FIT3077 - Assignment 2 Class Diagram

Yet Khai Bryan Wong, Matthew Jake Corpus Alix

Important note about ommissions in class diagram for the sake of readability and conciseness:

Get and set methods that simply return or set attributes are not shown.

Java packages that are used in this system and not shown include:
java.io for exceptions.,
java.net for URLEncoder.
java.util for List, Map, HashSet, Set, logging.
javax.xml.ws for WebServiceException.

GSON is the third-party library used to de-serialise JSON into POJOs (Plain Old Java Objects).

OkHttp is the third-party library used to make HTTP requests and retrieve data as JSON.

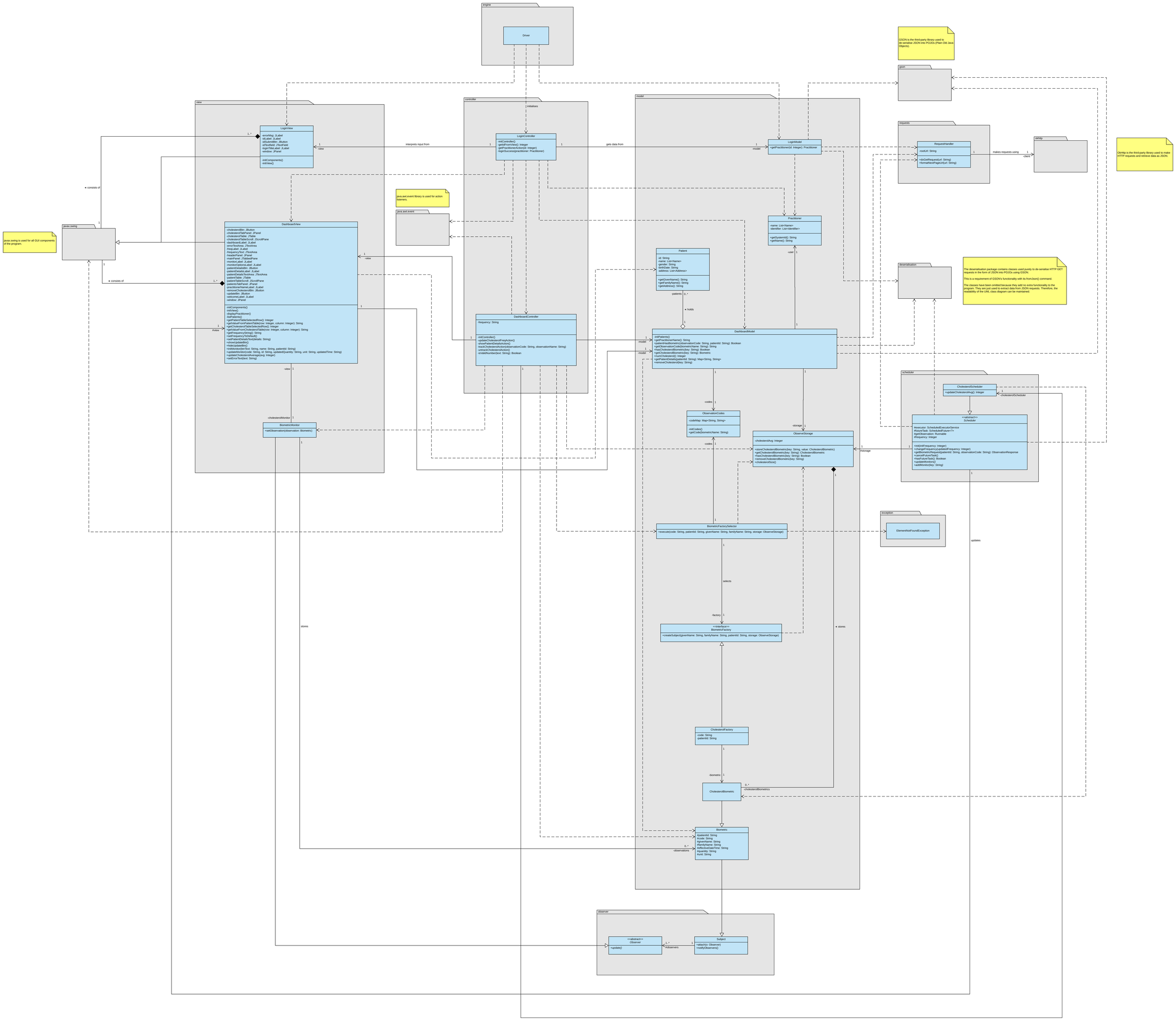
javax.swing is used for all GUI components of the program.

java.awt.event library is used for action listeners.

The deserialisation package contains classes used purely to de-serialise HTTP GET requests in the form of JSON into POJOs using GSON.

This is a requirement of GSON's functionality with its fromJson() command.

The classes have been omitted because they add no extra functionality to the program. They are just used to extract data from JSON requests. Therefore, the readability of the UML class diagram can be maintained.



Design Rationale

The software architecture used in the system was the active variant of the **Model-View-Controller (MVC) pattern**. One benefit of this was a separation concerns between each part of the application (e.g. the Login and the Dashboard) as well as within each part in respect to the GUI, user input logic, and business logic. This increased cohesion made it easier to test the application as each component could be debugged separately. The active variant of MVC was chosen because the Dashboard Model could change independently of the Dashboard Controller based on updates from the server every N seconds. One drawback of this pattern was that it added extra complexity to the design with more dependencies and more packages by making each individual class more cohesive.

The **Observer pattern** was used to monitor patient biometrics. Whenever the cholesterol observations (instances of CholesterolBiometric) are updated, this would notify the BiometricMonitor instance (which stores references to all cholesterol observations), which then updates the view. This pattern was used so that changes to observations would automatically update the monitors. This adds another layer of abstraction as the Subject and Observer objects do not need to know each other's concrete subclasses. This could be extended in the future to support new types of observations or observers simply by creating new classes which inherit the Subject and Observer classes.

The **Abstract Factory pattern** is used so that the implementation of instantiating biometric observations is separate from the dashboard and hidden behind the BiometricFactory interface. The BiometricFactorySelector class would only need to know the interface's execute() method to create an observation without needing to know the concrete implementation for specific types of observations. Another benefit of this pattern is that it creates a hinge-point in the design for future extensibility. New types of biometrics can be added by creating new classes inheriting the BiometricFactory interface and Biometric class. The drawback is that it requires many classes (the factory selector, concrete factories, and Biometric subclasses) just to create a single observation.

The **Dependency Inversion principle** is used so that the model and controller components of the Dashboard depend on the Biometric class, rather than subclasses which inherit Biometric. This creates a hinge-point so that new types of biometrics can be monitored by creating new classes inheriting the Biometric class without those components needing to know the concrete class implementations.

The **Liskov Substitution principle** is used in the BiometricFactorySelector class. The class calls the createSubject() method of the BiometricFactory interface without having to know the concrete implementation. Therefore, any subclass of BiometricFactory can be provided where BiometricFactory is expected and correctness is still preserved. The benefit of this is that the system can be extended with new factories without changing any modules dependent on the factory selector.

The **Acyclic dependencies principle** was supported through the use of the MVC pattern. By ensuring that the model package never depends on the view or controller packages, it prevents any cyclic dependencies between packages. Furthermore, most other packages outside the MVC pattern are stable making it less likely for a cyclic dependency to occur. The benefit is that as the system is extended, it reduces the likelihood that the failure of one module will trickle cyclically to other modules it is dependent on.

The **Common reuse principle** was used in creating packages outside of the MVC pattern. Classes were packaged together based on the likelihood that they would be reused together. Therefore, the requests, exception, scheduler, and observer packages were created separately as it is very unlikely that they would be reused with any other classes or packages. The benefit of this is that the packages have greater cohesion. However, this also grew the system complexity as there are now many small packages with only one or two classes in them.