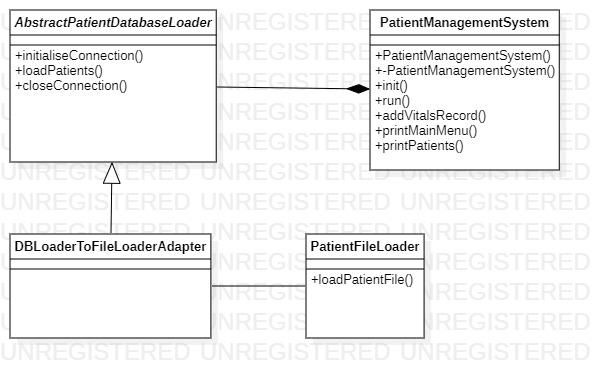
**COMP 3023 Design Patters with C++ Assignment 2 | Chris Taylor**

**Functional requirement FR1: Load patients from file**



**Design pattern: Adapter**

At present, the system loads patients from a (mocked) database but this needs to be modified whereas patients are loaded from a provided file in .txt format. To address this requirement, I apply the Adapter design patten which maintains the existing stable interface for the former (from a database) and new (from a file) patient loading methods. The Adapter is represented in Figure 1. The existing ‘AbstractPatientDatabaseLoader’ is extended to a new concrete class called ‘DBLoaderToFileLoaderAdapter’ class. This strategy allows us to easily switch between loading patients from a database or a file, and it is conducive to adding additional patient loading methods in the future (by creating more adapter classes). Maintaining our stable abstract interface eliminates any cascading issues that may be experienced if addressing the new requirements in absence of the Adapter pattern.

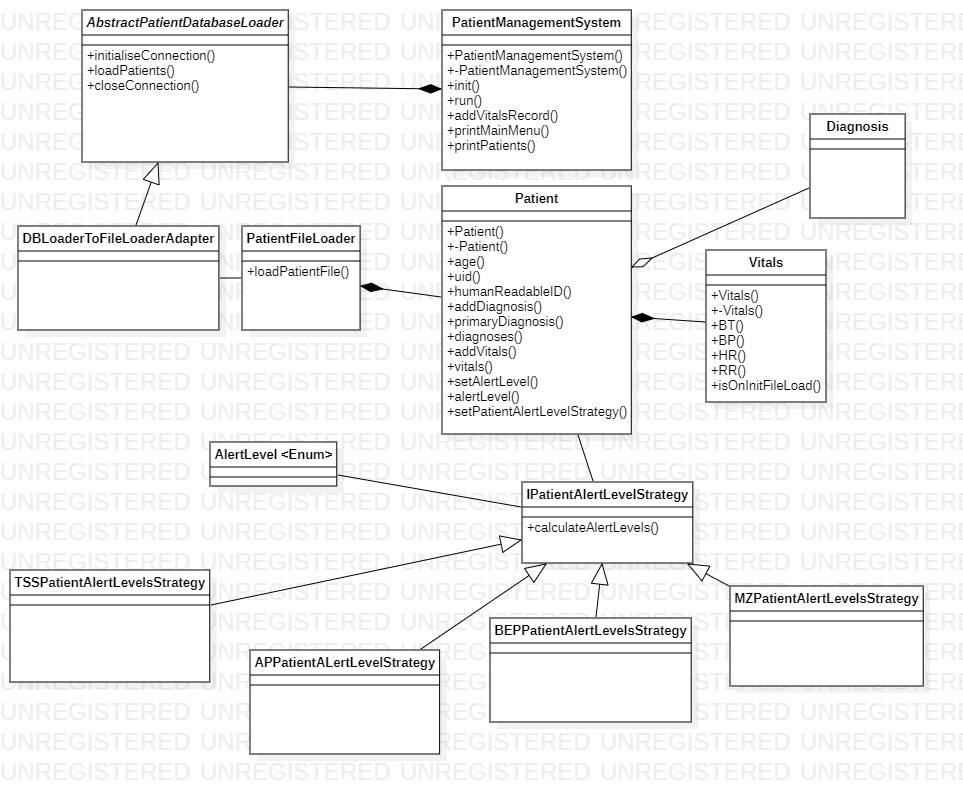
**How it works:**

1. The ‘PatientManagementSystem’ will initialise the ‘DBLoaderToFileLoaderAdapter’ by calling init().
2. The ‘DBLoaderToFileLoaderAdapter’ will call loadPatients(), passing in a vector which will be loaded with patients.
3. The concrete ‘DBLoaderToFileLoaderAdapter’ calls a specific loadPatientFile() member function of class ‘PatientFileLoader’, passing in a string with a path to the file with patients to load.
4. LoadPatientFile() parses each line of text in the file, instantiating new patients with their name and birthday, and adding vitals, diagnosis’, setting the patient alert level strategy, and adding to a vector of patients. This vector of patients is returned to loadPatients().
5. The original (empty) vector passed in is replaced by the returned (populated) vector.
6. The ‘PatientManagementSystem’ is then responsible to release Patient memory allocated by the ‘DBLoaderToFileLoaderAdapter’.
7. Finally, the ‘PatientManagementSystem’ must explicitly call closeConnection() to close the connection to the file.

**Git commits:**

* SHA-1: 5ed5c5b195df606612871deb1813566287ac5485 - I begin adding the design pattern by creating the DBLoaderToFileLoaderAdapter class and adding logic to the PatientFileLoader class.
* SHA-1: 398c469e2d379574f4cef571c28c926b79844696 – I refactor the adapter class and complete all logic for the PatientFileLoader class.

**Functional requirement FR2: Calculate the patient alert levels**



**Design pattern: Strategy**

When new vitals are added to a patient, the system needs to be able to calculate the patient alert levels based on the patient's primary disease. To address this requirement, I apply the Strategy pattern where each respective disease alert level trigger algorithm is defined in a separate class with a common interface. The Adapter is represented in Figure 2. An interface ‘IPatientAlertLevelsStrategy’ is extended by several concrete strategy classes ‘APPPatientAlertLevelsStrategy’, ‘BEPPatientAlertLevelsStrategy’, ‘MZPPatientAlertLevelsStrategy’ and ‘TSSPatientAlertLevelsStrategy’. This strategy reduces complexity in the calling member function and allows for easy modification of existing patient alert level strategies and/or the addition of new patient alert level strategies for new diseases that may be added to the system.

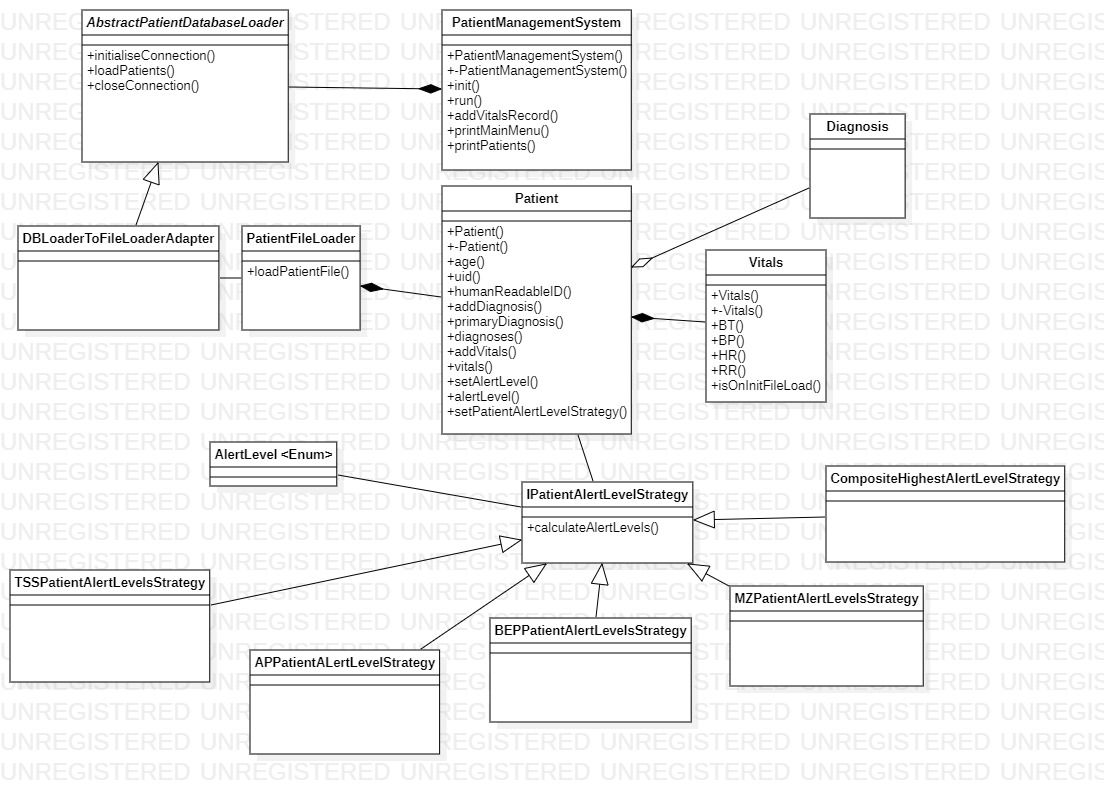
**How it works:**

1. When a new patient is created in the ‘PatientFileLoader’ class, an appropriate alert level strategy is set on the patient based on their single disease or primary disease (in the event they are suffering from more than one disease).
2. When new vitals are added to a patient using the addVitals() member function the system checks if the vitals are being added after the initial file load.
3. If they are, it calls calculateAlertLevels() member function, passing in a pointer to the patient itself and a pointer to the newly added vital measurements. This function returns the appropriate ‘AlertLevel’ enum.
4. Using polymorphism, the system executes the appropriate algorithm-based on the patients assigned alert level strategy.
5. Based on the returned ‘AlertLevel’, the patient's name and alert level is printed if it is more severe than green.

**Git commits:**

* SHA-1: eefaf64c1eb1844f072e2e5de16d98e7970be8fe – I begin by modifying logic in the PatientManagementSystem class when adding vitals. I pass in a strategy to each instance of a Patient and create all sub-classes of a newly named IPatientAlertLevelsStrategy interface.
* SHA-1: 5a1e01c9435e91f7814b8a9328126465a0fd5471 – I implement all separate alert level algorithms in the sub classes.

**Functional requirement FR3: Calculate the patient alert levels for all diseases a patient has**



**Design pattern: Composite**

If a patient is suffering from more than one disease at once, the system must calculate a patient’s highest Alert Level dependent on all their diseases. To address this requirement, I apply a Composite pattern where the system will determine the alert levels of all diseases the patient has and return the most severe alert level. The Composite pattern is represented in Figure 3. An additional concrete class that extends ‘IPatientAlertLevelsStrategy’ is created, called ‘CompositeHighestAlertLevelStrategy’. Future alert level strategies can be added to the system with no modification required to this composite class. Logic has been modified in the ‘PatientFileLoader’ class whereas patients that have more than one diagnosis use the ‘CompositeHighestAlertLevelStrategy’ rather than one of the specific disease-based alert level strategies.

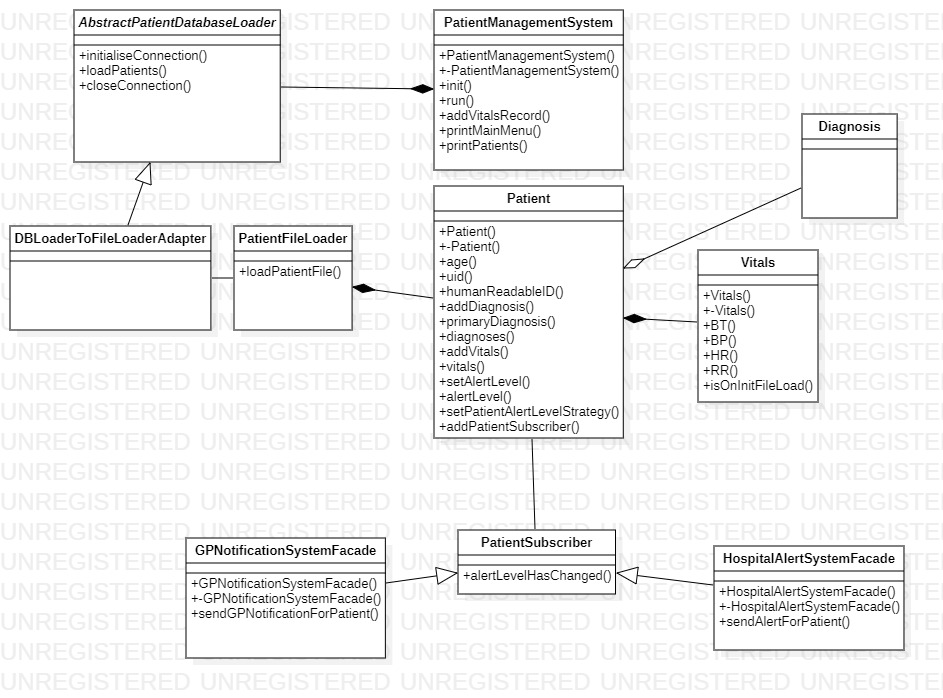
**How it works:**

1. If a new patient created in the ‘PatientFileLoader’ class has more than one diagnosis, then an instance of ‘CompositeHighestAlertLevelStrategy’ is instantiated.
2. For each string version of a diagnosis in a vector, check if it is equal to the value of a Diagnosis string constant. If so, add the strategy to the ‘CompositeHighestAlertLevelStrategy’ instance which maintains a vector of ‘IPatientAlertLevelsStrategy’ strategy elements.
3. When new vitals are added to a patient using the addVitals() member function the system checks if the vitals are being added after the initial file load.
4. If they are, it calls calculateAlertLevels() member function, passing in a pointer to the patient itself and a pointer to the newly added vital measurements. This function returns the appropriate ‘AlertLevel’ enum.
5. Using polymorphism, the system executes the appropriate algorithm-based on the patients assigned alert level strategy. In these cases, the appropriate strategy is always ‘CompositeHighestAlertLevelStrategy’.
6. Each algorithm that was added as a strategy element is executed via calculateAlertLevels() member function in ‘CompositeHighestAlertLevelStrategy’. The previous most severe alert level (0 – 4, the enumerator constant values) is compared against the current one in a loop.
7. The most severe alert level is returned. Based on the returned ‘AlertLevel’, the patient's name and alert level is printed if it is more severe than green.

**Git commits:**

* SHA-1: 6cac46c7835202f6555dad52712dff3f5d3f6fc1 – All functionality above. The additional class (‘CompositeHighestAlertLevelStrategy’) mentioned above is checked-in but not committed.

**Functional requirement FR4: Alert the hospitals and GPs**



**Design pattern: Observer**

The system needs to notify hospitals and GP’s when a patient’s alert level reaches red. To address this requirement, I apply an Observer design pattern that uses a ‘subscriber’ based interface. Patients act as ‘publishers’ who register subscribers (‘GPNotificationSystemFacade’ and ‘HospitalAlertSystemFacade’) who are notified each time a patient’s alert level changes. An interface ‘PatientSubscriber’ has been created and is extended by the two Façade-based classes mentioned above. It would be easy to modify the algorithm in each subscriber class based on future changes (I.e.: GP’s need to be notified if the alert level reaches yellow, but hospitals still need to be notified only when alert level reaches red). The Observer pattern also has extendibility in mind as any future subscribers (I.e.: Pharmacist) could be created in the same manner as the existing Facades. In the event of more complex notification sub-systems (I.e.: using API’s that send messages via SMS, pager or email), the Façade classes would abstract further classes beneath them. However, this is not necessary for the simplicity of our current notification requirements.

**How it works:**

1. When a new patient is added in the ‘PatientFileLoader’ class, an instance of ‘GPNotificationSystemFacade’ and ‘HospitalAlertSystemFacade’ is instantiated. Each instance is added as a subscriber via the addPatientSubscriber() member function in the ‘Patient’ class.
2. When setAlertLevel() is called after adding new vitals to a patient in the system, each subscriber calls the alertLevelHasChanged() member function, passing in a pointer to the patient.
3. As each of the Facades act as subscribers to this alert, alertLevelHasChanged() triggers a call to their own custom member functions that use separate logic to check if the alert levels have reached red. If so, they print custom messages to the console.

**Git commits:**

* SHA-1: 03c61bf21c93caf7badde3d6f08439752d2b7147 – All functionality above including the ‘PatientSubscriberClass’. ‘CompositeHighestAlertLevelStrategy’ is also committed.