Creating a Software Design For a Federated Bike Rental System

Coursework 2

Informatics 2C - Introduction to Software Engineering

Justin Howe
s1840358

Rohan Nittur
s1803949

Contents

1. Introduction

The following document describes how the system should meet the requirements for a federal bike rental system. The system state shows a unique relationship between the user's personal details, information about every booking and bike provider's shop, and the status of each bike in the shop. Modelling the system will be split into two parts: a static model and a dynamic model.

2. Static Model

2.1 UML Class Diagrams

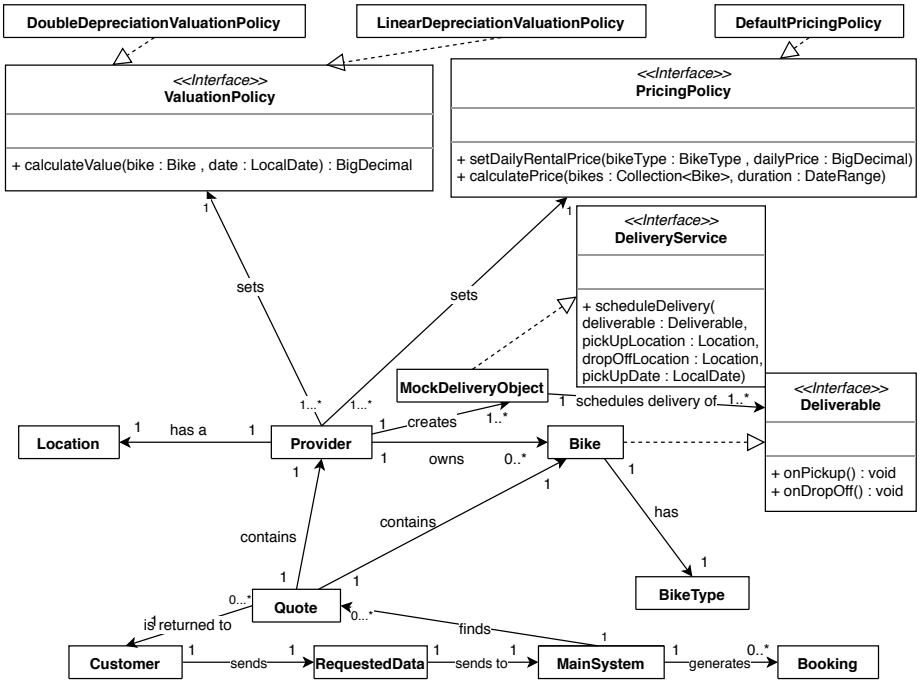

Figure 2.1: UML Class Diagram showing identified classes and associations corresponding to the 4 required use cases.

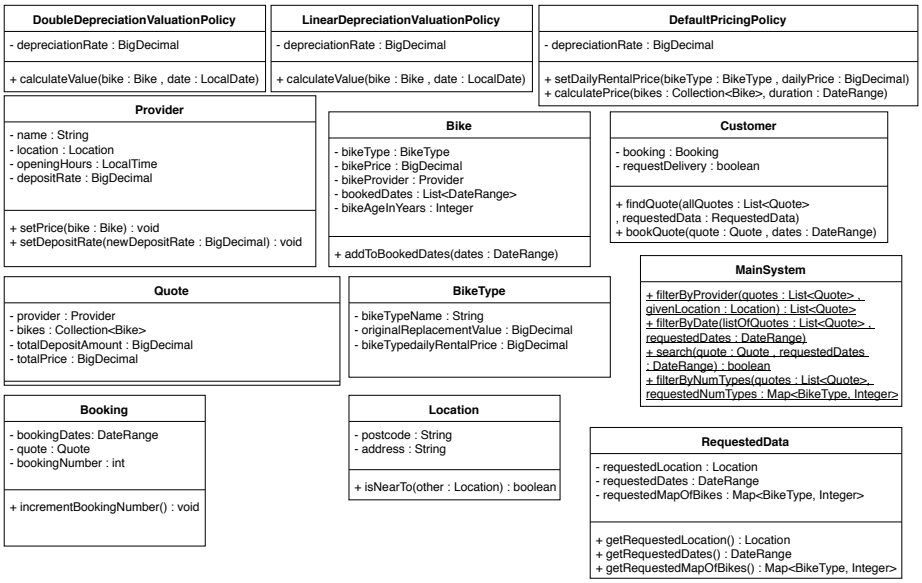

Figure 2.2: UML Class Diagram showing the fields and methods of classes corresponding to the 4 required use cases.

2.2 High Level Description

Modelling the UML Class diagram required additional attributes, operations and associations to other class objects in the diagram. In deciding on the classes to be used in the design, we have strived to keep to the object-oriented design principles of *abstraction*, *encapsulation*, *inheritance* and *polymorphism*.

One particular such design decision was the inclusion of a `MainSystem` class. The purpose of this class is to abstract away and encapsulate the details of performing most of the operations within the system such as searching and filtering. More generally, it acts as an intermediary between classes representing the user, and those intrinsic to the system so as to ease the readability and better conform to the principles. This can be demonstrated with the *Find Quotes* use case. It is the customer who carries the responsibility of requesting quotes so the method `findQuote()` therefore lies in the `Customer` class. However, the actual implementation of searching with respect to the customer's required search parameters is carried out by the filtering methods within the `MainSystem` class. Thus, the details of the actual search operation are abstracted away from the `Customer` class. Another scheme to consider in reality (not implemented in this coursework) is interaction with the system database. The `MainSystem` class would again act to abstract the details of doing such away, and encapsulating all methods for doing so. One advantage of this is that most of the code for system operations as described in the use cases is all in one place and is not repeated elsewhere in the code, as would be the case in a design without this class.

This is helpful not only for debugging, but also in a foreseeable scenario where the mechanism to connect with the database changes. Any and all methods that might require changing are all easily stored inside this class, making the job easier than if they were scattered around the other classes. In practice, this class would have several more methods than are outlined in this UML class diagram, but in the interests of keeping operations limited to only the 4 required use cases as the task described, only the methods specifically utilised by one of these 4 use cases are shown.

As seen in the UML Class Diagram, all of the methods in the `MainSystem` class are static. This ensures that all the methods belong to the class, rather than particular individual objects of the class, as `MainSystem` is only instantiated once. The system operation methods within `MainSystem` can then be called as class methods from any place in the system.

Two classes of system users were identified: the `Customer` and the `Provider`. These users will generally utilise the system for different purposes, so have their own classes that implement their respective methods. A design we considered was to have both these classes inherit from a general `User` class, a design which would allow further conformation to the principle of *inheritance* and *polymorphism*. However, we decided against such an implementation mainly due to the fact that the main functionality these classes would share in reality is access to the system database; this is not mentioned at all in the coursework so this idea was scrapped. To determine in which class certain identified methods would reside, the responsibilities of each type of user were considered. For example, it is the customer who physically requests and books quotes, so it would be sensible to include methods which pertain to these actions under the `Customer` class so that it more logically mirrors the real world workings of the system.

To represent the parameters that the user wishes to conduct their search for available bikes with (number, type, date range etc.), a separate class was created to model and encapsulate these requirements. The `RequestedData` class contains the required information ascertained from the customer to complete the search. The number and types of the bikes is stored in a `Map`, represented in the UML class diagram by the private variable `requestedMapOfBikes`. This stores the types of bike requested by the customer, along with the number of each type, and is used to match the particular bike requirements of the customer to a valid quote. Since this class only serves the purpose of encapsulating the data into a single object for readability, it contains no methods of its own, other than getters and setters.

The `Quote` class is utilised to bind together those details which constitute a quote as required by the description. It contains a `Provider` (we assume that all bikes in a quote come from the same provider), a `Collection<Bike>`, a deposit amount and a price. Similar to the `RequestedData` class, this class encapsulates the relevant data away into a class of its own for readability and adherence to the OOP principles. Hence, no methods are present in this class that affect the functionality of the system (only getters and setters).

An alternative to this overall design would be creating a `BookingController` class and a main `System` class and associate those two classes together along

with a `QuoteController` class. However, this would not be as effective as its overarching complexity would result in low cohesion and high coupling which contradicts our design philosophy.

A particular ambiguity that was discussed in the Requirements document was the lack of procedural details in the event of a physical transaction, i.e. if a customer walks into a store and requests a bike for rental. In such a case, we reasoned that a provider should have the option of carrying out the details of the request and then subsequently following through with the transaction. A large assumption that we made was that the provider would then limit the search to their own shop and those nearby (as defined by postcode), as the customer has walked in with the expectation of renting bikes at that very point in time, and would probably not be prepared to travel to farther store. However it is trivial to change these details. In this case, our implementation asserts that the provider uses the system to search for the requested bikes in their own store and nearby by supplying their own location in the search field. If the search returns a quote, payment would be made in-person and the provider would update the system manually using the `setBikeAvailability()` method to record its booking

3. Dynamic Models

3.1 UML Sequence Diagram

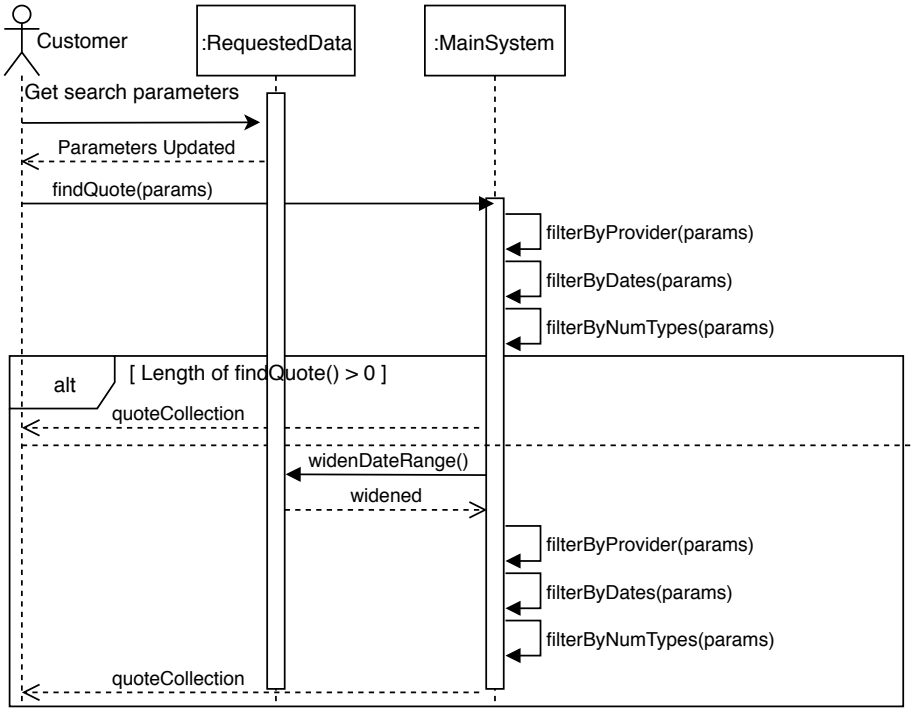

Figure 3.1: UML Sequence Diagram showing showing chronological messages for the Get Quotes use case.

3.2 UML Communication Diagram

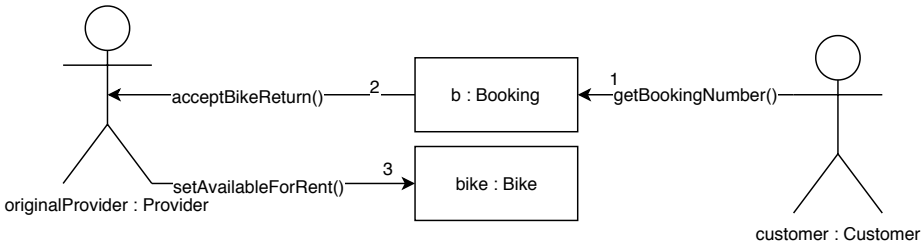

Figure 3.2: UML Communication Diagram for the Return Bike to Original Provider use case.

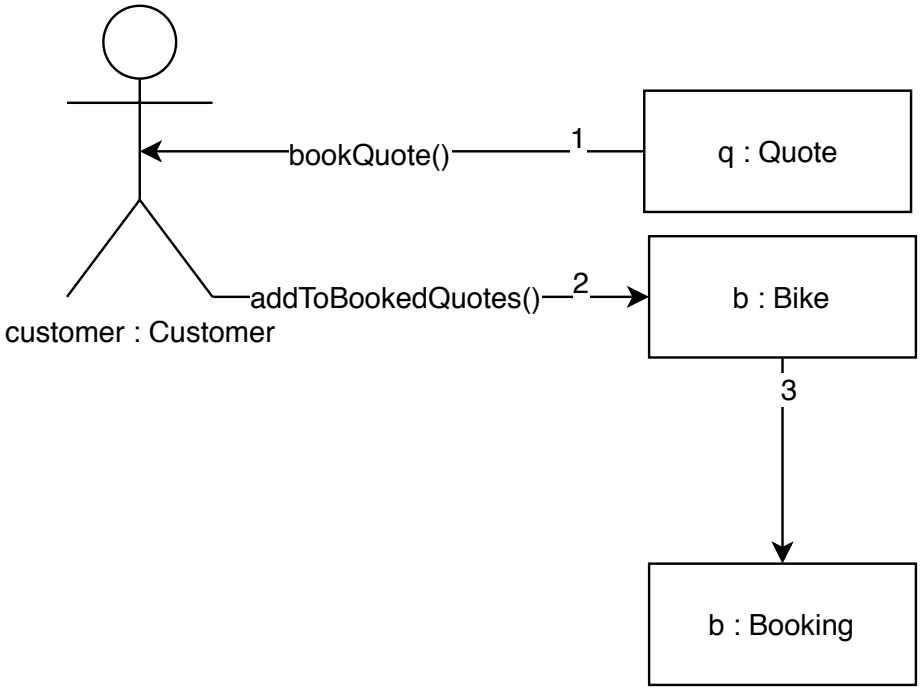

Figure 3.3: UML Communication Diagram for the Book Quote use case.

3.3 Changes to Diagrams

All of the diagrams were modified taking into account the comments of the marker. The sequence diagram now contains a label and a lifeline for the actor,

and the erroneous `not update()` label has been removed. The initial loop has also been removed due to its lack of purpose. The messages have been ordered such that the condition is known before going into the alternative frame and the classes have been updated to reflect the new classes and method names and functions.

The UML Communication diagrams were entirely re-done, after re-examination of the methodology of these diagrams and the use cases they apply to.

4. Conformance to Requirements

Overall the design produced in this document conforms well to the requirements specified in the requirements documents. The system stores most of the data outlined in the System State section in Coursework 1, with the few minor discrepancies and changes discussed below.

There were certain issues in the system requirements document that needed to be addressed and fixed. In particular, the system state needed to properly fulfil the information required from the system description. Such examples of information included the status of the bike availability, daily price and type of each bike from the providers, This information was then used to create the UML class model and sequence diagram in particular.

A particular detail that required changing was the method by which a provider could recored themselves partners with another. The broad design specified by the description in the requirements document ultimately produced duplication of information, as each of two providers records the same partnership. This has now been changed such that only one provider need record it, preventing this redundancy.

Further to this, the design of booking a quote was changed such that it does not return a confirmation object as was perhaps hinted to in the use case descriptions for Coursework 1. We found creating a whole class for the trivial amount of information necessary to be a useless errand, as the operability and readability of the system and system code respectively were not compromised.

In the use-case section, the full use-case template for Book Quote needed to include extensions and a fuller detail of the main scene scenarios. This was also then used to create the UML class model diagram. This would therefore show that the software design plan and the requirements documentation would be aligned.

5. Modified Design

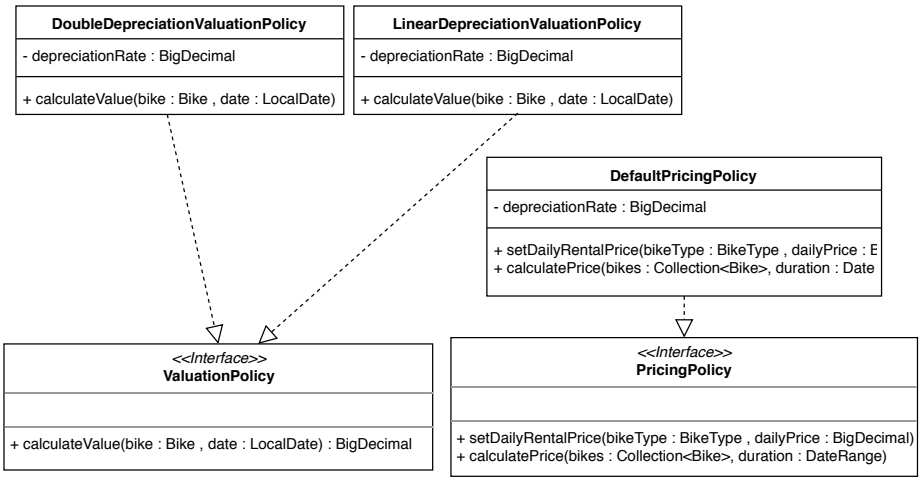

Figure 5.1: Part of the UML Class Diagram showing the extension interfaces and relevant classes that implement them. The connections between them is shown here, and the integration with the whole system is show in Figure 2.1, and described in the High Level Description

6. Self Assessment

6.1 Assessment

§1 Static Model **20%**

- Make correct use of UML class diagram notation 5%
- Split the system design into appropriate classes 3%
- Include necessary attributes and methods for use cases 3%
- Represent associations between classes 4%
- Follow good software engineering practices 5%

§2 High Level Description **10%**

- Describe/clarify key components of design 7%
- Discuss design choices 3%

§3 UML Sequence Diagram **15%**

- Correct use of UML sequence diagram notation 5%
- Cover class interactions involved in use case 7%
- Represent optional, alternative, and iterative behaviour where appropriate 3%

§4 Use Case Diagram **15%**

- Correctly use UML use case notation 5%
- Include key actors and use cases 5%
- Identify connections between actors and use cases 5%

§5 UML Communication Diagram **15%**

- Communication diagram for record bike return to original provider use case 8%
- Communication diagram for book quote use case 7%

§6 Conformance to Requirements **5%**

- Ensure conformance to requirements and discuss issues 5%

§7 Design Extensions **8%**

- Specify interfaces for pricing policies and deposit/valuation policies 3%
- Integrate interfaces into class diagram 5%

§7 Self Assessment **10%**

- Attempt a Reflective self-assessment linked to the assessment criteria 5%
- Justification of good software engineering practice 5%

6.2 Justification

This document paints a clear picture of how a federal bike rental system should be designed. Creating the UML class diagram to cover all use cases proved to be quite challenging as we had to use the CRC-card technique to find all valid classes for the model. In addition, the UML sequence diagram showed the process of obtaining the quotes in the system. Even though there were uncertainties in the diagram including the order in which the methods were called, our team was still able to accomplish the task. The design extensions having to read these specific use-cases and create multiple interfaces to finalise modifying the already-existing UML class model.

Our software engineering practice was certainly implemented in this documentation. We followed the design rules where strong cohesion and low coupling was significantly important for the UML class model. Our team did not want to implement unnecessary classes as keywords from the system description. In addition, we needed to use encapsulation for the class instance variables.