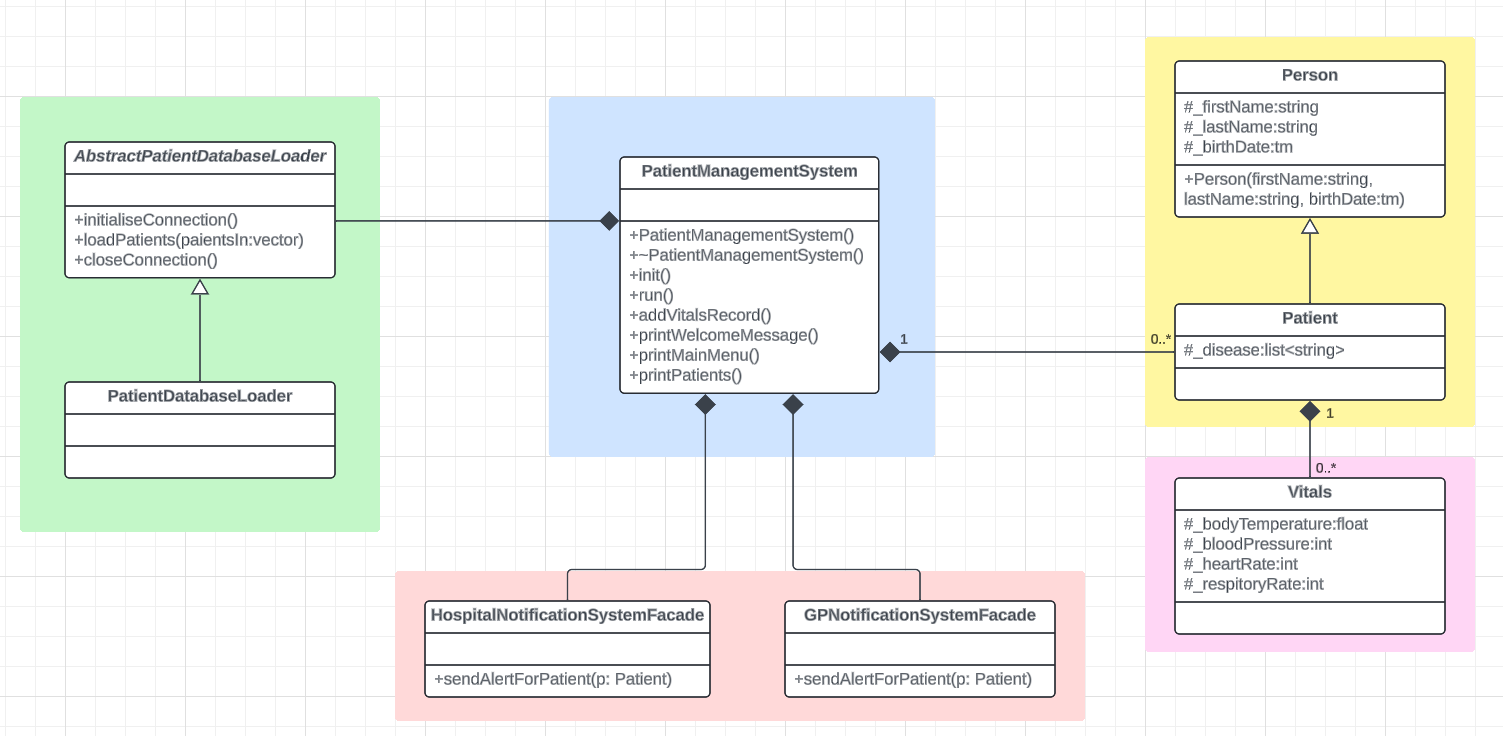
# **Patient Vitals Management System**

## 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]



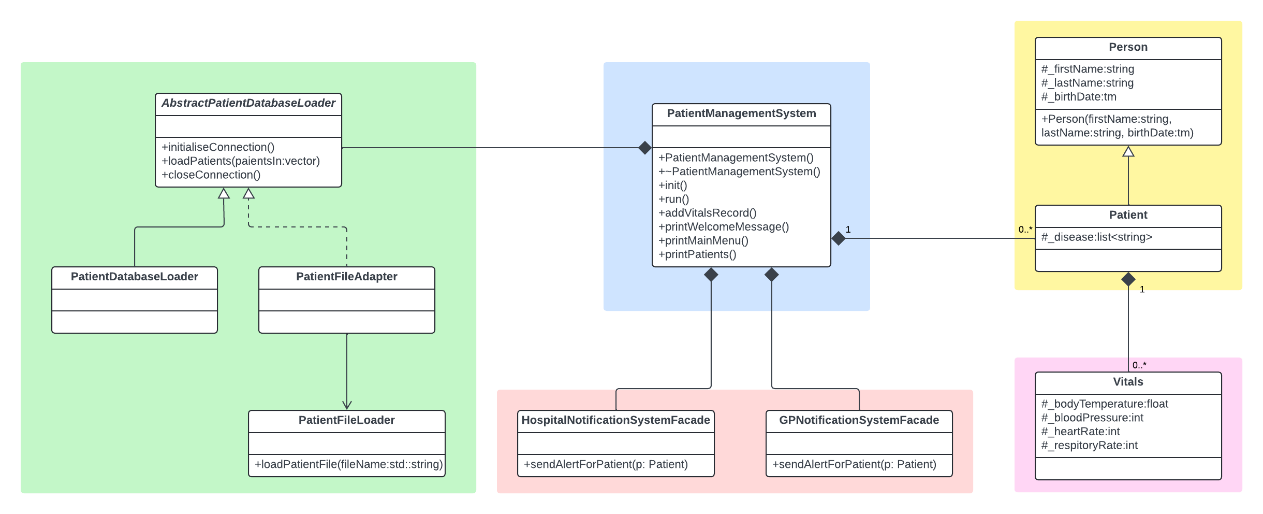
[The table below only summarizes the above diagram following the partial UML diagram in the Assignment Spec document.]

|  |  |
| --- | --- |
| **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 | Complete |
| FR4 | Alert hospitals and GPs when the patient alert level is Red. | Observer Pattern | Complete |

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



**Design Pattern:** 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.

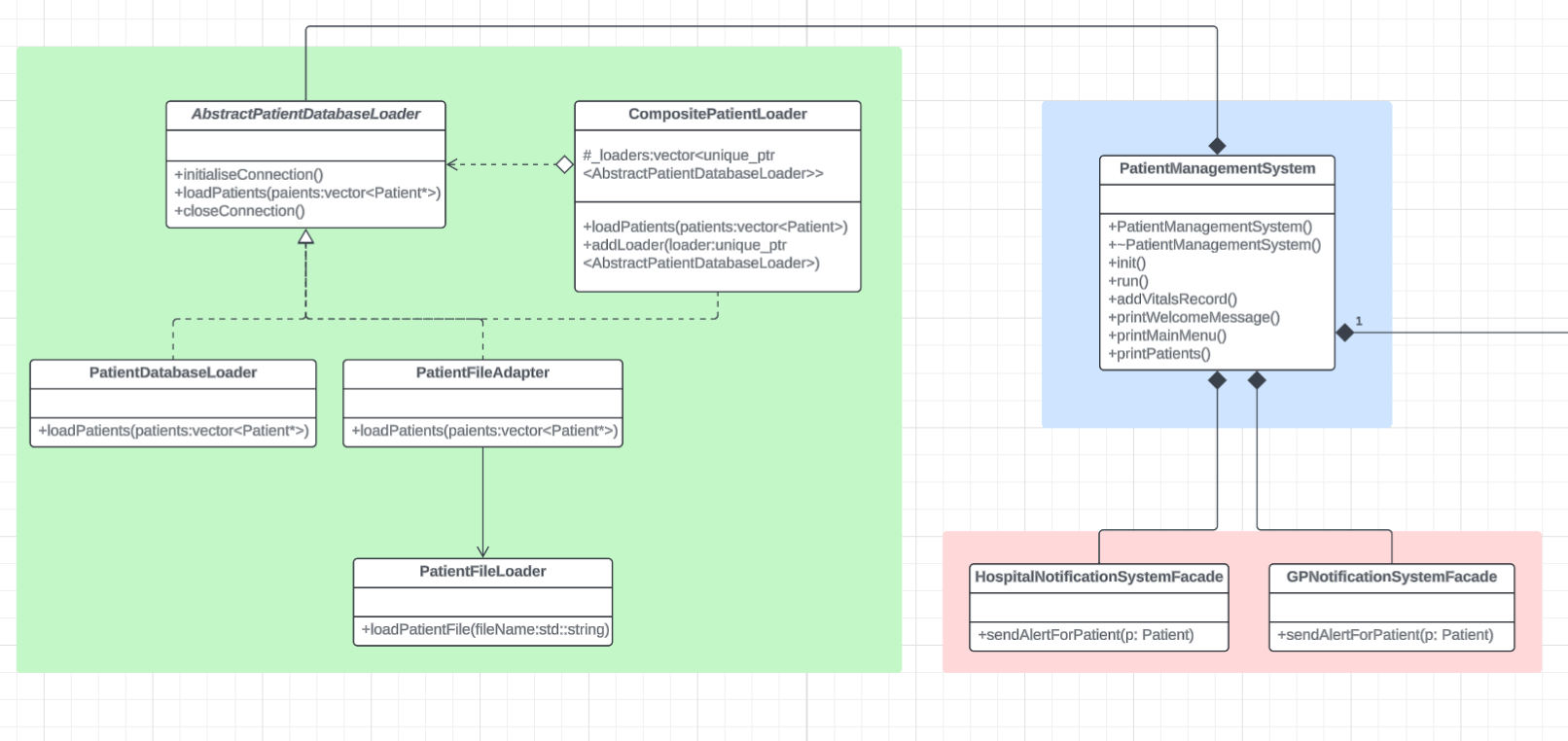
To solve this, I created an adapter class called PatientFileAdapter that wraps the PatientFileLoader and makes it compatible with AbstractPatientDatabaseLoader. By using this adapter, the PatientManagementSystem can now load patient data from a file without needing any changes to its own code. The Adapter Pattern helped connect otherwise incompatible components and ensures that future changes (like adding new data sources) can be supported more easily.

**How it works:**

1. PatientManagementSystem uses an AbstractPatientDatabaseLoader to load data.
2. PatientFileAdapter wraps PatientFileLoader and implements the abstract interface.
3. File-based patient data is loaded through the adapter without modifying the core system.

**Git Commits:**

* Introduced PatientFileAdapter to adapt PatientFileLoader to the AbstractPatientDatabaseLoader interface in commit 09c5e92.
* Integrated the adapter with PatientManagementSystem to enable file-based patient data loading without modifying existing system logic in commit 4d8fd2c.
  1. **FR2: Load Patients from Both the File and the Database using the Composite Pattern**

****

**Design Pattern:** Composite Pattern

The second functional requirement was to load patient data from both a file and a database. To handle this smoothly, I used the Composite Pattern. The idea was to let the system treat multiple data sources as if they were just one.

To do this, I created a class called CompositePatientLoader. It implements the same AbstractPatientDatabaseLoader interface that the system already understands. This loader doesn’t actually load any data itself—it just holds a list of other loaders, like PatientDatabaseLoader and PatientFileAdapter, and asks each one to do its job when needed.

This way, PatientManagementSystem doesn't need to care whether data is coming from a file, a database, or somewhere else. Everything gets bundled into one process, and the result is a complete list of patients.

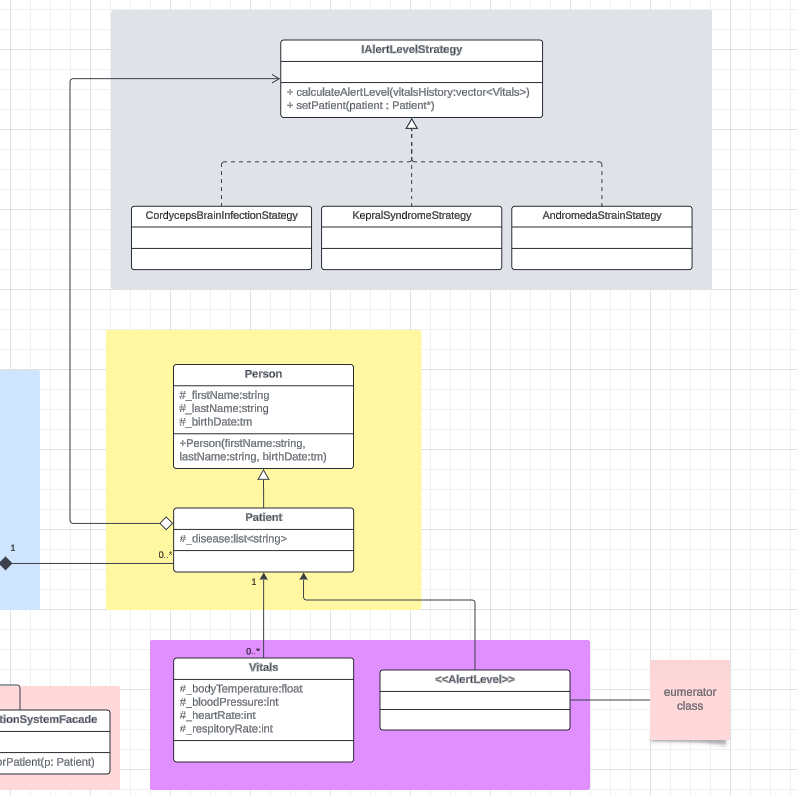
**How it works:**

1. CompositePatientLoader stores a list of loaders (file and database).
2. When loadPatients() is called, it delegates the call to each contained loader.
3. Both file and database patients are loaded into a single unified collection.

**Git Commits:**

• Added CompositePatientLoader and updated design doc in commit 1098b2b.  
• Integrated with PatientManagementSystem in commit 2fe1724.

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

****

**Design Pattern:** Strategy Pattern

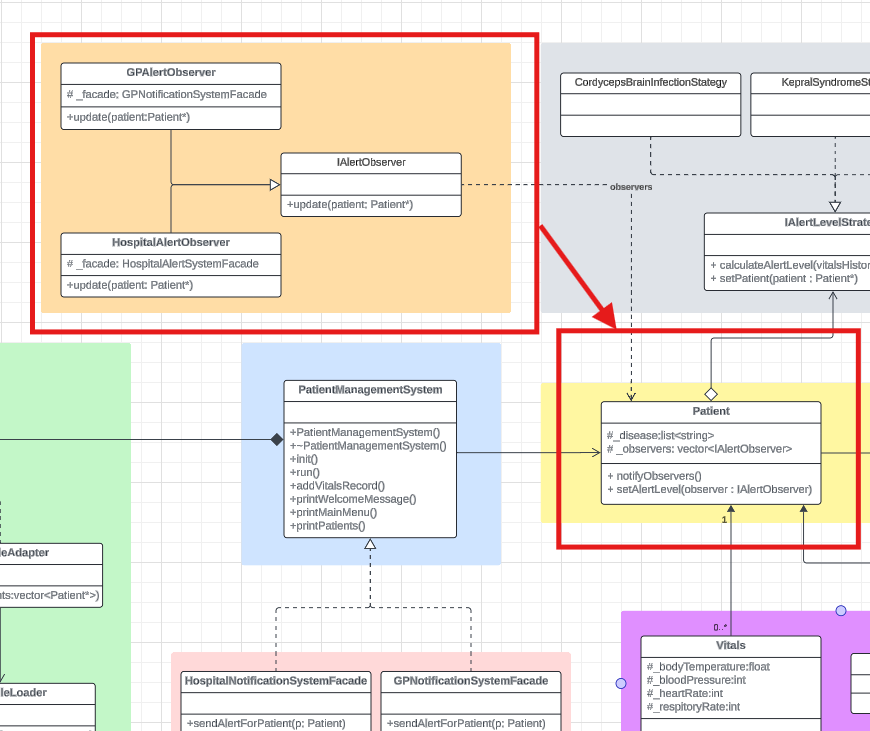
The third functional requirement focused on determining a patient’s alert level based on their diagnosis and most recent vital signs. To make this flexible and easy to extend, I used the Strategy Pattern. Each diagnosis, like Kepral’s Syndrome, Cordyceps Brain Infection, and Andromeda Strain, has its 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.

**How it works:**

1. Each diagnosis has its own strategy class implementing IAlertLevelStrategy.
2. The Patient class uses a strategy to calculate the alert level after receiving new vitals.
3. If the alert level is elevated, a warning is printed to notify the user.

**Git Commits:**

* Added IAlertLevelStrategy and diagnosis-specific strategies in commit e28a14d.
* Fixed logic and class declaration errors in commit 70cb715.
* Discovered the issue with undiagnosed test patients and confirmed correct strategy behavior with proper data in commit e14f4b5.
* Updated UML diagram and added early documentation in commit 31e75dc.
  1. **FR4: Alert hospitals and GPs when the patient alert level is Red using Observer Pattern**



**Design Pattern:** Observer Pattern

The fourth functional requirement was to alert hospitals and GPs when a patient’s alert level becomes Red. To make this possible, I used the Observer Pattern. The idea is to let the Patient object notify interested parties like the observers whenever its alert level changes. I introduced an interface called IAlertObserver, which is implemented by GPAlertObserver and HospitalAlertObserver. These observers use façade classes (GPNotificationSystemFacade and HospitalAlertSystemFacade) to send actual notifications. They're added to each patient using shared\_ptr so they can be reused across patients without duplication. When a patient's alert level is set, if it turns Red, the Patient object calls notifyObservers(). Each observer checks the alert level and responds accordingly, sending out the right alerts. This keeps the alert logic out of the core patient logic and makes it easier to add or change alert behavior later on. The pattern design helps decouple the Patient class from the components that need to react to changes in a patient's alert level. Without this pattern, the Patient class would need direct knowledge of every notification system leading to tight coupling.

**How it works:**

1. The Patient class holds a list of shared\_ptr<IAlertObserver>.
2. GPAlertObserver and HospitalAlertObserver are registered when the system initializes.
3. When setAlertLevel() is called, if it's Red, notifyObservers() is triggered.
4. Observers use facades to send notifications to GPs and hospitals.

**Git Commits:**

* Started design doc and initial diagram in commit 8f77a89.
* Added observer interface, concrete observers, and integration in commit 06b7fed.
* Finalised observer pattern documentation update in commit 6dc6f1b.

**Supporting resources:**

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