# SOFT252

## Object Orientated Software Engineering with Design Patterns.

## Assignment by

### Mike Jones

### Wing Chau

### Radu-Adrian Marcu

# The project.

For the assignment, we were required to use test driven development to develop a graphical application for managing a company’s vehicle pool.

Our application consists of a library to implement the data model and a graphical application to use the library. The graphical application was created using the built-in Swing MVC framework provided by the Netbeans IDE. Along with the Model View Controller design pattern we also used the Command, Decorator, Observer, Proxy, Strategy, Singleton and Visitor design patterns. In addition, we have elements of Lazy Initialisation throughout. We complied with SOLID principles by creating single-purpose objects that could be extended, and using interfaces instead of concrete implementations where applicable.

The library was the primary responsibility of Wing, whilst Radu mostly worked on the graphical application. Mike took on the role of project manager as well doing some of the integration between library and application. To aid in collaborative development we used an online source repository to manage the codebase.

# The process.

To begin with, we designed a rough data model by defining which objects we needed, roughly which properties each object required, and the relationships between objects. We began by implementing simple objects that had the minimum details required, and then added functionality to them. As an example, the Car isAvaliable method started off simply returning a Boolean that had been manually set. However, once we had implemented the AllocationRecord functionality the method was then updated to automatically calculate the vehicle status from the records.

We updated our unit tests after implementing each piece of functionality or operation to ensure that it worked properly, and to detect inadvertent code regression.

# The how.

For the graphical application, we tried to balance good object orientated design principles with ease of development. We created separate JPanel classes that correspond to the objects that they manage, without going over board and creating objects for everything. As an example, the Vehicle Management panel contains sub-panels to manage the Allocation/Service/Damage records for the currently selected vehicle. Each of these sub-panels are implemented in a separate class, and contain text fields and other graphical components to represent the relevant object from the data model. We use the Observer pattern to propagate events that alter the Datastore to the relevant interface components.

The undo/redo functionality had a major impact on the design of our application, as it required that all operations be represented as abstract state changes. When the application wishes to perform an operation on the Datastore it must first create an object that implements the ICommandBehaviour interface. An ICommandBehaviour object defines an operation that is to be performed such as AddCar. Once an instance of the AddCar behaviour has been created, it is decorated with a Command object and passed to the CommandTracker for execution. The CommandTracker contains a list of executed and undone commands, when Undo is called it inverts the previously executed instruction and moves it from the Undo list to the Redo list.

The Datastore and the Undo/Redo functionality are implemented using the Singleton design pattern because they need to maintain global state. The Datastore is opened using the LoadDatastore method and then operated on using static method calls. The open Datastore can be changed by calling LoadDatastore again with a new filename, this will cause it to replace the in-memory data with the contents of the selected file.

# The code.

One of the problems we encountered was that the Date object in Java has a resolution of milliseconds, however we wanted to perform Date comparisons based only on the date. This required that we write a custom date comparison method, however Java does not support global methods. To get around this limitation we created a Utils class and placed a static method inside that class. This solution allowed us to make a simple call along the lines of Utils.CompareDates(date1, date2) which would perform a date-only comparison of the objects. An alternative method would be to subclass the Date object and replace all the comparison methods, however we felt that the Utils class solution was an easier and cleaner way to implement this.

The Netbeans Swing framework used for our GUI does not include a date selection component. For date selection, we decided to use the JXDatePicker library. Unfortunately this requires installing an extra plugin to be able to edit the graphical components via the Netbeans IDE.

# The final touches.

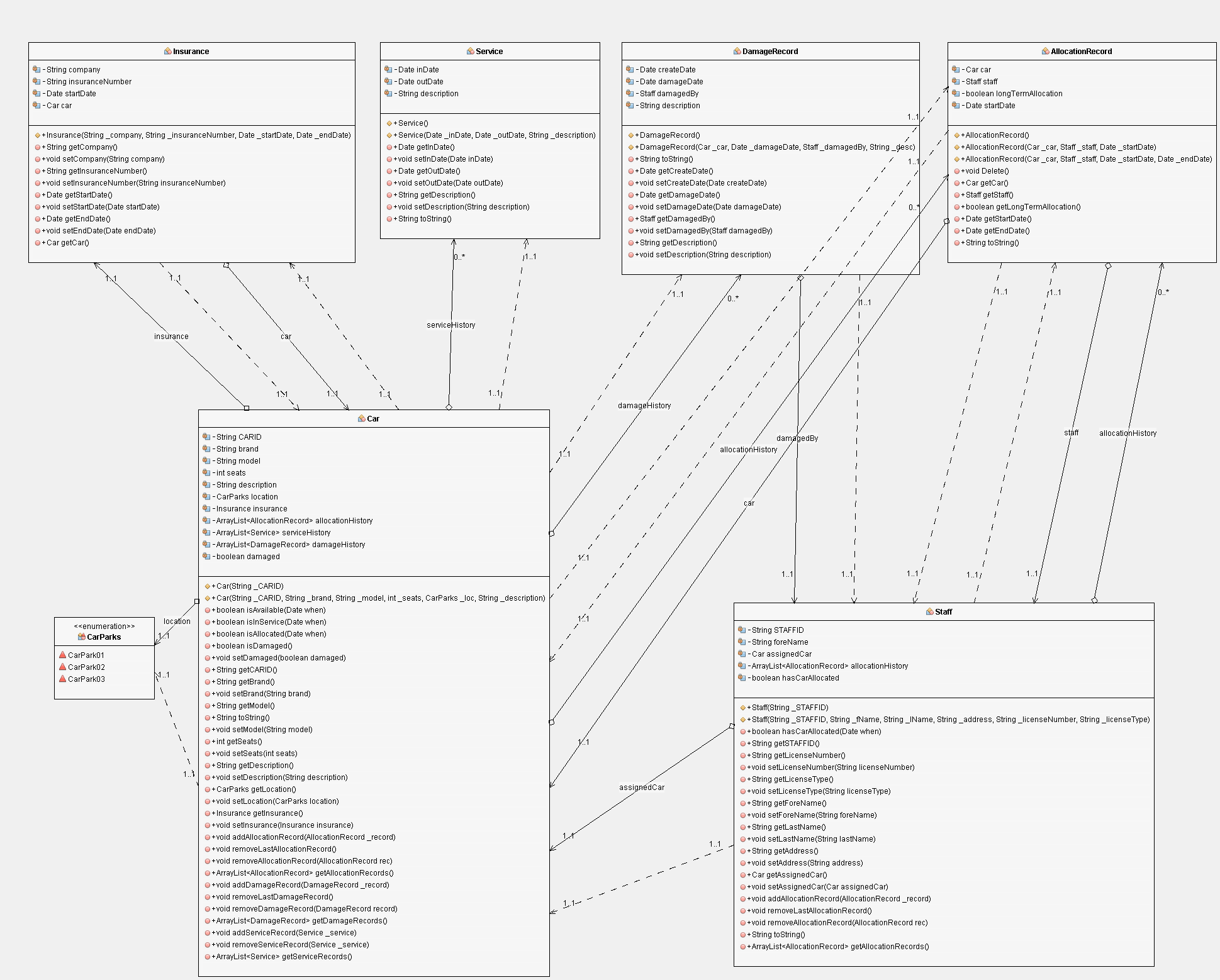
In addition to the functional requirements of the application we also put some consideration in to usability. Whilst the scenario does not provide us with actual users to give feedback on the system, there are certain things we felt were obvious additions. One example is that we implemented a custom cell renderer for the vehicle and staff lists to render them in a different colour depending on their status. Also, if the application cannot load the Datastore and needs to create it then it provides an option to pre-populate the Datastore with sample data, which may be useful for training.

# The flaws.

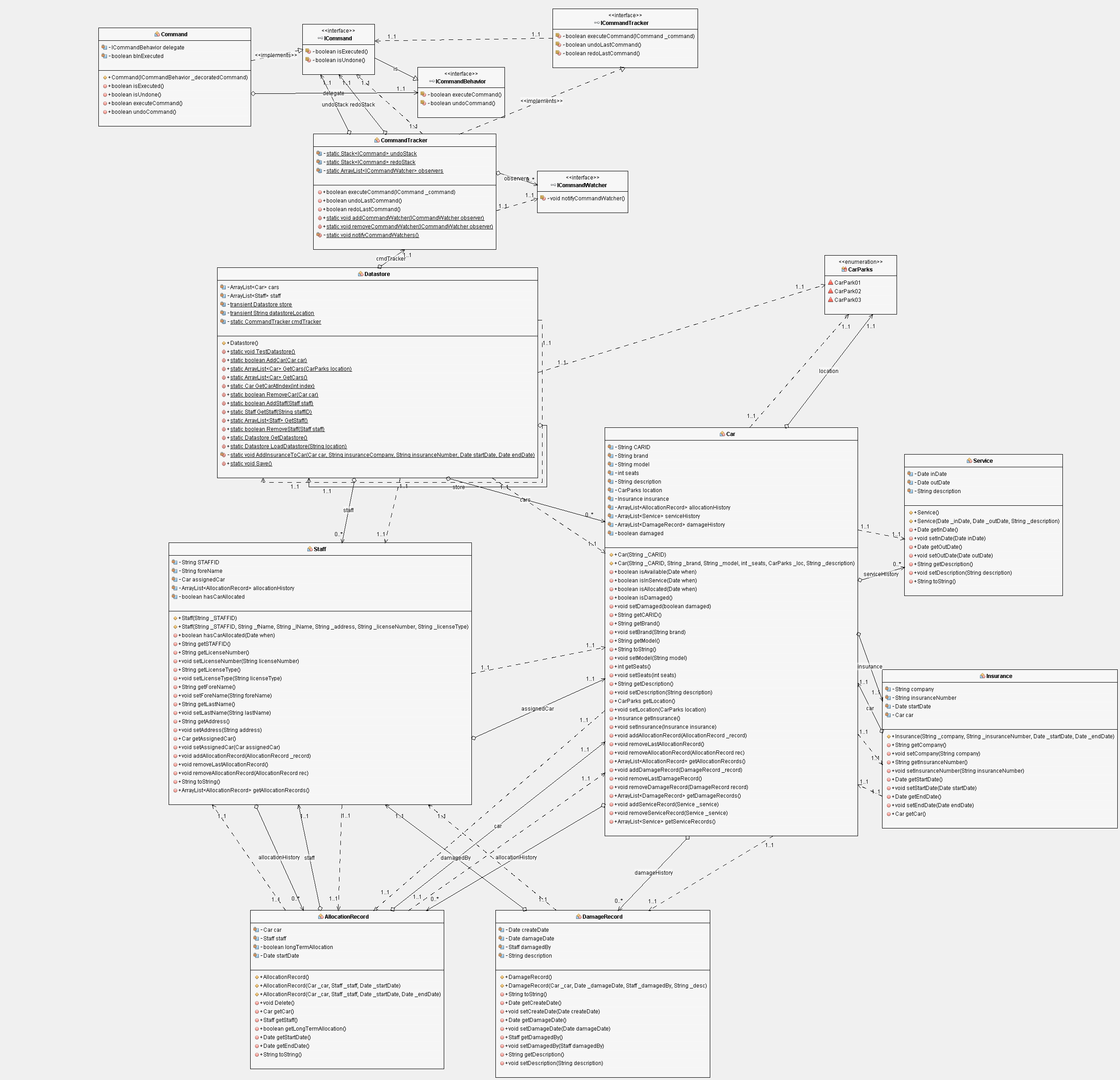
The assignment required us to implement our own datastore using object serialization. This method of storing data does not allow concurrent access and is easily susceptible to corruption. A relational database would probably be the preferred backend for this sort of application.

# Appendix A. UML Diagrams.

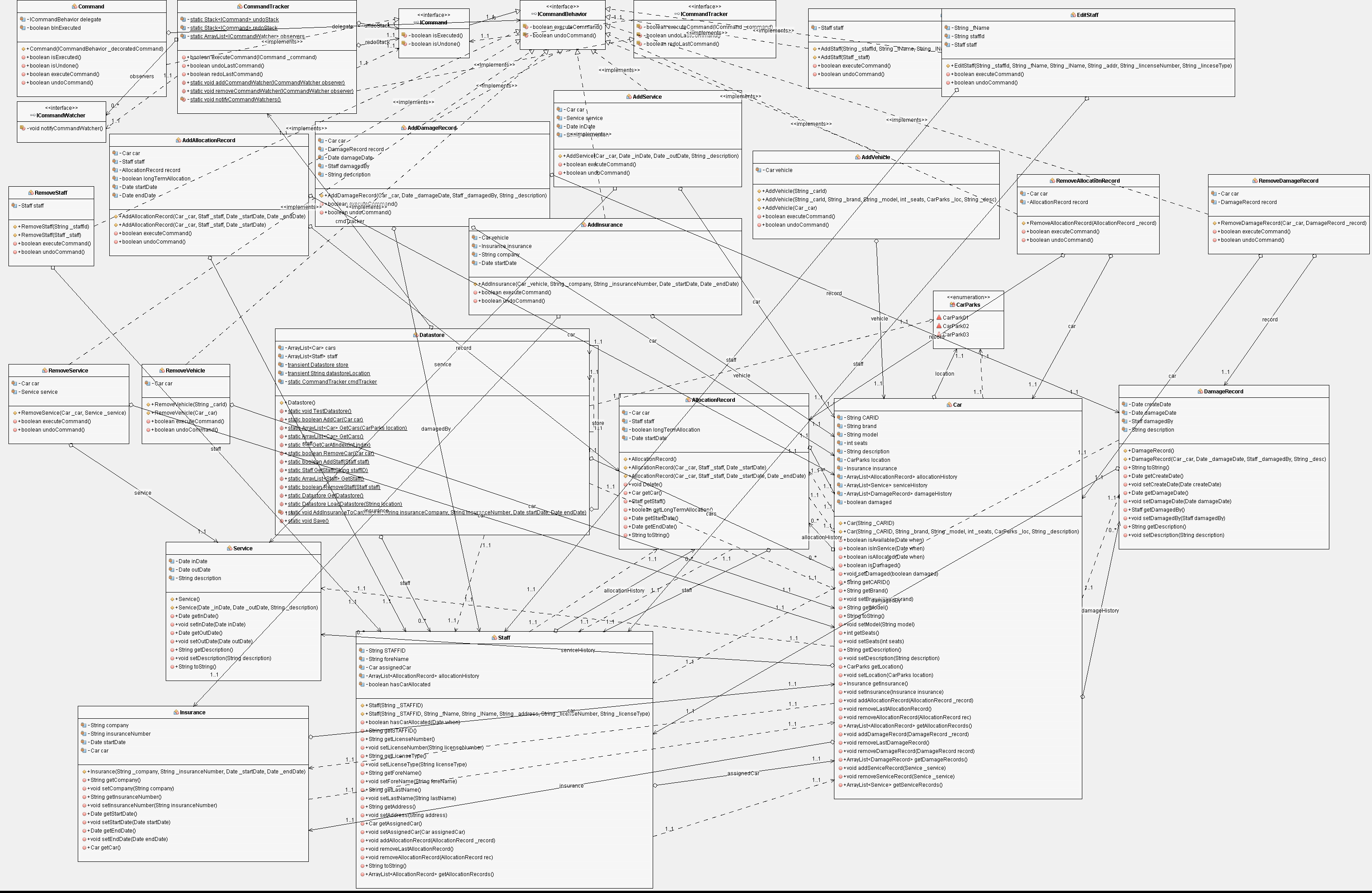
UML diagram of just the data model classes.



UML diagram of data model, datastore and command interfaces. This diagram clearly shows the code separation between the command execution logic that operates on the Datastore, and the data model stored within.

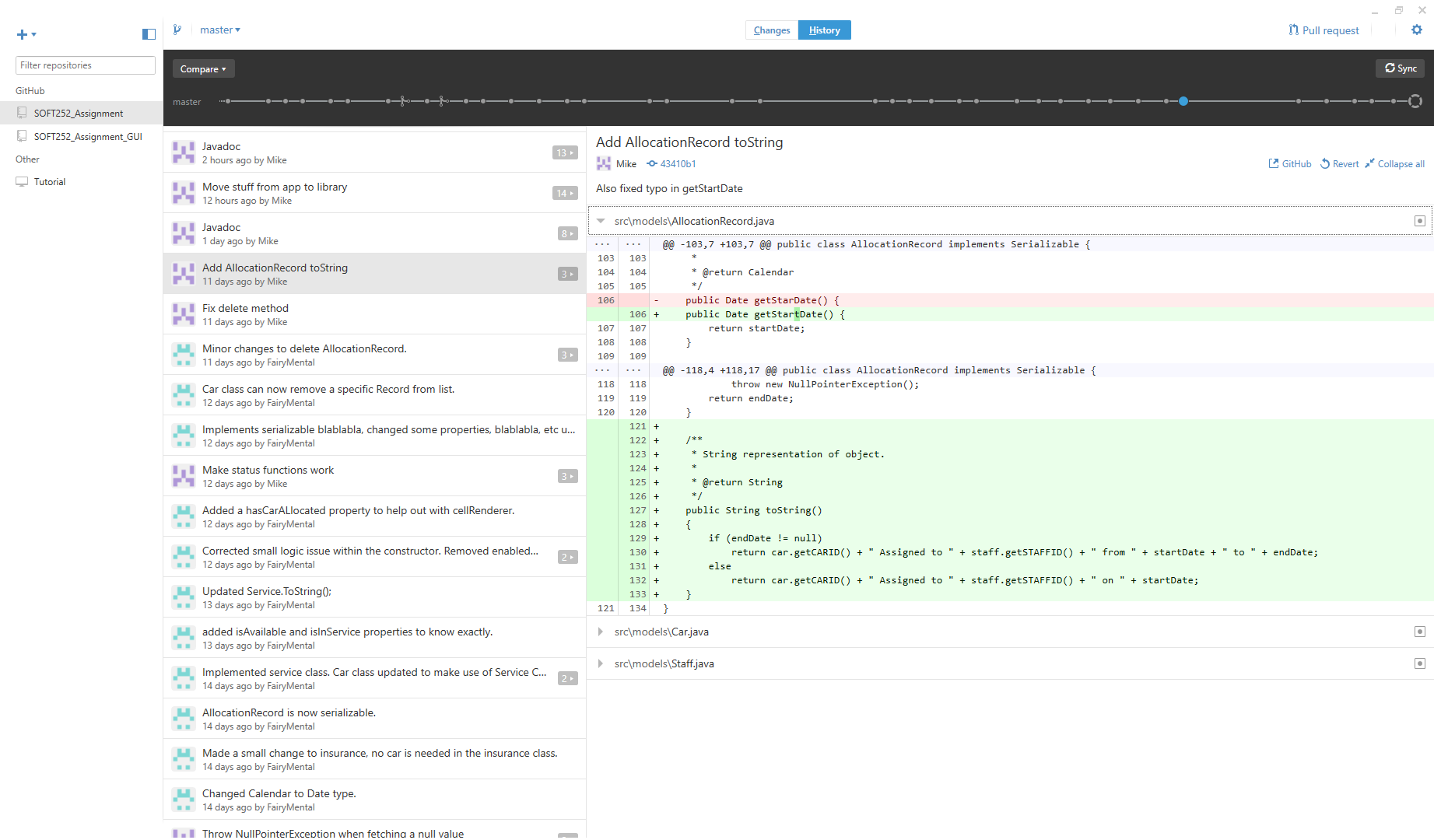


Full UML diagram of library. The classes that implement the ICommandBehaviour interface map between commands and various data model objects and attributes, resulting in a very difficult to read class diagram. Command Behaviours can be considered a separate category to the command and data model objects shown in the previous diagrams, and this diagram would be a lot easier to follow if rendered in 3 dimensions.



# Appendix B. Code repository.

Screen shot of git code repository used for collaborative development and code change monitoring. As well as helping to keep changes synchronised this also allows everyone to see exactly what has been done by others.



# Appendix C. The finished application.

See the included demo for full details.

