Inf2C Software Engineering 2019-20

Coursework 3

Creating an abstract implementation of a federalised bike rental system

1 Introduction

The aim of this coursework is to implement and test an abstract version of software for the federalised bike rental system, and to play the role of contractors implementing an extension to this system. This coursework builds on Coursework 1 on requirements capture and Coursework 2 on design. As needed, refer back to the Coursework 1 and Coursework 2 instructions.

In order to carry out this coursework you will need to know how to create and run tests using JUnit and how to use *interfaces* in Java. If you are not comfortable with either of these topics you should review the materials from the Inf1-OOP course before starting or look at some resources online[1] [2] [3].

2 Extension submodules (10%) and their peer review (up to 10% bonus marks)

In coursework 2, extensions to the price and deposit calculation part of the system were described, in the form of independent submodules implementing a predefined interface to add functionality to the system. In the first part of this coursework you will play the role of one of two external contractors whose role is to implement one of these two submodules: **attempt submodule 1 if your group number is odd or submodule 2 if your group number is even**. You should also develop unit tests for your submodule. Whilst in Coursework 2 you designed your own interfaces for these extensions, for this coursework you should use

[1] `https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-our-first-test-class/`

[2] `https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-writing-assertions-with-junit-5-api/`

[3] `https://www.w3schools.com/java/java_interface.asp`

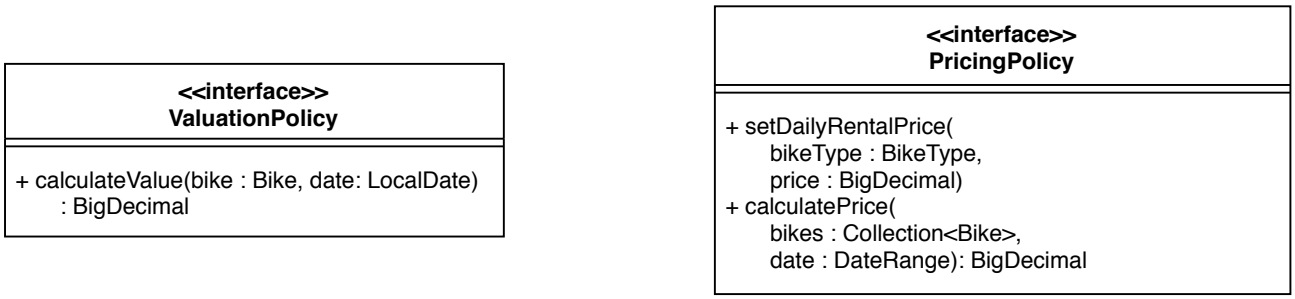

Figure 1: Interfaces for valuation and pricing policies

the interfaces provided in Fig. 1 so that, in theory, your extension submodules would be compatible with every group's system. Since these submodules are self-contained units, they can be developed independently from the rest of your system, so you do not have to finish implementing the system before attempting this task. To reduce coupling with the rest of your system, your submodule should only interact with the *mandatory classes* listed in Section 3.1 but may call any methods of these classes.

In the weeks' 10-11 labs you will be offered the opportunity to share your implementation of your submodule with one other group which is working on the other submodule and review each others' code and unit tests, for up to 10% bonus marks each (to be considered as long as your total for the 3 courseworks does not exceed 40% of the course mark). The bonus marks will be awarded based on the quality of your review (see Section 4.1). Only bring the source of your extension submodule and its tests for peer review to the lab, and not any other classes of your system. If your extension submodules subclass another of your classes (for example, your default pricing/valuation policy), only bring the extension submodules themselves for peer review, not the superclass or any tests concerning its behaviour.

When developing the rest of your system, assume the other submodule has not yet been implemented. However, you still need to implement the default pricing/deposit functionally regardless of which extension you implement (so prices are computed without any discounts and deposits are set to the deposit rate times the replacement value of the bike type without any depreciation). This means that for the task in crefsubsec:integration-and-mocking you must test your whole system, including parts whose full functionality will depend on the extension submodules.

2.1 Extension submodule 1: Multiday pricing discounts

For the existing design of the system, assume pricing is determined by adding up the prices of all the bikes in the order. However, many bike providers want to offer discounts to customers depending on the length of their booking. We want to do this by allowing them to define a *multiday discount policy* which offers a percentage discount which is determined by the duration of the booking in days (applied to the original price calculated based on the bikes in the order). The multiday discount policy should consist of a number of tiers of booking lengths (in days) for each of which a different percentage discount may be specified.

A sample multiday discount policy is shown in Fig. 2. This policy specifies that any bookings of 2 or fewer days should receive no discount, any bookings lasting between 3-6

Days	Discount
1-2	0%
3-6	5%
7-13	10%
14+	15%

Figure 2: A sample multiday discount policy.

days should receive a 5% discount, any bookings lasting from 7-13 days should receive a 10% discount, and any bookings lasting for at least 14 days should receive a 15% discount.

Implement a multiday rate discount policy class implementing the PricingPolicy interface (Fig. 1, Right). This should allow bike providers to define and update policies in the way shown in Fig. 2, calculate the discount rate for a given order, and take this into account when calculating the price of a given order. . All the code you introduce to implement this extension should be in new classes (comprising your extension submodule), although you may make use of any methods of the mandatory classes provided in the code skeleton.

2.2 Extension submodule 2: Custom bike deposit/valuation policies

In your existing design, the deposit amount for a booking should be calculated as a fixed percentage (the *deposit rate*) of the value of the bike, and the value of the bike should be set globally by the bike type's *replacement value*. However, due to accounting requirements, the bike providers need to calculate the value of a bike in a more sophisticated way, using a formula to calculate the *depreciation* of a bike's value over time, based on two factors: the *depreciation rate* (a percentage) and the *age of the bike* in whole years.

Write classes implementing the ValuationPolicy interface (Fig. 1, Left) to take into account different forms of depreciation when calculating the value of bikes. You do not need to implement support for arbitrary depreciation formulae, but should at least implement the following two popular methods for depreciation calculation, so that bike providers may choose between them based on their accounting requirements. All the code you introduce to implement this extension should be in new classes (comprising your extension submodule), although you may make use of any methods of the mandatory classes provided in the code skeleton.

Linear Depreciation Under linear depreciation the value of a bike decreases by the *depreciation rate* — a fixed percentage of its original value (the *replacement value*) — every year.

Double Declining Balance Depreciation Under double declining balance depreciation [4], the value of a bike starts at its *replacement value* in the first year of its life, and at

[4]More information about this depreciation method is available at `https://www.wallstreetmojo.com/double-declining-balance-method/`.

the end of each year the bike loses a percentage of its value in the previous year at a rate equal to *twice the depreciation rate*.

This means that if a bike was originally worth a replacement value of £900 and the depreciation rate is 10%, then after three whole years have passed, under linear depreciation its value would be $900 - 3 \times 0.1 \times 900 = £630$, whereas under double declining balance its value would be $900 \times (1 - 2 \times 0.1)^3 = £460.8$. Then, under a deposit rate of 20% the deposit amounts would be £126 and £92.16 respectively.

3 Further design details

You may follow your own design for the system as long as you address the requirements from Coursework 1. However, this section provides some extra directions which further restrict how you should implement the system to help you in this implementation and to make sure you can attempt all the sections of this coursework.

3.1 Mandatory classes

Your design should contain at least the following classes: Location, DateRange, Bike, and BikeType. Skeletons of these classes have been provided, however, you will need to supply implementations of their methods and you will need to introduce new attributes and methods. You will also need to introduce many of your own classes in order to realise your design.

- The Location class represents a spatial location described by a post code and a street address. This class must have a method isNearTo(Location other) : boolean which checks whether the location is near enough to another location, other, to allow for collection/delivery of bikes. Whilst you could imagine a sophisticated implementation of this method using an online mapping service, this is beyond the scope of this coursework. Instead you should simply compare locations by checking if they are in the same postal area. That is, you may assume two locations are 'near to' each other when the first two digits of their post codes are equal.

- The DateRange class represents a duration spanning (inclusively) a start date and an end date, both represented by Java LocalDate objects. This class should include getters getStart and getEnd respectively as well as an overlaps(DateRange) method, which checks if this date range overlaps with another. It may also contain any other methods you think are useful.

- Your should represent each bike in your system as an object of class Bike. Each bike should have a type, represented by a BikeType object.

3.2 Monetary values and percentages

You system should store any monetary values or percentages as Java BigDecimal objects rather than primitive doubles in order to ensure accurate and predictable monetary calcula-

tions. In your unit tests you can check if two BigDecimal objects a and b are equal by using the assertion[5]

```
assertEquals(a.stripTrailingZeros(), b.stripTrailingZeros());
```

3.3 DeliveryService and Deliverables

Rather than considering the exact details of how the system interacts with delivery drivers, you can assume you have been provided with a DeliveryService object which implements the interface shown in Fig. 3a. This includes a scheduleDelivery method which can schedule a delivery of an object implementing the Deliverable interface. An appropriate class in your system should implement the Deliverable interface (Fig. 3b) and its onPickup and onDropoff methods to update the status of bookings (and the bikes they contain) throughout the delivery process. An additional DeliveryServiceFactory class is provided to give your code access to a singleton delivery service object. Thus, your code can always get access to the DeliveryService via the static method call,

```
DeliveryServiceFactory.getDeliveryService();
```

You have also been provided with a mock implementation of the DeliveryService in the MockDeliveryService class (Fig. 3c). In your unit tests you should instruct the system to use this by calling

```
DeliveryServiceFactory.useMockDeliveryService();
```

before each test. You can use suitable methods on the MockDeliveryService class to simulate each step of the delivery and you should use this in your system tests to check your system is able perform updates during the delivery process.

[5]The reason why this is necessary is explained at `https://stackoverflow.com/questions/6787142/bigdecimal-equals-versus-compareto`.

```java
interface DeliveryService {
    public void scheduleDelivery(
        Deliverable deliverable,  // Deliverable to be delivered
        Location pickupLocation,  // Location to pickup deliverable
        Location dropoffLocation, // Location to dropoff deliverable
        LocalDate pickupDate      // Date for pickup to occur
    );
}
```

(a) DeliveryService interface

```java
public interface Deliverable {
    // Called by delivery service when deliverable picked up
    public void onPickup();

    // Called by delivery service when deliverable dropped off
    public void onDropoff();
}
```

(b) Deliverable interface

```java
public class MockDeliveryService implements DeliveryService {
    // carry out all pickups on the specified date
    // and schedule dropoffs
    public void carryOutPickups(LocalDate date);

    // Carry out all dropoffs
    public void carryOutDropoffs();
}
```

(c) MockDeliveryService methods

Figure 3: Provided interfaces and mock for deliveries.

4 Your tasks

There are several concurrent tasks you need to engage in. These are described in the following subsections.

Bear in mind that it is not enough to submit code claiming to implement your system without testing and other documentation demonstrating its effectiveness. As part of this assignment you are asked to implement system tests demonstrating how your system implements the key use cases of the system, and **these tests will be used as the primary means of marking this coursework**.

4.1 Extension submodules (10%) and their peer review (up to 10% bonus marks)

You need to prepare a submodule for peer review following the instructions in Section 2. You should do this before or in parallel with the implementation of the main system, and

you will exchange these submodules with a group working on the other submodule for a peer review during one of the labs in weeks 10-11.

In your peer review be sure to give specific feedback on how they could improve their submodule and tests. This may involve both the correctness of the code and its general quality (maintainability and readability). If you don't understand any aspect of their code you should ask for clarification and advise on how this could be made clearer to other readers of the code.

Include your extension submodule as part of your system after incorporating the feedback from the peer review. If you do not participate in the peer review, you still need to implement the extension submodule but you will not benefit from feedback from peers or the bonus marks.

4.2 Construct code (45%)

You should base your system on your design from coursework 2, although you may decide to deviate from your design, in which case you should update your design in coursework 2 in green to reflect the changes you make. Your implementation only needs to cover the three use cases listed in Section 4.3 as well as one of the two extensions listed in Section 2.

Follow good coding practices as described in the lecture. You should attempt to follow the coding guidelines from Google at

`https://google.github.io/styleguide/javaguide.html`

A common recommendation is that you never use tab characters for indentation and you restrict line lengths to 80 or 100 characters. You are strongly recommended to adopt both of these recommendations because it is good practice and, particularly, in order to ensure maximum legibility of your code to the markers. You should use an indent step of 4 spaces as in the Sun guidelines rather than the 2 spaces from the Google guidelines, for consistency with the supplied code.

Unfortunately the default in Eclipse is to use tab characters for indents. See the Coursework 3 instructions on Learn for advice on how to set Eclipse to use spaces instead and also how to get Eclipse to display a right margin indicator line.

To show you know the basics of Javadoc, add class-level, method-level and field-level Javadoc comments to the Location and DateRange classes. For your methods you only need to include @param and @return tags as well as a summary of the role of the method. Browse the guidance at

`https://www.oracle.com/technetwork/java/javase/tech/index-137868.html`

on how to write Javadoc domments. Follow the guidance on including a summary sentence at the start of each comment separated by a blank line from the rest of the comment. This summary is used alone in parts of the documentation generated by Javadoc tools.

Be sure you are familiar with common interfaces in the Java collections framework such as List, Set, Map, and Queue/Deque. and common implementations of these such as ArrayList, LinkedList, HashSet, TreeMap. Effective use of appropriate collection classes will help keep your implementation compact, straightforward and easy to maintain.

You are allowed to assume that the UI has already performed some input validation before calling the methods of your implementation. You should include assertions in at least

some of your methods as a means of catching faults and invalid states. You only need one or two assertions to show you know how to use them, but you may want to include more to help you develop and test your system.

4.3 Create system-level tests (20%)

Your are expected to use JUnit 5 to create system-level tests that simulate the scenarios of each of the use cases you implement. The provided ControllerTest class tests some but not all of the features described in Section 3 to get you started.

To get a good mark for this coursework you must test at least the scenarios in this section since your tests will provide the main means for assessing your implementation. Even if you have implemented a feature, you may not get full marks if you have not tested your system sufficiently to demonstrate your system successfully implements the feature.

You need to provide tests to cover at the following scenarios:

1. **Finding a quote**

   A customer wants to find a quote for a booking on a given date range. The system should return a list of all matching quotes including the provider, bikes, and the price and deposit amounts for the quote. You should check that this excludes bikes which are already booked during this period and providers which are not near enough to satisfy the order. You do not need to include special behaviour in the case no quotes are available for the requested dates[6].

2. **Booking a quote**

   A customer wants to book one of the quotes the system has returned. This should place a booking with the provider listed matching the details of the quote requested and return an object representing the details of the booking, including a unique *booking number*. If the customer has requested delivery, the DeliveryService should be used to schedule it.

3. **Returning bikes**

   A customer has arrived at a bike provider to drop off their bikes and the bike provider employee, whom they have provided their *booking number* to, wants to handle their return. You should test both the case in which they are the original bike provider, and in which they are a partner, and for the second the DeliveryService should be used to return the bikes to the original provider.

You are expected to add enough systems tests to completely test the above use cases.

Few tests are able to test single use cases by themselves. In nearly all cases, several use cases are needed to set up some state, followed by one or more to observe the state. You may want to include tests which perform more than one use cases in turn to check the integration of the modules of the system.

[6] In previous courseworks you were asked to include dates three days ahead/behind in this case; **this is not required for this coursework**.

You will also need to populate the system with enough data (bikes, bookings, etc) to adequately test its behaviour and should consider cases where different data leads to significantly different branches of your code. Whilst the coursework has directed you to focus on certain use cases, in order to test your system you will need to make sure your system includes enough other methods (such as getters and setters) so that your tests can insert data into the system and inspect the results. You will also need to implement equals or hashCode methods on many of your classes so that your tests can test objects and collections of objects are equal where required[7]. In use cases involving the DeliveryService you can use methods on the MockDeliveryService class to simulate the steps of a delivery and you should check the status of bookings/bikes are updated correctly during the delivery process.

Do not run all your tests together in one large single test. Where possible, check distinct features in distinct tests. Also, when a test involves exercising some sequence of use cases and features, try to arrange that this test is some increment on some previous test. This way, if the test fails, but the previous test passes, you know immediately there is some problem with the increment.

You are encouraged to adopt a test first approach — the tests are just as important a part of the assignment as is your implementation itself. Make use of the provided tests to help you develop your code, and when tackling some feature not already tested for, write a test that exercises it before writing the code itself.

Include all your additional tests as JUnit test methods in the SystemTests.java class, or as unit tests corresponding to other classes.

Your SystemTests.java file should serve as the primary evidence and documentation for the correct functioning of your system. To this end, take care with how you structure your test methods and add appropriate comments, for example noting particular design features being checked.

4.4 Unit testing (5%)

As well as system tests you may find it useful to create unit tests for specific classes. These should focus on checking whether the implementation of each specific class is correct, rather than considering the correctness of other modules of the system. The tests for a class named MyClass should be in a file named `TestMyClass.java` and should use JUnit 5 assertions and annotations similarly to the examples provided.

For your coursework submission you are only required to provide such unit tests for the provided Location and DateRange classes in the provided files `TestLocation.java` and `TestDateRange.java` respectively (in addition to the system tests and extension submodule tests you are asked to write in other sections).

4.5 Integration and mocking of pricing and valuation (10%)

Your will need to develop your system so it will work with and without the extension submodules providing custom pricing and valuation behaviour. This means your system tests

[7] See `https://www.sitepoint.com/implement-javas-equals-method-correctly/`, and the DateRange class for an exampl e. If you are using Eclipse you can get it to write these methods for you: `https://www.baeldung.com/java-eclipse-equals-and-hashcode`

should test that your system is able to generate quotes including correctly calculated prices and deposits.

In particular, regardless of which submodule you develop, you will still need to implement the default pricing and valuation behaviour for the system. That is, the calculation of prices without applying any discounts and the calculation of deposits based directly on the *replacement value* of a bike type and the *deposit rate* of the provider.

You should have at least system test(s) which test pricing/valuation calculation using the extension submodule you have implemented.

If you want to test your system more thoroughly you could also create a mock object which mimics the potential behaviour of the extension submodule which you **did not** implement (**without implementing its full functionality!**) and write a few tests to check your code correctly interfaces with these modules (that is, the prices in your quote generation take into account the extension modules). **This is a more challenging task and you should not attempt it until you are comfortable with your solution to the rest of the coursework or if you feel it will lead you to spend an excessive amount of time.** The aim of these tests is not to test the behaviour of the extension module (since you are assuming that it has not yet been implemented), but rather, to check your code will call them in the correct way and correctly make use of their results as part of the pricing/deposit calculation process. You can look at the MockDeliveryService class as an example of how to create a mock object in Java.

4.6 Update design and requirements documents (5%)

When you submit this coursework you should also update your design and requirements documents (courseworks 1 and 2) in green with any changes to the design or clarifications to the requirements you made based on your experience implementing the system. The report submitted as part of this coursework should include a section summarising any changes to your design and requirements during the implementation, and your reasons for these changes. If you were not able to implement any aspects of your design due to practical limitations, keep them as part of your design (in coursework 2) but include a comment in this coursework about the difficulties you encountered.

4.7 Self-assessment (5%)

The assessment criteria are detailed in Fig. 4. Include a self-assessment similarly to previous courseworks in this section of your report.

For this section, consider to what extent you have met the assessment criteria for this assignment. For each criteria, indicate the mark which you believe your answer merits and justify your judgement. Format your answer as a two-level nested list, providing your predicted mark, and a brief justification of your answer.

The correctness of your code will be assessed primarily via the system tests you provide, since your systems will vary widely depending on your design. This means that in order to get the marks you must both provide tests which adequately test your system correctly implements the use case and your system should pass these tests. If you do not provide system tests for all of the required use cases you should expect to loose quite a few marks since there will be no way for your markers to tell if your system works correctly.

5 Practical details

5.1 Working practices

This coursework is a small-scale opportunity to try out ideas that have been mentioned in lectures. For example, you could try writing tests before code and using the tests to drive your design work. You could also try pair programming, whereby, in a session, one of you programs and the other gives feedback.

You are strongly encouraged to take an incremental approach, adding and testing features one by one as much as possible, always maintaining a system that passes all current tests.

Sometimes it is seen as important to have development teams and testing teams distinct. This way there are two independent sets of eyes interpreting the requirements, and problems found during testing can highlight ambiguities in the requirements that need to be resolved. As you work through adding features to your design, you could alternate who writes the tests and who does the coding.

You may also find it useful to collaborate on your code via a shared git repository[8].

5.2 Getting started

You will need access to a Unix command line with the make and git commands (as are available on DICE) and Java 8. Make sure you are able to get the skeleton code and run the provided sample tests as soon as possible so you can get help with any issues in the labs or on Piazza.

Start by cloning the skeleton code repository:

```
https://github.com/twright/Inf2C-SE-Bike-Rental-Skeleton
```

You can either create your own (**private!**) fork of this repository, download it as a zip file, or clone it using the command,

```
git clone https://github.com/twright/Inf2C-SE-Bike-Rental-Skeleton.git
```

This contains a top-level directory `Inf2C-SE-Bike-Rental-Skeleton` containing sub-directories `src` and `tests`.

The repository comes with a simple Makefile allowing you to compile the code and run tests from the commandline. First `cd` to the `Inf2C-SE-Bike-Rental-Skeleton` directory. Then run:

```
make test
```

If all goes well your will see the test results

```
╷
├─ JUnit Jupiter ✔
│  ├─ TestDateRange ✔
│  │  ├─ testOverlapsFalse() ✘ org.opentest4j.AssertionFailedError
│  │  ├─ testToYears1() ✔
│  │  ├─ testToYears3() ✔
```

[8] https://guides.github.com/introduction/git-handbook/

```
|   |   └─ testOverlapsTrue() ✘ org.opentest4j.AssertionFailedError
|   └─ SystemTests ✔
|       └─ myFirstTest() ✘ expected: <The moon> but was: <cheese>
└─ JUnit Vintage ✔
...
```

followed by more details of the failing tests. You will see more results as you add your own unit tests and system tests.

It is also possible to import this code into Eclipse and run the tests within the IDE.

The code is provided in the form of a GitHub repository. You may find it useful to use git to collaborate with your teammate, but **any repositories you create must be private**.

5.3 Provided code

The provided code includes the following.

- The DeliveryService and Deliverable interfaces and a DeliveryServiceFactory class maintaining a singleton instance of a delivery service.
- A MockDeliveryService class containing a mock implementation of the delivery service.
- An incomplete SystemTests class for the tests for your whole system.
- Incomplete PricingPolicyTests and ValuationPolicyTests classes for tests covering your extension modules.
- Skeleton Location, DateRange, Bike, and BikeType classes for you to complete.
- Incomplete TestLocation and TestDateRange classes for unit tests covering locations and date ranges.

The provided code should compile fine. However, all the provided tests will fail.

5.4 What can and cannot be altered

- Do **not** alter the provided DeliveryService, Deliverable interfaces, or the DeliveryServiceFactory and MockDeliveryService classes.
- The coursework requires you to add Javadoc comments to the Location class and to the DateRange class. You should provide the necessary methods but should not change the interface of the constructors or the provided methods.

It is expected you will introduce several new classes in addition to the provided classes. While it is possible to produce a functioning implementation with very few classes, such an implementation would have very poor object-oriented design.

6 Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work in some web repository, then you must ensure that access is restricted to only members of your coursework group. Be aware that not all free repository hosting services support private repositories, repositories with access restrictions. There are some more details about this on the Coursework Learn page.

7 What to submit

7.1 Working code, tests, and an output log

Your code **must** compile and run from a DICE command line, **exactly** as described above in Section 5.2. Please be aware that the default ECJ compiler used in Eclipse is a *different* compiler from the command line `javac` compiler. ECJ will compile and run code that `javac` will not compile. If you develop your code in Eclipse or some other IDE, or if you use some other non-DICE platform, you must double check it runs OK under DICE from a DICE command line before submitting.

We will assume you are using Java 8 and JUnit 5. Do not use any libraries outside of the standard Java 8 distribution other than the JUnit libraries provided in the lib subdirectory.

Create a zip file submission.zip of your code and tests, the output.log file, and your report.pdf file by setting your current directory to the main project directory, and running the command

```
make submission
```

You can capture the output of running your JUnit tests using the command

```
make testlog
```

This output.log file is a required part of your submission. The `make submission` command will automatically compile your code, create the test output log, and add this all to your submission.zip file, but you should check the submission.zip contains all of the files you want to be part of your submission. This zip file will also include your report.pdf file if you include it in your project directory.

7.2 Report

This should be a PDF file named report.pdf. Please do not submit a Word or Open Office document.

This only needs to include the written part of this assignment, consisting of a description of any design/requirements changes you made during the implementation of this coursework, and a self-assessment.

The report should include **a title page with names and UUNs of the team members**.

8 How to submit

You need to submit coursework 3, as well as any new iterations of courseworks 1 and 2, to receive your final marks. If you do not re-submit courseworks 1 or 2, the provisional mark you received for them on your latest previous submission at a previous deadline (if any) will become their final mark.

Your submission for coursework 3 should consist of the submission.zip file generated following the instructions in Section 7.1, and must include your code, tests, output.log, and your report.pdf. Please make sure your code compiles and runs at the time of submission and you can at least run your system tests for the main use cases.

For your submissions, ensure you are logged into MyEd. Access the Learn page for the Inf2C-SE course and go to "Coursework and labs" - "Coursework submission". For the re-submission of courseworks 1 and 2, use their own submission instructions (i.e. submit coursework 1 in "Coursework 1" and coursework 2 in "Coursework 2"). For the submission of coursework 3, use "Coursework 3".

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.

Only one group member should submit each of the 3 courseworks.
This coursework is due

16:00, Friday 29th November

The coursework is worth 50% of the total coursework mark and 20% of the overall course mark.

You must also have your extension submodule and its unit tests ready for peer review in the week 10-11 labs (before the coursework deadline!) otherwise you will not receive marks for the peer review.

Thomas Wright 8th November 2017.

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%
  - ◇ 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
- Unit tests for Location and DateRange 5%
- Systems test including implemented extension to pricing/valuation 5%
- Mock and test pricing/valuation behaviour given other extension **(challenging)** 5%

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
- 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**
- Quality of design and implementation 5%
  - ◇ Your implementation should follow a good design and be of high quality
  - ◇ Should include some assertions where appropriate
- Readability 5%
  - ◇ Code should be readable and follow coding standards
  - ◇ Should supply javadoc comments for Location and DateRange classes

4. Report **10%**

- Revisions to design 5%
  - ◇ Design document class diagram matches implemented system
  - ◇ Discuss revisions made to design during implementation stage
- Self–assessment 5%
  - ◇ Attempt a reflective self-assessment linked to the assessment criteria

Figure 4: High-level marking criteria.