Inf2C - Software Engineering 2019-20

Coursework 3

- ❑ Daniel Wilks (UUN: s1851664)
- ❑ Eshaan Manglik (UUN: s1892592)

Q 4.6 Update design and requirements documents

We changed our class diagram by adding a MainSystem class in which most of the functionality occurs like creating a customer and invoking the quoteController class in which the quotes are generated on the bases of the rental needs of the customer. It helps us increase cohesion in each of our classes. We also changed some functionality in the classes like Bike Rental shop and the bikes in order to reduce the coupling.

We changed our class sequence diagram to represent our updated code. Before we were iterating over all bike types requested by the customer and were creating quotes if any of the bikes requested were available in each bike rental shop. The output would then include quotes which did not include all of the specified bikes required by the customer. In our updated version, we made sure that only quotes that had all of the requested bikes without any overlaps with the other reservations were created and returned to the customer.

Before, we included functionality to allow interaction with banks for the payment. We had functions such as validatePayment in booking. We removed these, as we did not need to include such functionality. Furthermore, the removal of this allowed us to increase cohesion for our booking class.

Q 4.7 Self-assessment

1. Extension submodules 10%

• Implementation of extension submodule 10%

◊ Should implement extension submodule

◊ Should include unit tests for extension submodule

8%, Our implementation of the submodule extension adhered to the specification. Also, we included good unit tests which covered most cases although there could be some edge cases we have missed.

• Peer review of other group's submodule (**up to 10% bonus marks**)

8%, We gave an honest reflection of their work, highlighting areas that could be improved, e.g. there were some instances strong coupling. We also highlighted more areas of improvement leading to a detailed report. Although, there could be some software engineering practises that we are unaware of and did not comment about in the report.

2. Tests 35%

• System tests covering key use cases 20%

◊ Should have comments documenting how tests check the use cases are correctly implemented
◊ Should cover all key use cases and check they are carrying out the necessary steps
◊ Should have some variety of test data
◊ Should use MockDeliveryService

17%, Our tests cover all the appropriate use cases and all the necessary steps using junit assert statements. We also use a variety of test data to cover lots of scenarios and edge cases for our testing. We feel we did this adequately and have covered most cases that could occur although we understand there could be some edge cases that we have not tested. We also have appropriate comments which document how the tests check each use case. This allows for clarity and makes the code easy to understand. Furthermore, we used MockDeliveryService in our code.

• Unit tests for Location and DateRange

5%, We created adequate testing for Location and DateRange, making sure to account for edge cases. Our tests included a variety of data that allowed us to be confident that our implementation of Location and DateRange are correct and work properly.

• Systems test including implemented extension to pricing/valuation 5%

• Mock and test pricing/valuation behaviour given other extension **(challenging)** %5

8%, We implemented tests for the extension for pricing and valuation. We thought about edge cases possible for the code and implemented the necessary tests to make sure our code was running for said edge cases. This allowed us to ensure our code was correct. However, there could be more edge cases we have not thought about and did not test for.

3. Code 45%

• Integration with pricing and valuation policies 10%

◊ System should correctly interface with pricing and valuation policies

◊ System should correctly implement default pricing/valuation behaviour

9%, We implemented the pricing and valuation policies correctly according to the specification. We did this with the least amount of coupling as we saw feasible and tried to make the cohesion as strong as possible. We believe we did a good job in this regard.

• Functionality and correctness 25%

◊ Code should attempt to implement the full functionality of each use case

◊ Implementation should be correct, **as evidenced by system tests**

21%, We fully implemented the functionally for each use case according to our system design and have also provided system tests to ensure that the code is correct and implements the functionality as desired. Although, there could be edge cases in our system tests that we have not covered in our testing.

• Quality of design and implementation 5%

◊ Your implementation should follow a good design and be of high quality

◊ Should include some assertions where appropriate

3%, Our implementation follows good design and is of high quality. This is evident as we have tried to reduce coupling and increase cohesion within our classes and system. Although, there could be some areas in which there is coupling and this could be improved by altering the classes and how they interact.

• Readability 5%

◊ Code should be readable and follow coding standards

◊ Should supply javadoc comments for Location and DateRange classes

5%, We follow the same coding practise, e.g. indentation, '{' placement etc. through-out the codebase to ensure readability. We adhere to the coding guidelines from Google as specified in the coursework spec. Our javadoc style comments were adequate and followed the specification.

4. Report

• Revisions to design 5%

◊ Design document class diagram matches implemented system

    - ◊ Discuss revisions made to design during implementation stage

5%, We have revised the design document class to match the implemented system. We have made these revisions in green. We have discussed the revisions in detail and have given reasons for our decisions. We believe the decisions we made were helpful in allowing our code to be more coherent and to reduce the coupling and increase the cohesion.

- Self–assessment 5%

    - ◊ Attempt a reflective self-assessment linked to the assessment criteria

5%, We have attempted a full reflective self-assessment in detail. We reflected on our work using the specification and how closely our implementation matches the spec.