Inf2C Software Engineering 2019-20

Coursework 2

Creating a software design for a federated bike rental system

1 Introduction

The aim of this coursework is to create and document a design for the software part of a federated bike rental system. To create the design you are encouraged to experiment with the class identification technique discussed in lectures and with the approach of using CRC Cards you were asked to read up about.

This coursework builds on Coursework 1 on requirements capture. Please refer back to the Coursework 1 instructions for a system description. The follow-on Coursework 3 will deal with implementation and testing.

2 Design document structure

Ultimately, what you need to produce for this coursework is a design document that combines descriptive text with UML class, sequence and communication diagrams. The following subsections specify what should be included in this document. The percentages after each subsection title show the weights of the subsections in the coursework marking scheme.

There is no requirement for you to use a particular tool to draw your UML diagrams. The *draw.io* tool[1] is one easy-to-use tool you might try. If you draw your diagrams by hand, be sure to include a high-quality scan in your report.

2.1 Introduction

Start your document with a few sentence description of the whole system and then refer the reader to these instructions and the previous instructions for further general

[1] `http://draw.io`

information.

2.2 Static model (UML class diagram and high-level description)

This section must contain a complete UML class model and a high-level description of the model.

2.2.1 UML class model (25%)

Construct a UML class model for the system. Include operations only when you consider them important for the execution of one of the following use cases:

1. **Get quotes**
2. **Book quote**
3. **Register bike return to original provider**
4. **Register bike return to partner of original provider**

Do not include simple operations such as attribute getters and setters. The operation descriptions must include types of any parameters and the type of the return value. You may also ignore attributes which are not directly involved in the above use cases such as customer details (apart from their location) or the phone number of a bike provider. Associations must include multiplicities at each end. Some objects may use maps to look up information of some type `T` according to a key of some type `K` — you should depict these as attributes of type `Map<K, T>` instead of representing these as associations. Some methods may take or return collections of objects of a given class `T`; represent these as of type `Collection<T>` rather than specifying a concrete collection type.

Leave navigation arrowheads off associations: at this abstract level of design, this information is not needed.

Some classes will model utility concepts such as a location or the date range of a booking. Represent such classes, but to keep things simple you do not need to represent their associations to other classes in the class diagram.

You might find it useful to express the class model using more than one UML class diagram. For example, one diagram might show just class names and the associations and other dependencies such as generalisation dependencies between classes. Other diagrams might omit the associations, but show for each class its attributes and operations.

You may find the following reference for drawing class diagrams in `draw.io` useful[2].

[2] `https://about.draw.io/uml-class-diagrams-in-draw-io/`

2.2.2 High-level description (15%)

The high-level description should expand upon the details of the design you chose. Focus on clarifying aspects which may be unclear from your use case diagram and explaining where you have made design decisions. What alternatives did you consider and why did you make the choices you made?

You discussed assumptions concerning ambiguities and missing information in the system description during the first coursework. Be sure to indicate how these assumptions have impacted your design.

A viewer of your class diagram should quickly form an impression of the structure of your design. But obviously the diagram leaves out many details. If you have attributes or operations of classes whose purpose is not clear from their names and associated types alone, add a few explanatory notes.

2.3 Dynamic models

In this section you should produce UML sequence diagrams and communication diagrams to illustrate how your design implements the key use cases of the system. In doing so, please consider the feedback from the markers on the interactions from your use cases from coursework 1. Also, if during your work for this section you change your mind about any interactions in your use cases, please update your use cases for coursework 1 highlighting your changes in green for its future resubmission (see Q 2.4).

2.3.1 UML sequence diagram (20%)

Construct a UML sequence diagram for the *Get Quotes* use case. Do show message names, but there is no need to include some representation of any message arguments. As needed, use the UML syntax for showing optional, alternative and iterative behaviour.

2.3.2 UML communication diagram (15%)

While sequence diagrams focus more on the sequential exchange of messages between objects, the alternative notation of UML *communication diagrams* (also sometimes called *collaboration diagrams*) can be used to focus on the organisation of the objects and their interaction. Communication diagrams were introduced in the lecture on UML interaction diagrams and its associated resources. You may find this link useful for constructing communication diagrams in *draw.io* [3].

Produce UML communication diagrams for the *return bikes to original provider* and *book quote* use cases.

[3] `https://about.draw.io/uml-communication-diagrams/`

2.4 Conformance to requirements (5%)

Your static and dynamic models should both be consistent with the system requirements, as captured in coursework 1. If you discover any issues with your requirements documents you should update it before resubmitting coursework 1. Please highlight any changes you make to coursework 1 in green. You should also comment here on any apparent divergences with the requirements or issues you encountered with your requirements documents (particularly those which required changes).

2.5 Design extensions

Upon receiving your initial design and consulting with the bike providers who will participate in the system, whilst largely satisfied, your client has expressed concern that the design does not make it possible to implement the newly planned *pricing* and *deposit* policies of existing bike providers. The client has decided to hire external contractors to implement both custom pricing policies and custom deposit calculation policies as separate submodules. As such, you need to modify your design to allow extensions to the price calculation formula and the deposit calculation formula.

As part of the final coursework, you will play the role of one of these contractors and implement a custom pricing or deposit policy.

2.5.1 Custom pricing

Whilst your design should already allow providers to set their own daily prices for each bike type, in reality providers may have much more complex pricing policies, which offer customers discounts based on the duration of hire or the number of bikes being hired. As such the system should be extensible, providing an interface describing a *pricing policy* for a provider which calculates a price for a given collection of bikes and date range of hire. Custom pricing policies can then be implemented by the contractors by providing classes which implement this interface.

2.5.2 Custom deposits/valuation

Your existing design should allow providers to calculate deposit amounts by multiplying the bike type's *replacement value* by their custom *deposit rate*. However, in practice, bike providers have decided to use more complicated formulae for calculating the value of a bike based on its original value and its age to take into account the depreciation of the bike's value over time — this valuation is used in place of the bike type's replacement value when calculating the deposit. As such the system should be extensible, providing an interface for *deposit or valuation policies* which can change how the value for a given bike is calculated when determining the deposit. Custom deposit policies can then be implemented by the contractors by providing classes which implement this interface.

2.5.3 Modified design (10%)

You should modify your UML class diagram to add interfaces for pricing policies and deposit/valuation policies and to make use of them in calculating the price and deposit of quotes. You do not need to redraw the whole diagram — you can just provide the parts to it which will need to change to incorporate custom pricing and deposit/valuation policies.

2.6 Self-assessment (10%)

For this section you should consider to what extent your have met the assessment criteria for this assignment. The main criteria for each question are listed in Fig. 1. For each criteria you should indicate the mark which you believe your answer merits and justify your judgement.

You should format your answer as a two-level nested list (mirroring that below), providing your predicted mark, and a brief justification of your answer.

You should pay particular attention to *justifying how your system design follows good software engineering practice*, and dedicate at least one paragraph elaborating this aspect of your design.

3 Further information

3.1 Modelling using message bursts

Create a system design with a single thread of control. Do *not* try to make it concurrent.

System activity consists of bursts of messages passed between objects, each triggered initially by some actor instance sending a message to some system object. Each burst corresponds to the execution of some fragment of a scenario of a use case. Assume that the only messages sent from system objects back to actor instances are the final reply messages of bursts. Such a message corresponds to the return of the method invoked by the trigger message sent by the actor. Imagine each burst completes relatively quickly, in well under a second.

Because of the single-threaded nature, the system does not handle further input messages during a burst. This should not be too much of a restriction, because of the assumed short duration of each burst.

3.2 Abstract inputs

Consider input messages at an abstract level. Do not model the user interface (online or in store) and other possible sources of the input messages. Here are some examples:

1. To search for quotes, a *Customer* actor might send in a *find quotes* message, perhaps identifying their rental requirements. You need to think of some way to represent their rental requirements (including their location, date range, and the

Q 2.2.1 Static model **25%**

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

Q 2.2.2 High-level description **15%**

- Describe/clarify key components of design 10%
- Discuss design choices/resolution of ambiguities 5%

Q 2.3.1 UML sequence diagram **20%**

- Correctly use of UML sequence diagram notation 5%
- Cover class interactions involved in use case 10%
- Represent optional, alternative, and iterative behaviour where appropriate 5%

Q 2.3.2 UML communication diagram **15%**

- Communication diagram for *return bikes to original provider* use case 8%
- Communication diagram for *book quote* use case 7%

Q 2.4 Conformance to requirements **5%**

- Ensure conformance to requirements and discuss issues 5%

Q 2.5.3 Design extensions **10%**

- Specify interfaces for pricing policies and deposit/valuation policies 3%
- Integrate interfaces into class diagram 7%

Q 2.6 Self-assessment **10%**

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

Figure 1: High-level marking criteria.

number of bikes they require of each bike type) using appropriate classes of objects — these will form the arguments of this method. For the required numbers of bikes of each type perhaps you could use an appropriate collection type[4] [5]? The system should then return a collection of objects representing matching quotes.

2. To create a booking, a *Customer* actor instance might send in a *create booking* message, perhaps identifying a quote previously returned by the system. The system should then return an object representing an invoice.

3. To record a bike return, a *Bike Provider* might send a *record return* message specifying the order number of the booking to be returned.

3.3 Abstract data

When handling monetary values or percentages your design should use an appropriate numerical type.

Your design should create a Location class which is used to specify the details of a location (which consists of a street address and a postcode). This will need to include a method for testing whether two locations are near to each other — do not worry about how this should be implemented yet.

Your design will also need to handle dates and ranges of dates for creating bookings. Dates should be represented using Java's `LocalDate` class[6]. You may find it useful to introduce a class to represent ranges of dates.

3.4 Abstract outputs

Do not model the complexities of a the user interface (online or in store) that might be displaying results. You should focus on designing the software system.

For example, Quotes can be returned as objects with appropriate attributes containing their details. Similarly, when booking a quote, an object should be returned representing an invoice, which should contain the details of the quote and an order number identifying the booking.

3.5 Delivery Service

A key part of the bike rental system is represented by the delivery drivers who deliver bikes to customers' locations (if this has been requested) and by partners who return bikes to the original provider.

You do not need to be concerned with the details of how this works. Instead assume this is modelled by a single class, `DeliveryService`, with a method `scheduleDelivery` which schedules a delivery for a booking from a start location to an end location on a

[4] `https://www.javatpoint.com/collections-in-java`
[5] `https://www.javatpoint.com/java-map`
[6] `https://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html`

given date. Rather than considering the various steps of delivery, you will simply call this class to schedule a delivery at some point in the future. The delivery service will have to update the status of bookings (and the bikes therein) when they are collected for transit and when they arrive — you should provide suitable methods on your other classes to allow their status to be updated. You should include a class for the delivery service in your design which you may need to include in your dynamic models — such a class will be provided for you during the final coursework.

4 Asking questions

Please ask questions on the course discussion forum if you are unclear about any aspect of the system description or about what exactly you need to do. For this coursework tag your questions using the `cw2` folder. As questions and answers build up on the forum, remember to check over the existing questions first: maybe your question has already been answered!

5 Submission

Please submit a PDF (not a Word or Open Office document) of your design document. The document should be named **design.pdf** and should include **a title page with names and UUNs of the team members who have worked on this coursework**.

As a group, please only submit the coursework solutions ONCE.

How to Submit

Ensure you are logged into MyEd. Access the Learn page for the Inf2C-SE course and go to "Coursework and labs" - "Coursework submission" - "Coursework 2".

Submission is a two-step process: upload the file, and then submit. This will submit the assignment and receipt will appear at the top of the screen meaning the submission has been successful. The unique id number which acts as proof of the submission will also be emailed to you. **Please check your email to ensure you have received confirmation of your submission**.

If you do have a problem submitting your assignment try these troubleshooting steps:

- If it will not upload, try logging out of Learn / MyEd completely and closing your browser. If possible try using a different browser.
- If you do not receive the expected confirmation of submission, try submitting again.
- If you cannot resubmit, contact the course organiser at Cristina.Alexandru@ed.ac.uk attaching your assignment, and if possible a screenshot of any error message which you may have.

- If you have a technical problem, contact the IS helpline (is.helpline@ed.ac.uk). Note the course name, type of computer, browser and connection you are using, and where possible take a screenshot of any error message you have.
- Always allow yourself time to contact helpline / your tutors if you have a problem submitting your assignment.

6 Deadlines

The two deadlines for this coursework are as follows.

- **Deadline 1 (formative): 16:00, Tuesday 5th Nov**

  A bonus of 2.5% of the final mark will be awarded to students who submit a **full attempt** at this coursework at the above deadline. You will get formative feedback and provisional marks back.

- **Deadline 2: 16:00, Friday 29th Nov (compulsory, summative, marks which count!)**

  For this final deadline, please consider the feedback received from the markers for the first deadline and highlight your changes to your solutions in green. Also, adjust your self-assessment justifications to explain how you have addressed marker comments, also as highlighted in green.

The coursework is worth 30% of the total coursework mark and 12% of the overall course mark.

Thomas Wright, 14th October 2019.