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 Diagram

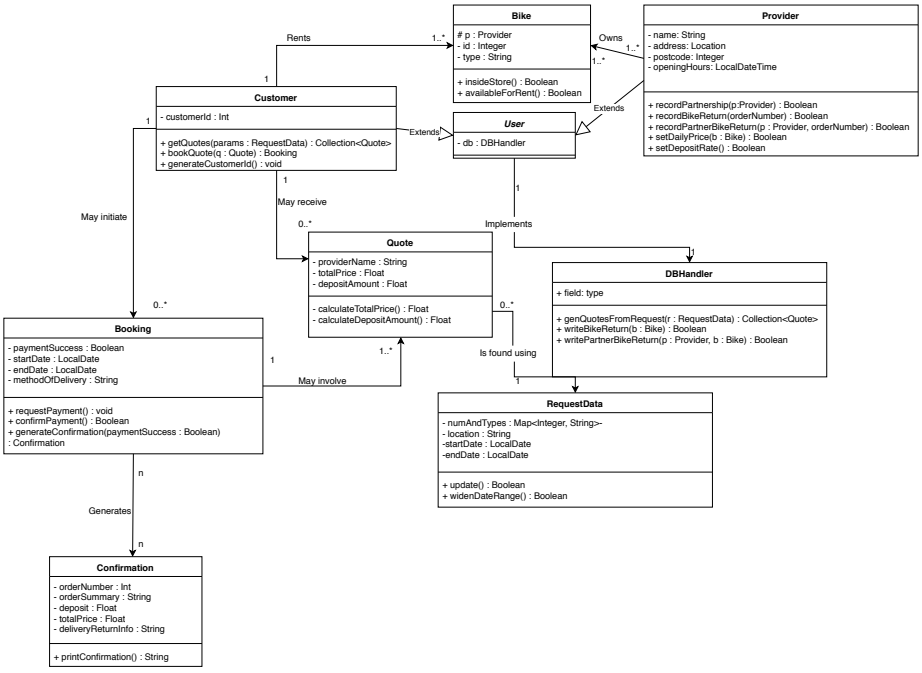

Figure 2.1: UML Class Diagram showing identified classes and associations along with fields and methods corresponding to the 4 required use cases.

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 `DBHandler` class. The purpose of this class is the abstract away and encapsulate the details of writing to and reading from the system database. Thus, the actual interaction with the database is reserved for methods of the `DBHandler` and no methods external to this class can directly access or modify the database. One advantage of this is that all the code for reading from and writing to the database 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. An example of how this class may be utilised is the following: Suppose the customer wishes to request quotes. The customer object would call the `getQuotes()` method with the appropriate argument (covered below). This method will then invoke the `genQuotesFromRequest()` method of the instantiated `DBHandler` object that the `Customer` class contains, which performs the prats of the search relating to the database, meaning none of the `Customer` class methods need actually access that database, as intended.

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 subclasses that implement their respective methods. However, both (every user) will need to be able to access the database, and so the superclass `User`, from which both these classes inherit, contains an instance of the `DBHandler` class for this purpose. 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 these parameters. The `RequestData` 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 `numAndTypes`. This stores the types of bike requested

by the customer, along with the number of each type. The method `update()` in this class serves the purpose of updating the data in the `RequestData` object before a search is conducted, so that the search uses the most recently input parameters.

An alternative to this 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 it would result in low cohesion and high coupling which contradicts our design philosiphy.

3. Dynamic Models

3.1 UML Sequence Diagram

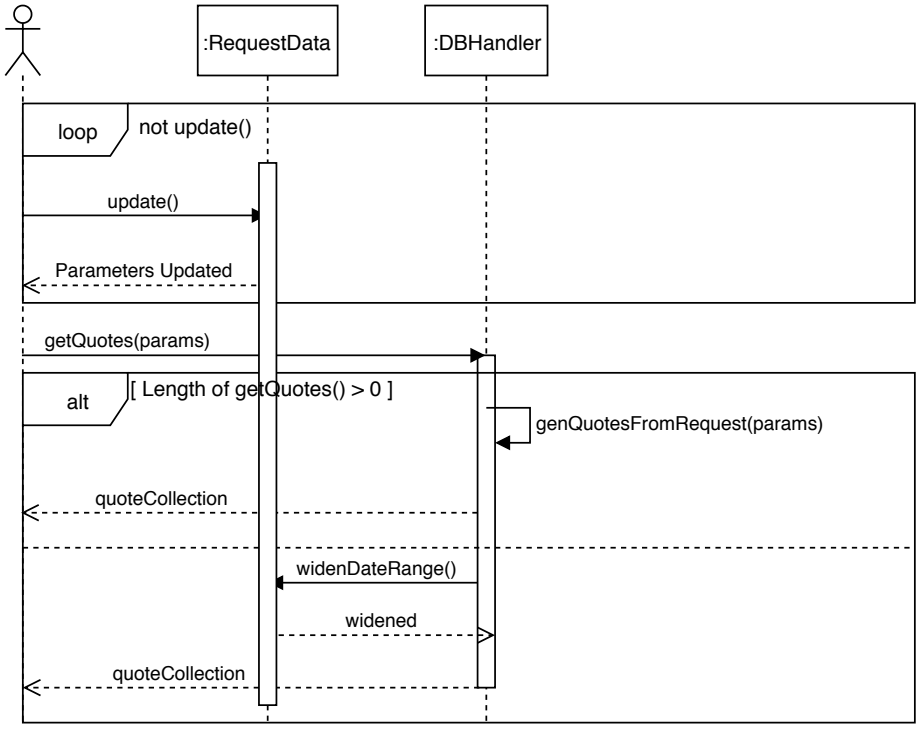

Figure 3.1: UML Sequence Diagram showing showing chronological messages for the Get Quotes use case.

3.2 UML Communication Diagram

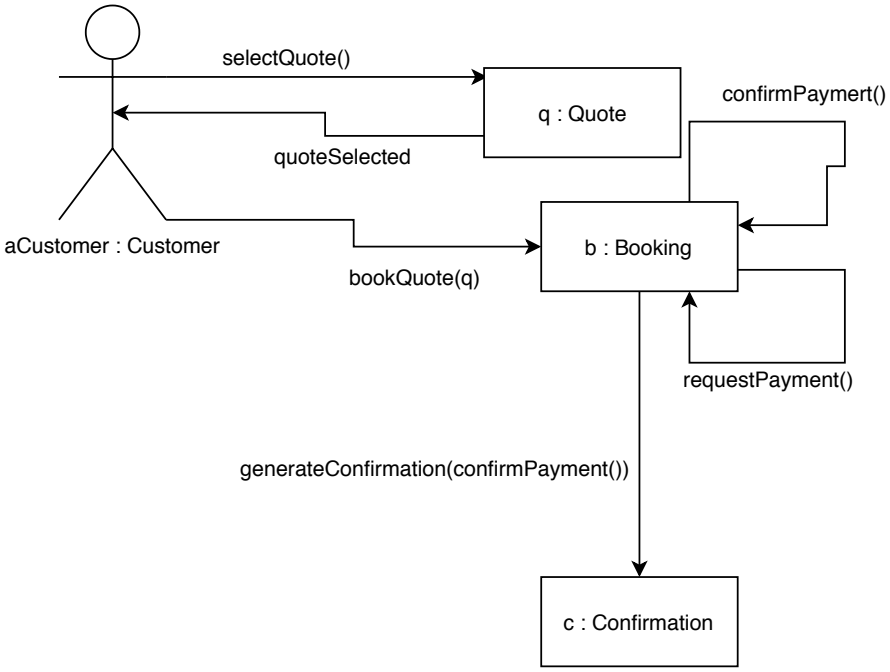

Figure 3.2: UML Communication Diagram for the Book Quotes use case.

4. Conformance to Requirements

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. 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

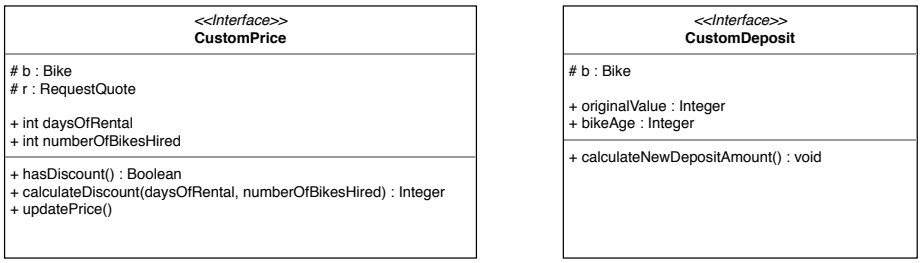

Figure 5.1: Part of the UML Class Diagram showing the additional parts.

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%**
• Include state essential to the operation of the system	5%
• Include additional state mentioned in the description	5%
§3 Describe Use Cases	**37%**
• Identify use cases	10%
• Describe use cases using the appropriate templates	27%
§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 Describe Non-Functional Requirements	**10%**
• Identify non-functional requirements within the context of the system	7%
• Provide reasoning for assessing non-functional requirements	3%

§6 Ambiguities and subtleties **5%**

- Identify some ambiguities in the system description 3%
- Discuss potential options for resolution in ambiguities 2%

§7 Self-assessment **5%**

- Attempt a reflective self-assessment linked to the assessment criteria 5%

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 finalize 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.