Inf2C - Software Engineering Coursework

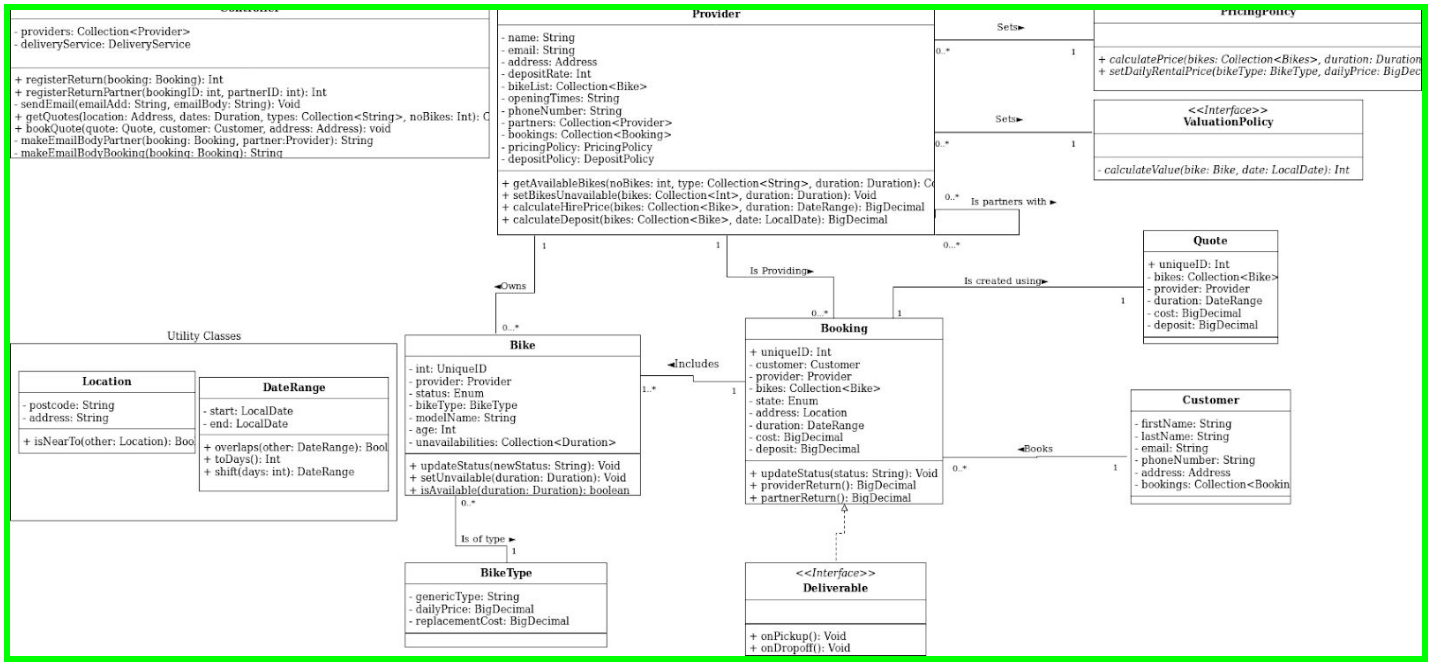
Software Design Document

Team members:
Martin Lewis - s1824863
Alasdair MacGillivray - s1837803

This design document covers the design for a federalised bike hire system allowing for the hiring of bikes to people online and for them to be returned to partner locations. This should allow for a much easier experience in hiring bikes and returning them. See the coursework documents and the previous requirements document for a more detailed explanation of the system.

Static Model

UML class diagram



High level description

General decisions
All attributes relating to price are BigDecimals, representing price in pounds

We have seperated utility classes, used for simply encapsulating information as it is passed between other classes, into their own corner of the diagram.

The “is created using” dependency in the diagram, is simply intended to show that the attributes of the leftmost class in the relation are somehow decided upon in part thanks to the rightmost class

Controller class
We decided to create the controller class in order to have all of the control logic for the system in a single place. It contains methods, intended to be called by some abstract user interface (not modelled in this coursework), that operate on the system model.

The bookQuote method includes an address parameter, which is intended to be left as Null unless the customer wants the bikes delivered. It it is given an actual object, then the bookQuote method will go through additional functionality relating to delivery scheduling. This also means that address is not associated with any customer object, which is important if delivery to an address other than the customer's normal residence is desired.

The registerReturn and registerReturnProvider methods return a BigDecimal specifying how much deposit is owed - the UI can then display this.

Provider
The provider class models information relating to bike providers, and has methods allowing the controller to extract and alter specific pieces of that information.

The getAvailableBikes method is called by the controller as it is searching for quotes, and returns a set of bikes suitable for the quote. Each provider only returns one set, even if others might have been suitable, in order to reduce the information a user has to take in.

We made the decision that a provider is allowed to have zero bikes associated with it, as it's possible to imagine a situation where this might arise (for example, a small provider that exclusively operates during summer sells off their stock at the end of the season, intending to buy newer bikes at a later date).

We decided that all providers would be associated with pricing and deposit policies, and that default ones would simply be provided on first constructor call, in order to simplify the system.

Providers only have a single address, as we assumed in ambiguity 2

Bike
Each bike object models an individual bike within the system, and contains information specific to that bike, including times when it is booked out and its current status.

As we stated in ambiguity 3, we have extended upon the idea oa f bike type - bikes have an actual type, what we referred to as a 'model' in ambiguity 3, and a generic type ("Mountain Bike", for example), . Bike Type will be implemented as a utility class (and each bike type stores generic type as an attribute), a list of them would be stored in the system, and they would be passed to the bike constructor as a parameter - this was left out of the UML diagram as it was not relevant to the use cases we were asked to consider.

Unavailabilities is simply a collection of DateRange objects for which the bike is unavailable.

Booking
The booking class is how we record the bookings our customers make.

The address attribute stores the address that the booking is to be delivered to, and is left Null if delivery is not required

The providerReturn and partnerReturn methods are called by the registerReturn methods in controller. The BigDecimal they return is directly returned by the method calling them, and corresponds to the deposit owed.

Address:
The address class stores a collection of information that is intended to allow for a specific location to be located.

The distanceTo method was added as a result of our decision in ambiguity 7, as we would need it for determining if an address is in deliverable range. Coursework 3 clarifies this to mean that two address are in range if the two if the first characters of the postcodes are the same, so a method is used isNear() to check this.

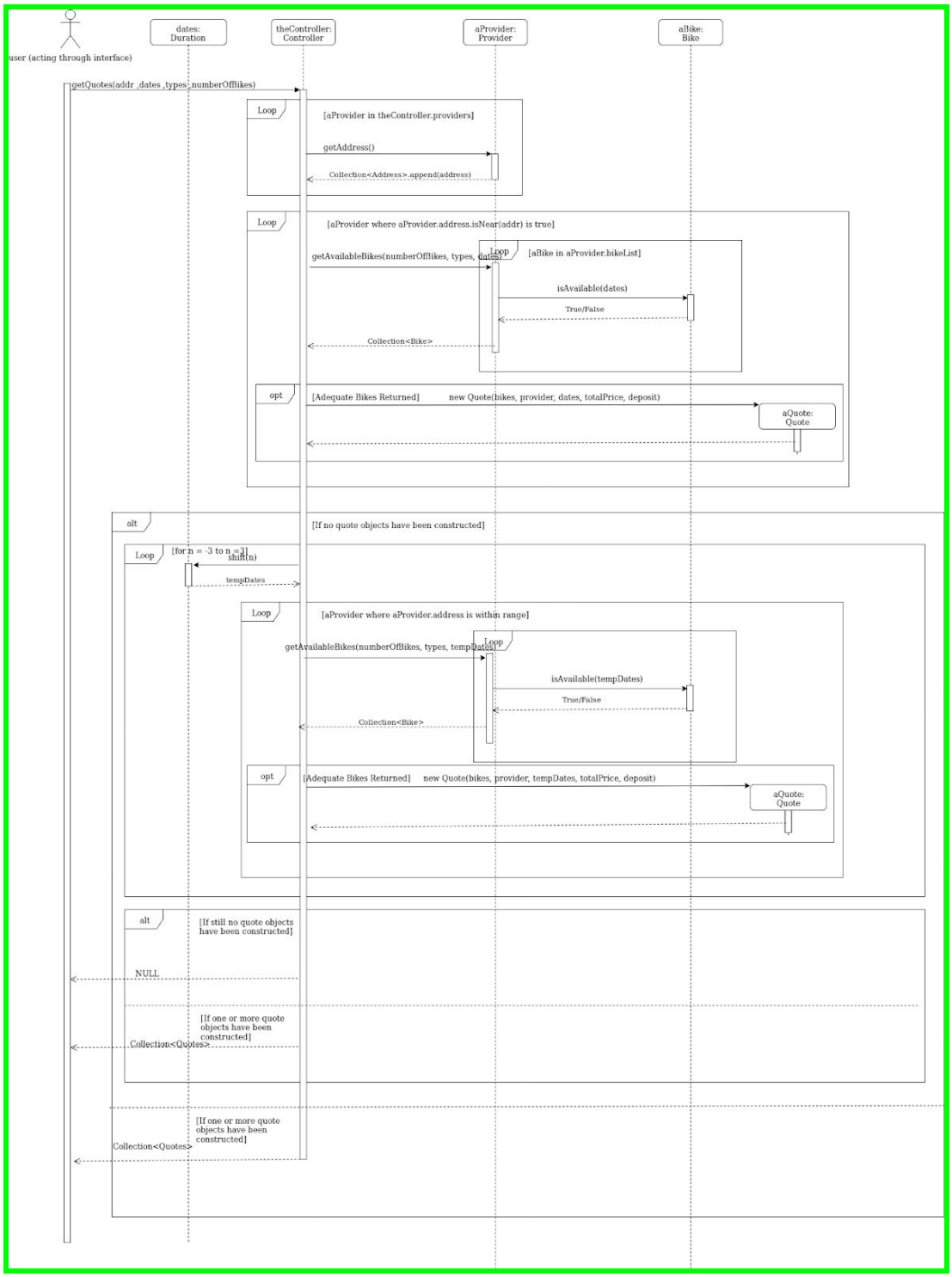
DateRange:
The DateRange class stores a start and a end date, and contains methods for getting information about them, and comparing to other dates

The shift method within duration returns the duration it was called on, but shifted by a number of days specifies by the days parameter.

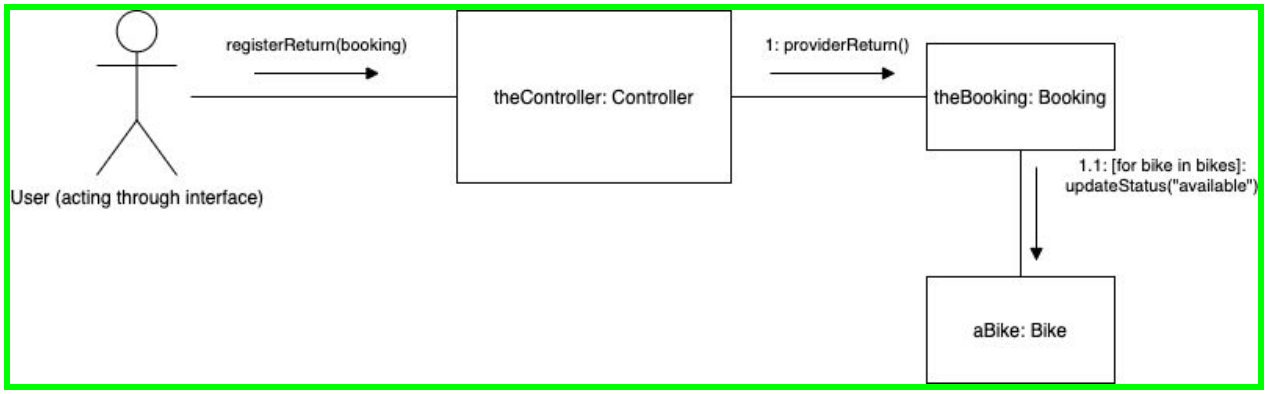
The Overlaps method takes another date range, and returns true if any days contained within that dateRange (including start and end) are contained within the DateRange it is called on.

Dynamic Models

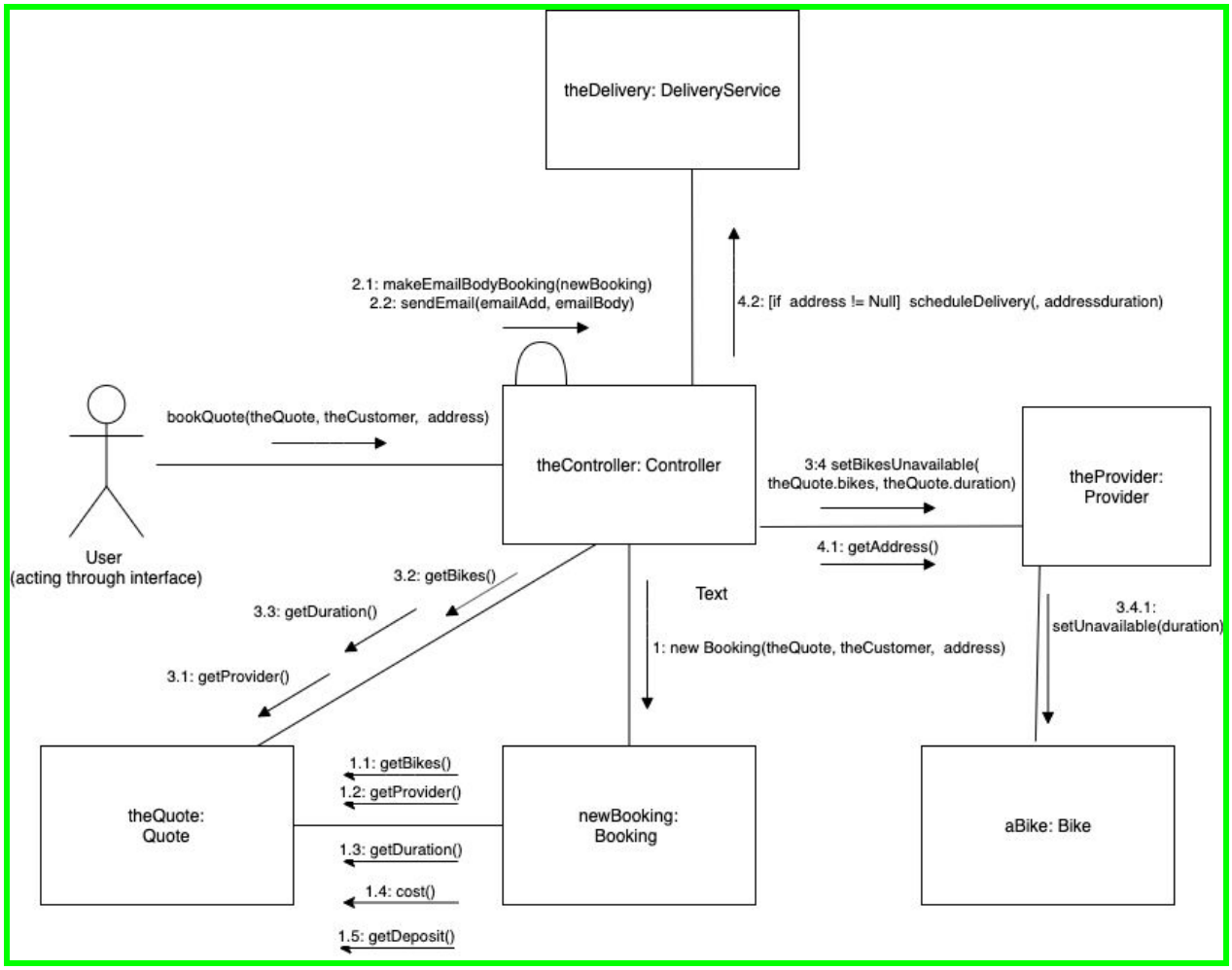
UML Sequence Diagram - Get Quotes



UML Communication Diagram - Register Bike Return



UML Communication Diagram - Book Quotes



Justification of good software engineering practices

The controller class encapsulated the model, and gives a specific interface that our UI is allowed to use when acting on on it. This will prevent misuse of the model. For example, the UI should be unable to directly set statuses of bikes as available - it would have to the use the registerReturn method which would trigger the correct procedures to ensure the system remains valid.

We have also added encapsulation by using getters and setters in all of our classes and setting the attributes as private allowing us to validation.

We have followed the Law of Demeter in our design. For example, the getQuotes method in the controller class asks each individual provider to find fitting bikes themselves, instead of reaching through the provider class to the bikes.

Further notes

After changing our initial requirements document in accordance with feedback, we found that we did not have to make any further changes when creating this design document.

We wrote our design document from the start to include the extensions, and so our class diagram did not require any changes

Self Assessment

Q 2.2.1 Static Model: 21%

Make correct use of UML class diagram notation: 4%

- We feel that we correctly used UML notation for the classes and associations.

Split the system design into appropriate classes: 5%

- Uur central controller system is efficient, and all real life elements are individually modelled in their own classes

Include necessary attributes and methods for use cases: 5%

- We have included all attributes and methods necessary for carrying out our use cases

Represent associations between classes: 4%

- We represented associations between classes and made a good attempt as modelling dependencies

Follow good software engineering practices: 3%

- We tried to follow software engineering practices to the best of our understanding
- We enforced encapsulation across the class diagram, by requiring the user to go through the controller class, and ensured that coupling was minimal by having a single class for each “real life” object/concept (for example, each bike is modelled by an object)
- We may, however, have violated the law of demeter in certain cases

Q 2.2.2 High-level description: 14%

Describe/clarify key components of design: 9%

- We gave a brief description of each class and described any methods/attributes that we considered not immediately obvious

Discuss design choices/resolution of ambiguities: 5%

- We brought up the places where our decisions regarding ambiguities in the requirements report impacted our design decisions here, and pointed the reader to the specific ambiguity involved.

Q 2.3.1 UML sequence diagram: 18%

Correct use of UML sequence diagram notation: 4%

- UML sequence syntax used mostly correctly

Cover class interactions involved in use case: 9%

- All class interactions covered with method calls and details about return values

Represent optional, alternative, and iterative behaviour where appropriate: 5%

- These behaviours were covered correctly with loop and alt frames
- These covered the three cases: found quotes on correct day, found quotes three days on either side, and no quotes found

Q 2.3.2 UML communication diagram: 13%

Communication diagram for record bike return to original provider use case: 7%

- Communication diagram was included with correct method calls

Communication diagram for book quote use case: 6%

- Communication diagram was included with method calls

Q 2.4 Conformance to requirements: 5%

Ensure conformance to requirements and discuss issues: 5%

- Our design conforms to the requirements set out in our first requirements report, and we had no issues to discuss

Q 2.5.3 Design extensions: 10%

Specify interfaces for pricing policy and deposit/valuation policies: 3%

- We fully specified the interfaces

Integrate interfaces into class diagram: 7%

- We integrated our interfaces

Q 2.6 Self-assessment:

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

- We attempted a self assessment and linked it to the assessment criteria

Justification of good software engineering practice: 3%

- We discussed where we felt our design followed software engineering practice, and gave examples