Initial Design

Please find ‘UC_RSS.jar’ in the main Directory. This is the executable Jar file demonstrating the application.

Overview:
The first design decision, before I even began to think about unit tests, class structure etc, was the focus on designing the application for mobile phones under the assumption that this is the end device that is targeted.
This meant that we had finite screen real estate. I had to create a page manager of some description, as many unique screens are needed in order to display the many requirements. This meant that I had to figure out not only how to use javafx (let alone Cucumber and Mockito and JUnit) But I had to do so in a way that allowed me to build additional screens easily and efficiently.

Each page consists of a fxml file, outlining the javafx user interface controls, a corresponding controller class, to handle page specific logic, and most also required a model specific to that page. (For instance the stop point model for each page manipulating stop points in one way or another).

The central controller class I have used is built in the main class. I thought about separating the entry point of the program and the main controller class but felt this was acceptable given the already narrow focus of the class. Main is where program starts javafx and caches all the static strings (links) of both controllers and relevant fxml files for use elsewhere in the program. Later these variables can perhaps be stored in an external file as being static is an ideal candidate for external storage. The Main class purpose is an entry point, initialization of javafx and it’s scene/stage control and the link or address to the various pages within the application.

The point of all this is that now in each controller class, I can easily load another page simply by first loading the data with Main.mainContainer.loadScreen(*link*) then, switching actual page content by calling the setScreen method in the local ScreensController myController.

Adding a new page after this setup was complete only takes a few steps. An extensive problem I faced when making this was the way javafx uses cellValueFactories, simply making a value display in a Table View was far more convoluted than perhaps necessary and I dwelled far too long on the multiple issues I had with Javafx having had only one or two weeks prior experience with it.

The second most important design decision was the use of a fake (faux) database.
This is simply a static collection of values that can later be rewritten as a real database. My FauxDatabase class contains a Set of StopPoints and a List of Rides.

The classes which make use of (either obtaining values from or writing to) the database will not have to be rewritten, only refactored.

These prerequisites are both important as if this was a real software development project for a company, it would save both time and effort and therefore capital for the company. Which is why I perhaps made this project a little more complex than strictly necessary yet resulted in it being incomplete.

I had a go at Cucumber and Mocking late in development to no avail. It was assumed that we knew this prior to the delivery of this assignment to the students. While the due date was extended, I was struggling having had only one semester of java experience and one lecture on mocking/cucumber is not enough. I was simply falling behind in other classes so decided to implement my isValid method of controlling tests instead rather than implement this.

Justifications

I created an interface for my objects that are required to be checked for user entry.
For the majority of my objects i had them implement iValidatable which simply forces a contract to implement a isValid method which returns a boolean. Since the validity will be varied for each object, I wanted this ensurance that each could be easily called from anywhere to determine whether they're valid objects. For similar contractual reasons I used an interface for setting a screens parent called iControlledScreen.

BackLog Details

1. As a driver, I want to register a car so that it can be used for sharing a ride.

Design Decisions:
The Car class was reasonably straightforward. It consisted of two public enums Type and Colour which I can add to later or perhaps store elsewhere. A colour chooser was first considered, then dismissed as needlessly excessive. It would be a nice addition on a final product however.
The class members consist of a SimpleObjectProperty of Type Car.Type and SimpleIntegerProperty NumSeats and Year and various SimpleStringPropertys: Model, Colour, and RegistrationNumber.

I also provided a section of private variables entitled 'Validity Criteria' Which i did not adopt for other classes that implements iValidatable due to time constraints but would be good to provide a consistent area to constraint validity checks of instantiated model classes. (See the code details [Class:Car lines 30-35] for constraints.)

Initially, before I realised I did not actually have to have any of my model collections as orderable (As using a TableView in JavaFX provides this functionality) I had the Car class implement Comparable<Car> which requires the overridden implementation of [**int compareTo(Car)**, **int hashCode()** and **boolean equals(Object)**] I also included an override of **toString()** that the compareTo method uses to compare between two cars for it's place in an ordered list.
This was not used however. It was over complex and not required.

The Driver maintains a Set of Cars as I want each to be unique. No two Cars should be the same. They all have differences. On thinking this through again, only the registration needs to be compared between two vehicles, not the complete string representation of the vehicle. Another note, I used the term 'Car' interchangeably with 'Vehicle'. It was my intent to change this to Vehicle yet at was understandably lower on the list than adding new features.

Why I chose the specific scenarios for each of your acceptance tests:
I used JUnit to test the constructor by instantiating one Car with given parameters, then a second one instantiated with the previous Car as a parameter. I then asserted that all class members were equal. I tested the setting and getting of the number of seats by providing variables outside the scope of the Validity Criteria section in the class. These ensured that a Car would fail the isValid() check built into the class. I also tested the setting and getting of the registration number by asserting that given parameters were equal with retrieved values.

For the steps I used the scenarios: ‘I am a driver’, ‘I register a car’ as this is done frequently and represents one of the core functions of the application - the ability to create your profile as a driver and the ability to provide a car for your trips / routes.

Why I did or did not unit test:
This one I did unit test as I wanted to ensure it was working correctly given that it used user input for its instantiation.

2. As a driver, I want to create stop points so I can specify where I will pick up or drop off passengers.

Design Decisions:
The StopPoint class utilizes SimpleStringProperty type for its class members. At different times throughout programming this, I required several methods of instantiating a StopPoint hence the Four constructors. This was the first model class I implemented and should have stuck to a solid method of creating a new StopPoint. But some are justified. As no definitive specification was given as to how a StopPoint was to be implemented, I separated the Number, Street, Suburb and Address as separate variables the class contains. The Address is unique in that it is a full string representation of the StopPoints location. This is created after a StopPoint is instantiated given only it’s Number, Street and Suburb and uses them to create an Address, although I had to account for the fact that a street or suburb may have one to two words in their particular string.

I decided to store the StopPoints as a set as we do not require multiple identical elements in the collection. Perhaps naive of me, in my code I still check for an element’s existence before attempting to add it to any collection. If I have time, (unlikely) i’ll refactor the getters for various collections as unmodifiableSets. I believe this is called an encapsulation leak.

The Passenger and the Driver store StopPoints as HashSets as I wanted an orderable yet unique collection. This was decided prior to creating the FauxDatabase (Which stores them as a simple set) It was later discovered after implementing a JavaFX TableView that I do not actually have to have them orderable at all. The UI takes care of displaying the Table’s elements in ascending or descending order given their string representation.

Why I chose the specific scenarios for each of your acceptance tests:
I used a scenario ‘Create a stop point’ that creates a new empty StopPoint, then fills it with an Address, it is then added to a route and we attempt to add it again, checking that we cannot add duplicates to the routes StopPoint Set.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

3. As a driver, I want to add a route so I don’t have to re-enter the stop points I will pass for every trip.

Design Decisions:
The Route class contains a Set of StopPoints, this was originally a hashSet yet I recalled that we should store the most high level collection as possible. Also included is a SimpleStringProperty ID to display the route’s ID in JavaFX and a default value as I enable the user to create a route without providing a title. This is a classic example of feature creep, it is NOT in the specification. It was implemented because I was neck deep in implementation and not considering the overall progress of the project and sounded (at the time) like a good feature.

The Validity Criteria provided is simply a minimum size that if a route is to be valid, it must have at least two StopPoints. (otherwise by definition, it is not a route.)

The route is stored only in a Trip as a class member.

Why I chose the specific scenarios for each of your acceptance tests:
Since the Route is simply a collection of StopPoints, not much testing was required. It is included in the scenario ‘Create a Stop Point’ where I add a StopPoint to the route. It is then tested whether the size of the route’s StopPoint collection matches the amount provided.

Why I did or did not unit test:
Due to the simplicity of this class, I did not spend much time testing it.

4. As a driver, I want to add trips so that I can set up all the information for the rides that I will share later on.

Why I did or did not unit test:
I figured the trips would be easy to test, I tested the constructors, ensured they instantiated equal objects. I ensured expiry dates were valid and couldn’t be before now() and after five years. Any other date between was ok. In the application, an error in defining the date is displayed to the user. I also ensured instantiated routes, days list and route stop points were correct.

Design Decisions:

The Trip class was a complex one. It contains a route, Direction (which is a public static enum that is used elsewhere in code - This should be removed from here and placed in somewhere more ‘central’ perhaps) a boolean denoting whether its recurring or not, an expiryDate (stored as a LocalDate) a Car and the following as SimpleStringPropertys: TripID, CarID, RouteID, DirectionID. I understand that the direction and various other ID’s can be obtained from the previous class members yet I had a lot of trouble pinpointing exactly why JavaFX refused to populate various UI elements. This was the result and removing these additional members and refactoring / tidying the class was a lower priority than adding new features. As a result, they remain.

I have several constructors as Java does not support default parameters for constructors, however the builder pattern may have helped yet I didn’t know it at the time. Again i’ve required throughout the program to be able to instantiate a Trip given various parameters. Again, the way in which we create a Trip should be standardized. This was also on the To Do list.

The Trip class implements my interface iValidatable which provides a contract to implement an isValid() method return a boolean. I used these for any model classes which were to be instantiated given user input. I felt this was necessary as it enforced a standard to be followed for any new classes. Here in the Trip class, I forced any Trip that was created to a route that was not empty, it had to have a direction. On thinking this through, (away from the complexity of coding), I realize I could have chained together isValid() calls and simply requested whether route.isValid() and iterated route.getStopPoints() checking for validity also.

5. As a driver, I want to share a ride so that it can be displayed to potential passengers.

Design Decisions:
Originally I did not understand what this requirement implied. So when the time came to implement this, my code was not compatible.. Numerous refactoring and rewriting was required and a reassessment of the difference between a Trip and a Ride was undertaken. The Ride is simply an instance of one StopPoint in a Trip with a given Time and place. When a Trip is shared, each StopPoint in that Trip essentially becomes a Ride. I have not, like others, added a separate JavaFX page for this purpose. **To share a ride you log in as a Driver, navigate to View Trips, select a Trip and click ‘Share Trip’.** Had I time I would have implemented this for ease of use.
The actual Ride class contains SimpleStringPropertys Day and Time, it contains a Driver, a Car a StopPoint and a Direction. The Ride class also implements iValidatable as we must ensure that a Ride is valid. It also implements Comparable<Ride> as although given its collection type (Set) duplicate elements shouldn’t exist, I created this class before FauxDatabase which is the only place Rides are stored and was at that time, unsure of the collection type.

Why I chose the specific scenarios for each of your acceptance tests:

I did not. See below.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

6. As a user, I want to search for existing stop points so that I can define or find rides.

Design Decisions:
The search is within the class [**PassengerViewStopPointsController.java : public void search()**] I retrieve the text from the user, they may search for the number, the street or the suburb or any combination of the three. I then create a temporary ArrayList of StopPoints, and cache the trimmed and convert-to-lowercase user input.
I then iterate over all StopPoints in the table (All StopPoints recorded in the FauxDatabase) and check whether the user text is contained within either the number, street or address for the current StopPoint. If so, it is added to the temp ArrayList and the display is updated. I overengineered here also. The StopPoint could have been a complete address as a string and simply checked if it exactly matched a given user input. Yet at the time I felt this was important, and that the user had more control over their search.

Why I chose the specific scenarios for each of your acceptance tests:
I did not. See below.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

7. As a passenger, I want to see the available rides at a specific stop point so that I can choose one.

Design Decisions:
The user first selects to View Rides and is taken to a page displays all StopPoints in the FauxDatabase, the page allows the user to search for a particular StopPoint given Number, Street or Suburb. When clicking on the Find Ride button the application takes the user to the View Ride page displaying all individual Rides pertaining to the selected StopPoint. I implemented a toggle button in order to filter by direction. Clicking the Book Ride button takes us to the ‘Book A Ride’ page which is incomplete. But it is there that the user may book a ride and this event would notify the driver and update the Database relating to the number of seats available.

Why I chose the specific scenarios for each of your acceptance tests:

I did not. See below.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

8. As a passenger, I want to see the details of a specific ride so that I can choose the best ride for me.

Design Decisions:
This is unimplemented yet the foundation and structure is all there. Just requires building and testing.

Why I chose the specific scenarios for each of your acceptance tests:
I did not. See below.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

9. As a passenger, I want to book a ride so that I can get to where I am going.

Design Decisions:
This is unimplemented.

Why I chose the specific scenarios for each of your acceptance tests:
I did not. See below.

Why I did or did not unit test:
Out of time. Deathmarch. Stress.

Release 2 Tasks were not completed.

Summary

Even though I didn’t not get around to doing the second part of this assignment or even completely finish the first section and did not understand how to correctly implement mocking or Cucumber, I learned a great deal about software development. What it is really like to work under pressure to specific user story requirements. I also now know the effects of feature creep and now know the feeling of what a deathmarch actually means for the developer. (something which surely is critical knowledge for anyone entering a development team) and can appreciate TDD now that i’ve had my first experience of this methodology.

I think everybody must have underestimated the depth of this assignment and Java, JavaFX knowledge required. I feel that we were not ready for this undertaking and both the lecture notes and lecture content, nor the previous Seng201 content prepared us for this increased level of complexity. This is a perfect example of a deathmarch where the team (me) was undertrained and the scope of the project, given the time constraints, was too high. Perhaps a requirement of Seng202 would be useful for future Seng301 classes, or weekly online Java tests to ease students into an assignment of this scope rather than produce extreme stress and causing the student to burn out.

Prior to handing this assignment in, while attempting to solve an fxml path name issue in netbeans, I accidently deleted my latest copy, which included the handling of rides and about ~30% of other additions.. Given the due date and the other assignment I will hand in my old copy.