Inf2C Software Engineering 2015-16

Coursework 2

Creating a software design for a city bike-hire scheme

1 Introduction

The aim of this coursework is to create a design, documented using UML class and sequence diagrams, for the software part of a new selve-serve bike-hire scheme for Edinburgh. This coursework builds on Coursework 1 on requirements capture. Please refer back to the Coursework 1 instructions for a description of the scheme. The follow-on Coursework 3 will deal with implementation and test.

2 Your task

Your job for this coursework is to create a design document for the software. The following subsections specify what should be included in your design document.

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 Design assumptions

Record here the main design assumptions you are making in your design. Many of these will concern your choice of classes and operations that abstract the input and output devices. See Section 3.2 for details on how to do this abstraction.

Many assumptions will rule out extension scenarios of use cases. When this is so, identify the relevant use case by number: see Section 2.7 below.

If you get up to 15-20 design assumptions, stop. There is no need to list more.

2.3 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.

2.3.1 UML class model

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 should include multiplicities. The class model must include any interface classes you introduce.

Some classes will model utility concepts such as time, location and bank card authorisation code. Because of their simplicity, they will likely appear as the argument or return types of operations and as attribute types, but not as classes with association links 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 @16 on the course discussion forum for further advice on *draw.io* and other tools.

2.3.2 High-level description

The high-level description should include justification for the design you chose, including specific rationale for the decisions made in the design. For example, you could describe why your design may be better than another or why you chose to implement a specific design pattern in one of your subsystems.

2.3.3 Further class documentation

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.

For each operation, indicate by number which use-cases motivate the inclusion of the operation: part of your report will be a summary of use cases. See Section 2.7

For classes modeling I/O (Input/Output) devices, make clear that they are doing so. For example, a heading for a Button class might be **Button class (input device)**. When relevant, also include in the heading for a class description if it is a utility class and if it is associated with a particular system component such as the hub system or a docking station. Or you might find it sensible to group related classes together under particular sub-headings.

2.4 Dynamic models (UML sequence diagrams)

This section should contain a few scenarios captured as sequence diagrams illustrating major use-cases of the system. Specifically, you should construct UML sequence diagrams for at least the following use cases:

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

- Register user.
- Hire bike.
- Return bike.
- Generate activity summary for user.
- Charge user.
- Show nearby docking stations with free points.

You may include a couple more diagrams, if you wish.

The *draw.io* tool is also an easy tool to use for drawing sequence diagrams.

2.5 Extension scenarios

There are a large number of possible extension scenarios. Identify here 4 or 5 you consider most important and that you have ensured your design can accommodate. For each describe in at most a couple of sentences how the design does so. These extension scenarios need not be the same as those you identified in Coursework 1.

2.6 Changes in requirements and use-cases

Describe briefly how your understanding of the system requirements and use cases has changed because of going through the design process and because of any feedback provided by markers on your Coursework 1 submission.

You may find it useful to document for your own purposes your revised use cases, but there is no need to submit filled-in templates for these use cases or a detailed list of any revised requirements.

2.7 Appendix summarising use-cases

End your report with an appendix that summarises the use cases you considered when creating your design. For each use case, just give its name and perhaps a sentence or two describing what it involves, if this is not obvious from its name. Ensure each use case is numbered for ease of reference (see the instructions above in Section 2.3).

There is no need for this list to be restricted to the use cases you included in Coursework 1 in your use-case diagram or your use-case descriptions.

3 Further information

3.1 Design architecture

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. Classes for input devices should have operations corresponding to input events: a Button class should have a press() operation. An actor

pressing a button is modeled by the invocation of this press() operation on a Button object representing the button. Classes for output devices should have operations for output events: a Light class might have an turnOn() operation. The software system turns on a particular light by invoking the turnOn() operation on the object representing the light.

Create a system design with a single thread of control. Do *not* try to make it concurrent. System activity then consists of bursts of messages passed between objects, each triggered initially by some input event, some actor 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. 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. There is the possibility that non-trigger input events might occur in the middle of bursts, because of the way in which input/output behaviour is made abstract. See below.

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 input from actors.

This single-threaded assumption is unrealistic, but it keeps the design and implementation much simpler and manageable in the time available.

3.2 Level of abstraction of inputs and outputs

In a real-world implementation there would be a tremendous amount of detail in 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.

This abstraction consists of abstracting sequences of input and/or output events that might always occur together at some input/output device into single events or pairs of events. Input events in the middle of bursts (what were called *non-trigger* input events above) will be initiated by system activity rather than by actors. Sometimes input events might be ignored altogether.

Here are some examples of abstractions you are recommended to make which illustrate these general abstraction ideas.

1. Flashing a light consists of two output events, one to turn the light on, one to turn it off a short time later. These two events can be abstracted by a single output event modeled by providing a flash() operation for the Light class.

2. At a key reader, the input events of *detect key insertion*, *read key data*, *detect key removal* could all be merged into a single abstract *read key* input event.

   This abstraction makes an assumption about the behaviour of the bike hirer inserting the key, that they don't just leave their key in the slot and walk away. In general, many abstractions will make assumptions about the behaviour of actors, and your report should document a representative sample of these assumptions. See Section 2.2 above for where to include this documenation.

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

4. Consider the touch-screen of a docking station terminal as a 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 abstract task-level input events such as *start registration* and *enter personal details* that in a real-world implementation will involve many input and output events. Even better, as the above two events always occur together, fuse them together so the *start registration* operation has the personal details as its return value.

   An abstract output event that a touch-screen should support is *display docking point availability at nearby docking stations*. The operation modeling this output event should take as arguments the data to display, but there should be no concern about how this data might be presented on the screen, for example how it might overlay a map of the area surrounding the docking station.

5. Similarly, the keyboard, mouse and monitor that a hub operator uses to interact with the hub system should be a single device with operations at the abstraction level of the tasks the operator is performing.

6. Group together the bank card reader and PIN entry pad as a single *card and* PIN *reader* device modeled by a single class. Assume this device also has a small screen for displaying prompts such as *insert card*, *enter* PIN and *remove card*, and information messages such as *card valid*.

   One modeling option is to give the class distinct operations for an abstract output event *prompt for card and* PIN and abstract input event *read card and* PIN. With such operations the output event ends one burst of system activity and the input event triggers a new one.

   However, it is tedious to design and code when there are such breaks in activity at a single object. An alternative recommended approach is instead to provide a single operation combining these events, say *get card and* PIN *info*, that is invoked by the system and that combines prompt and information output events, and card and PIN input events.

   For convenience, assume the card and PIN reader device also takes care of checking card validity, maybe via a wireless link to some banking server. Do *not* explicitly model this validity checking. Assume the *get card and* PIN *info* operation returns either an *authorisation code* or some token indicating a problem, e.g. that the PIN was wrong or authorisation failed.

7. 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 the authorisation code for some card and the amount to charge.

3.3 Interface between hub system and docking stations

In the first draft of your design, you need not consider explicitly how the hub system and docking stations connect.

Some classes will naturally be associated with the hub system and some with docking stations, and there then will be one or more associations between the classes in one group and the classes in the other. Assume for sake of argument that there is only one association between a hub class and a docking station class, and the classes involved are called Hub and DStation respectively.

It is desirable that the interface that the hub system or docking station present to each other is explicitly represented in the design. To this end, introduce UML interface classes HubInterface and DStationInterface. The idea is that a DStation object only ever invokes operations on the Hub object by invoking operations specified in HubInterface. Likewise, the Hub object only ever invokes operations on a DStation object that are specified in the DStationInterface.

Figure 1 shows the association for the first draft of this example. Figure 2 show how this association can expressed as two directed associations when the interfaces are included too.

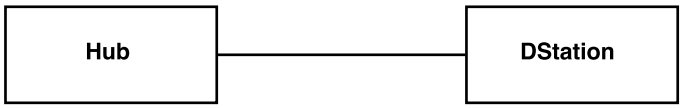

Figure 1: Hub-DStation association

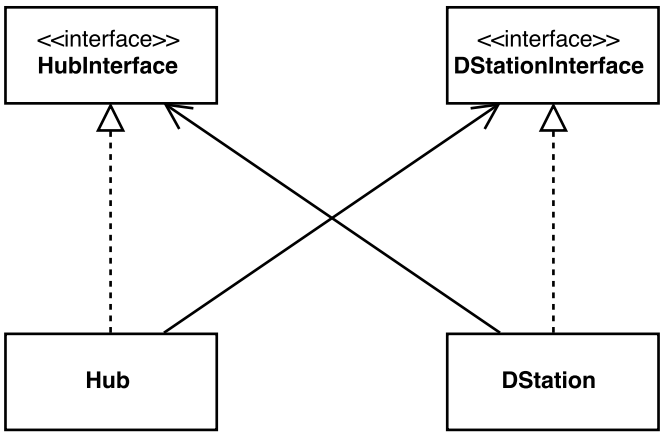

Figure 2: Hub-DStation association with interfaces

In general, your initial design might have multiple associations between multiple classes in each component. In this case, an interface is needed for each of the involved classes.

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 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

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place your PDF requirements document and your team.txt file in a directory called `Application`. Then ensure the current directory is set to the directory containing this `Application` directory and submit your work using the command:

```
submit inf2c-se 2 Application
```

This coursework is due

Noon on **Tuesday 10th November**

The coursework is worth 30% of the total coursework.

Paul Jackson, 28th October 2015.