Inf2C Software Engineering 2018-19

Coursework 3

Creating an abstract implementation of a auction house system

1 Introduction

The aim of this coursework is to implement and test an abstract version of software for a system for managing auctions at an auction house. 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.

2 Further design details

The auto-marking of your submitted code will assume you have precisely implemented further design details as described below.

2.1 AuctionHouse interface and implementation class

Your auction house implementation class should be named AuctionHouseImp and should implement the interface specified in Java interface AuctionHouse shown in Figure 1. The interface methods define the input messages your implementation must accept, specifying both arguments these messages carry and the types of return values. There is one method for each of the use cases identified in Appendix A of the Coursework 2 instructions. For simplicity String is used in a number of places where abstract types such as LotDescription and MessagingAddress were advocated in Coursework 2.

In addition AuctionHouseImp should define a constructor

```
public AuctionHouseImp(Parameters p) {...}
```

where the Parameters argument type

```
public interface AuctionHouse {

    Status registerBuyer(
            String name,
            String address,
            String bankAccount,
            String bankAuthCode);

    Status registerSeller(
            String name,
            String address,
            String bankAccount);

    Status addLot(
            String sellerName,
            int number,
            String description,
            Money reservePrice);

    List<CatalogueEntry> viewCatalogue();

    Status noteInterest(
            String buyerName,
            int lotNumber);

    Status openAuction(
            String auctioneerName,
            String auctioneerAddress,
            int lotNumber);

    Status makeBid(
            String buyerName,
            int lotNumber,
            Money bid);

    Status closeAuction(
            String auctioneerName,
            int lotNumber);
}
```

Figure 1: AuctionHouse Interface

```
public class Parameters {
    public final double buyerPremium;
    public final double commission;
    public final Money increment;
    public final String houseBankAccount;
    public final String houseBankAuthCode;
    public final MessagingService messagingService;
    public final BankingService bankingService;

    ...
}
```

collects together configuration parameters for the system.

2.2 Reporting status of method execution

A class Status is provided whose objects are used as return values of methods in order to indicate how methods have executed. It is used by methods declared in the AuctionHouse and BankingService interfaces. A fragment of its definition is

```
public class Status {
    public static enum Kind {
        OK,
        ERROR,
        SALE,
        SALE_PENDING_PAYMENT,
        NO_SALE
    }

    public Kind kind;
    public String message;
    ...
}
```

Enumerated types in Java are classes and the enumerated type Kind is an example of a nested class definition. The constants are referred to by names such as Status.Kind.OK and can be compared with the == operator.

Most methods should just make use of the OK and ERROR kinds. One exception is that the closeAuction() method in the AuctionHouse interface should use SALE, SALE_PENDING_PAYMENT and NO_SALE rather than OK. The scenarios for the return of each of these are as follows:

- NO_SALE: Hammer price is below the reserve price or no bids at all have been received on the lot.
- SALE: Hammer price is at least the reserve price and both payments from buyer and to seller succeed.
- SALE_PENDING_PAYMENT: Hammer price is at least the reserve price, but one of payments from buyer or to seller has failed.

The message component of a Status object is optional and is for providing further information about the status. See the Status class definition for the available constructors and some static methods for constructing common cases.

You should design your code so that input messages always return a Status object with `ERROR` kind if they are invalid. Scenarios when input messages are invalid include when repeat registering with the same name, an unregistered seller trying to add a lot, an attempt to note interest or open an auction on a non-existent lot. Sending in an input message should never result in your code throwing an exception. In addition, if an input message is invalid, the state of the system should always be unchanged. Your input message processing code should therefore always check validity first before going on to code that updates the state. These requirements makes it straightforward for test code to check whether your implementation is correctly identifying invalid messages. These checks of valid input are examples of *precondition* runtime checks, as discussed in class.

2.3 Lot status

A lot can have one of four statuses. A lot's status should indicated by an element of the LotStatus enumerated type:

```
public enum LotStatus {
    UNSOLD,              // Either not auctioned or unsold after auction
    IN_AUCTION,          // In a currently running auction
    SOLD,                // Sold in an auction
    SOLD_PENDING_PAYMENT // Sold, but attempt to collect money from buyer
                         // or pay money to seller has failed
}
```

Changes in status are triggered by the openAuction() and closeAuction() input messages. Valid status transitions are

- From `UNSOLD` to `IN_AUCTION` on openAuction()
- From `IN_AUCTION` to `UNSOLD` on closeAuction() when the sale does not go through. As a consequence a given lot may be in an auction multiple times.
- From `IN_AUCTION` to `SOLD` on closeAuction() when the sale does go through, payment is successfully collected from the buyer, and payment is successfully made to seller.
- From `IN_AUCTION` to `SOLD_PENDING_PAYMENT` on closeAuction() when the hammer is sufficient for a sale, but the buyer payment fails (the transfer() call drawing money from the buyer's account returns an `ERROR` status) or the seller payment fails (the transfer() call paying money to the seller's account returns an `ERROR` status). How such situations are subsequently resolved is beyond the scope of this coursework.

No other status transitions are allowed. A openAuction() message received for a lot that does not have `UNSOLD` status is an error. Likewise a closeAuction() message for a lot that does not have the `IN_AUCTION` status is an error. Also makeBid() messages are only valid for lots with `IN_AUCTION` status.

2.4 Messaging and Banking

As with Coursework 2, your design should make use of provided MessagingService and BankingService objects for notification messages and bank transfers respectively. With this coursework, MessagingService and BankingService are interfaces.

```
public interface MessagingService {

    void auctionOpened(String address, int lotNumber);

    void bidAccepted(String address, int lotNumber, Money amount);

    void lotSold(String address, int lotNumber);

    void lotUnsold(String address, int lotNumber);
}

public interface BankingService {

    Status transfer(
            String senderAccount,
            String senderAuthCode,
            String receiverAccount,
            Money amount);
}
```

Previously, the instructions were not precise on who exactly should receive each kind of notification. Your design should have the following behaviour:

- auctionOpened() messages should be sent to all buyers who have noted interest and to the seller of the lot.
- bidAccepted() messages should be sent to all buyers who have noted interest except the buyer making the bid, to the seller of the lot and to the auctioneer of the lot.
- lotSold() and lotUnsold() messages should be sent to all buyers who have noted interest and to the seller.

In each case, the sender of the input message triggering these notification messages does not need to receive such a notification message because they receive a reply message when the operation invoked by the input message returns.

The lotSold() and lotUnsold() messages should be sent out during closing of an auction when the lot status transitions to `SOLD` and `UNSOLD` respectively. If the lot status transitions to `SOLD_PENDING_PAYMENT`, no notification messages should be sent out.

Objects implementing these interfaces are passed to your system via the Parameters argument of the AuctionHouseImp constructor.

For test purposes, test code constructs *mock objects* [1] implementing each of these interfaces. These mock objects record calls to the interface methods and support comparison of

[1] `https://en.wikipedia.org/wiki/Mock_object`

these records with expectations. In addition, the BankingService mock object allows the designation of some bank accounts as bad. When a transfer is attempted from a bad account, the status returned by the transfer() method has kind ERROR rather than OK.

When an auction closes and the hammer price is at least the reserve price, your design should always collect payment from the buyer before making payment to the seller. If the collection of payment from the buyer fails, no attempt should be made to pay the seller; the auction house does not want the possibility of losing money when there are payment problems.

See the MockMessagingService and MockBankingService class code for further information.

2.5 Catalogue entries

The CatalogueEntry class has definition:

```
public class CatalogueEntry {

    public int lotNumber;
    public String description;
    public LotStatus status;
    ...
}
```

The description strings are as passed in by addLot() messages. The list of CatalogueEntry objects returned by the viewCatalogue() input message should always place the objects in increasing lot-number order.

2.6 Money

A Money class is provided. A fragment of its definition is as follows:

```
public class Money implements Comparable<Money> {

    private double value;
    ...
    public Money add(Money m) { ... }
    public Money subtract(Money m) { ... }
    public Money addPercent(double percent) {...}
    public int compareTo(Money m) {...}
    public Boolean lessEqual(Money m) {...}
    public boolean equals(Object o) {...}
}
```

The definition uses a double value to represent monetary values in pounds. A class invariant is that the stored value is always close to a monetary amount with a whole number of pence and is considered to represent that amount.

3 Your tasks

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

3.1 Construct code

There is no requirement that you stick to the design you produced for Coursework 2.

Follow good coding practices as described in lecture. Consult the coding guidelines from Google at

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

or Sun at

`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`.

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 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 in 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 web page for advice on how to set Eclipse to use spaces instead and also how to get Eclipse to display a right margin indicator line.

Be sure you are familiar with common interfaces in the Java collections framework such as List, Set and Map and common implementations of these such as ArrayList, LinkedList, HashSet and TreeMap. Effective use of appropriate collection classes will help keep your implementation compact, straightforward and easy to maintain. If you start wanting to implement some sorting algorithm to put CatalogueEntry objects in order, you have not been looking closely enough at this framework.

3.2 Write Javadoc comments

To show you know the basics of Javadoc, add Javadoc comments to the provided Money class. In particular, add a class Javadoc comment and method and field Javadoc comments for the methods and field shown in Section 2.6. The only tags you need to use are the @param and @return tags for the methods. 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.

3.3 Create unit tests

Add tests to the JUnit 4 test class MoneyTest to check the correct functioning of the Money class methods listed in 2.6.

3.4 Create system-level tests

Your are expected to use JUnit 4 to create system-level tests that run scenarios of each of the use cases you implement. The provided AuctionHouseTest class tests some but not all the features described in Section 2. You are expected to add further tests to more completely test all the features.

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.

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. The provided tests show one way of organising tests so that writing lots of incremental tests is easy. It is recommended that, when you can, you should have your tests build on auxiliary method runStory() in the same way that the provided tests do.

Include all your additional tests as JUnit test methods in the AuctionHouseTest class. Feel free to add extra further auxiliary methods that handle repeatedly needed sequences of input and output events. Your AuctionHouseTest.java file should serve as the primary documentation for your system-level tests. To this end, take care with how you structure your test methods and add appropriate comments, for example noting particular design feature being checked.

You are encouraged to adopt a test first approach. 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.

3.5 Add logging code

Add logging code to your code using logging support from the standard java.util.logging package. The provided code for the AuctionHouseTest, AuctionHouseImp, MockMessagingService and MockBankingService classes gives examples of the use of this package. Note that each class using logging includes a field

```
private static Logger logger = Logger.getLogger("auctionhouse");
```

that sets up a field logger that allows for easy reference to a Logger object. Logging messages are then created by statements like

```
logger.finer("Entering");
```

The logging messages automatically include the enclosing class and method names. It therefore can be sufficient for the logging text to be very brief, say just the word `"Entering"` as above, indicating the logging message is generated at the entry-point of a method. The message might also provide information about the object or arguments the method is called on or might generate a banner to improve log readability. See the provided code for examples of such logging messages.

Each log message is generated at a *logging level* with higher levels being for more important messages. Level names in increasing order of importance include *finest*, *finer*, *fine*, *info* and *warning*. Note how test-cases start with banners at the level *info*, but the main methods

in AuctionHouseImp for handling input messages use level *fine* and those in MockMessagingService and MockBankingService use level *finer*.

The static variable loggingLevel in the AuctionHouseTest class controls the minimum level that at which messages are reported. Have a brief browse of documentation on the web concerning this package to learn more about what is going on with logging.

Not every method you write needs logging. Your aim is to add enough logging that both you and the markers can follow the flow of control around your implementation objects and methods when each of your tests is run. Experiment with generating logging messages at different levels, so the volume of log messages generated can be controlled be varying the minimum logging level. See Section 4.2 for how to control this minimum logging level when running JUnit tests from a command line.

You are required to submit a file of the logging output from running your tests. This serves as documentation of the dynamic behaviour of your system. You are not otherwise required to submit any UML sequence diagrams or text descriptions of behaviour.

3.6 Keep a project log

Each member of a group is expected to keep and submit their own independent project log. This is the only task that should be done independently. All other tasks should be completed jointly.

This project log should contain an entry for each day you spend a significant time on the coursework. It should contain the following elements.

Plans What are your plans for completing the project? What activities are you going to do when? What times do you have available? As the project progresses, you can make plans for how you are going to complete the next individual tasks you have to work on.

Achievements Describe what you have achieved through each day you put in time on the coursework. Summarise outcomes of meetings with your partner. Note how much time you are putting in.

Reflection Has everything gone to plan or not? Were you optimistic or pessimistic? Did unforeseen issues come up? Are there ways in which you could improve your own pattern of work or your pattern of work with your project partner? How is your understanding evolving of the concepts involved in the coursework? Have you resolved issues that were puzzling you? Do you have new questions you need to find answers to?

Be sure to consider including remarks on working practices you experiment with. See Section 4.1.

Do not include details of design and implementation decisions in your log. Instead, refer to your report for this information. However, note that generally it is appropriate to include such information in project logs.

Your project log is expected to be no more than 2 or 3 pages. You can edit your project log in order to improve ease of reading and keep it to length.

Mention in your project log the overall time you have spent on this coursework and give some idea of how this broke down into different activities.

3.7 Write a report

This report is expected to be significantly shorter than that you produced for Coursework 2. A title page plus 2 or 3 further pages is fine. The report should have the following sections:

UML Class diagram Include boxes for the provided interfaces AuctionHouse , MessagingService and BankingService, provided class AuctionHouseImp and any other classes you create. Boxes are not required for other provided design classes; they can just appear in attribute and operation types. Test classes AuctionHouseTest, MockMessagingService and MockBankingService should not appear.

Do not try to exhaustively list all the methods and fields of the classes. Instead, only list a few of the most important ones. As is commonly the convention, do not show fields that implement associations drawn in the diagram. Do show the navigability of associations and do include multiplicities on their ends.

As with Coursework 2, try to avoid explicitly using container types. Instead use appropriate UML notation that suggests them. The one exception to this recommendation is with the catalogue returned by viewCatalogue(). For this it is fine to use List<CatalogueEntry>. To indicate the use of maps, you might consider making use of UML *qualified associations*. To indicate that an association suggests an ordered container, add the UML property "{ordered}" next to the end of the assocation that has multiplicity *.

High-level design description This could include some of what you produced for Coursework 2. However, only cover the parts you have actually implemented. Be sure to include discussion of your design decisions. Include remarks about how your design might have changed since Coursework 2.

Implementation decisions Discuss the main implementation decisions you have made. For example, state the kind of Java collection class you have used to implement each of the one-to-many associations in your design, and explain why.

4 Practical details

4.1 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.

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.

4.2 Getting started

Download the skeleton code zip file auctionhouse-skeleton.zip from the Coursework 3 section of the *Coursework and Labs* page on Learn. This can be placed anywhere, a temporary directory for example.

Unzip the file. This creates a top-level directory `AuctionHouse` containing sub-directories `src`, `bin` and `lib`. The code can be compiled and run from the command line. First `cd` to the `AuctionHouse` directory. Then run:

```
export CLASSPATH=bin:lib/junit-4.12.jar:lib/hamcrest-core-1.3.jar
javac -sourcepath src -d bin src/auctionhouse/*.java
java auctionhouse.AllTests
```

If all goes well you will see a bunch of logging messages followed by the text

```
TEST RESULTS
Number of tests run: 11
SOME TESTS FAILED
Number of failed tests: 5
```

and some details about each of the failed tests. To run with reduced or no logging messages, try

```
java auctionhouse.AllTests info
```

or

```
java auctionhouse.AllTests off
```

Other options for the logging level argument include `fine` and `finer`.

Instructions on how to import this code into Eclipse are available from the Coursework 3 section on Learn.

4.3 Provided code

The provided code includes the following.

- The AuctionHouse interface and associated classes such as Status, Money, CatalogueEntry and LotStatus.
- An AuctionHouseImp class providing a skeleton implementation.
- Interfaces BankingService and MessagingService.
- A Parameters class for passing in configuration information to AuctionHouseImp, including test objects with BankingService and MessagingService types.
- A partially complete JUnit test class AuctionHouseTest for your whole system.
- Test classes MockBankingService and MockMessagingService implementing the BankingService and MessagingService interfaces.

- A skeleton JUnit test class MoneyTest for the Money class.
- A class AllTests with a main method which enables all JUnit tests in the AuctionHouseTest and MoneyTest classes to be run from the command line.

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

4.4 Modes of testing

In Eclipse you can run individual JUnit tests in either the AuctionHouseTest class or MoneyTest class, all tests in one of these classes, or all tests in both classes using the JUnit test suite defined in the AllTests class.

It is also possible to run the AllTests class as a Java application, as it defines a static main method. See Section 4.2.

4.5 Singleton classes

Your AuctionHouseImp class and perhaps others too will be singleton classes; you will only intend ever to create one instance of each such class. Do *not* do anything special to realise the Singleton design pattern. For example, do *not* use static fields to hold a singleton's data, a static field to refer to the single instance of a singleton class or a static method to retrieve the single instance. Just create one instance of the class when setting up your system.

This ensures that each of your tests starts with a completely fresh state and you do not encounter subtle testing bugs when state from one test persists into the next.

4.6 What can and cannot be altered

- Do **not** alter the provided AuctionHouse, MockMessagingService and MockBankingService interfaces or the provided Status, Parameters, CatalogueEntry, LotStatus, MockMessagingService and MockBankingService classes. Auto-marking compilation and test will rely on these being as supplied.
- In order to complete the coursework you will need to modify the code in the AuctionHouseImp class, ensuring the class still implements the AuctionHouse interface. However, do not change the provided constructor's arguments; testing assumes that the AuctionHouseImp class defines a constructor of form:

```
public AuctionHouseImp(Parameters parameters) {...}
```

- The coursework requires you to add Javadoc comments to the Money class. However, do not change the fields and methods of this class.
- You are free to modify the provided AuctionHouseTest, MoneyTest and AllTests test classes.

It is expected you will introduce several new classes in addition to the provided classes. While it is possible to produce a functioning implementation by adding no further classes, such an implementation would have very poor object-oriented design.

5 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 web page.

6 What to submit

6.1 Working code, tests and an output log

Your code **must** compile and run from a DICE command line, **exactly** as described above in Section 4.2. This is essential for the auto-marker. 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.

Auto-marking will use Java 1.8.

Do not use any libraries outside of the standard Java 1.8 distribution other than the JUnit libraries provided in the lib subdirectory.

Create a zip file src.zip of your code and tests by setting your current directory to the main project directory containing the src, bin and lib directories, and running the command

```
zip -r src.zip src
```

Again, it is important you follow these instructions **exactly** so that the auto-marking scripts can run without problems.

Capture the output of running your JUnit tests using the command

```
java auctionhouse.AllTests &> output.log
```

This output.log file is a required part of your submission.

6.2 Report

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

If you are using draw.io to draw diagrams, export your diagrams as PDFs, not as bit-maps (e.g. .png files). Make sure your PDF is viewable using the evince application on a DICE machine. The report should include **a title page with names and UUNs of the team members**.

6.3 Project log

Keep this as say a Word or Open Office document, but convert it to PDF for submission. Group members should submit their project logs separately and name their project log files project-log.pdf. See the submission instructions below.

6.4 A team members file

Create a text file named team.txt with only the UUNs of the team members (one UUN on each line) as shown,

```
s1234567
s7891234
```

Do **not** include any other information in this file, such as the names of the team members. At the start of marking a script has to be able to process these files.

7 Marking Scheme

1. **Code quality**: Use of coding standards, ease with which code can be read and understood. Use of Javadoc for Money class. (5%)
2. **Report**: This mark is based on the quality of the design and implementation information. (35%)
3. **Code output log**: Is it easy to follow the execution of your use cases by reading the log along with your class diagram? (5%)
4. **Code correctness and completeness**: This will be assessed by auto-marking, followed by a review of the auto-marking results. Auto-marking will be more thorough than the provided tests. of the phases. It will only test features described in this or previous Coursework handouts. (30%)
5. **Project log**: (20%)
6. **Quality of unit tests**. Have you written unit tests in the javaMoneyTest class to adequately test the methods of the provided Money class? (5%)

8 How to submit

Each member of a group has to submit some files.

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place all the requested files in the same directory and cd to this directory.

Only one group member should submit the code, output log and report. That group member should run the command

```
submit inf2c-se cw3 src.zip output.log report.pdf project-log.pdf team.txt
```

If the group has two members, the second group member should run the command

```
submit inf2c-se cw3 project-log.pdf
```

This coursework is due

16:00 on **Tuesday 27th November**

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

A Document history

v1. (8 Nov) Initial release.

v2. (13 Nov).

- Added information to Section 2.2 on exactly when closeAuction() should return status kinds `NO_SALE`, `SALE` and `SALE_PENDING_PAYMENT`.
- Added information to Section 2.3 on what should happen when payment from the auction house to the seller fails.
- Added information to Section 2.4 on which notifications and banking transfers to make in the different scenarios of the *Close auction* use case.

Paul Jackson. v2. 13th November 2018.