Inf2C Software Engineering 2018-19

Coursework 2

Creating a software design for an auction house system

1 Introduction

The aim of this coursework is to create and document a design for the software part of a system for managing auctions at an auction house. To create the design you are encouraged to experiment with the class identification technique discussed in lecture 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 test.

To ensure some uniformity in the starting points for both this Coursework 2 and Coursework 3, these instructions include in Appendix A an outline of a set of use cases. Your Coursework 2 design should conform to this information.

Unless otherwise specified here, your design should conform to the Coursework 1 system description and be at a similar level of detail. You are free to follow assumptions and clarifications you have adopted in your Coursework 1 submission. However, there is no requirement for you to do so. Markers will not be referring back to your Coursework 1 submission when marking this coursework.

2 Design document structure

Ultimately, what you need to produce for this coursework is a design document that combines descriptive text with UML class and sequence 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.

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

2.1 Introduction

Start your document with a sentence or two describing the whole system and then refer the reader to these instructions and the previous instructions for further general 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 (30%)

Attribute descriptions must include their types and operation descriptions must include types of any parameters and the type of the return value, if relevant. When the intended use of a parameter is not clear from the type alone, do also include a suggestive parameter name. Include operations when you consider them important for the execution of the use cases listed in Appendix A. Otherwise do not include them. Do not include simple operations such as attribute getters and setters. Do not show access control information for attributes and operations.

Because operation descriptions in full might be rather long, consider describing the class model with more than one UML class diagram. For example, one diagram might show all classes and include associations and other dependencies such as generalisation dependencies between classes, but leave out attribute and operation type information and parameter names. One or more other diagrams might show for each class full information about its attributes and operations, but omit all associations and other dependency information.

Associations must include multiplicities each end. Leave navigation arrowheads off associations: at this abstract level of design, this information is not needed. As needed, give associations names or add role names to association ends. When a relationship between two classes is indicated by an association, do not also include attributes in either class for handling the association. This is redundant information and potentially confusing. Only show associations when they have long lifetimes, when they persist between use case executions. Do not show associations when they only exist for the duration of an operation execution.

Do not use any collection classes in types (e.g. Set, List or Map), unless otherwise recommended below. The need for such classes in an implementation should be inferrable from appropriate use of multiplicities.

One or more classes will model utility concepts such as monetary amounts. Because of their simplicity, such *utility classes* will likely appear as the argument or return types of operations and as attribute types, but not as classes with associations to other classes in the class diagram.

2.2.2 High-level description (25%)

The high-level description should include justification for the design you chose, including specific rationale for the decisions made in the design. What alternatives did you consider and why did you make the choices you did? How have you tried to make your design adhere the principles of good design outlined in the lecture on software design?

It is likely that you will need to make assumptions concerning ambiguities and missing information in the system description provided for the first coursework. Be sure to record the assumptions you make in this section.

A viewer of your class diagrams should quickly form an impression of the structure of your design. But obviously the diagrams leave 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

2.3.1 UML sequence diagram (25%)

Construct a UML sequence diagram for the *Close auction* 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 Behaviour descriptions (20%)

UML sequence diagrams could be used to illustrate all the use cases. However, it can be rather time consuming to draw all these diagrams. Instead, for the use cases *Add lot*, *Note Interest* and *Make bid*, produce textual descriptions of the objects involved and the flows of messages.

Here is an example of such a textual description. It illustrates a fragment of the behaviour of the *call lift* use case explored in Tutorials 1 and 2.

```
aUser -- press() --> aFloorButton
   aFloorButton -- requestFloor() --> theController
      theController -- close() --> theDoors
      theController <- - - - - - - theDoors
      theController -- startMovingLiftUp() --> theMotor

LOOP [while lift not at destination floor]

theLift -- reportPosition() --> thePositionSensor // considering lift as an actor
 LOOP [for each position display]
   thePositionSensor -- update() --> aPositionDisplay
 ENDLOOP
   thePositionSensor -- recordPosition() --> theController

ENDLOOP
```

The notation `a -- m() --> b` is for object `a` sending call message `m()` to object `b`. The notation `a <- - v - - b` is for object `b` sending a reply message carrying data value `v` to object `a`. Reply messages need not always be shown and operation arguments and return data values need only be named if it adds clarity. As shown above, feel free to add comments to these descriptions.

For both your sequence diagram and these behaviour descriptions, you may consider including some getter or setter messages if doing so clarifies what is going on. For example,

one of your classes A might have an attribute foo and your sequence diagram might send a message getFoo()" to an A object.

You may for your own benefit create and include textual descriptions for the other four use cases. However these will not be considered during marking.

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.

View system activity as consisting of bursts of messages. Consider each burst as started by some actor instance sending a message to some system object and then consisting of a sequence of messages between system objects and from system objects to actors in the outside world. Each burst corresponds to the execution of a scenario of a use case. 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 Messages back to actors

Messages are either

- *call messages*, which cause the invocation of an operation on the message destination object,
- *reply messages*, which correspond to the return of an operation invoked by some earlier call message.

Actors sending messages into the system at the start of a burst get a reply message at the burst end. However, several use cases also require that actors other than the one triggering a burst also get notification messages. For example, the *Open auction* use case requires all buyers who have noted interest in a lot to be notified of the opening of the lot's auction. To model this include in your design a special MessagingService class that has a single instance (i.e. it is a *singleton class*, it realises the *Singleton* design pattern). For each kind of message sent out to actors modelling people acting in particular roles (e.g. a buyer or a seller), add an operation to this class that includes an argument of type MessagingAddress for specifying who the message should be sent to, as well as one or more further arguments specifying the information to be contained in the method. Do not be concerned with how these further arguments might be formatted into a single string. In a real system, this messaging service might be realised using some bought-in instant messaging package.

A further case when messages need to be sent to actors is for supporting financial transactions. For this, assume there is a singleton class BankingService with an operation

```
transfer(
    senderAccount : String,
    senderAuthorisation : String,
```

receiverAccount : String,
amount : Money) : Status.

Here, the senderAccount and receiverAccount arguments provide full information on the respective accounts, senderAuthorisation is some code the sender provides to authenticate the payment request (e.g. like the 3 digit security code on the back of most bank cards), Money is a utility class for representing currency values, and Status is some type which indicates whether the transaction was successful. Assume the classes Money, Status and String are provided; they do not have to be defined in the class diagram and they can be used throughout the design description.

3.3 Abstract inputs

In practice users might interact with the auction house system via a web server that can generate various pages that include buttons and text boxes for users to provide further input. Do not model any of this. Instead just model interactions between users and the system using abstract messages. For each use case, create one kind of message that a user sends in to trigger the use case execution. Include as arguments to this message all the information the user needs to provide to the system as part of the use case; do not model some dialogue between the system and users where the system is prompting for further information.

When the user who is the source of an input message is important, include a user-name string as one of the arguments to the message. In a real system users might use passwords to prove their identity to the system. For simplicity, do not try to model user authentication by passwords or any other means.

Assume that each lot is uniquely identified by a number which is included in the input message for the *Add lot* use case. We can imagine that auction house staff might pass sellers this information just before they trigger the *Add lot* use case.

3.4 Abstract data

Assume that provided classes include PersonalDetails for the *Register buyer* and *Register seller* use cases and LotDescription for the *Add lot* use case. Treat these classes as abstract; assume nothing in particular about their content. Consider it possible that LotDescription objects include low and high estimates of the lot's value. Assume these objects do not include reserve prices; these must be provided as distinct arguments for *Add lot* use case input messages. Remember that execution of the *Close auction* use case needs to compare the hammer price with the reserve price.

3.5 Abstract outputs

There are ways in which almost every use case can go wrong. A user-name provided with the input message for a *Note interest* use case might not correspond to a previously-registered user-name of any buyer. A lot number provided in the *Make bid* use case might be invalid. To handle such situations, assume the reply messages associated with input messages carry values that are instances of this class Status mentioned above.

Assume that a class CatalogueEntry is provided for holding information about a lot such as its identification number, its description and its status. Assume that the reply message back to the member of public user for the *View catalogue* use case returns an object of type CatalogueEntry List.

3.6 Auction House class

A major design task is deciding which system class or classes will be responsible for executing the scenarios triggered by each input message. Further, such classes might be distinct from the classes chosen as the initial entry points for these input messages. In some cases, messages could be sent first to an object of one class and then forwarded on via perhaps one or more intermediate objects to an object of a class that has an operation for executing the heart of the use case.

A very simple option is for each use case to designate a single class both as the initial entry point and as provider of code for executing the use case. This can be suitable for an early design phase, as it completely postpones the question of how input messages get routed from some user interface objects to objects from classes that have operations for executing the corresponding use cases.

The strong recommendation is that you do not follow this option. Instead, include in your design a central singleton AuctionHouse class that all input messages are directed to. This is in preparation for Coursework 3 where tests will be provided for your implementation. These tests will assume that all implementations include such a class. The advantage of this approach is that it means the tests make relatively few assumptions about the structure of the rest of your system design. This gives you a high degree of design and implementation freedom. This approach also models at an abstract level how user interface code often has a central component for deciding how to act on user input.

It is strongly recommend that you do not also site the operations which execute the bulk of use case scenarios in this AuctionHouse class. Rather, use this class's singleton object as a portal into the system and have the operations handling input messages mostly just forward on those messages to suitable other classes. This avoids the AuctionHouse class becoming very large and having poor cohesion. This use of an AuctionHouse class is an example of the *Facade* design pattern [2].

When spreading around responsibility for executing use cases, it is usually a bad idea to create a separate class for each use case. This results in a large number of classes with lots of coupling and generally poor object-oriented design structure.

3.7 Classes associated with human actors

It is natural to consider introducing a class corresponding to each actor. For this coursework, you might introduce Buyer, Seller and Auctioneer classes, for example. Such classes can have several perhaps-distinct responsibilities. These include

1. Storing data associated with the actor, the user-name and bank account number of a buyer or seller, for example.

[2] `https://en.wikipedia.org/wiki/Facade_pattern`

2. Handling messages coming in from the outside world from the corresponding actor.

3. Sending messages from the system to the corresponding actor in the outside world.

A class that combines two or three of these responsibilities might have poor cohesion; the complexities of interfacing with the outside world might have little to do with storing some pieces of data. Data objects might end up coupled to objects handling interfacing that they do not have a strong natural association with. In some cases it might be worth instead having distinct classes for the different responsibilities. For example, one could have a BuyerInfo class for information about buyers, BuyerInput class for handling input messages and a BuyerInput class for handling output messages to buyers. Then again maybe this would result in too many classes.

Do consider whether other classes in your design could better handle these interface messages. For example, the instructions above specify you should use a single MessagingService class for ultimately handling many output messages. Is there any need for such messages to previously be routed via say Buyer or Seller objects?

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 two files

1. 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**.

2. 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.

How to Submit

Only one member of each coursework group should make a submission. If both members accidentally submit, please alert the course organiser so confusion during marking is avoided.

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place your *design.pdf* requirements document and your *team.txt* file in the same directory, ensure this directory is set as your current directory (i.e. `cd` to it), and submit your work using the command:

```
submit inf2c-se cw2 design.pdf team.txt
```

This coursework is due at

16:00 on **Monday 5th November**

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

A Use cases

Here is a mostly-complete list of use-case titles and primary actors.

1. *Register buyer* (Buyer)
2. *Register seller* (Seller)
3. *Add lot* (Seller)
   Coursework 1 suggested that this use case would involve auction house staff triggering this use case, perhaps including information provided by the seller of the lot. For Coursework 2 for simplicity, consider this use case triggered by the seller directly. There is therefore no need for auction house staff actors to be considered in this coursework.
4. *View catalogue* (Member of Public)
5. *Note interest* (Buyer)
6. *Open auction* (Auctioneer)
7. *Make bid* (Buyer)
   For simplicity, just handle the *jump bid* scenario described in the Coursework 1 instructions.
8. *Close auction* (Auctioneer)
   As recommended in the Coursework 1 instructions, consider this use case to include the financial transactions that follow if the lot is sold.

Paul Jackson, 22nd October 2018.