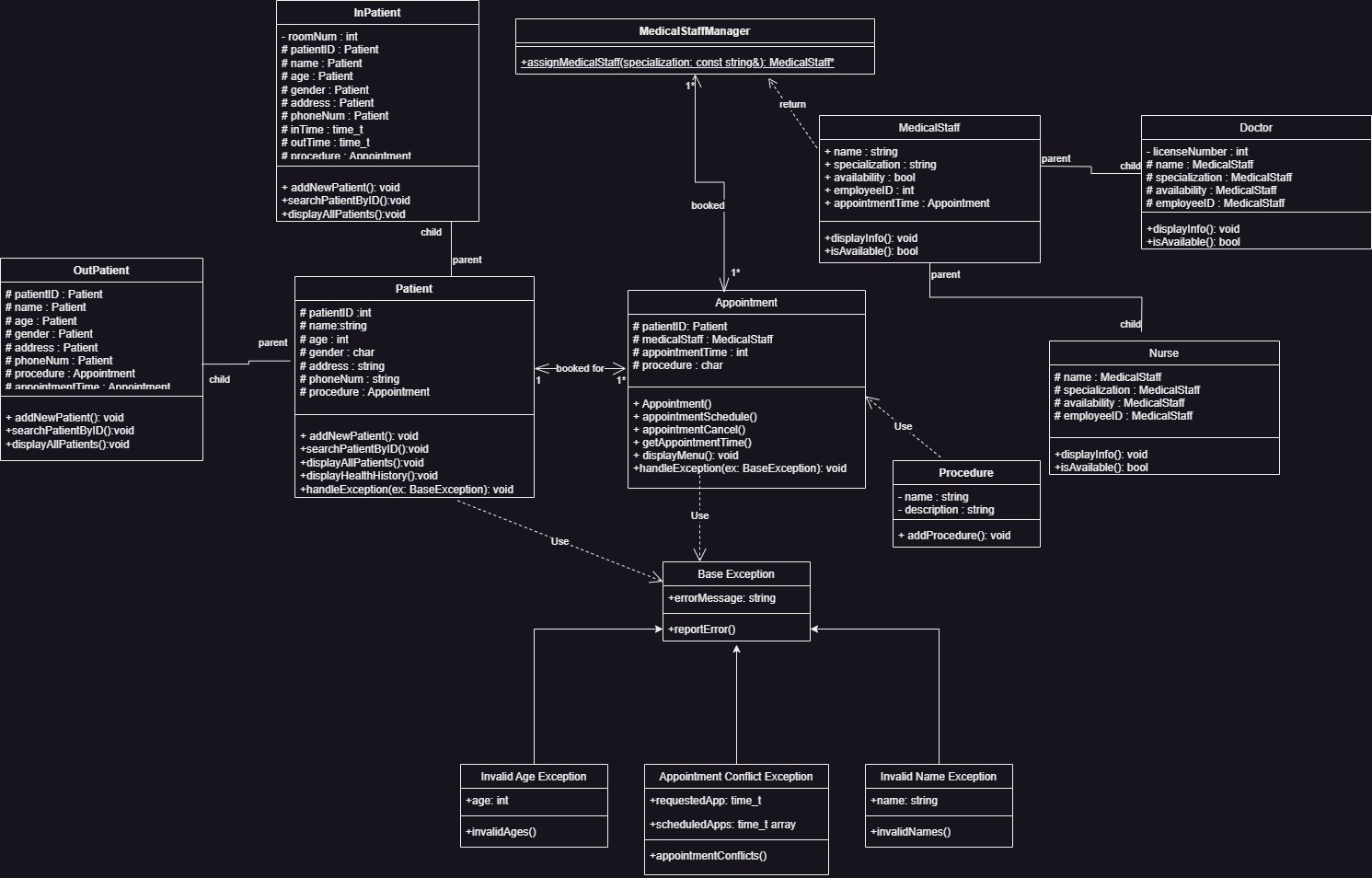
8. \*\*Error Reporting\*\*: Informative error messages are provided to users in case of invalid inputs or errors, guiding them on how to correct the issues and interact effectively with the system.

9. \*\*Code Readability\*\*: Our code is well-commented, making it easy to understand and maintain. Clear and descriptive variable/function names enhance code readability and comprehension.

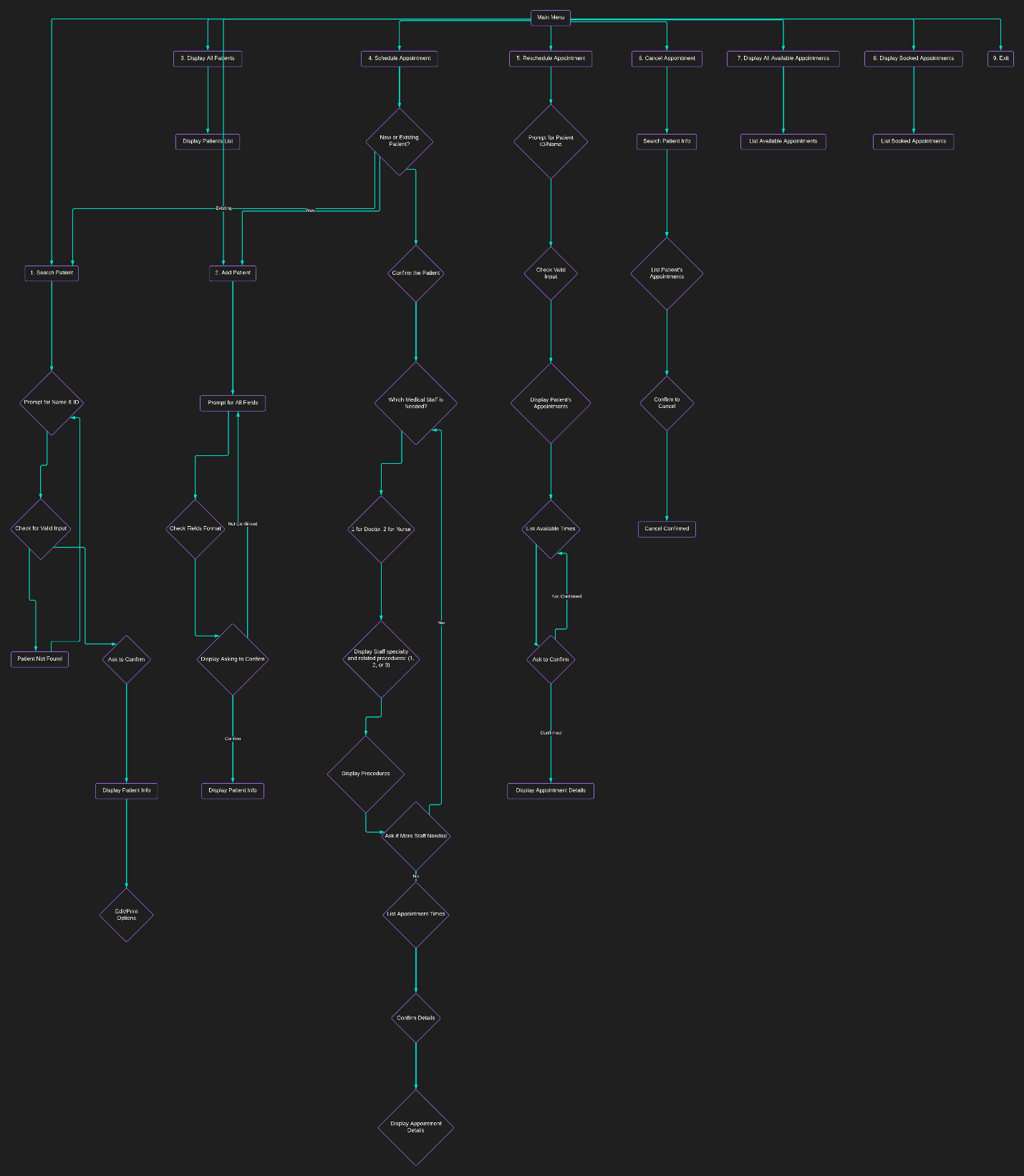
Overall, our `main (userinterfaceg1.cpp)`integrates the program altogether and demonstrates a structured, functional, albeit simplified hospital management system, with a focus on modularity, exception handling, and user interaction.

# Design

## UML Class Diagram



## Data Flow Diagram



# Testing and Evaluation

## Features to be tested.

* Add patient:  
  文本

  描述已自动生成文本

  描述已自动生成  
  文本

  描述已自动生成
* Search patient:

文本

描述已自动生成

* Display all patients
* Schedule appointment
* Reschedule appointment
* Cancel appointment
* Display booked appointments
* Exit

## Pass/Fail Criteria

1. **Add Patient:**

**Pass:** Prompts for all required fields with correct formats, confirms input, displays patient info upon confirmation, and allows editing if needed.

**Fail:** Does not prompt for required fields, lacks input validation, or does not display patient info.

2. **Search Patient:**

**Pass:** Successfully prompts for name and ID, validates input, displays patient info if found, and allows editing/printing options.

**Fail:** Fails to prompt for necessary input, does not validate input, or does not display patient info.

3. **Display All Patients:**

**Pass:** Displays all patients with names and IDs.

**Fail:** Does not display all patients or displays incorrect information.

4. **Schedule Appointment:**

**Pass:** Successfully schedules an appointment for both new and existing patients, prompts for necessary information, confirms details, and lists available appointment times.

**Fail:** Fails to schedule appointments, does not prompt for required information, or lists incorrect appointment times.

5. **Reschedule Appointment:**

**Pass:** Allows rescheduling of appointments, prompts for patient ID/name, displays appointments, lists available times, confirms changes, and updates the appointment accordingly.

**Fail:** Unable to reschedule appointments, does not prompt for required information, or does not confirm changes.

6. **Cancel Appointment:**

**Pass:** Successfully cancels appointments, prompts for patient info, lists appointments, confirms cancellation, and updates the appointment list accordingly.

**Fail:** Unable to cancel appointments, does not prompt for required information, or does not confirm cancellation.

7. **Display Booked Appointments:**

**Pass:** Lists all booked appointments showing patient ID and name, appointment time, and medical staff assigned.

**Fail:** Does not display booked appointments or displays incorrect information.

8. **Exit:**

**Pass:** Allows the user to exit the program gracefully.

**Fail:** Does not provide an option to exit or does not exit properly.

# Project Issues

## Lessons Learned

During the development of the patient management system, several key considerations emerged, shedding light on best practices and challenges encountered in the project.

The utilization of a container of pointers posed a notable challenge. By combining STL containers with pointers, complexities arose regarding ownership and manipulation of the underlying objects. For instance, employing a smart pointer, such as unique\_ptr, to manage objects of the Patient class and its derivatives, led to the transfer of ownership upon insertion into the container. Consequently, the pointer would become null, akin to a deep copy scenario. To access stored values thereafter, the implementation of iterators or search functions became necessary.

Another significant aspect pertained to the selection of appropriate STL containers aligned with business requirements. Matching container types with the characteristics of the data entities proved crucial for operational efficiency. For instance, assigning medical staff to a vector container suited their stable nature, minimizing real-time updates. Conversely, patients were assigned a list container due to dynamic characteristics necessitating real-time updates. Employing suitable container types based on business contexts enhanced operational robustness and streamlined future development efforts.

Maintaining data integrity through comprehensive validation procedures constituted another focal point. Defining reasonable ranges for variables, such as patient ages, and implementing exception handling for out-of-range inputs ensured data reliability. Additionally, validating temporal data and resolving scheduling conflicts contributed to the overall integrity of the system.

An intriguing challenge encountered during development related to user input handling within encapsulated function bodies. The iterative nature of function calls and loops sometimes resulted in repetitive usage of user inputs. Research revealed that the input stream object, cin, retained excess input, affecting subsequent executions. Mitigation strategies involving cin.clear() and cin.ignore() functions were implemented to manage input stream memory and facilitate user interactions.

Reflection on project management practices highlighted areas for improvement, particularly in work distribution and time management. Regular check-ins and reassessment of roles could have ensured alignment with initial commitments, preventing disruptions, delays, imbalance of work, and enhancing team efficiency.

In summary, addressing these challenges and implementing best practices contributed to the successful development of the patient management system, fostering operational reliability, and facilitating future enhancements.

# Conclusion

In conclusion, the development of the Hospital Management System presented challenges in managing pointers within STL containers, aligning container selection with business needs, and ensuring data integrity. Mitigation strategies and best practices were employed to navigate these challenges successfully. Reflection on project management highlighted areas for improvement, emphasizing the importance of effective communication and time management. Ultimately, the project culminated in a robust solution poised to enhance operational efficiency and patient care within healthcare institutions.

# References

Charming Data. (2023, May 29). Push code to your GitHub account – Under 3 minutes [Video]. YouTube. https://www.youtube.com/watch?v=vpRkAoCqX3o

Fergusson, K. (2018, March 1). UML class diagrams in Draw.io. https://drawio-app.com/blog/uml-class-diagrams-in-draw-io/

Lucidchart. (n.d.). Lucidchart. https://www.lucidchart.com/pages/

Pixabay. (n.d.). Health care medicine healthy. [Illustration]. Pixabay. https://pixabay.com/illustrations/health-care-medicine-healthy-2082630/

Portfolio C. (2023, March 26). Exception handling | C++ tutorial [Video]. YouTube. https://youtube.com/watch?v=5nCXSDv6e4I

Simplilearn. (2021, December 2). Exceptions handling in C++ | What is exception handling in C++ | C++ programming | Simplilearn [Video]. YouTube. https://www.youtube.com/watch?v=7hcQQEHZPiQ

Spaceo. (n.d.). Hospital management system. Spaceo. https://www.spaceo.ca/blog/hospital-management-system/

The Cherno. (2017, September 3). Operators and operator overloading in C++ [Video]. YouTube. https://www.youtube.com/watch?v=mS9755gF66w