For the features of Assignment 3, our group did not have to apply any refactorization techniques, as the system was, from the beginning, conceived so that adding new graphs/panels/data monitors would be easy and efficient. Most of the work was creating the graphs – adding to the table and adding filters was extremely easy with the centralised observable/observer pattern that we implemented in Assignment 2. It was as simple as adding new variables, adding new columns, etc.

As was written in the last design rationale, the observable class demonstrates DRY as any observer can use a method that is written in the observable class. This actually is extremely important for this set of features, and was a good thing to keep in mind when we were designing the system. Adding new GUI elements and even new data was trivial.

The GUI elements and views still adhere to the Liskov Substitution Principle, as all views observe the data source, and can make changes as per the pre-written methods. Any class that inherits GUI element could substitute for one another, therefore the set of classes complies with the LSP. The GUI element class itself is also Open for extension but Closed for modification – it inherits the abstract method update() all the way from Observer, so all the views must implement this, but each and every single view extends on update() in its own way – this showcases how the class is open for extension. There are also methods setDataSource() and setFrame() which are required to set the observable/observer relationship up. These are generic enough to not need to be changed, and therefore are closed for modification.

The system that we designed without any knowledge of the Model-View-Controller design pattern actually somewhat resembles it, with a few changes. The model would be the observable here, which is the large table that holds all the information required for all the views. The biggest difference is that through the Java Swing package, the Controller and the View are integrated into one another. For example, clicking an element in the table view under certain conditions could cause an update to the model, updating all views and causing a graph to appear. The Model here is the observable patient table that calls update() for every observer that is currently observing it, which is the same as the example given in the MVC active model. This simplification is what makes the design extremely easy to extend and control, but there are some downsides. The event-driven way the code runs made it extremely hard to debug – there were many, many System.out.println() statements required to figure out what was going wrong. Another issue is how slow the application could get from redrawing all the views constantly – right now every single update() has a frame.pack() method call in it, which basically repaints all the views, even the ones that have already been repainted and haven’t changed and squishes as close together as possible. Calling this multiple times an update is just wasted computing power, and this problem would only grow as views are added over time. This could be fixed with a refactor for the current views, but some extremely complex views with many, many data points would not appreciate being repainted 10 times a frame, which is not so simple to fix. However, for simple views such as text and 5 point graphs, this model is extremely simple and effective to implement. In fact, layout out the groundwork in Assignment 2 made it extremely easy to implement the features in Assignment 3. We estimate it took nearly ¼ the time, or less, to implement and test all 5 features, with less than half the number of commits, and around 10% of the stress. These statistics support the idea that this model was simple to use and extend.