## FR1: load patients from file

**Design Pattern:** Adapter Pattern

A diagram of a company

AI-generated content may be incorrect.**Why I am using this pattern:** The System is required to take data from the text file. The PatientFileLoader class is already given to me to read the patient data from a local file (patients.txt in this case). However, it is not compatible with the interface expected from the class (AbstractPatientDatabaseLoader), since it was designed for the database. Adapter pattern is used to solve this problem; it allows incompatible interfaces to work together by creating a class which convert one interfaces into another. In this case, the adapter is (FilePatientDatabaseLoader), it adapts the PatientFileLoader to conform to AbstractPatientDatabaseLoader interface.

**How it works:**

1. The PatientManagementSystem initializes a PatientFileLoaderAdapter by calling the constructor with the file path "patients.txt".
2. The PatientFileLoaderAdapter wraps the PatientFileLoader using the Adapter pattern to make it compatible with the AbstractPatientDatabaseLoader.
3. The PatientManagementSystem calls initialiseConnection() on the adapter, which prepares the file loader for reading.
4. During init(), the PatientManagementSystem calls loadPatients(…) passing in a vector which will be populated with patients.
5. The PatientFileLoaderAdapter delegates to the PatientFileLoader to read the file line by line, parse patients’ data, and construct Patient objects which added to the vector.
6. The PatientManagementSystem builds a lookup map from the loaded patients for efficient retrieval.
7. Finally, the PatientManagementSystem calls closeConnection() in its destructor to properly clean up file resources.

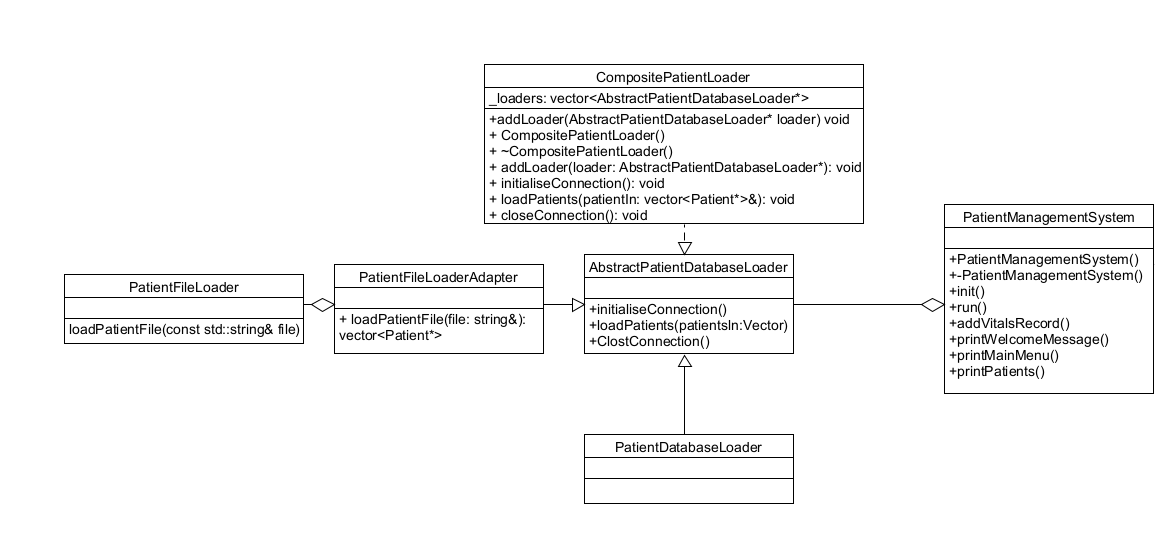
**Git Commits:**

* **I first implemented the PatientFileLoader in commit f52438.**
* **I created the adapter PatientFileLoaderAdapter in commit 42df08.**
* **The adapter was implemented in 4f473**

## FR2: load patients from file and database

**Design Pattern:** Composite Pattern

**Why I am using this pattern:** The Composite pattern is used to implement this function requirement. The system is required to load patients from three ways: from database, from text file and from both. And is required to switch among these configurations easily with a line of code change. Composite is ideal for dealing this case because it can help treat different loaders (PatientDatabaseLoader and PatientFileLoader) and the combination (CompositePatientLoader) evenly through a common interface which is AbstractPatientDatabaseLoader. This enables us to create a tree-like structure where the composite can contain individual loaders or other composites, executing them in a specific order while hiding the complexity of managing multiple data sources.



**How it works:**

1. The PatientManagementSystem initializes a CompositePatientLoader using the Composite pattern.
2. The CompositePatientLoader is configured with multiple loaders: first a PatientDatabaseLoader, then a PatientFileLoaderAdapter, ensuring database patients are loaded before file patients.
3. The PatientManagementSystem calls initialiseConnection() on the composite, which calls initialiseConnection() on each individual loader in sequence.
4. During init(), the PatientManagementSystem calls loadPatients(…) on the composite.
5. The CompositePatientLoader executes each loader in order: first the database loader adds patients to the vector, then the file loader adds additional patients to the same vector.
6. Finally, closeConnection() is called on the composite, which properly closes both database and file connections through each individual loader.

**Git Commits:**

* **I created the composite PatientFileLoader in commit 53fb4e.**
* **The adapter was implemented in 486abb.**

## FR3 calculate the patient alert levels

**Design Pattern:** Strategy Pattern

**Why I am using this pattern:** The system must calculate a patient’s Alert Level using disease-specific algorithms that vary depending on the patient's diagnosed diseases. Each patient’s Alert Level is recalculated whenever new vital signs data are input, but **not** during initial data loading from historical sources such as databases or files. I choose strategy pattern for this requirement because it allows the system to treat each disease-specific alert calculation algorithm independently. This approach enables flexible, extensible alert level calculations without the need to modify patients’ code. The system can also dynamically choose the appropriate algorithm at runtime based on the patient’s disease.

A diagram of a company

AI-generated content may be incorrect.

**How it works:**

1. Creates an interface AlertLevelCalculator for calculating alert levels for different diseases.
2. Create the three calculator classes, each of them is corresponding to one disease.
3. The new calculator classes will inherit from the abstract class AlertLevelCalculator.
4. Calculators are assigned based on the patient’s diagnosis which allows switching between configurations automatically based on the patient’s diagnosis
5. When new vitals for patient are added, the calculator will automatically detect the diseases with diagnosis and choose the correct calculator.
6. The new alert level is calculated, and the updated alert level is set for the patient.

**Git Commits:**

* The base interface AlertLevelCalculator is created in commit d61ff1.
* The specific calculators are created and implemented as follows:

Cordyceps Brain Infection created & implemented in 47431f.

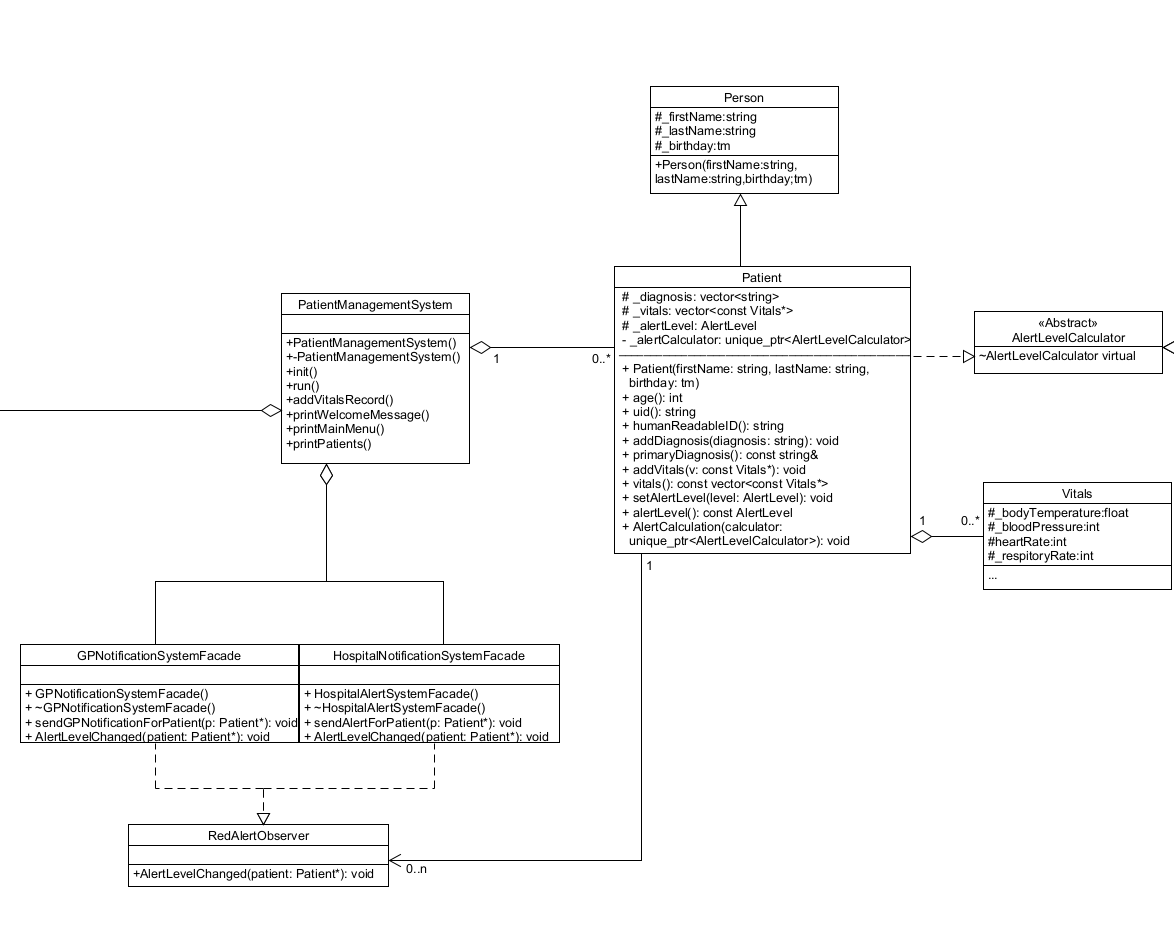
Kepral’s Syndrome Created in 72a2ba, implemented in e9f8a8

Andromeda Strain Created and implemented in 72a2ba

## FR4: alert the hospitals and GPs

Design Pattern: Observer Pattern

**Why I am using this design pattern:** To satisfy this functional requirement, I decided to use the Observer design pattern because it fits well with the need to automatically and instantly alert stakeholders—like hospitals and GPs—when a patient’s alert level reaches Red. By treating the two façade systems as observers, the notifications happen in real time without directly tying them into the core patient logic. This keeps the code cleaner and easier to manage. It also makes future changes, like adding more notification systems, much simpler without needing to rewrite the core functionality.



**How it works:**

1. When PatientManagementSystem initializes, HospitalAlertSystemFacade and GPNotificationSystemFacade instances are created as unique\_ptr members.
2. Each patient has both facade systems registered as observers using addSubscriber(), implementing the Observer pattern.
3. When a patient's alert level is set via setAlertLevel(), the method checks if the new level is Red, if so the alertSubscribers() is called to notify all registered observers.
4. Each observer's AlertLevelChanged() method is called with the patient as a parameter.
5. The HospitalAlertSystemFacade responds by calling sendAlertForPatient() which sends critical alerts only for Red level patients.
6. The GPNotificationSystemFacade responds by calling sendGPNotificationForPatient() which sends follow-up notifications for Red level patients.
7. Both facades use the Facade pattern to provide simplified interfaces to complex hospital and GP notification systems, with actual notifications currently implemented as console output for demonstration purposes.

**Git Commits:**

* The base interface RedAlertObserver is created in commit abd610.
* The methods for managing the subscribers are created in commit 4961d8,
* The above methods are implemented in commit 0da51a.
* The trigger to call the alert the stakeholders is added to the alert level class in commit 42ac4f.
* The alert method for notifying GP and hospital facades is created in

6669dd and are implemented in the same commit.