Inf2C Software Engineering 2016-17

Coursework 3

Creating an abstract implementation of a restaurant order-management system

1 Introduction

The aim of this coursework is to implement and test an abstract version of software for managing orders in a restaurant. 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 all 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 a demonstration 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 PassButton objects, and many output-capable device objects such as PassLight objects.

A test might run as follows:

1. The SystemTest object sends an press() 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 PassButton object named in the event.

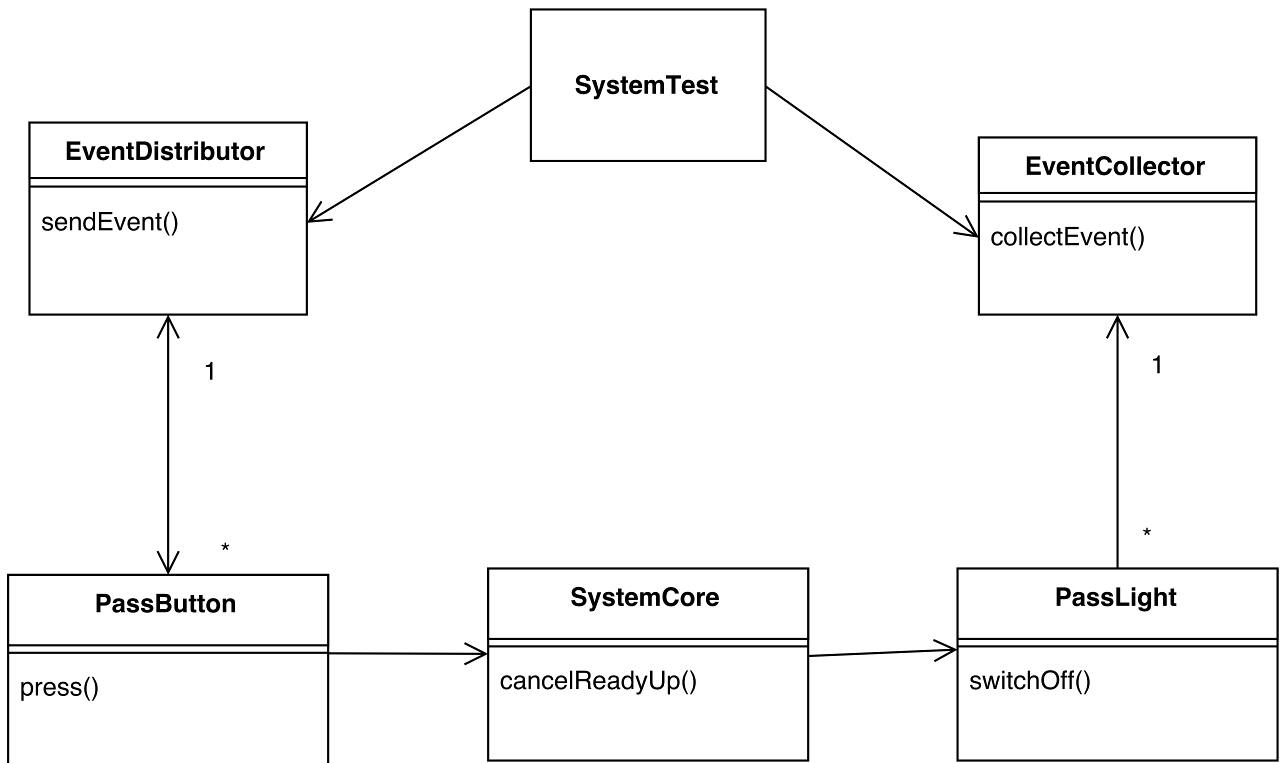

Figure 1: Framework Overview
Unless otherwise stated, all associations are 1-1

3. The event is received at the named PassButton object, resulting in the press() 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 cancelReadyUp() method in a SystemCore object which in turn invokes a switchOff() method on a PassLight object. The class PassLight is an example of a class for output-capable interface devices.

5. The switchOff() method invocation results in output event being created by the PassLight 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 switchOff() method, so the code associated with the system can execute further.

7. Further execution of system code might generate further output events. The EventCollector object stores all collected output events.

8. When the invocation of sendEvent() on the EventDistributor object finally returns, the SystemTest code then checks whether the actual output events collected match up with a sequence of expected output events that were previously stored in the SystemTest object.

2.2 Trigger and non-trigger input events

In general, in the middle of an execution started by some *trigger* input event such as press(), 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 core code of the system under test wishes to obtain further input data, it calls a method in the relevant input-capable device that fetches the next input event stored in the EventDistributor's input event queue. As noted in the Coursework 2 instructions, we call such input events *non-trigger* input events.

2.3 Abstract interface device classes

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

Here the abstract classes (AbstractDevice, AbstractInputDevice, AbstractIODevice, AbstractOutputDevice) capture common implementation features of input, output and input-output device classes. The class AbstractInputDevice is an observer class in a realisation of the Observer design pattern. It enables the EventDistributor class to be designed without prior knowledge of the specific sub-classes of AbstractInputDevice and AbstractIODevice it passes information on to.

Ideally AbstractIODevice would inherit code from both AbstractInputDevice and AbstractOutputDevice. However, Java does not allow muliple inheritance, so instead AbstractIODevice has a copy of code introduced in AbstractOutputDevice.

The operation receiveEvent() in AbstractInputDevice and subclasses is for receiving trigger input events from the distributor and dispatching them to operations for specific event kinds, e.g. the press() operation in the PassButton class. The operation fetchMatchingMessage() in AbstractInputDevice is for retrieving non-trigger input events from the distributor.

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,
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.

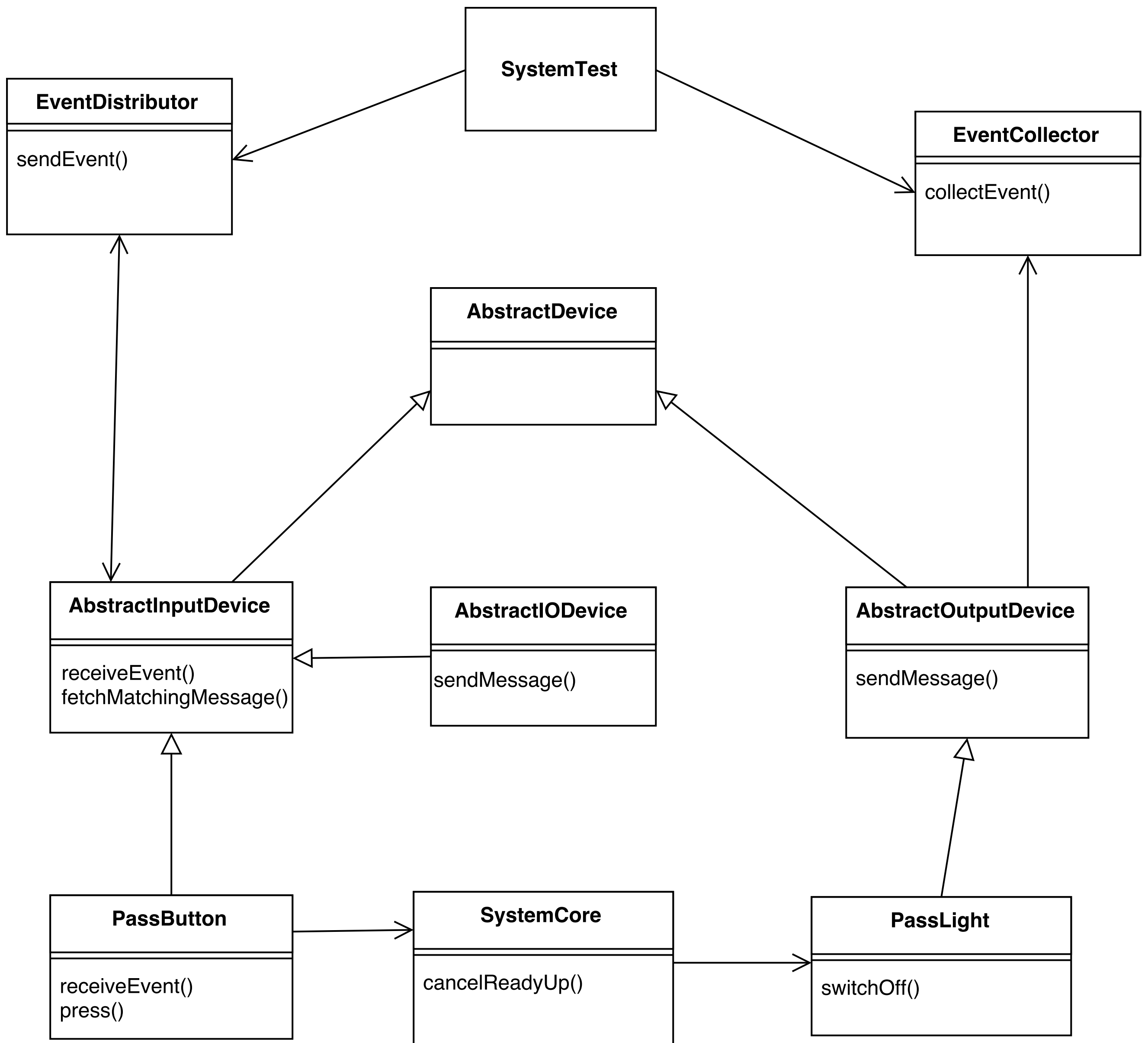

Figure 2: Framework Details

2.5 Coursework Phases

As a starting point, you are provided with code which includes a generic test framework and classes for all the interface devices you will need. You then need to implement code that realises functionality much like that which you explored in Courseworks 1 and 2. Almost all the code you add will be to new classes that you create. A small amount of code also needs to be added to the interface device classes you use.

The implementation goals are split into 3 phases:

Phase I This involves adding code for supporting the creation of menus in the office. The implementation should require writing no more than 100 lines of Java code (excluding comments and whitespace). Further code is needed for testing. The marking scheme allows for students to score up to 78% if they complete this phase and progress no further.

Phase II This is for adding features for supporting the customer interactions with the system at a restaurant table. The additional code for this phase is maybe 50% larger than the Phase I code. However, much of the new code required is similar to that needed for Phase I, as in both cases the features are concerned with building a data-structure for finite sequences (Lists in Java terminology). The marking scheme allows for students to score up to 93% for completing Phases I & II.

Phase III This is for adding features involving activity in the kitchen and at the pass. The amount of code required is comparable to that for Phase II.

The marking scheme allows for students to score up 100% for completing Phases I, II and III.

The marking scheme emphasises the value of working code, and full marks for a phase are only obtainable if code fully passes tests during marking.

It is strongly recommended that you complete well all tasks involved with a given phase, before progressing on to the next. The marking scheme is intended to reward students for doing this, rather than racing ahead on the coding tasks for later phases, and then only having time for mediocre work on other tasks.

It is up to each coursework group to decide how many phases they wish to tackle and how much time they wish to spend on this coursework. The hope is that virtually all students should be able to complete Phase I within the nominal 20 hour time budget each has for this coursework. If any coursework groups are finding they are struggling to achieve this, they are encouraged to contact the course organiser to discuss their situation.

See Section 7 for a full description of the marking scheme.

3 Your tasks

There are several concurrent tasks you need to engage in. These are described in the following sub-sections.

3.1 Construct code

Appendix B describes the use cases that have to be implemented for each of the phases. Appendix A summarises the interface device classes and Appendix C gives details on the input and output messages that each interface class handles.

As recommended in the Coursework 2 instructions and on the discussion forum, do introduce distinct classes for domain entities such as menus, tickets and the rack, do not put the data-structures for these inside the interface device classes. Also, consider putting responsibilities for coordinating use case activity and linking together system components into classes realising the Mediator design pattern, rather than including these responsibilities in the provided interface device classes. Whatever you decide, be sure to explain your deceisions in your report (see Section 6.2).

There is no requirement that you stick to the design you produced for Coursework 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`.

A common recommendation is that you never use tab characters for indentation and you restrict line lengths to 80 or 100 characters. You are strongly recommended to adopt both these recommendations because it is good practice and, particularly, in order to ensure maximum legibility of your code to the markers. You should use an indent step of 4 spaces as in the Sun guidelines rather than the 2 in the Google guidelines, for consistency with the supplied code.

Unfortunately the default in Eclipse is to use tab characters for indents. See the Coursework 3 web page for advice on how to set Eclipse to use spaces instead and also how to get Eclipse to display a right margin indicator line.

To show you know the basics of Javadoc, add class-level, method-level and field-level Javadoc comments to your Menu class.

3.2 Create unit tests

The provided code includes a class Money for monetary amounts. Monetary amounts are represented using arbitrary precision decimal numbers. Arithmetic on them is exact. However, string presentations of them are always rounded and trailing-zero-extended to 2 decimal places. For example, the amount 0.127 is presented as the string `"0.13"` and the amount 3.1 as `"3.10"`.

Complete the provided JUnit 4 test class MoneyTest, adding appropriate tests for the class constructor and each of the methods in the Money class. A single basic test for the constructor and for each the methods is adequate, though you should add a few more tests to check that the rounding is correctly done and that formatting of strings always adds trailing zeroes, as appropriate.

Ensure your tests are commented so it is clear what each is checking.

3.3 Create system-level tests

Your are expected to use JUnit 4 to create system-level tests that run scenarios of each of the use cases you implement.

The provided code includes an example system-level test on a demonstration system.

For this design, it doesn't make much sense to create each test for each use case alone. In nearly all cases, several use cases will be needed to set up some state, followed by one or more to observe the state.

Do not run all your tests together in one large single test. Where possible, do test distinct features and use cases in distinct cases. Also, when a test involves exercising some sequence of use cases and features, try to arrange that this test is some increment on some previous test. This way, if the test fails, but the previous test passes, you know immediately there is some problem with the increment.

You are encouraged to adopt a test first approach, where you write the test for some use case or feature first before writing the code itself.

Include all your tests as JUnit test methods in the SystemTest class. Feel free to add extra auxiliary methods that handle repeatedly needed sequences of input and output events; avoid cut and pasting the same sequences multiple times.

Your SystemTest.java file should serve as the primary documentation for your system-level tests. To this end, take care with how you structure your test methods and add appropriate comments.

In addition, in your report (see Section 4.7), add a table cross-referencing your tests and the use cases each exercises. If you have tests that exercise some features of a use case, but not others, add a remark about this in the appropriate box in the table, rather than simply putting an X. In practice, test documentation often cross-references tests to feature lists, like that you produced for Coursework 1. For simplicity, this is not required in this coursework.

3.4 Add assertion checking

Nearly all of the use cases have preconditions; assumptions they make about the state of the system or the input the system receives from actors. For example, the use case to remove a menu item from the menu, given an ID for that item, might reasonably assume that the ID provided as input matches the ID of some item in the current menu.

What might your code do if a precondition of a use case is violated?

- It might terminate normally, giving no indication that anything is wrong. The violation might be due to a bug in the system behaviour prior to the use case or in the code generating the test input. Possibly the violation may have no adverse effect, but possibly it might corrupt part of the system state. Either way, the problem might not be discovered until some time later.
- It might throw some random exception, e.g. a null pointer exception. This signifies immediately something is wrong, but it could be hard to diagnose the cause.

More generally, methods often have preconditions for their proper behaviour. Often if a use-case precondition is violated, some method during some execution of the use-case will also have its precondition violated.

A helpful practice is to add checks to your code that use-case preconditions and/or individual method preconditions are not violated. On detecting a violation, a check could issue a warning message or throw an exception.

Consider the use case for removing a menu item from the menu. Assume this use case has a precondition stating that the ID provided as input matches that of some item on the menu. Consider how this use case executes when an ID is provided that is not on the menu, and identify a method that could be given a precondition that also is violated in this case. It is likely that your Menu class will have such a method.

1. Add a precondition check to this method that makes use of a Java assert statement. Use the syntax

   `assert` *condition* : *message* ;

   for your check. This incorporates a *message* object, usually a Java String object, into the AssertionError object that is thrown if condition is false. This message object gives information on what caused the condition to be false.

2. Enable assertion checking in your Java runtime system. E.g. run the `java` command with the option `-ea`. If you are using Eclipse, you can add `-ea` as a VM (Virtual Machine) argument to test configurations accessible from Eclipse's *Run* menu.

3. Add a JUnit test in your `SystemTest` class that causes this assertion to fail. Make sure you comment this method appropriately, explaining what is going on.

4. Use the appropriate JUnit @Test annotation for this method in order to check the AssertionError exception is indeed thrown. Look up the needed annotation in the JUnit documentation at `http://junit.org`. Most JUnit 4 tutorials also will explain what is needed.

3.5 Add logging code

Add logging code to your code using logging support from the standard java.util.logging package. The provided code for the interface device classes gives examples of the use of this package. See the AbstractDevice class for how to set up a reference to the test framework Logger object. See the press() method in the PassButton class for one example of how to create a log message. Have a brief browse of documentation on the web concerning this package to get an idea of what is going on with logging.

Your aim is to add enough logging that both you and the markers can follow the flow of control around your implementation objects and methods when each of your tests is run.

For most methods it will be fine to add just a single logging line at its start. For some it might be useful to log too when control flow reaches intermediate points in the methods.

The logging messages automatically include the enclosing class and method names. It therefore can be sufficient for the logging text to be very brief, say just the word *Entry*, indicating the logging message is generated at the entry-point of a method, or a string identifying the particular object the method is called on. All the provided interface device classes already have logging calls that do just this.

The provided system test framework also writes log messages listing input events, expected output events and actual output events. If you have a test that fails, it is worth checking this information to find out why the test failed.

You are required to submit a file of the logging output from running your tests. This serves as documentation of the dynamic behaviour of your system. You are not otherwise required to submit any UML sequence diagrams or text descriptions of behaviour. To aid in making this legible, be sure to generate highly-visible log messages at the start of each of your tests. See the provided example test for an example of this.

3.6 Keep a project log

Each member of a group is expected to keep and submit their own independent project log. This is the only task that should be done independently. All other tasks should be completed jointly.

This project 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?

Be sure to consider including remarks on working practices you experiment with. See Section 4.1.

Do not include details of design and implementation decisions in your log. Instead, refer to your report for this information. However, note that generally it is appropriate to include such information in project logs.

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

Mention in your project log the overall time you have spent on this coursework and give some idea of how this broke down into different activities.

3.7 Write a report

This report is expected to be significantly shorter than that you produced for Coursework 2. A title page plus 2 or 3 further pages is fine. The report should have the following sections:

Work completed Make clear which phases you have attempted, how complete the code is for those phases, and the extent to which you believe that code is working properly. A couple of paragraphs is fine.

UML Class diagram Include the interface-device classes you use and the new classes you add. Theres no need to represent any of the other provided classes. Do not try to exhaustively list all the methods and fields of the classes. Instead, only list a few of the most important ones. As is commonly the convention, do not show fields that implement associations drawn in the diagram. Do show the navigability of associations and do include multiplicities on their ends.

High-level design description This could be much the same as what you produced for Coursework 2. However, only cover the parts you have actually implemented. Be sure to include discussion of your design decisions. Include remarks about how your design might have changed since Coursework 2.

There is no need to provide further documentation on any of your classes, as was requested in Coursework 2.

Implementation decisions Discuss the main implementation decisions you have made. For example, state the kind of Java collection class you have used to implement each of the one-to-many associations in your design, and explain why.

Tests The report need only include a table cross-referencing your tests and the use cases they exercise. See Section 3.3. Otherwise, the markers should find all your test documentation in your provided SystemTest.java and MoneyTest.java files.

4 Practical details

4.1 Working practices

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

You are strongly encouraged to take an incremental approach, adding and testing features one by one as much as possible, always maintaining a system that passes all current tests.

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.

4.2 Getting started

Download the test framework and demonstration system zip file rom-demo.zip from the Coursework 3 web page:

`http://www.inf.ed.ac.uk/teaching/courses/inf2c-se/Coursework/2016/cw3.html`

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

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

The code can be compiled and run from the command line. First `cd` to the `ROMSDemo` directory. Then run:

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

Instructions on how to import this code into Eclipse are available from the Coursework 3 web page.

4.3 Provided code

The provided code includes the following.

- A full implementation of the event-passing test framework introduced in Section 2.
- Virtually complete implementations of all the device interface classes needed for all three implementation phases.
- A demonstration system implementation which is put together in the method setupSystem() in the SystemTest.java file. This system is essentially that shown in Figures 1 and 2.
- A system-level test that exercises the demonstration implementation. This is defined by JUnit test method cancelReadyUpLight() in SystemTest.java.

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

Note that the demonstration design creates interface device objects for all three phases. You should not need to create any further device objects. If you are working on Phase I or Phase II, you will have unused interface device objects. This is fine. Just ignore them. Do **not** alter the instance names assigned by the demonstration code to these interface device objects. Auto-marking tests will assume your design uses these names and will fail if you change any names. Also note that this code introduces IDs for each table which, if you get to Phase III, should be included on the tickets printed at the pass printer.

4.4 Modes of testing

There are three test files.

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

In Eclipse you can run individual JUnit tests in either SystemTest.java or MoneyTest.java, all tests in one of these files, or all tests in both files using the JUnit test suite defined in AllTests.java.

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

1. If it is run as above, where it is passed no arguments, it calls a JUnit test runner and 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. Example data files are provided in the `data` subdirectory. Run

   ```
   java roms.AllTests data/cancelReadyUpLight
   ```

   to read in the input event file `data/cancelReadyUpLight.in.txt` and compare the results with the expected output event in file `data/cancelReadyUpLight.expected.txt`. This is the same demonstration system test as is defined in SystemTest.java. This `data` directory also includes some basic tests for each of the implementation phases.

   - For Phase I: `addShowMenu.*`
   - For Phase II: `addShowTicket.*` and `addTicketPayBill.*`.
   - For Phase III: `addTicketsShowRack.*`.

   You are strongly recommended to check that your implementation passes the relevant tests.

4.5 Modifying provided classes

The following sections describe the provided files you can modify. Do *not* modify any other provided classes and do *not* change the package name the provided code uses.

4.5.1 Device interface classes

All input-capable device classes have a method for each kind of trigger input event they can receive, which gets called on event reception. For example, the PassButton class has a press() method.

You need to customise each of these methods to determine what behaviour your system should exhibit for each of these input event kinds. For example, they might send call messages to other objects in your implementation, they might call methods in the same class for generating output events, or both. For communication with other objects you need to add associations from these device classes to the message destination classes. You also need to add method calls which generate the call messages to objects in these destination classes. All such customisation of the device classes should be done within the areas indicated by comments in these classes. You should not have to modify any code outside of these areas. If you feel the need to, please first discuss what you wish to do with the course organiser.

4.5.2 Domain entity classes

Skeleton Menu, Ticket and Rack classes are provided which include dummy implementations of methods required by output-capable interface device classes. You have to provide proper implementations of these methods and can add whatever fields and other methods you wish.

Comments in these files indicate the regions in which you should place your code. Again, you should not have to modify any code outside of these areas. If you feel the need to, please first discuss what you wish to do with the course organiser.

4.5.3 Test classes

You should implement tests in the provided SystemTest and MoneyTest classes, and also include code for building your system in the SystemTest class.

4.6 Singleton classes

Several of your classes might be singleton classes; there might be only one object of its type created. Do *not* do anything special to realise the Singleton design pattern. For example, do *not* use static fields to hold the singleton's data or a static field to refer to a single instance of the class. Just create one instance of the class when setting up your system.

A special case is the Clock class. See the next section.

4.7 The Clock class

A single object of the provided class Clock holds the current *test-time*, a time determined by the time-stamp of the most recent event sent into your system. The method getDateAndTime() can be run on this object to retrieve a java.util.Date object holding this time.

The Clock class implements the Singleton design pattern by using a static field to hold the Clock instance. A static method initialiseInstance() creates a new instance and the static method Clock.getInstance() retrieves this instance. This approach is taken for convenience. It allows any method in any class to retrieve the current time without the method or the object it is called on having to have an explicit reference to the clock.

See the Clock file, in particular the comment at the class start, for more about this class.

5 Good Scholarly Practice

Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that, in particular, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work in some web repository, then you must ensure that access is restricted to only members of your coursework group. Be aware that not all free repository hosting services support private repositories, repositories with access restrictions. There are some more details about this on the Coursework web page.

6 What to submit

6.1 Working code, tests and an output log

Your code must compile and run from a DICE command line, as described above in Section 4.2. This is essential for the auto-marker. Please be aware that the default ECJ compiler used in Eclipse is a *different* compiler from the command line `javac` compiler. ECJ will compile and run code that `javac` will not compile.

Auto-marking will use Java 1.8.

Do not use any libraries outside of the standard Java 1.8 distribution other than the JUnit libraries provided in the lib subdirectory.

Create a zip file src.zip of your code and tests by setting your current directory to the main project directory containing the src, bin, lib and data directories, and running the command

```
zip -r src.zip src
```

Capture the output of running your JUnit tests using the command

```
$ java -ea roms.AllTests &> output.log
```

as described above in Section 4.4. This output.log file is a required part of your submission.

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**.

6.3 Project log

Keep this as say a Word or Open Office document, but convert it to PDF for submission. Each member of a group should name their project log project-log.pdf and should submit it separately. See the submission instructions below.

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
```

7 Marking Scheme

1. **Code quality**: Use of coding standards, ease with which code can be read and understood. Use of Javadoc for Menu class. (10%)

2. **Report**: This mark is based on the quality of the design and implementation information. The requested test table is assessed as part of the next mark. (35%)

3. **System test quality**: How complete are your tests? How easy is it to read and understand your tests? Did you include a test of the assert statement, as requested? (25%)

4. **Code output log**: Is it easy to follow the execution of your use cases by reading the log along with your class diagram? (5%)

5. **Code correctness and completeness**: This will be assessed by auto-marking, followed by a review of the auto-marking results. Auto-marking will be more thorough than the provided tests for each of the phases. It will only test use-case features described in this or previous Coursework handouts.

   Auto-marking will determine a scaling factor that will be applied to the previous 4 items. For example, the factor will be:

   - 0.3 if no tests pass,
   - 0.7 if all Phase I tests pass,
   - 0.9 if all Phase I & II tests pass,
   - 1.0 if all tests pass.

6. **Project log**: (20%)

7. **Quality of unit tests**. These are the tests of the Money class you insert in the MoneyTest class. (5%)

Given the auto-marking-based scaling scheme, the maximum possible coursework mark is:

- 48% if no tests pass,
- 78% if all Phase I tests pass,
- 93% if all Phase I & II tests pass,
- 100% if all tests pass.

8 How to submit

Each member of a group has to submit some files.

Ensure you are logged onto a DICE computer and are at a shell prompt in a terminal window. Place all the requested files in the same directory and `cd` to this directory.

Only one group member should submit the code, output log and report. That group member should run the command

```
submit inf2c-se 3 src.zip output.log report.pdf project-log.pdf team.txt
```

If the group has two members, the second group member should run the command

```
submit inf2c-se 3 project-log.pdf team.txt
```

This coursework is due

16:00 on **Tuesday 29th November**

The coursework is worth 50% of the total coursework mark and 20% of the overall course mark.

A System interface devices

A complete set of interface device classes is shown in Table 1.

Phase	Name	Kind	Description
I	OfficeKVM	IO	Keyboard, video display and mouse for office computer
II	TableDisplay	IO	Touch-screen display for a table
	CardReader	IO	Contactless card reader
	BankClient	IO	Internet connection for card payment processing
	ReceiptPrinter	O	receipt printer
III	Clock	I	Trigger for timed tasks
	KitchenDisplay	IO	Large touch-screen display in kitchen
	TicketPrinter	O	Ticket printer at the pass
	PassLight	O	Order ready-up light at the pass
	PassButton	I	Press button for cancelling ready-up light

Table 1: Interface device classes

B Use cases to realise

The use cases listed here and their numbering are the same as in the Coursework 2 handout. Each description lists a use-case name, the primary actor triggering the use case, the message kinds for the input and output events involved, and sometimes adds some extra remarks. Only main success scenario or scenarios need to be realised.

B.1 Phase I

In the office. All messages here only involve the OfficeKVM object.

- 4(a) ShowOfficeMenu. (*HeadChef*). showMenu (I), viewMenu (O).
- 4(b) AddToMenu. (*HeadChef*). addToMenu (I). This input event specifies a *menu ID* for each menu item. Several kinds of output event use menu IDs to name menu items and order ticket items.
- 4(c) RemoveFromMenu. (*HeadChef*) removeFromMenu (I). A menu ID is used to indicate which item to remove.

B.2 Phase II

At a table. Unless otherwise stated, all use cases here only involve a TableDisplay object.

- 1(a) StartOrder. (*Customer*) Create new empty order ticket.
- 1(b) ShowTableMenu. (*Customer*). showMenu (I), viewMenu (O). Display menu items along with IDs.
- 1(c) ShowTicket. (*Customer*). showTicket (I), viewTicket (O). Display order items along with a menu ID and a count for each.

- 1(d) AddToTicket. (*Customer*). addMenuItem (I). Add one count of a menu item to the ticket. A menu ID is used to indicate what to add. There are two MSSs here, depending on whether the item is already named on the ticket.
- 1(e) RemoveFromTicket. (*Customer*). removeMenuItem (I). Take away one count of a menu item from a ticket. A menu ID is used to indicate which item to take away.
- 1(f) SubmitOrder. (*Customer*). submitOrder (I). Tickets should be assigned a *Ticket number* on submission, starting from 1. These ticket numbers then are used in the ticket rack display to name tickets.
- 1(g) PayBill. (*Customer*). payBill (I), approveBill (O), acceptCardDetails (I), makePayment (O), acceptAuthorisationCode (I), TakeReceipt (O). This use case also involves a CardReader object, the BankClient object and a ReceiptPrinter object.

B.3 Phase III

In the kitchen and at the kitchen pass.

- 2(a) ShowRack. (*Clock*). tick (I), viewRack (O). Involves the Clock and KitchenDisplay objects.
- 2(b) IndicateItemReady. (*Chef*). itemReady (I), takeTicket (O), viewSwitchedOn (O). If item is first of an order, print an order ticket. If last, switch on ready-up light. Items are selected using a combination of a ticket number and a menu ID. Involves the KitchenDisplay, PassLight and TicketPrinter objects.
- 3(a) CancelReadyUpLight. (*Waiter*). press (I), viewSwitchedOff (O). In this and the previous use case, the light output messages should not take account of the current state of the light. If a light is told to switch off and it is already switched off, it should still send out a viewSwitchedOff message. Involves the PassButton and PassLight objects.

C Message processing by I/O devices

This appendix documents the input and output message processing for the input/output devices.

This processing is sufficient to support all use cases in all three phases. You should *not* add any further processing. 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*
Declaration for handling method in device class

Declaration for method in device class that fetches input event
NTI *Input message name and arguments*

Declaration for method in device class that generates output

O   *Output message name and arguments*

Input and output message arguments are always strings, so their types are not explicitly shown.

C.1 Output message argument formatting

Several methods generating output messages add some kind of formatting. For example constant strings are included that explain values included in the output string sequences. A common kind of formatting used considers the output string sequence as a series of tuples, with each tuple printed on a separate line and with corresponding tuple elements aligned vertically. For example, the displayMenu(Menu menu) method in the OfficeKVM class initially generates a list of strings for the menu using a toStrings() method on the menu object. This might generate the list of strings

```
"D1", "Soft Drink", "1.50", "D2", "Wine", "3.25"
```

The displayMenu() method adds formatting information to this list, inserting the following strings as the arguments of the generated viewMenu() message.

```
"tuples", "3", "ID", "Description", "Price", "D1", "Soft Drink", "1.50",
"D2", "Wine", "3.25"
```

The Event class toString() method then formats the whole output event as:

```
1 19:15, OfficeKVM, okvm, viewMenu, tuples, 3,
    ID, Description, Price,
    D1,  Soft Drink,  1.50,
    D2,        Wine,  3.25,
```

Below some hints are given about the formatting. For further information, read the code of the output message generating methods and of the provided domain entity classes that define toStrings() methods.

C.2 Phase I

In the office.

OfficeKVM

TI   showMenu()
      void showMenu()

      displayMenu(Menu menu)
O   viewMenu("tuples","3","ID", "Description", "Price" + triples)

TI   addToMenu(menuID, description, price)
      void addToMenu(String menuID, String description, Money price)

TI removeFromMenu(menuID)
void removeFromMenu(String menuID)

C.3 Phase II

At a table.

TableDisplay

TI startOrder()
void startOrder()

TI showMenu()
void showMenu()

void displayMenu(Menu menu)
O viewMenu("tuples","3","ID", "Description", "Price" + triples)

TI showTicket()
void showTicket()

void displayTicket(Ticket ticket)
O viewTicket("tuples","3", "ID", "Description", "Count" + triples)

TI addMenuItem(menuID)
void addMenuItem(String menuID)

TI removeMenuItem(menuID)
void removeMenuItem(String menuID)

TI submitOrder() : void
void void submitOrder()

TI payBill()
void payBill()

CardReader

String waitForCardDetails()
NTI acceptCardDetails(cardDetails)

BankClient

String authorisePayment(String cardDetails, Money amount)
O makePayment(cardDetails, amountString)
NTI acceptAuthorisationCode(authorisationCode)

ReceiptPrinter

void printReceipt(Money total, String authCode)
O takeReceipt("Total:", totalStr, "AuthCode:", authCode)

C.4 Phase III

In the kitchen and at the kitchen pass.

Clock
TI tick()
void tick()

KitchenDisplay
TI itemReady(ticketNumberString, menuID)
void itemReady(int ticketNumber, String menuID)

void displayRack(Rack rack)
O viewRack("tuples","6", "Time","Ticket#", "MenuID","Description", "#Ordered","#Ready"
+ 6-tuples)

TicketPrinter
void printTicket(Ticket ticket)
O takeTicket("tuples","3", "Table:", tableID, "_", "MenuID", "Description", "Count" + triples)

PassLight
switchOn()
O viewSwitchedOn()

switchOff()
O viewSwitchedOff()
PassButton
TI press()
void press()

Paul Jackson. v1: 14th November 2016, v5: 27th November 2016.