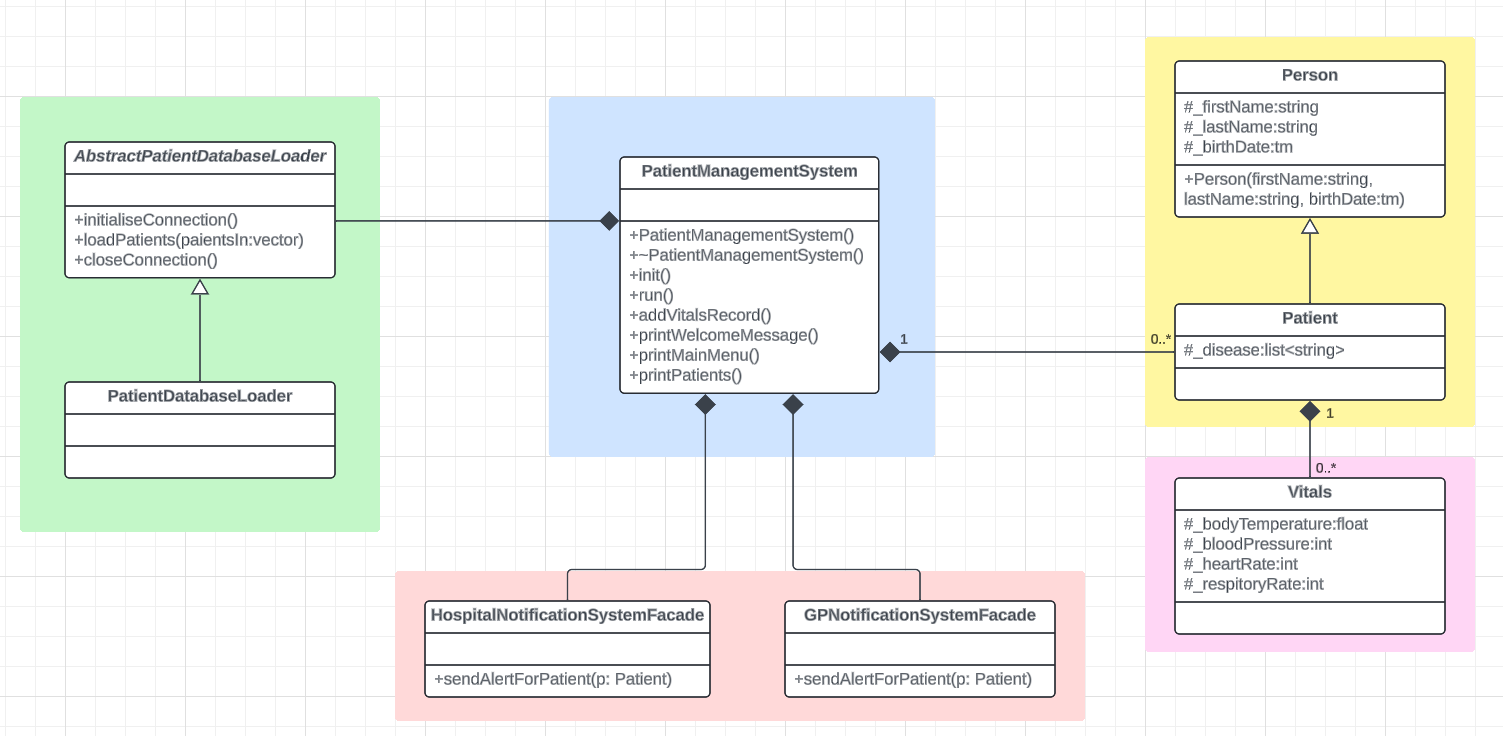
# **Patient Vitals Management System**

## System Design Documentation

1. **System Overview**

The Patient Vitals Management System is designed to manage patient information, vital recording, and notifications for healthcare providers with several core components.

[Figure 1.1 – Initial State]



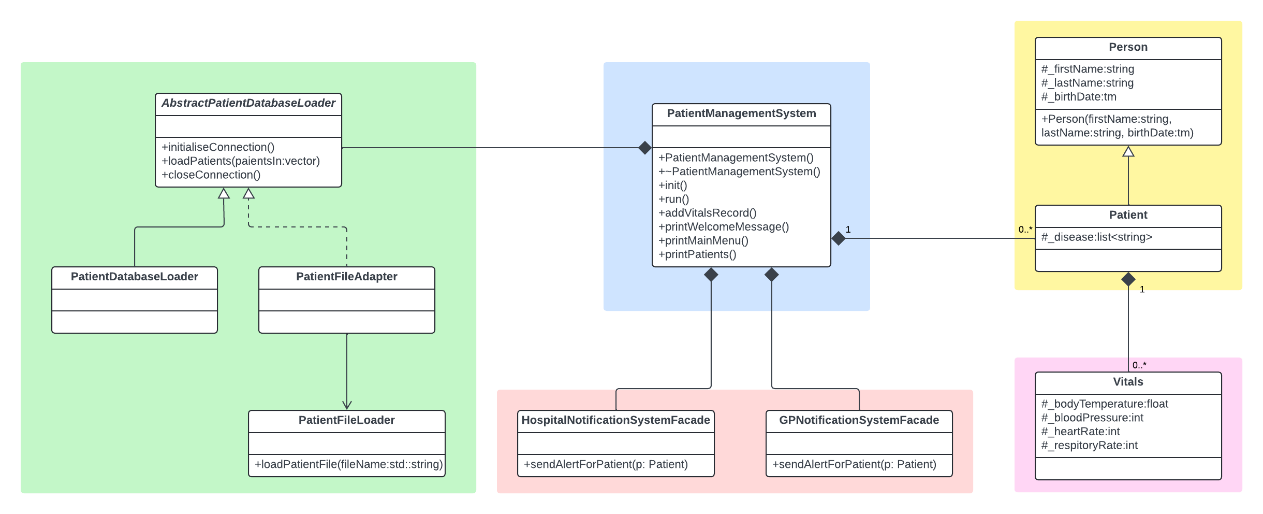
[The table below only summarizes the above diagram following the partial UML diagram in the Assignment Spec document.]

|  |  |
| --- | --- |
| **Main Component** | **Description** |
| Main Driver | * PatientManagementSystem (A) controls the program. It starts and runs the system, adds vitals, shows menus, and tracks patients. |
| Patient Data Loading | * Patient data is loaded using an interface called AbstractPatientDatabaseLoader (B). * The concrete implementation PatientDatabaseLoader simulates database access with a mocked connection. |
| Patient Model | * The Patient class (C) inherits from the Person class, encapsulating personal information such as first name, last name, and birthday. * Each patient maintains a list of diseases (represented as strings) and can record multiple Vitals instances (D). * Vitals include measurements such as body temperature, blood pressure, heart rate, and respiratory rate. |
| Notification System | * The system provides two facade interfaces, HospitalNotificationSystemFacade and GPNotificationSystemFacade (E). * Designed to send alerts about patient conditions to hospitals and general practitioners, respectively. (Not yet integrated into the main system workflow) |

1. **Mapping Functional Requirements to Design Patterns**

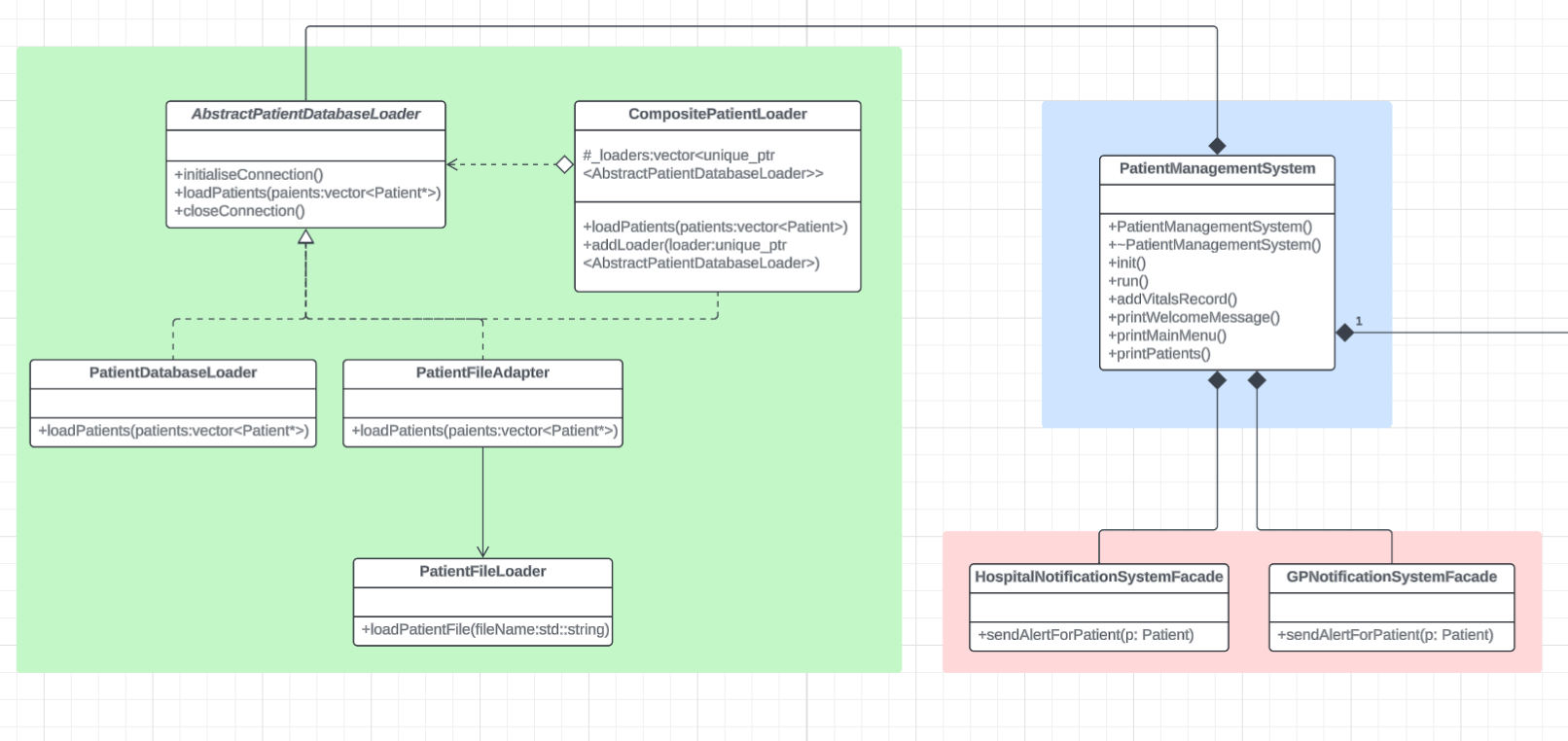
|  |  |  |  |
| --- | --- | --- | --- |
| **Functional Requirements #** | **Description** | **Design Pattern** | **Progress** |
| FR1 | Load patients from a file. | Adapter Pattern | Complete |
| FR2 | Load patients from both the file and the database. | Composite Pattern | Complete |
| FR3 | Implement algorithms to identify patient alert levels when new vitals are recorded. | Strategy Pattern | In progress |
| FR4 | Alert hospitals and GPs when the patient alert level is Red. | Observer Pattern |  |

1. **Design and Implementation Details**
   1. **FR1: Load patients from a file using the Adapter Pattern**



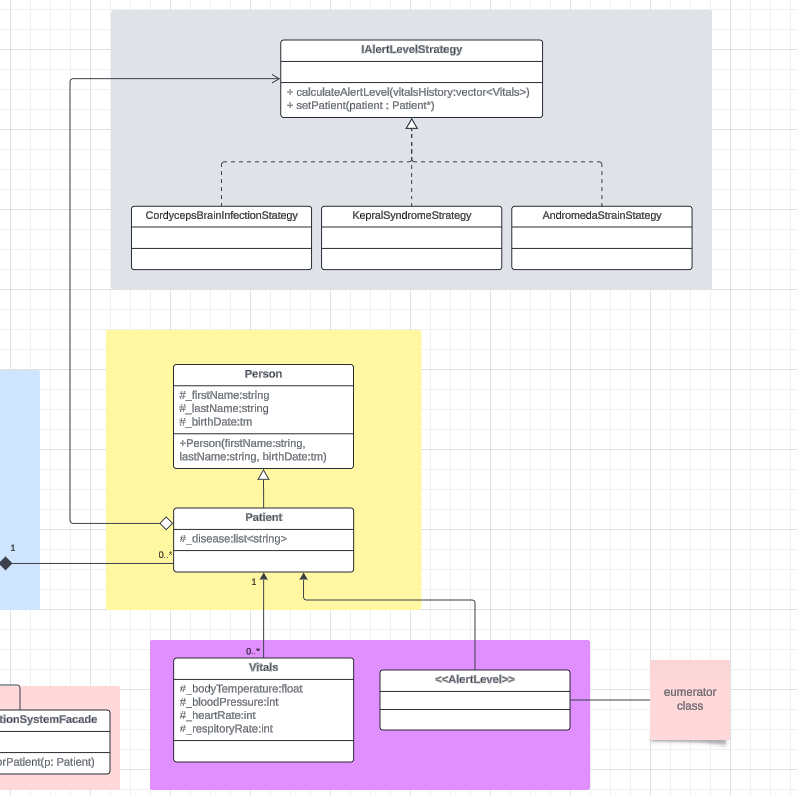
The first functional requirement was to load patient data from a file. This was successfully implemented using the Adapter Pattern. The PatientManagementSystem is designed to depend on an abstract interface called AbstractPatientDatabaseLoader, which allows flexibility in loading data from different sources. However, the file loader PatientFileLoader did not originally implement this abstract interface. To solve this, I created an adapter class called PatientFileAdapter that wraps the PatientFileLoader and makes it compatible with AbstractPatientDatabaseLoader. By using the adapter, the PatientManagementSystem can now load patient data from a file without changing any existing code. Following the Adapter Pattern allow incompatible interfaces to work together and supports any changes in the future.

* 1. **FR2: Load Patients from Both the File and the Database using the Composite Pattern**

****

The second functional requirement was to load patient data from both the database and a file source. This was successfully implemented using the Composite Pattern. The PatientManagementSystem is designed to depend on the abstract interface AbstractPatientDatabaseLoader, which allows for flexible loading from different data sources. To meet this requirement, I created a new class called CompositePatientLoader that also implements the AbstractPatientDatabaseLoader interface. The CompositePatientLoader acts as a container that holds multiple loader components (in our case, we got PatientDatabaseLoader and PatientFileAdapter). When the system runs, the composite loader calls each child loader in sequence, ensuring that patients from the database are loaded first, followed by patients from the file. Both PatientDatabaseLoader and PatientFileAdapter serve as leaf classes in this design, performing the actual data loading.

* 1. **FR3: Implement algorithms to identify patient alert levels when new vitals are recorded using the Strategy Pattern**

****

The third functional requirement focused on determining a patient’s alert level based on their diagnosis and most recent vital signs. To make this flexible and easy to extend, I used the Strategy Pattern. Each diagnosis, like Kepral’s Syndrome, Cordyceps Brain Infection, and Andromeda Strain, has its unique criteria for what counts as a critical condition. Instead of putting all that logic into the Patient class, I created separate strategy classes that each handle one diagnosis or disease. These classes all follow a common interface IAlertLevelStrategy, which defines how to calculate an alert level. Inside the Patient class, there’s a pointer to one of these strategy objects. When a patient is first diagnosed, the system assigns the right strategy based on their condition. Then, whenever new vital signs are added, the patient uses the strategy to figure out whether their alert level has changed. If the alert level is anything other than Green, a warning message is printed to let the user know. This design makes it easy to add new diagnoses later, just create a new strategy class, and the system will work without needing to touch any existing code.

* 1. **FR1: Load patients from a file using the Adapter Pattern**

**Supporting resources:**

1. Refactoring Guru - [Design Patterns](https://refactoring.guru/design-patterns)