Design Rationale

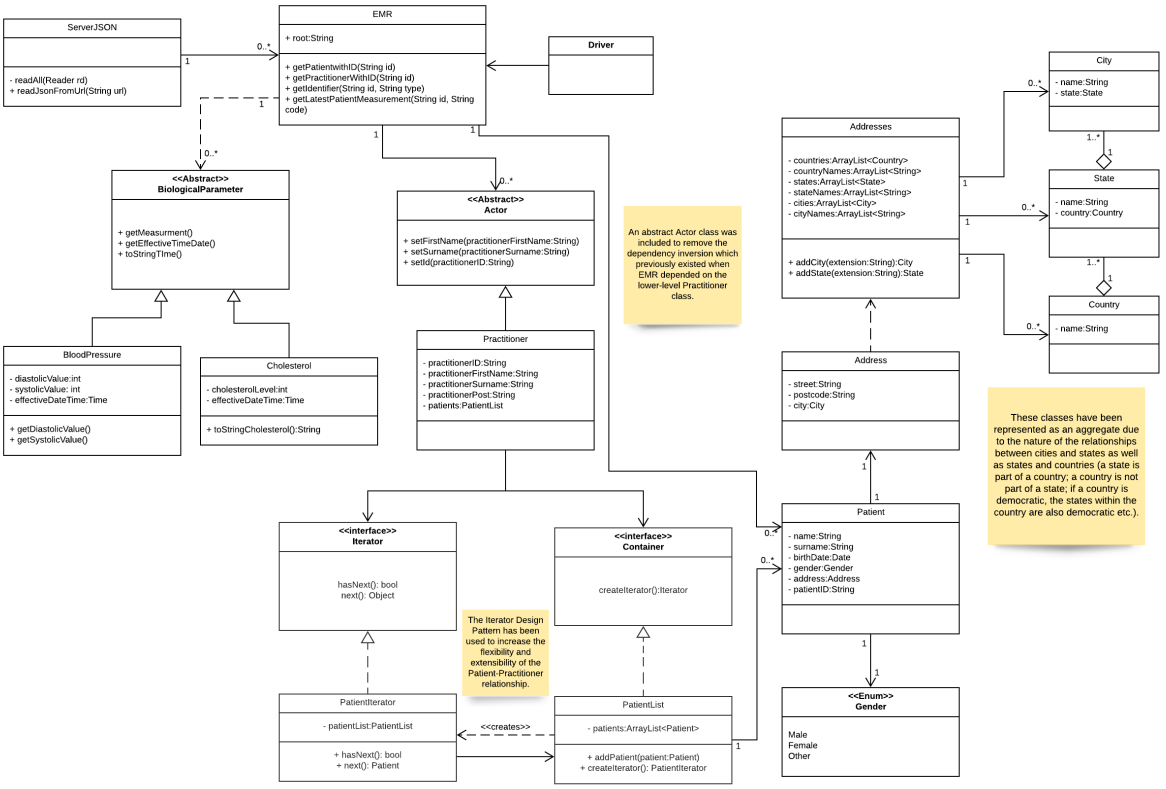

Figure 1 - UML Class Diagram

As shown in Figure 1, every class has a specific purpose and is required to fulfil the requirements of the application. Note that throughout this document, there are references to classes by name as well as the concept they represent. A capitalised noun refers to the class representing the lowercase concept.

The heart of the app is the patient, with all other functions relating to a patient either directly or indirectly. To store the data for each patient, the Patient class requires name, surname, birthDate, gender, address and patientID attributes. Name and surname are strings, gender is an enum and birthDate is a Date for obvious reasons, patientID is a string instead of an integer for flexibility but the Address class requires a detailed explanation. Each address contains a street name, a postcode, a city, a state and a country. The decision to make these aggregations of separate classes was made for several reasons. Some cities, states and countries share similar or identical names (e.g. Abbotsford, Victoria and Abbotsford, New South Wales or Georgia the country and Georgia, USA) so by making a class for each, any overlap of names doesn't cause confusion as they will be separate classes and every created Country-State-City set is stored in Addresses. These classes also allow for potential extension in Assignment 3 as they could be modified to contain information about the patients they contain. They have also been represented as an aggregate due to the nature of the relationships between cities and states as well as states and countries (a state is part of a country; a country is not part of a state; if a country is democratic, the states within the country are also democratic etc).

Each Practitioner has a list of Patients which they manage so the Iterator design pattern has been employed for this functionality. The Iterator design pattern makes use of two design principles. Splitting the iteration through the collection over multiple classes which all have a

specific purpose adheres to the Single Responsibility Principle (SRP) and the Open Closed Principle (OCP) is utilised by allowing for potential extension of other classes which could contain collections of Patients or allowing Practitioner to be extended to include collections of other classes. Since neither of these uses of OCP are necessary for the app yet, it could be argued that the Iterator design pattern is overkill and inefficient in this scenario but employing this design pattern ensures that the design is extendable and flexible which are more important in the context of this application.

The whole system is run by the Driver class which utilises the EMR, Calculations and Display classes to perform the main functionality of the app. EMR interacts with the server, Calculations has methods which analyse the data from the server, and Display handles the UI. Spreading these tasks across these classes utilises the SRP as each responsibility is handled by a single class.

The EMR class manages Practitioners and their Patients which requires the EMR class to depend on Practitioners which violates the Dependency Inversion Principle (DIP) as Practitioners is a lower-level class than EMR. This issue was solved by creating an abstract Actor class which Practitioner extends. This solution can also be easily extended by creating other classes which extend Actor.

For exactly the same reasons, the abstract class BiologicalParameter was created as a prototype for all biological measurement classes. Cholesterol and BloodPressure extend BiologicalParameter, employing the DIP to ensure that the design is extensible and flexible. This arrangement also makes use of the Liskov Substitution Principle (LSP) as BloodPressure and Cholesterol replace BiologicalParameter without sacrificing the correctness of the program.

There are only two packages in this system: frontEnd and model. The model package contains all of the server-side classes, dealing with acquiring the data from the server and parsing it for use by the frontEnd package. The frontEnd package deals with displaying and analysing the data acquired from the server. Each of these packages has high cohesion – all of the classes within them perform a single purpose – and low coupling – minimal dependencies between them.