INF2C - SE Coursework 3

Michal Glinski s1813305
Siang Jun Teo s1800918

Michal Glinski s1813305, Siang Jun Teo s1800918

Design Changes:

Bike can have multiple booking at the same time for different DateRange, instead of previously only being able to have one booking.
Bike is not holding bookings for themselves, they are instead holding the date ranges for which they are taken, this reduces coupling.
Bike now has BikeType which holds the replacement value of the bike, it is used to calculate the value of the bike with ValuationPolicyInterface.

ServerDataInterface is now the thing that holds all the necessary data such as bookings and providers, to aid in testing and allow for flexible implementation (i.e. Usage of databases or custom save files)

getQuotes
- isTaken now takes a parameter of query
- matchesQuery now takes query and bike parameters

bookQuote
- This now also calls customer.addBooking on success

Provider now is the only class that is the owner of the bikes, where previously the ServerData was also in possession of all bikes.

Make booking deliverable, as it will be easier to determine what booking the delivery it talking about and every booking has exactly one bike

There are no default behaviours for when policies are not provided, it is assumed they are always provided, but they may be the default implementation of the policy that simulates the default behaviour. It removed unnecessary checks for edge cases when policies are not provided and in fact makes code easier to read.

Michal Glinski s1813305, Siang Jun Teo s1800918

Self Assessment:

1. Extension submodules 10%

- Implementation of extension submodule 10%
    - Should implement extension submodule
    - Should include unit tests for extension submodule
- Peer review of other group's submodule (up to 10% bonus marks)

2. Tests 35%

- System tests covering key use cases 20%
  18/20
    - Should have comments documenting how tests check the use cases are correctly implemented
      Comments for the most part felt unnecessary as the test names were pretty self explanatory. For testIntegration() however, there were comments explaining how we expect the system to run step by step from beginning to end. Small comment on how setUp() was set up.
    - Should cover all key use cases and check they are carrying out the necessary steps
      These were carried out.
    - Should have some variety of test data
      There was a good variety.
    - Should use MockDeliveryService
      MockDeliveryService was used.
- Unit tests for Location and DateRange 5%
  5/5
  Unit tests were manually made, and the code written for Location and DateRange passes them.
- Systems test including implemented extension to pricing/valuation 5%
  5/5
  There were tests for valuation policy.
- Mock and test pricing/valuation behaviour given other extension (challenging) 5%
  5/5
  We used mock policies for the systems test.

3. Code 45%

- Integration with pricing and valuation policies
  10/10
  - System should correctly interface with pricing and valuation policies
    System uses only ValuationPolicy and PricingPolicy when trying to get values represented by them, when something should not use one, it is instead using the default one
  - System should correctly implement default pricing/valuation behaviour
    Both provider and bikes can be constructed without the policies, and then when creating they will be initiated with default behaviour of policies
- Functionality and correctness 25%
  23/25
  - Code should attempt to implement the full functionality of each use case
    It implements the functionality off 3 use cases that were to be implemented, while keeping in mind even small details and possible exceptions.
  - Implementation should be correct, as evidenced by system tests
    Tests have high coverage of the code and were designed by hand so possible coding errors are avoided. Additionally the codebase passes all of them
- Quality of design and implementation
  4/5
  - Your implementation should follow a good design and be of high quality
    The implementation is still high level, which was achieved in parallel with encapsulation. The code avoids what could be considered bad coding practices
  - Should include some assertions where appropriate
    Assertions were used to check for things that were designed as invariants, so when the system is running, they are not slowing the execution
- Readability 5%
  5/5
  - Code should be readable and follow coding standards
    Code is highly readable, and execution is self explanatory when being read. When there could be exceptions there are comments in place to help with reading
  - Should supply javadoc comments for Location and DateRange classes
    JavaDoc heavily documents pretty much all of the codebase without the exception of Location and DateRange. When code was changed the JavaDocs were updated with the code

4. Report 10%

- Revisions to design 5%
  5/5
  - Design document class diagram matches implemented system
    We've updated the diagram to match the system.
  - Discuss revisions made to design during implementation stage
    Revisions were discussed and documented.
- Self–assessment 5%
  5/5
  - Attempt a reflective self-assessment linked to the assessment criteria
    There was much effort to self assess our work to ensure it meets the criteria.