Inf2C Software Engineering 2016-17

Coursework 2

Creating a software design for a restaurant order-management system

1 Introduction

The aim of this coursework is to create and document a design for the software part of a new restaurant order-management system. 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. The follow-on Coursework 3 will deal with implementation and test.

Please refer back to the Coursework 1 instructions for a system description. To ensure some uniformity in the starting points for both this Coursework 2 and Coursework 3, these instructions include appendices which sketch part of a solution to Coursework 1. Appendix A covers the chief application domain entities, Appendix B lists the input-output devices and Appendix C outlines a set of use cases. Your Coursework 2 design should conform to this information.

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.

2.1 Introduction

Start your document with a few sentence description of 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 diagrams and class descriptions)

This section must contain a complete UML class model, a high-level English description of the model, and some further documentation for each class.

2.2.1 UML class model (30%)

The class model may use more than one UML class diagram. For example, one diagram might show just class names and the associations and other dependencies such as generalisation dependencies between classes. Other diagrams might omit the associations, but show for each class its attributes and operations. The operation descriptions must include types of any parameters and the type of the return value, if relevant. Associations must include multiplicities each end. When a multiplicity is possibly greater than one, consider adding the *UML property* text '{ordered}' also next to the association end. This property indicates that it is important that instances of the class at the association end be maintained in some order.

Leave navigation arrowheads off associations: at this abstract level of design, this information is not needed.

Some classes will model utility concepts such as time and authorisation codes for bill payments. 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.

There is no requirement for you to use a particular tool to draw your class diagrams. The *draw.io* tool[1] is one easy-to-use tool you might try. See post @40 on the course discussion forum for advice on *draw.io* and other tools.

2.2.2 High-level description (15%)

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. For example, it is reasonable to assume that order tickets are removed from the kitchen display when the orders are completed. Be sure to record the assumptions you make in this section.

2.2.3 Further class documentation (15%)

To provide documention for each class beyond what is shown in the class model, provide a brief description of the class, giving some indication of the purpose of each attribute and operation when this is not obvious from its name in the class model. Also describe the expected behaviour of any operation if this not obvious. For example, if the operation involves some calculations, it could be worth explaining these calculations.

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

For each operation, indicate by number which use-cases from Appendix C motivate the inclusion of the operation. Do not include operations if they are not needed by any of these use cases.

Indicate when a class is modelling an input and/or output device and when it can be considered a utility class.

2.3 Dynamic models

2.3.1 UML sequence diagrams (20%)

Construct UML sequence diagrams for the following use cases from Appendix C.

- 1(d): *Add menu item to order ticket*
- 1(g): *Pay bill*
- 2(b): *Indicate order item ready*

To simplify your diagrams, do not include lifelines for actors, but do draw short message lines for the messages exchanged between actors and interface objects. In the case of messages sent from actors to objects handling input, draw a black filled-in circle at the tail end of the message arrow. In UML, this is called a *found* message. 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 behaviour. Again, see post @40 on the course discussion forum for further advice on *draw.io* and other tools that could be used for drawing these diagrams.

2.3.2 Behaviour descriptions (20%)

UML sequence diagrams could be used to illustrate all the use cases listed in Appendix C. However, it can be rather time consuming to draw all these diagrams. Instead, for this coursework it is sufficient to produce textual descriptions of the objects involved and the flows of messages for the following use cases:

- 1(b): *Show menu*
- 1(f): *Submit order*
- 2(a): *Show current order*
- 3(a): *Cancel ready-up light*
- 4(c): *Remove item from menu*

Describe important operation arguments carried by call messages and important operation return values carried by reply messages.

3 Further information

3.1 IO interface classes

As with the Lift system covered in Tutorial 2, you should have a class for each kind of device that handles input and/or output. The devices you need are listed in Appendix B.

3.2 Input and output events

Let us call the messages exchanged between actors (strictly-speaking, actor *instances*) and objects representing interface devices *events*. *Input events* are sent by actors to interface objects and *output events* are sent by interface objects to actors. These event messages are rather special because they cross the system boundary and model interaction between the system and actors rather than one operation invoking another wholly within the system.

Classes for devices handling input should have operations corresponding to input events: a Button class should have a press() operation. The input event of an actor pressing a button is modelled by the invocation of this press() operation on a Button object representing the button. Classes for devices handling output should have operations for generating output events: a Light class might have an switchedOn() operation. The software system generates the output event of a particular light being switched on by invoking the switchOn() operation on the object representing the light. Usually the output event message has a related, but not identical name to the operation that generates it. It might describe the action just performed at the output interface object or might be a command to whatever actor is receiving the output event. For example, when a light is switched on, the event set out by the Light object might be called switchedOn() or viewLightOn().

3.3 Modelling using message bursts

Create a system design with a single thread of control. Do *not* try to make it concurrent. This single-threaded assumption is unrealistic, but it keeps the design and implementation much simpler and manageable in the time available.

System activity consists of bursts of messages passed between objects, each triggered initially by some input event, some actor instance sending a message to some object representing a device handling input. The messages in a burst in general will include some sent to objects representing devices handling output, and so some messages will generate output events. Each burst generally corresponds to the execution of some scenario of a use case or some fragment of a scenario. Because of the single-threaded nature, the system does not handle further trigger input events during a burst. Further trigger input events are only handled after any current burst has completed.

In general, it is useful to consider bursts which involve input events after their start. For example, in a burst corresponding to a scenario of the *Pay bill* use-case, a customer might tap their bank card against a bank card reader, after being prompted to do so on the touch-screen display of a table. Model the handling of the card-reading input event by having a call message being sent to the card reader object and having the reply message corresponding to this call carry some bank card details. Let's call input events that occur in the middle of bursts *non-trigger* input events.

We imagine bursts complete relatively quickly, sometimes in well under a second, other times in at most a small number of minutes. The duration of a burst will be made up of not only time when the system is executing, but also time when it is waiting for non-trigger input events from actors.

3.4 Level of abstraction of inputs and outputs

In a real-world implementation there would be a tremendous amount of detail in the structure and timing of the input and output events handled. Both for design purposes and to keep the time required for this coursework reasonable, you must abstract away from most of this detail.

Here are some examples of abstractions you are recommended to make.

1. The input event of releasing a previously pressed button can probably be ignored if the system's Button class does handle a press() input event. If this event is ignored, the Button class does not need even provide a release() operation.

2. As recommended previously for Coursework 1, consider a touch-screen display as a single single input-output device, i.e. do not create separate classes modelling the transparent touch position sensor and the screen display underneath the touch sensor. Create single abstract task-level input events for use case steps that conceptually involve input being provided to the system, but that in practice might involve a number of input and output events. For example, the *Add menu item to order ticket* use case might involve the step of navigating through various menu pages before selecting some item to order. We can abstract this step to some single input event which involves a customer sending an addMenuItem() message to the touch-screen display object with, as argument, some identifier for the menu item. To support this abstraction, the customer needs access to an identifier for each menu item. We imagine the touch screen display object having say a showMenu() operation capable of generating a showMenu() output event that carries as an argument a list of all the menu items along with their identifiers. We ignore all details about how the menu might be visually presented on the display itself.

   As touch-screen display devices are modelled at this abstract task level, distinct classes are needed for the touch-screen displays at the tables and for the touch-screen display in the kitchen. Your designs for these classes will share few, if any, common operations.

3. Similarly, the keyboard, mouse and monitor in the restaurant office should be a single device with operations at the abstract level of the functions the office computer user requires.

4. For making charges to a bank card, assume there is a class for the banking server interface supporting a charge() operation that takes as arguments some bank card details and an amount to charge, and returns some authorisation code for the payment. This is an example of a single operation taking care of both an output event and a (non-trigger) input event.

3.5 Abstract identifiers

As remarked in the previous section, some use cases need to have a way of referring to components in the output of other use cases.

- When adding a menu item to an order ticket, there must be a way of referring to the menu item observed in the display of the menu.
- When subtracting an item from an order ticket, there must be a way of referring to an item observed in the display of the current order ticket.
- When removing a menu item from the menu, there must be a way of referring to an item observed in the display of the current menu.

To refer to an order item or menu item, we need a name, an *identifier* for it. An identifier here could be a number indicating the position of the item in a display list, some unique short string associated with the item at some stage, or the reference (opaque address) of the object in the system representing the item.

To simplify working with identifiers, it is recommended that you make the following assumptions:

1. Every object has a unique ID (identifier) assigned to it when it is created.
2. The ID of an object never changes as the object is passed around the system you are designing.
3. The type of identifiers for objects in a class Foo is FooID.
4. Implementations of output-event generating operations in interface classes can lookup IDs of objects and add this ID information into their output events.
5. Let there be an association between classes A and B that has multiplicity 1 at the A end and multiplicity * or similar at the B end. It is always straightforward to implement operations for class A similar to
   - getB(i : BID) : B that, when invoked on object a from class A, returns any B object linked to a with ID i, and
   - deleteB(i : BID) that, when invoked on object a from class A, deletes any link from a to a B object with ID i.

3.6 Separating IO classes from the domain model

In graphical user interface design, the Model-View-Controller pattern and its variants and descendants all provide advice on how to separate classes handling input and formatting output from those modelling the application domain under consideration. This enables multiple views onto the same underlying domain data structures and simplifies keeping the underlying domain data and the presentations of it synchronised. More generally, this separation of concerns results in classes with better cohesion and that are easier to understand and maintain.

Please do *not* try implementing the MVC pattern or some similar pattern: they are not appropriate for the level of abstraction of the design you need to create. However, in your design, I do recommend you adopt this separation. Do not use your input/output interface classes to model the central domain entities of menus, order tickets and the kitchen order rack, as described in Appendix A. Rather create distinct classes for these domain entities that stand alone, separate from the I/O classes corresponding to the devices listed in Appendix B.

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
   ```

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 2 design.pdf team.txt
```

This coursework is due at

16:00 on **Tuesday 8th November**

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

A Application entities

1. *The menu.* The menu describes the items available for ordering in the restaurant. For each item it identifies a price.

2. *An order ticket.* An order ticket describes the set of items that a group of customers wish to order. Orders in general might have multiple requests for a given item, so by each item the ticket specifies a quantity, rather than listing the item multiple times. Each order ticket also records information on the table submitting the order and on the time of submission.

3. *The order rack.* The order rack maintains the list of current order tickets, ordered by submission time.

B Input-output devices

The input (I), output (O) and input-output (IO) devices mentioned or directly implied by the description are as follows:

At each table in the dining area

- touchscreen display (IO)
- receipt printer (O)
- contactless card reader (I)

In kitchen:

- Large touchscreen display (IO)

At pass (pass-through area between kitchen and dining area):

- Ticket printer (O)
- Order ready-up light (O)
- Press button to cancel light (I)

At restaurant office:

- Office computer, including keyboard, mouse and display (IO)
- Internet connection (for card authorisation) (IO)

C Use cases

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

1. **At table**
   Here the activities involved in building an order are broken down into several use cases. In practice, when interacting with the touch-screen display, some of these use cases might be integrated: e.g. part of the screen might show the menu, another the current order ticket, and adding a menu item to the order might involve tapping some button shown by the item. However, for design purposes, it is easier to separate out each of these activities.

   The step of starting an order was not made explicit in the previous instructions. However, it useful to consider the table touch-screen unit initially displaying some welcome screen, perhaps with some instructions on what to do, and a new set of customers at a table tap some *Start order* button on this screen before being able to add items to the order.

   After the *Pay bill* use case is executed, we can imagine the touch-screen display returned to the welcome screen for the next round of customers.

   (a) *Start new order ticket* (Customer)
   (b) *Show menu* (Customer)
   (c) *Show order ticket* (Customer)
   (d) *Add menu item to order ticket* (Customer)
   (e) *Subtract item from order ticket* (Customer)
   (f) *Submit order* (Customer)
   (g) *Pay bill* (Customer)
   This includes viewing the bill, paying with a bank card and possibly collecting a receipt.

2. **In kitchen**

   (a) *Show current orders* (Clock)
   We imagine this use-case triggered frequently, several times a minute.
   (b) *Indicate order item ready* (Chef)

3. **At pass**

   (a) *Cancel ready-up light* (Waiter)

4. **In office**

   (a) *Show menu* (Head chef)
   (b) *Add item to menu* (Head chef)
   (c) *Remove item from menu* (Head chef)

Paul Jackson, 24th October 2016.