Inf2C Software Engineering 2015-16

Coursework 3

Creating an abstract implementation of a city bike-hire scheme

1 Introduction

The aim of this coursework is to implement and test an abstract version of software to support a new selve-serve bike-hire scheme for Edinburgh. 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 Framework description

2.1 Overview

As a starting point, you are provided with a skeleton implementation and a test framework. The skeleton implementation includes code for most of the input/output device classes. The test framework enables test code to distribute input events to input device objects and collect and check output events generated by output device objects. See Figure 1 for a class diagram sketch of the framework connected to an example system.

When the framework is configured, there is only one SystemTest object, one EventDistributor object, and one EventCollector object, but there can be many input-capable device objects such as KeyReader objects, and many output-capable device objects such as OKLight objects.

A test might run as follows:

1. The SystemTest object sends an *insertKey()* input event to the EventDistributor object by passing the event as an argument to the sendEvent() method of the EventDistributor class.

2. Each event includes not only a message but also the name of some particular input-capable interface device it is associated with. The EventDistributor object passes the event on to the KeyReader object named in the event.

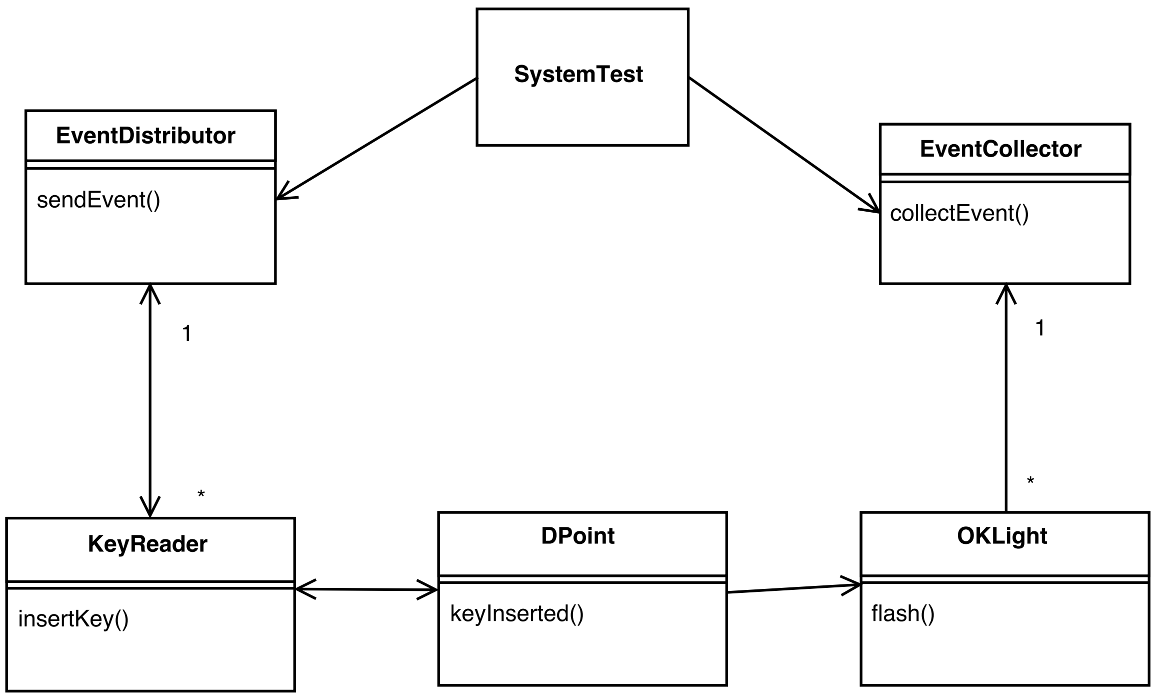

Figure 1: Framework Overview
Unless otherwise stated, all associations are 1-1

3. The event is received at the named KeyReader object, resulting in the insertKey() method being invoked on the object.

4. The implementation of this method models how the input event is handled by the designed system. Here we imagine that it invokes a keyInserted() method in a DPoint (docking point) object which in turn invokes a flash() method on an OKLight object. The class OKLight is an example of a class for output-capable interface devices.

5. The flash() method invocation results in output event being created by the OKLight object and sent to the EventCollector object.

6. The EventCollector object stores the output event received and the thread of control returns back to the flash() method, so the system code can execute further.

7. When the invocation of sendEvent() on the EventDistributor object finally returns, the SystemTest code then checks whether the actual output events queued up and stored at the EventCollector object match with its expectations.

2.2 Triggering and non-triggering input events

In general, in the middle of an execution started by some *trigger* input event such as *insertKey()*, the designed system might want to obtain further input data. To support this, the EventDistributor code is slightly more complicated than described above. What happens is that a test first fills an input event queue in the EventDistributor object without any being sent on to input-capable devices. When all the input events for a test are loaded, the test signals to the EventDistributor to successively remove input events from this queue and send each on to the input-capable device it names.

When the designed-system code wishes to obtain further input data, it calls a method in the relevant input-capable device which fetches the next input event stored in the EventDistributor's input event queue. We call such input events *non-triggering* input events.

2.3 Java interfaces and abstract classes

The class diagram in Figure 1 does not show relevant parent classes and implemented interfaces. A more accurate depiction of the same example is shown in Figure 2.

Here the abstract classes (AbstractInputDevice, AbstractIODevice, AbstractOutputDevice) capture common implementation features of input and output device classes. The interfaces InputCapableDevice and KeyInsertionObserver are both parts of realisations of the Observer design pattern, enabling the EventDistributor and the KeyReader classes to be designed without prior knowledge of the specific classes they pass information on to.

2.4 Events

The Event class in the provided code is used for both input and output events. An event contains the following:

1. a *timestamp* for when the event occurred,

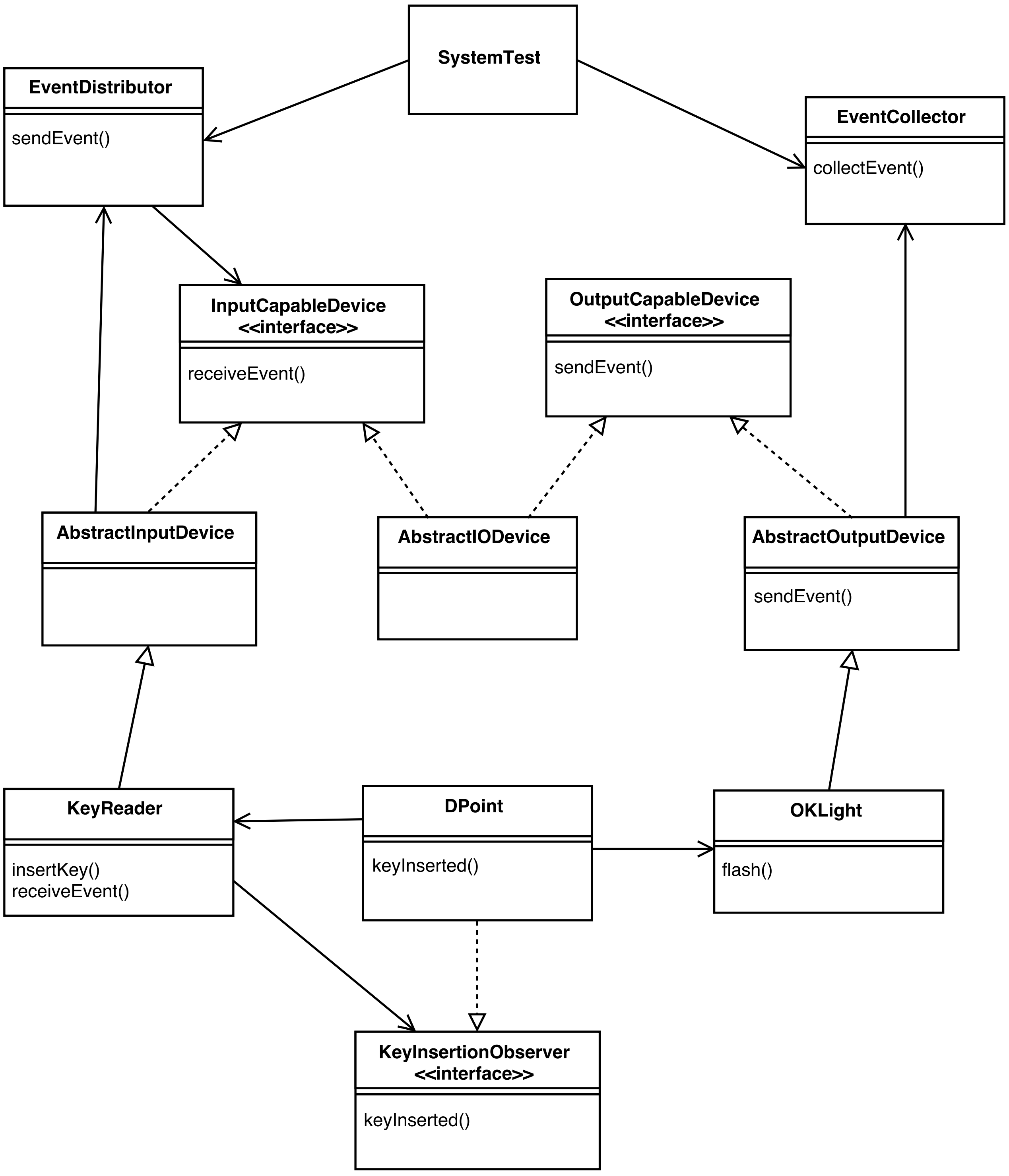

Figure 2: Framework Details

2. a *device identifier* consisting of a class name and an instance name which together uniquely identify an interface device associated with the event,
3. a *message* which has a name and 0 or more arguments. Each argument is a string.

See the Event.java file for further details.

3 Your tasks

There are several concurrent tasks you need to engage in. Your approach should be an incremental one, adding and testing features one by one as much as possible, always maintaining a system that passes all current tests.

Tasks are categorised as *target* or *supplementary.* Most marks will be awarded for reasonable realisations of the target tasks. You are expected to complete all target tasks. Additional marks will be awarded for realisation of one of the two supplementary tasks. These are described in Sections 3.2.3 and 3.1.

3.1 Construct code

The target task is to complete an implementation of the use cases AddDStation, AddBike, RemoveBike, RegisterUser, HireBike, ReturnBike, ViewUserActivity and ViewOccupancy, as described in Appendix B.1.

One supplementary task is to complete an implementation of the additional use cases RemoveBike, ReportFault, FindFreePoints, ChargeUser and ViewStats, as described in Appendix B.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`.

3.2 Create tests

The first two tasks here on system-level and unit testing are required target tasks. The third is a supplementary task.

3.2.1 System-level tests

Your are expected to use JUnit 4 to create system-level tests for each of the use cases you implement. The provided code gives examples of tests for the AddDStation, RegisterUser and ShowOccupancy use cases, as well as a basic test exercising docking point interface devices.

Also, add logging code to your code using logging support from the standard java.util.logging package. The provided code gives examples of the use of this package. The aim here is to have sufficient logging that one can view the flow of control around your implementation objects and methods resulting from running each test. This logging output enables you and the markers to understand how use cases execute.

3.2.2 Unit tests

The provided Event class has methods equals() and listEquals() with slightly non-trivial implementations. See the comments in the Event.java file, particularly at the start, for further details.

Your target is to create a set of tests for these methods that checks these methods are working as expected. Ensure your tests are commented so it is clear what each is checking. An outline test file Event.java is provided to get you started.

3.2.3 Component-level test

A supplementary task is to create a component-level test for a docking station. To do this, first ensure you've characterised the hub and docking station interfaces as requested in Coursework 2. Then create an implementation where you remove all the code for the hub system and associated devices, and replace the Hub object (or whatever objects you have on the hub side of your interface), with a special *stub* object or *mock* object which in turn connects to a special input/output device. The idea is that, through a combination of the stub functionality and appropriate input and output events to the special I/O device, one mimics the behaviour of the hub sufficiently to be able to test the docking station in isolation.

3.3 Keep a project log

This 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?

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

Each member of a group can keep and submit their own independent project log. Alternatively, a group can submit a joint log.

3.4 Working practices

This coursework is a small-scale opportunity to try out ideas that have been mentioned in the course. For example, you could try writing tests before code and using the tests to drive your design work. You could also try pair programming.

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 Practical details

4.1 Getting started

Download the framework and demonstration system code from:

`http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Coursework/2015/bikescheme.zip`

This can be placed anywhere, a temporary directory for example.

Unzip the file. This creates a top-level directory BikeScheme containing sub-directories src, bin, lib and data.

The code can be compiled and run from the command line. First cd to the main project directory called BikeScheme. Then run:

```
$ export CLASSPATH=bin:lib/junit.jar:lib/org.hamcrest.core_1.3.0.v201303031735.jar
$ javac -sourcepath src -d bin src/bikescheme/*.java
$ java bikescheme.AllTests
```

where $ is the shell prompt.

Instructions on how to import this code into Eclipse wil be provided shortly.

4.2 Provided code

The provided code includes a full implementation of the event-passing testing framework, a demonstration system implementation, and system-level tests that exercise the demonstration implementation. The main classes of the demonstration design are sketched in the Figure 3 class diagram.

This demonstration design exercises many of the provided device classes and contains a dummy implementation of the real system: there are Hub, DStation (docking station) and DPoint (docking point) classes, which have just enough functionality that one can run tests for the use cases AddDStation, RegUser and HireBike that have much of the expected input/output behaviour.

You are strongly recommended to study carefully this demonstration design and the tests provided for it.

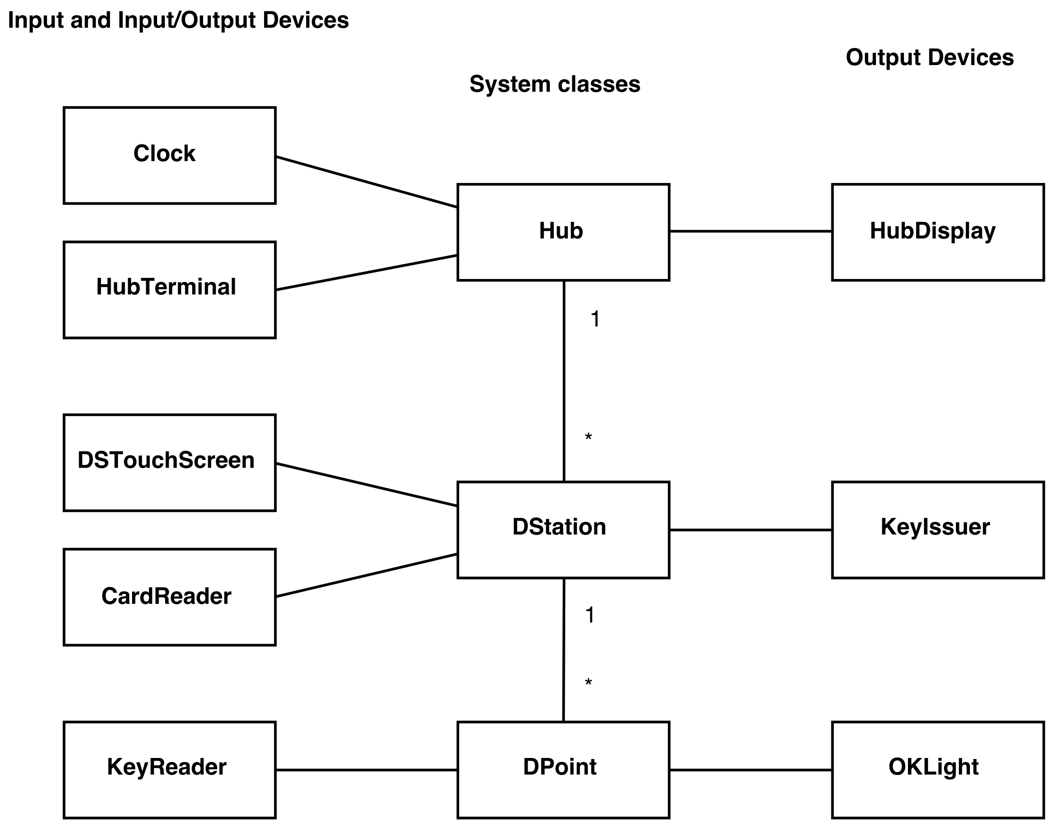

Figure 3: Demonstration design

4.3 Modes of testing

There are three test files.

- SystemTest.java defines some basic JUnit 4 system-level tests of the provided system. You should put your own system tests in this file.
- EventTest.java is for JUnit unit tests of the Event class.
- AllTests.java defines a JUnit test suite grouping together the tests in the SystemTest and EventTest classes.

In Eclipse you can run individual JUnit tests in either SystemTest.java or EventTest.java, all tests in one of these files, or all tests in both files using AllTests.java.

There are also several options for running the AllTests class as a Java application, as it defines a static main method.

1. If it is run as above, where it is passed no arguments, all tests in the test suite are run.
2. If it is run with one argument, that argument specifies files for input data and expected output data. Demonstration data files are provided in the `data` subdirectory. One can run

   ```
   java bikescheme.AllTests data/bike
   ```

   To read in the input event file `data/bike.in.txt` and compare the results with the expected output events in file `data/bike.expected.txt`.
3. If it is run with two arguments, the first specifies the input event file as previously, and the second is ignored. With this option a test is run which reads in the input events and writes the output events to the standard output stream. As all logging output is written to the standard error stream, the output events are easily separated off. For example, one could run:

   ```
   $ java bikescheme.AllTests data/bike run 1> output.log 2> error.log
   ```

5 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 cw3 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!

As questions and answers build up on the forum, remember to check over the existing questions first: maybe your question has already been answered!

6 What to submit

6.1 Working code and tests

Your code must compile and run from a DICE command line. There should be no need for external libraries other than the JUnit libraries provided in the lib subdirectory.

Capture the output of running your JUnit tests using a command similar to:

```
$ java bikescheme.AllTests &> output.log
```

as described above in Section 4.3. Include output.log in your submission. This file should show the logging messages tracing the execution of your tests and should include at the end the test results summary.

Record exact instructions on how to compile and run your code in a section of your report (see below).

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**.

The report should make clear what you have implemented. Your report needs to explain your design and tests. You can make use of class diagrams, but the diagrams don't need to be exhaustive. Describe and justify your design decisions.

6.3 Project log

Keep this as say a Word or Open Office document, but convert it to PDF for submission. If each member of a pair is submitting their own log, then the pair member submitting the rest of the files should name their log first-project-log.pdf and submit it with the files. The other pair member should name their log file second-project-log.pdf and include it by itself in an Application directory which they independently submit as described below.

If a joint log is submitted or you are working solo, just name the file project-log.pdf.

Please follow these naming guidelines: they will save time when marking.

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
```

Place this in your Applications directory.

7 Marking Scheme

- Code and tests for target use cases (35%)
  - Independent testing by markers of your implementation of the target use cases (10
  - The quality of your system-level tests for the target use cases. Do they test well the desired functionality? Are they easy to read and understand? Does the log information look sensible? (10%)
  - The quality of your code. Is it easy to read and follow? Have you followed coding guidelines well? (10%)
  - Your unit tests of the `Event` class. (5%).
- Report on the design realising the target use cases and summarising the tests. (25%)
- Project log (20%)
- Completion of one of the supplementary tasks. This will be assessed according to similar criteria as above, but a single mark will be given. (20%).

8 How to submit

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place all your files in a directory named `Application` or in subdirectories of this directory (for your Java files). Then ensure your current directory is set to the directory containing this `Application` directory and submit your work using the command:

```
submit inf2c-se 3 Application
```

This coursework is due

16:00 on **Tuesday 1st December**

The coursework is worth 50% of the total coursework.

A Bike scheme interface devices

A reasonably-complete set of interface device classes is shown in Table 1. This set should be sufficient to realise both the target and supplementary use cases described in Appendix B.

Name	Kind	Component(s)	Status	Description
Clock	I	All	I	Trigger for timed tasks
HubTerminal	IO	Hub	I	Keyboard, mouse, monitor
HubDisplay	O	Hub	I	Large wall display
BankServer	O	Hub		For charging hires to user's bank cards
DSTouchScreen	IO	DST	I	Including touch sensor and display itself
CardReader	IO	DST	I	Bank card reader, mini-display & PIN pad
KeyIssuer	O	DST	I	Smart key dispenser
KeyReader	I	DST, DP	I	Smart key reader
BikeSensor	I	DP	I	Bike present & bike ID sensor
OKLight	O	DP	I	Green OK light
BikeLock	O	DP	I	
FaultButton	I	DP		
FaultLight	O	DP		Red light

Table 1: Interface device classes
Status: I=Implementation provided sufficient for target use cases *Components*: DST=Docking Station Terminal, DP=Docking Point

The KeyReader is listed as associated with a Docking Station Terminal as well as Docking Points. This was not explicitly stated in the Coursework 1 instructions. This association is to enable users to quickly authenticate at Docking Station terminals in order to access information about their recent trips, and get extensions on their current charging period when all points are full.

Each interface device supports one or more kinds of event messages. Details on these message names and arguments is provided in AppendixC. The provided message support should be sufficient for completing the implementation of the target use cases described in Appendix B.1.

To complete an implementation of the supplementary use cases as described in Appendix B.2, a few further devices need to be implemented as indicated in Table 1. Also some extra message kinds need to be added to the existing device implementations.

B Use cases to realise

B.1 Target use cases

Here is a list of target use cases your implementation should support, with an indication of the trigger actor and which component (Hub, Docking Station Terminal or Docking Point) handles the use case, and a summary of the expected exchanges between the external world and the system. These are mostly as suggested by the Coursework 1 and Coursework 2 descriptions.

- AddDStation. *Operator at Hub.* For coursework 3, this use case folds in adding docking points, so a separate use case is not needed.
- AddBike. *Service staff member at DP.*
- RegisterUser. *User at DST.* User enters personal info at DSTouchScreen, CardReader prompts for card and PIN, User enters card and PIN and CardReader validates, KeyIssuer issues key.
- HireBike. *User at DP.* User inserts key in DP KeyReader, DP unlocks bike and flashes OK light,
- ReturnBike. *User at DP.* BikeSensor senses bike inserted in dock, DP flashes OK light and locks bike.
- ViewUserActivity. *User at DST.* User requests view of activity at DSTouchScreen, DSTouchScreen prompts for key insertion into reader, KeyReader at DST reads key, DSTouchScreen presents a summary of journeys completed since prior midnight when key owner was last charged.
- ViewOccupancy. *Clock at Hub* and/or *User and Service staff member at DP.* Show on large display at Hub how fully occupied each station is, highlighting those with high or low occupancy.

For the target task you need only implement support for the main success scenario of the use case. There is no need to consider extension scenarios. These target use cases and the associated interface devices are fairly tightly specified, to ensure that markers can run tests to verify the functionality of your implementation.

B.2 Supplementary use cases

The supplementary task is to realise the main success scenarios of the following use cases.

- RemoveBike. *Service Staff Member at DP.*
- ReportFault. *User at DP..* If user presses fault button within 2 minutes[1] of returning bike, the bike is registered as faulty and the fault light is turned on.
- FindFreePoints. *User at DST.* User is shown on touch screen display the locations of nearby docking stations with free points.
- ChargeUser. *Clock at Hub.* The clock triggers this event every 24 hours at midnight for each user who has incurred charges.
- ViewStats. *Operator at Hub.* Many possibilities here. For each day over some period, could show number of journeys, number of users, total distance travelled, average journey time. Also could show locations of faulty bikes.

These aspects will not be independently tested by markers, so you have more freedom in how you realise them.

[1]Not 10 seconds as in Coursework 1. 2 minutes is needed because finest time resolution of events is 1 minute.

C Message processing by implemented I/O devices

This appendix documents the input and output message processing for the implemented input/output devices. This processing should be sufficient to support all the target use cases. We use the abbreviations TI for *triggering input*, NTI for *non-triggering input* and O for *output*. The format of the processing information for each of the three kinds of messages is as follows.

```
TI   Triggering input message name and arguments (UML syntax, Java types)
     Declaration for handling method in device class
     Observer interface name
        Declaration for observing method in interface

     Declaration for method in device class that fetches input event
NTI  Input message name and arguments (UML syntax, Java types)

     Declaration for method in device class that generates output
O    Output message name and arguments (UML syntax, Java types)
```

The processing information for the implemented devices is as follows:

```
Clock
  TI   tick() : void
       void tick()
       TimedNotificationObserver
          void processTimedNotification()

HubTerminal
  TI   addDStation(name : String, eastPos : int, northPos : int, numPoints : int)
       addDStation(String name, int eastPos, int northPos, int numPoints)
       AddDStationObserver
          void addDStation(String name, int eastPos, int northPos, int numPoints)

HubDisplay
       void showOccupancy(List <String> occupancyData)     See below for list format
  O    viewOccupancy( occupancyData : List<String>)

DSTouchScreen
  TI   startReg(personalInfo : String)
       void startReg(String personalInfo)
       StartRegObserver
          void startRegReceived(String personalInfo)

  TI   viewActivity()
       void viewActivity()
       ViewActivityObserver
```

```
             void viewActivityReceived()

          void showPrompt(String prompt)  e.g. "insert key"
   O      viewPrompt(prompt : String)

          void showUserActivity(List <String> activityData)     See below for list format
   O      viewUserActivity( activityData : List<String>)

CardReader
          void requestCard()
   O      enterCardAndPin()

          String checkCard()
   NTI    checkCard(authCode : String)

KeyIssuer
          String issueKey()  returns new unique Id
   O      keyIssued(keyId : String)

KeyReader
   TI     insertKey(keyId : String)
          void insertKey(String keyId)
          KeyInsertionObserver
             void keyInserted(String keyId)

          String waitForKeyInsertion()
   NTI    keyInsertion(keyId : String)

BikeSensor
   TI     dockBike(bikeId : String)
          void dockBike(String bikeId)
          BikeDockingObserver
             void bikeDocked(String bikeId)

OKLight
          void flash()
   O      flashed()

BikeLock
          void lock()
   O      locked()

          void unlock()
   O      unlocked()
```

Formats for some outputs consisting of lists of tuples:

- HubDisplay class, showOccupancy method. List<String> argument is expected to be a series of 6 tuples with fields for

  - Docking station name
  - Position in metres East (+) or West (-)
  - position in metres North (+) or South (-)
  - Status (HIGH, OK or LOW)
  - Number of docking points occupied
  - Total number of docking points

  When generating the message arguments, this method adds to the front of the list the elements "unordered-tuples","6" indicating the list structure and a header tuple "DSName","East","North","Status","#Occupied","#DPoints". Testing makes no assumptions about the order in which the tuples are presented.

- DSTouchScreen device, showUserActivity method. List <String> argument is expected to be a series of 4 tuples with fields for

  - Hire start date and time
  - Name of docking station where bike hired
  - Name of docking station where bike returned
  - Hire duration (in minutes)

  When generating the message arguments, this method adds to the front of the list the elements "ordered-tuples","4" indicating the list structure and a header tuple "HireTime","HireDS","ReturnDS","Duration (min)". Tuples are expected to be in ascending time order.

Paul Jackson, 17th November 2015.