Inf2C Software Engineering 2017-18

Coursework 2

Creating a software design for a tour guide app

1 Introduction

The aim of this coursework is to create and document a design for the software part of a new tour guide app. 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.

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.

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.

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

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%)

The operation descriptions must include types of any parameters and the type of the return value, if relevant. Include operations when you consider them important for the execution of one of the three main use cases from Coursework 1. Otherwise do not include them. Do not include simple operations such as attribute getters and setters. 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 the physical location of a waypoint. 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.

You might find it useful to express the class model using 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.

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 diagram should quickly form an impression of the structure of your design. But obviously the diagram leaves 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 follow tour 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 browsing tours and creating a tour use cases, it is sufficient to produce textual descriptions of the objects involved and the flows of messages. In these descriptions, include important operation arguments carried by call messages and important operation return values carried by reply 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`. As with sequence diagrams, reply messages need not always be shown and data values need only be named if it adds clarity. As shown above, feel free to add comments to these descriptions.

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.

System activity consists of bursts of messages passed between objects, each triggered initially by some actor instance sending a message to some system object. Each burst corresponds to the execution of some fragment of a scenario of a use case. Assume that the only messages sent from system objects back to actor instances are the final reply messages of bursts. Such a message corresponds to the return of the method invoked by the trigger message sent by the actor. 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 Abstract inputs

Consider input messages at an abstract level. Do not model the touch-screen user interface and other possible sources of the input messages. Here are some examples:

1. To start the creation of a tour, a *Tour Author* actor instance might send in a *create tour* message, perhaps carrying information in arguments such as a title for the new tour.

2. During tour creation, the *Tour Author* actor instance repeatedly might send in an *add waypoint* message.

3. Both tour creation and tour following rely on knowing the current location, perhaps obtained from a GPS receiver built in to the smartphone or tablet running the app. Assume there is some *Location Service* actor external to the system that can regularly send in *set location* messages carrying as arguments information about the current location.

Sometimes there is a need for input messages to refer to system objects. In particular, a *start following tour* input message might want to indicate which tour should be followed. Do this by associating some short identifier with each such object. `Tour1` could identify one of the tours. Such identifiers can be used as input message arguments and should be included in appropriate data output by the system. For example, if the system sends an output message to a *Tourist* actor instance that lists the titles of tours on offer, each tour title should be accompanied by the corresponding tour identifier.

3.3 Abstract data

As mentioned in Coursework 1, whole tours, tour waypoints and tour legs might be annotated with combinations of formatted text, pictures, video and audio. Continue to ignore all details of annotations. Just include in your design an abstract *Annotation* class whose objects can hold such data. Messages sent into the system during tour creation can carry *Annotation* objects as arguments. And an annotation is presented to a tourist following a tour by having a message burst end with a reply output message including an *Annotation* object in its return value.

3.4 Abstract outputs

Do not model the complexities of a screen that might be displaying a formatted combination of text, pictures and video. Nor should you model any audio output device.

As suggested In the Coursework 1 instructions, the app must be capable of presenting the user simultaneously with several pieces of data. For example, when following a tour, the app might present simultaneously

- the bearing of and distance to the next waypoint of the tour,

- information about the waypoint currently being visited,

- information about the current tour leg.

Let us call such pieces of presented data *output chunks* or just *chunks*. Imagine having a *Chunk* abstract class. Then let us have for each kind of chunk a sub-class, each defining appropriate fields for its objects. One sub-class for the direction of the next waypoint might have fields of type `double` for the bearing and distance. A sub-class for information about a waypoint would have a field of *Annotation* type.

Multiple pieces of data can be presented simultaneously to the app user by having the return value of reply messages carry multiple *Chunk* objects.

3.5 Separating input and output

Which message bursts should terminate with a return value carrying chunks? It is strongly recommended that your design supports an input message *getOutput()* which returns with the chunk or chunks that currently should be presented to the app user, but otherwise does not change the internal state of the app. All other input messages should cause bursts that end with no return data.

We can imagine that the part of a real app responsible for presenting data periodically sends *getOutput()* messages to the core design this coursework is concerned with in order to update the presentation.

This separation of input and output is reminiscent of what happens with the *Model-View-Controller* design pattern and its derivatives which are widely used in user interface design. With this design pattern user inputs such as presses of buttons on a GUI are handled by the *Controller* object which then modifies a *Model* object (or collection of objects) holding the main internal state of the system. The *View* object governs how information is presented in the GUI. It gets updated from the *Model* object.

3.6 Further requirements and design constraints

1. Ignore any step(s) at the start of the follow tour use case concerning collecting payment for a tour.
2. Do not explicitly model time or include a clock object.
3. For the tour browsing mode imagine there are two sub-modes, one where a list of all the available tours is displayed, each being described by just a short title and an identifier, and another where the full annotation of a particular selected can be viewed.

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 6th November**

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

Paul Jackson, 23rd October 2017.