UECS2354 Software Testing
Lab 08: Integration Testing and System Testing

In this lab, we will use a simple complete application to demonstrate the concept of integration and system testing. We will also discuss in detail how to automate system testing based on use cases for two versions of the application: one involving a command line UI and another utilising a Java Swing GUI.

Integration testing

Test double is useful because it helps to **isolate the SUT** during the testing process. This ensures that if a failure occurs in the test, we can be **assured that the source of the defect or bug is in the SUT**.
Without the use of mocks or stubs, the test failure of a SUT could be also due to a bug in the DOCs. This will increase the amount of code that needs to be checked by the developer in the process of tracking down the source of the bug. As debugging is a tedious and time consuming activity, limiting the amount of code that needs to be checked is very helpful indeed.

Once we have completed an entire set of unit tests for all the basic components (which are nearly always methods in classes) that constitute our application, the next step will be integration testing. Here, we **remove the test doubles** and allow the various SUTs to directly interact with the DOCs in a normal manner when we are testing.

For example, consider that we have method `A` in class `X` that invokes method `B` in class `Y` in order to use a result returned from `B`. In integration testing, we will test method `A` as it runs normally without replacing class `Y` with a stub.

Before doing this, we are likely to have already unit tested method `A` and method `B` in isolation by themselves successfully. Therefore, if we observe an **error when testing both of them together**, we can conclude that the bug is **due to the nature of their interaction**; which basically refers to how the method arguments (or parameters) are passed from `A` to `B`, and how the results returned from `B` are used in `A`.

Integration testing can range from the most basic types to more complicated ones; where a large module of code (consisting of many classes and methods) are tested when they interact with another equally large module of code. Often, these two large modules are produced by two different software teams and will usually involve some access to external resources such as file systems, databases, web services or network sockets. Integration testing however usually does not involve testing the GUI portion of an application.

System and acceptance testing

System testing is concerned with the **behaviour of the whole system**/product as defined by the scope of a development project or product. This is usually the final stage of testing before the product is actually released to the customer or into the commercial marketplace. Therefore it is usually supplemented with acceptance testing activities as well, which seeks to validate the testing of the system with regards to whether it actually meets the targeted end user needs and requirements.

Testing at this stage usually involves use cases that document the most common uses of the software product by its targeted end users, and will involve significant testing of the GUI aspect of the application. To quickly revise, consider a complete application such as Eclipse. A sample use case to test Eclipse might be the sequence of instructions to import a project folder into the Eclipse workspace, which goes something like this:

1. Import the *XXXXX* project from the *resources* subfolder into the Eclipse IDE.
2. From the main menu, select *File* → *Import*.
3. In the *Import* dialog box, select *General, Existing Projects into Workspace*, and click *Next*.
4. In the *Select root directory* textbox, click on *Browse*, and navigate to the *resources* subfolder to select the *XXXXX* folder.
5. Check the *Copy projects into workspace* in the *Options* list.
6. Click *Finish*.
7. This project should now be imported into your workspace.

The **use case is specified in terms of steps** to be performed by the user when interacting with the system, and it does not require any programming knowledge to create.

For a complex application like Eclipse, there may be hundreds to thousands of use cases possible: for examples, refactoring code, debugging a program, adding in new classes and packages, etc. Therefore, automating the testing of use cases (so that they need not be done repeatedly by a user) will be very useful indeed.

In this lab, we are going to see how we can perform both integration testing as well as system testing for a very simple application using the testing frameworks that we have worked with so far (JUnit and Mockito). At this point it is important to note that although JUnit is designed as a unit testing framework, it can be just as easily used to create integration and system tests based on use cases as well. The difference lies in how we set up the classes and methods to be tested; rather than on any special functionality within JUnit or Mockito itself.

UECS2354 Software Testing
Lab 08: Integration Testing and System Testing

Running the lab

Student Record application

We will demonstrate the testing activities at the various test levels (unit, integration and system) that can be carried out on a software product through the use of a simple application called the *Student Record* application. The functional specifications of this application are as follows:

It reads in a text file with various lines of text that represent the student marks for several subjects. The contents of a sample text file is shown below:

```
Vicky     english:10    geography:20   malay:30  art:40
Peter     english:50    art:90         malay:15  maths:50
Carmen    english:80    maths:15       malay:50  art:90
Melanie   science:10    maths:60       malay:80  art:20
Albert    science:30    geography:25   malay:70  art:10
```

Each text line in the file represents an individual student record. The first item is the student name, and the remaining 4 items are a subject name / mark pair; with the subject name separated from its associated mark by the ":" separator.

Each student will take exactly 4 subjects out of a total of 7 possible subject choices (`english`, `geography`, `malay`, `art`, `science`, `maths` and `history`). The subject name / mark pair appears in random order for each given student (for example `art` appears as the 2nd item for `Peter`, but the 4th item for all other students).

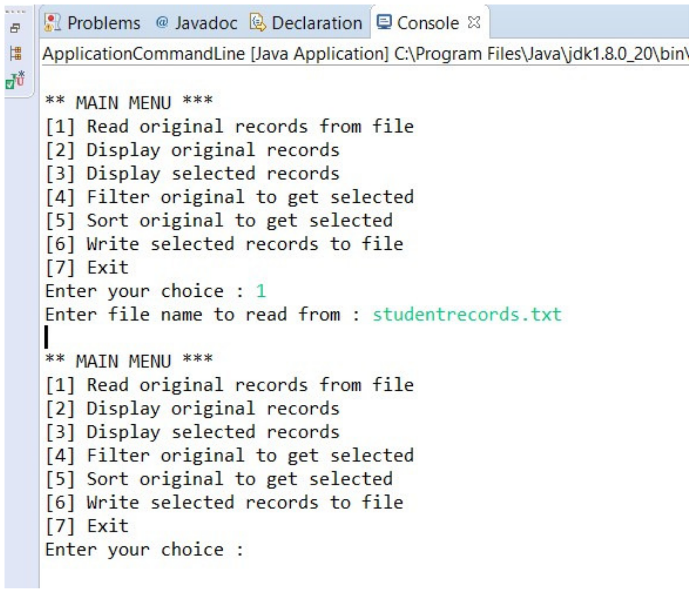
The application is capable of performing two types of operations on the initial set of records read in from the text file. The first is to create a sorted order on the records based on the following items:

- the name of the students (sorted in ascending order from a – z)
- the total mark of the students; given by the sum of all the 4 subject marks for each student (sorted in descending order)
- the mark of a specific subject; which can be any one of the 7 possible choices mentioned earlier (sorted in descending order)

The second is to filter these initial set of records to create a smaller subset of selected records based on the specific subjects taken by a student. This subset will not be in a sorted order. The initial set of records read in, as well as the list of sorted or filtered records can then be displayed via an appropriate user interface (UI). The sorted or filtered records can additionally be saved to a text file in the same format shown earlier.

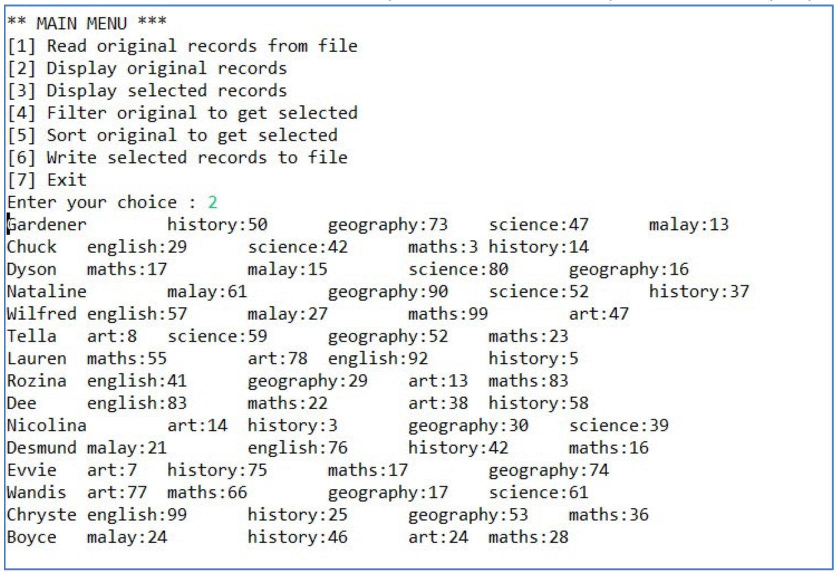
There are two versions of this application implemented with identical functionality: one is based on a command line UI (`DemoCommandLine`), while the other is based on a GUI (`DemoSwingGUI)` that is constructed using components from the Java Swing library.
There are 2 test files in the project root folder `DemoCommandLine`: we will use `studentrecords.txt` for demonstration. Run `ApplicationCommandLine.java` as Java application.

A main menu display will be shown in the console view. Double click on the view tab to maximise it. Type in the following entries followed by the enter key. User input will usually be shown in a different colour from the output text.



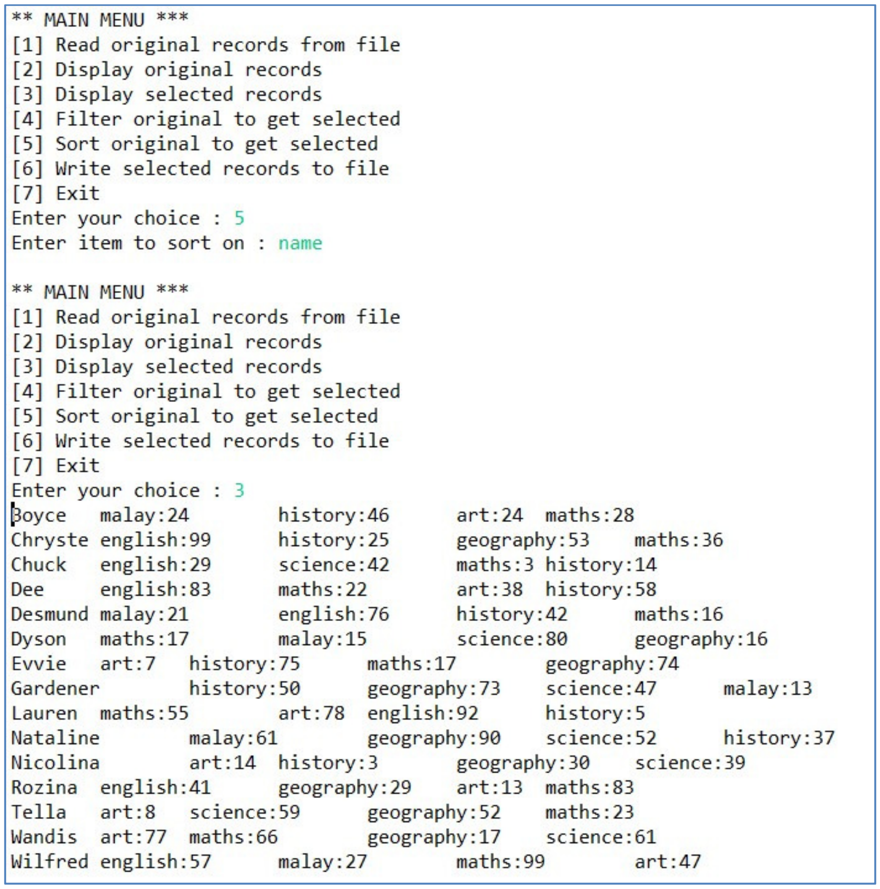
```
ApplicationCommandLine [Java Application] C:\Program Files\Java\jdk1.8.0_20\bin\

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 1
Enter file name to read from : studentrecords.txt

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice :
```

Now that we have read in the records from the input text file, use option 2 to display them.



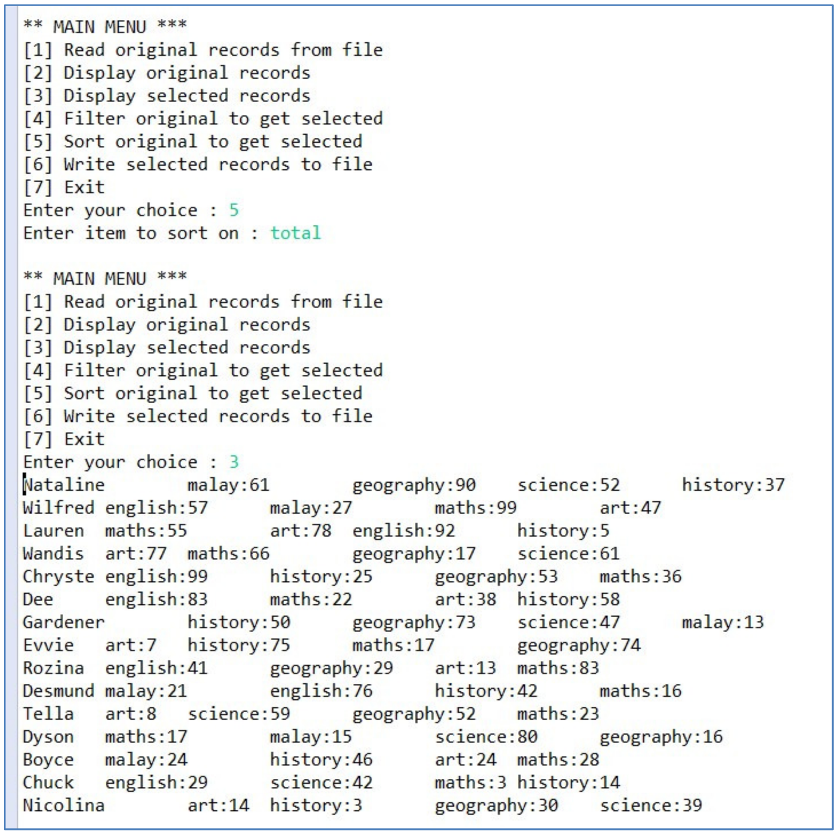
```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 2
Gardener        history:50      geography:73    science:47      malay:13
Chuck   english:29      science:42      maths:3 history:14
Dyson   maths:17        malay:15        science:80      geography:16
Nataline        malay:61        geography:90    science:52      history:37
Wilfred english:57      malay:27        maths:99        art:47
Tella   art:8   science:59      geography:52    maths:23
Lauren  maths:55        art:78  english:92      history:5
Rozina  english:41      geography:29    art:13  maths:83
Dee     english:83      maths:22        art:38  history:58
Nicolina        art:14  history:3       geography:30    science:39
Desmund malay:21        english:76      history:42      maths:16
Evvie   art:7   history:75      maths:17        geography:74
Wandis  art:77  maths:66        geography:17    science:61
Chryste english:99      history:25      geography:53    maths:36
Boyce   malay:24        history:46      art:24  maths:28
```

We can now perform a sort on the name of the students and display the selected records. Every time we perform a sort or filter operation on the original set of records, the results will be stored as the selected records; overwriting the results from the previous operation.

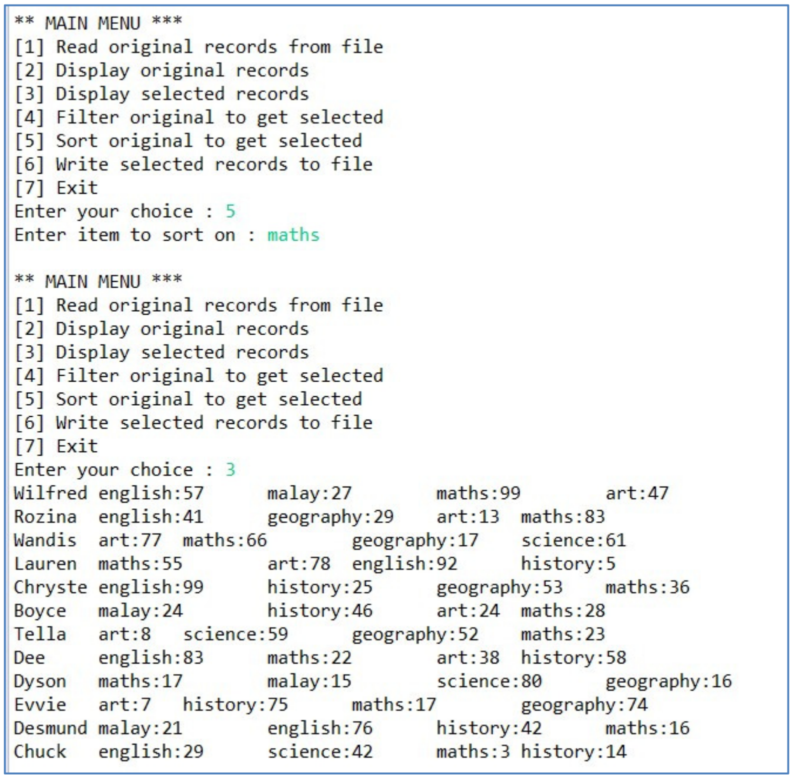


```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 5
Enter item to sort on : name

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 3
Boyce   malay:24        history:46      art:24  maths:28
Chryste english:99      history:25      geography:53    maths:36
Chuck   english:29      science:42      maths:3 history:14
Dee     english:83      maths:22        art:38  history:58
Desmund malay:21        english:76      history:42      maths:16
Dyson   maths:17        malay:15        science:80      geography:16
Evvie   art:7   history:75      maths:17        geography:74
Gardener        history:50      geography:73    science:47      malay:13
Lauren  maths:55        art:78  english:92      history:5
Nataline        malay:61        geography:90    science:52      history:37
Nicolina        art:14  history:3       geography:30    science:39
Rozina  english:41      geography:29    art:13  maths:83
Tella   art:8   science:59      geography:52    maths:23
Wandis  art:77  maths:66        geography:17    science:61
Wilfred english:57      malay:27        maths:99        art:47
```

Experiment the sorting on other items, such as the total mark or the mark for a specific subject. If we choose the mark for a specific subject, only students who have taken that particular subject are included in the sort and displayed in the selected records.

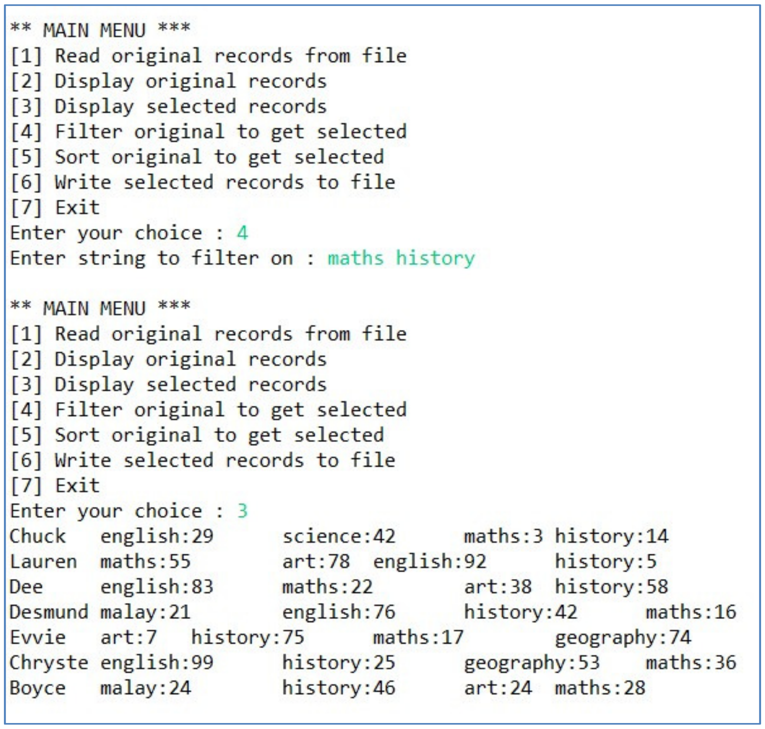


```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 5
Enter item to sort on : total

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 3
Nataline        malay:61        geography:90    science:52      history:37
Wilfred english:57      malay:27        maths:99        art:47
Lauren  maths:55        art:78  english:92      history:5
Wandis  art:77  maths:66        geography:17    science:61
Chryste english:99      history:25      geography:53    maths:36
Dee     english:83      maths:22        art:38  history:58
Gardener        history:50      geography:73    science:47      malay:13
Evvie   art:7   history:75      maths:17        geography:74
Rozina  english:41      geography:29    art:13  maths:83
Desmund malay:21        english:76      history:42      maths:16
Tella   art:8   science:59      geography:52    maths:23
Dyson   maths:17        malay:15        science:80      geography:16
Boyce   malay:24        history:46      art:24  maths:28
Chuck   english:29      science:42      maths:3 history:14
Nicolina        art:14  history:3       geography:30    science:39
```



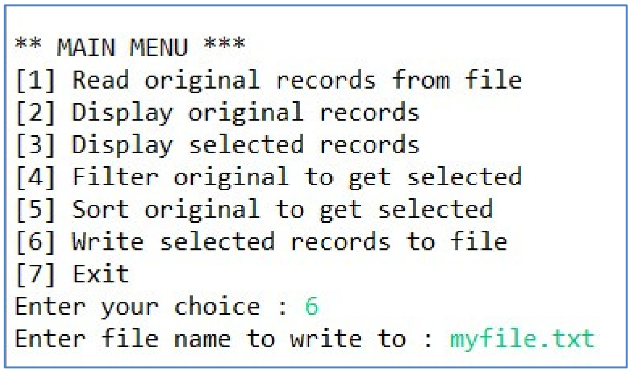
```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 5
Enter item to sort on : maths

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 3
Wilfred english:57      malay:27        maths:99        art:47
Rozina  english:41      geography:29    art:13  maths:83
Wandis  art:77  maths:66        geography:17    science:61
Lauren  maths:55        art:78  english:92      history:5
Chryste english:99      history:25      geography:53    maths:36
Boyce   malay:24        history:46      art:24  maths:28
Tella   art:8   science:59      geography:52    maths:23
Dee     english:83      maths:22        art:38  history:58
Dyson   maths:17        malay:15        science:80      geography:16
Evvie   art:7   history:75      maths:17        geography:74
Desmund malay:21        english:76      history:42      maths:16
Chuck   english:29      science:42      maths:3 history:14
```

We can also choose to filter the original records based on specific subjects taken by the students. You may specify one or more subjects as the filter string.



```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 4
Enter string to filter on : maths history

** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 3
Chuck   english:29      science:42      maths:3 history:14
Lauren  maths:55        art:78  english:92      history:5
Dee     english:83      maths:22        art:38  history:58
Desmund malay:21        english:76      history:42      maths:16
Evvie   art:7   history:75      maths:17        geography:74
Chryste english:99      history:25      geography:53    maths:36
Boyce   malay:24        history:46      art:24  maths:28
```

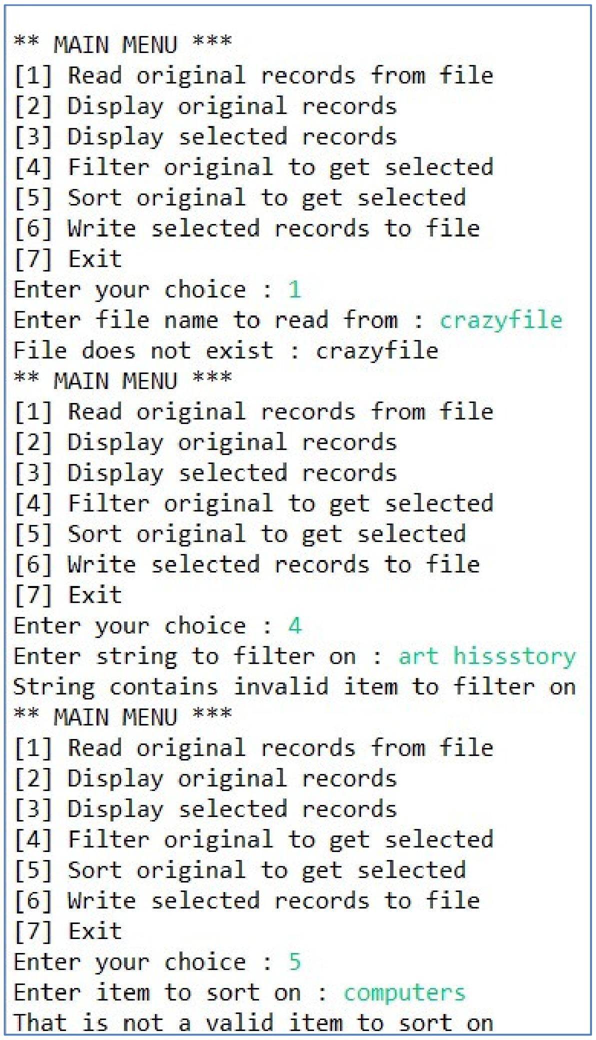
Choose to save the selected records to a file, which will by default be in the project root folder.



```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 6
Enter file name to write to : myfile.txt
```

The application also performs input validation and displays appropriate error messages in response to various conditions such as:

- Attempting to read from a non-existent input file
- Attempting to sort on an invalid item or on more than one valid item. Valid items include name, total and one of the 7 possible subject names.
- Attempting to filter on an invalid item in the filter string
- Attempting to sort or filter on a valid item (for example, a subject name) which however does not exist among the current set of original records

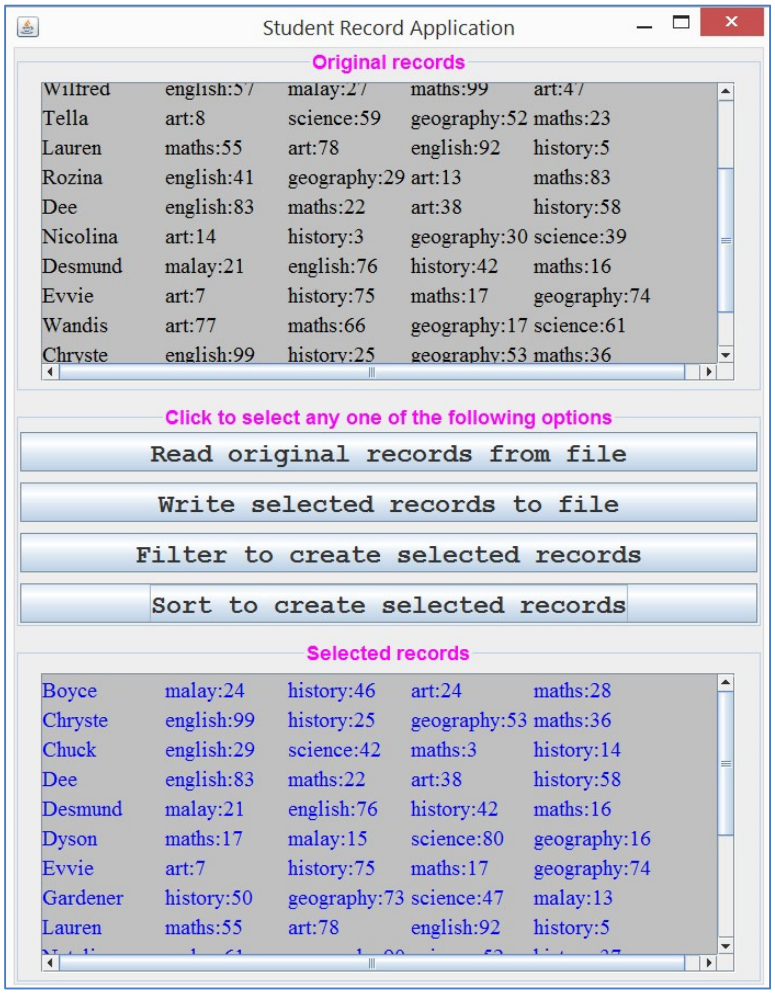


```
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 1
Enter file name to read from : crazyfile
File does not exist : crazyfile
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 4
Enter string to filter on : art hissstory
String contains invalid item to filter on
** MAIN MENU ***
[1] Read original records from file
[2] Display original records
[3] Display selected records
[4] Filter original to get selected
[5] Sort original to get selected
[6] Write selected records to file
[7] Exit
Enter your choice : 5
Enter item to sort on : computers
That is not a valid item to sort on
```

Enter 7 to exit.

UECS2354 Software Testing
Lab 08: Integration Testing and System Testing

The Swing GUI version is `ApplicationMainGUI.java` in `DemoSwingGUI`. Run it as Java Application. There are 4 options available which can be selected by clicking on the appropriate buttons; the list of original records and selected records are displayed in the scrollable text area at the upper and lower area of the application window respectively.



Student Record Application
Original records
Click to select any one of the following options
Read original records from file
Write selected records to file
Filter to create selected records
Sort to create selected records
Selected records

```
Wilfred     english:57  malay:27     maths:99     art:47
Tella       art:8       science:59   geography:52 maths:23
Lauren      maths:55    art:78       english:92   history:5
Rozina      english:41  geography:29 art:13       maths:83
Dee         english:83  maths:22     art:38       history:58
Nicolina    art:14      history:3    geography:30 science:39
Desmund     malay:21    english:76   history:42   maths:16
Evvie       art:7       history:75   maths:17     geography:74
Wandis      art:77      maths:66     geography:17 science:61
Chryste     english:99  history:25   geography:53 maths:36
```

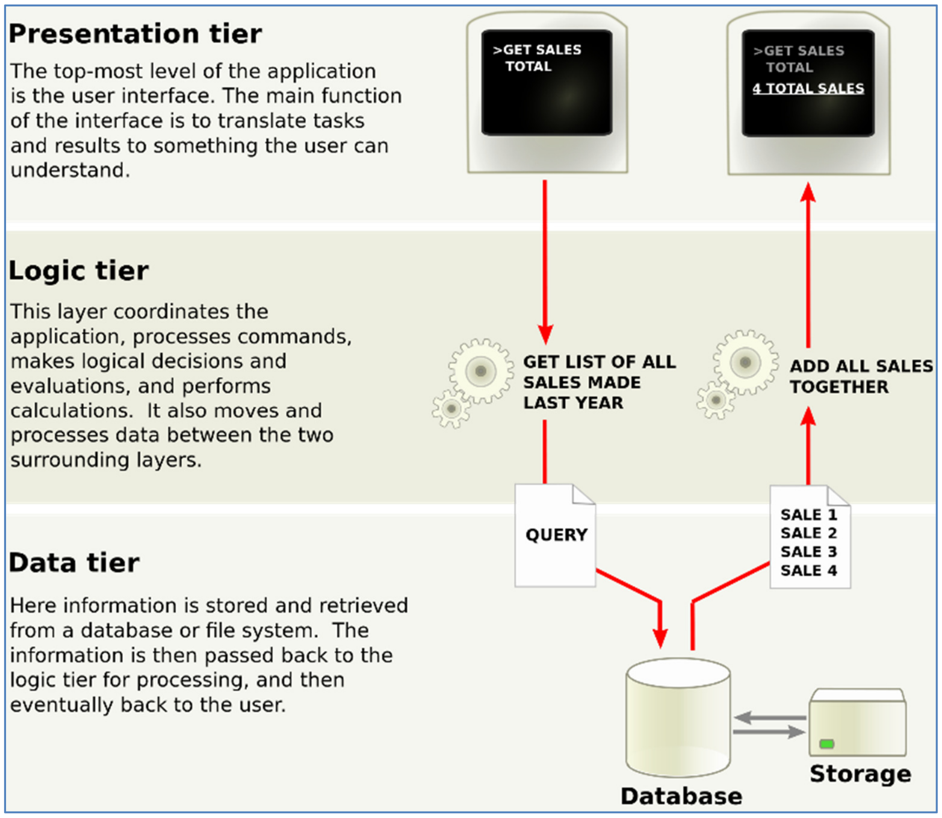
```
Boyce       malay:24    history:46   art:24       maths:28
Chryste     english:99  history:25   geography:53 maths:36
Chuck       english:29  science:42   maths:3      history:14
Dee         english:83  maths:22     art:38       history:58
Desmund     malay:21    english:76   history:42   maths:16
Dyson       maths:17    malay:15     science:80   geography:16
Evvie       art:7       history:75   maths:17     geography:74
Gardener    history:50  geography:73 science:47   malay:13
Lauren      maths:55    art:78       english:92   history:5
```

UECS2354 Software Testing

Lab 08: Integration Testing and System Testing

Student Record application structure

This application is structured on the basis of a 3-tier architecture, which is a very common architectural approach for many applications ranging from large scale web applications to mobile applications.



Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

The two versions of the Student Record application implement two different presentation tiers (command line and Swing GUI) but share identical functionality and code implementation for their logic and data tiers. There is an important design principle that should be adhered to as much as possible when implementing code for multi-tier applications: **maintain a clean separation** (minimal or no coupling) between the code for each tier.

The code that implements data tier functionality is found in `Student.java` and `FileUtilites.java`. The code that implements logic tier functionality is found in `StudentRecordApplication.java` and `SortUtilities.java`. Finally, the code that implements the presentation tier is found in `ApplicationCommandLine.java` and `ApplicationMainGUI.java`. The clean separation between the different tier code components has the desirable result of facilitating the porting of tier functionality between different implementations.

Look at both `DemoSwingGUI` and `DemoCommandLine` projects. They have exactly the same source code files for data and logic tier functionalities, with the exception of `ApplicationMainGUI.java` and `ApplicationCommandLine.java` (presentation tier functionality). As we examine the source code files for the data and logic tier functionalities, there is no GUI related code at all.

This allows us to easily swap in different presentation tier implementations without the need to modify code in lower level tier implementations. The same concept is also equally applicable as well if we wanted to have different data tier implementations; for example, one that interacts with basic text files, and another interacts with a SQL database through the Java JDBC library.

UECS2354 Software Testing

Lab 08: Integration Testing and System Testing

Student Record unit tests

Look at the unit tests in the `DemoCommandLine` project, located in `src/test/java`, bearing in mind that the unit tests in the `DemoSwingGUI` project are also identical as both projects share identical source code for the data and logic tier functionalities.

`FileUtilitiesUnitTests.java` contains the unit test methods for the `writeStringsToFile()` and `readStringsFromFile()` methods. We will not use test doubles for these tests since the code in these methods interact directly with the Java API I/O libraries; which we can safely assume are working correctly and have already been thoroughly tested.

In that case, we can combine the unit tests for both methods into a single test method, `testReadAndWrite()`, by using `writeStringsToFile()` to write an array of random Strings (`testStrings`) out to a file, then reading back from this file into another String array and then comparing both these arrays.
We can opt to make this test more thorough by parameterising it so that this testing sequence is run with a large number of String arrays instead of just one. We also include one more test to ensure that the correct Exception is thrown if a file name that does not exist is specified.

`SortUtilitiesUnitTests.java` contains unit tests for the methods in `SortUtilities.java`. 5 `Student` objects are instantiated with various values for their member variables. There is no fixed rule for the optimal number of objects to create for our tests; it should be sufficiently large enough to demonstrate all (or most) possible input partition combinations, but not so large that the test code becomes difficult to read.

In the `getParamsForTestSortOnSubject()`, we create 6 arrays that contain some or all of these 5 `Student` objects in different sequence combinations; corresponding to the expected results of specific sort operations that are to be executed on an array containing the initial 5 objects.

These 6 arrays, along with the String specifying the sort item to be used; are passed to the parameterised test method `paramTestSortOnSubject()`. At the start of that method, we create a new array containing the initial 5 `Student` objects in a specific sequence (`s1,s2,s3,s4,s5`). We then determine the desired method to be invoked (either `sortOnTotalMark`, `sortOnName` or `sortOnSubject`), passing ArrayList created from this new array. We then compare the result obtained with the expected result arrays passed in from `getParamsForTestSortOnSubject()`.

Notice that we have to convert an array into an ArrayList

```
ArrayList<Student> inputList = new ArrayList<Student>(Arrays.asList(inputArray));
```

and an ArrayList into an array.

```
Student[] outputArray = new Student[outputList.size()];
outputArray = outputList.toArray(outputArray);
```

This is because the **parameterised test methods can only accept array parameters**, whereas the various methods being tested in `SortUtilities` accept `ArrayList` parameters instead. The parameterised test method `paramTestFilterOnSubjectNames()` is based on the same concept; and the code in it is similarly structured as well.

Look at StudentRecordApplicationUnitTests. At the start of the class, we instantiate 3 Student objects (s1, s2, s3) that will be used in many of the test methods in this class.

In the @Before method (setupForAllTests()), we create two mocks based on the FileUtilities and SortUtilities class that can later use as either mock objects or stubs; and pass these to the constructor of a StudentRecordApplication object. This object will be subsequently used in all the remaining test methods of this class.

The first parameterised test method, testIsValidSubjectName(), is reasonably straight forward and checks that isValidSubjectName() returns the correct result (either true or false) corresponding to different possible combinations of input String arrays.

The testPerformSortOperation() test method makes extensive use of Mockito facilities. First, ArrayList is created, containing the sample 3 Student objects, and suMock is set up to function as a stub to return this ArrayList when any of its methods (sortOnName, sortOnTotalMark or SortOnSubject) are invoked.

We are performing a unit test; so when a call is made to any one of the methods in SortUtilities (sortOnName, sortOnTotalMark or SortOnSubject) from within performSortOperation in StudentRecordApplication.java, this call must be redirected to a stub (suMock) which returns a predictable result.

Next, verify that certain methods are not called depending on the value of itemSort (all those verify() methods in testPerformSortOperation); these corresponds to the if-else statement logic in performSortOperation in StudentRecordApplication.java.

Then, make sure that the selected records are now equal to the ArrayList containing the sample 3 Student objects that the stub suMock is expected to return.

```
ArrayList<Student> selectedRecords = sra.getSelectedRecords();
Student[] selectedRecordArray = new Student[selectedRecords.size()];
selectedRecordArray = selectedRecords.toArray(selectedRecordArray);

assertArrayEquals(listToUseArray, selectedRecordArray);
```

The testPerformSortOperationError1() method checks that an expected exception is thrown when an error condition occurs.
testPerformSortOperationError2() also checks that the expected exceptions are thrown when invalid sort item arguments are specified.
The testPerformSortOperationError3() test method then checks for a thrown exception when the last possible type of error condition occurs.

The parameterised test method testperformFilterOperation() has a similar code structure and logic to that of testPerformSortOperation(), and works on a variety of valid input filter String combinations. There are also test methods (testPerformFilterOperationError1, testPerformFilterOperationError2 and testPerformFilterOperationError3) to verify that exceptions are thrown for the various error conditions that can occur. This are very similar in structure to the error checking methods for testPerformSortOperation.

The private method `getParamsForTestGetRecordsAsString()` is used to provide the parameters for the parameterised test methods `testGetOriginalRecordsAsString()` and `testGetSelectedRecordsAsString()`. Both are concerned with verifying that an array (which is subsequently converted into an ArrayList) of `Student` objects can be converted to a correct string representation of those objects.

```
ArrayList<Student> studentRecords = new
   ArrayList<Student>(Arrays.asList(inputArray));

sra.setStudentRecords(studentRecords);
String displayString = sra.getOriginalRecordsAsString();
String expectedString = "";

for (String s : expectedResults)
   expectedString += s + "\n";

assertEquals(expectedString, displayString);
```

```
ArrayList<Student> studentRecords = new
   ArrayList<Student>(Arrays.asList(inputArray));

sra.setSelectedRecords(studentRecords);
String displayString = sra.getSelectedRecordsAsString();
String expectedString = "";

for (String s : expectedResults)
   expectedString += s + "\n";

assertEquals(expectedString, displayString);
```

We also check that when the student or selected records is null, the String object returned for their representation is null as well.

```
if (inputArray == null) {
   sra.setStudentRecords(null);
   String displayString = sra.getOriginalRecordsAsString();
   assertNull(displayString);
}
```

```
if (inputArray == null) {
   sra.setSelectedRecords(null);
   String displayString = sra.getSelectedRecordsAsString();
   assertNull(displayString);
}
```

In `testWriteSelectedRecordsToFile()`, we set up `fuMock` to function as a mock object to verify that the `writeStringsToFile()` method is called with a correct String representation (`verify(fuMock).writeStringsToFile(expectedResults, "dummyfilename");`) of the ArrayList of sample `Student` objects.

```
ArrayList<Student> selectedRecords = new ArrayList<Student>();
selectedRecords.add(s1);
selectedRecords.add(s2);
selectedRecords.add(s3);
```

This is followed by two more test methods (`testWriteSelectedRecordsToFileError1()` and `testWriteSelectedRecordsToFileError2()`) that check that exceptions are thrown corresponding to the two possible error conditions that can occur in `writeSelectedRecordsToFile()`.

In `testInitializeRecordsFromFile()`, we set up `fuMock` to function as both a stub and a mock.
As a mock, we want to verify that `readStringsFromFile()` was called with the value `dummyfilename`.

```
verify(fuMock).readStringsFromFile("dummyfilename");
```

As a stub, we want `readStringsFromFile()` to return an array of Strings that describe our 3 initial Student objects, so that we can verify that this array has been used correctly by `initializeRecordsFromFile()` to construct an array of 3 Student objects.

```
when(fuMock.readStringsFromFile("dummyfilename")).thenReturn(testFileStrings);
```

The last test method, `testInitializeRecordsFromFileError()`, checks that an exception is thrown if an error is detected in the file name.

`RegressionUnitTestSuite.java` is a test suite that contains all the unit tests; which cover the classes that implement the data and logic tier functionalities. The idea behind regression testing is that every time we make a change to existing code base; whether to correct an existing bug or to introduce a new feature, we should run the unit tests for all existing methods or components. This is because the introduction of new code or modification of existing code can occasionally introduce unintended side effects on other existing parts of the code base. Regression testing will help pick up bugs or faults related to these side effects.

As application code base grows in size as we progress through the project life cycle, the number of unit tests also grows significantly as well. If we were to include every single unit test we write into the regression test suite, running this test suite might consume a significant amount of time. This is further compounded by the fact that we are obliged to do regression testing even if we change just a tiny portion of our code base.

In light of this, we may choose to only include in the regression test suite the unit tests for methods that implement core application functionality. That way, we can still afford to do regression testing multiple times throughout the day whenever we make a change to ensure we don’t break critical functionality. The comprehensive, time consuming testing involving all unit tests can be run overnight or over the weekend in an automated fashion.

UECS2354 Software Testing
Lab 08: Integration Testing and System Testing

Student Record integration tests

`StudentRecordApplicationIntegrationTests` contain the integration tests for `StudentRecordApplication`. Notice that this test class DOES NOT involve the use of any test doubles (stubs or mocks). Integration testing involves testing the interaction of the various SUTs with their DOCs in a **normal manner**. Thus, all the calls to methods in `FileUtilities` and `SortUtilities` from within `StudentRecordApplication` will be allowed to proceed in the usual manner.

In the set up method, we utilise real objects from these classes to be passed to the constructor of `StudentRecordApplication`.

```
public void setupForAllTests() {
   fu = new FileUtilities();
   su = new SortUtilities();
   sra = new StudentRecordApplication(fu,su);
}
```

The logic and code structure of `testPerformSortOperation()` are similar to the one of `StudentRecordApplicationUnitTests`, except that here we do not set up mocks or stubs to be used. As a result, the test method implementation is actually simpler to read and understand. The same observation applies as well to `testperformFilterOperation()`.

Finally, `testWriteSelectedRecordsToFile()` combines the testing of `writeSelectedRecordsToFile()` and `initializeRecordsFromFile()` in a single method. This is done by writing out an ArrayList of known `Student` objects to a file, reading back from this same file into an ArrayList of `Student` objects and then comparing the array equivalents of these two ArrayLists. Running this test method will create a file `dummytest1.txt` in the project root folder.

```
sra.setSelectedRecords(selectedRecords);

sra.writeSelectedRecordsToFile("dummytest1.txt");
sra.initializeRecordsFromFile("dummytest1.txt");

ArrayList<Student> studentRecords = sra.getStudentRecords();
Student[] studArray = new Student[studentRecords.size()];
studArray = studentRecords.toArray(studArray);

assertArrayEquals(inputArray,studArray);
```

Notice that there are some methods that are tested within `StudentRecordApplicationUnitTests` that are not found in `StudentRecordApplicationIntegrationTests` (e.g. `testIsValidSubjectName`).

This is because these methods are standalone methods that do not interact with any other methods. Hence, it is sufficient to only have a unit test for them. Methods that have code which call other methods can be tested in isolation (unit test) or normally with the methods they call (integration test).

Notice that integration testing does NOT involve the use of complicated or additional functionality from the JUnit or Mockito frameworks.

UECS2354 Software Testing

Lab 08: Integration Testing and System Testing

Student Record system tests

System or acceptance tests are normally based on use cases. Use case testing usually involves the user manually interacting with the system in accordance to the sequence of instructions outlined in the use case, while verifying that appropriate output appears on the UI for each instruction executed. For example, a simple use case for the Student Record application that verifies it can sort a set of student records correctly is outlined below:

1. Start application: verify main menu appears
2. Type 1 at choice prompt: verify file name prompt appears
3. Type studentrecords.txt at file name prompt: verify main menu appears
4. Type 5 at choice prompt: verify enter item prompt appears
5. Type name at enter item prompt: verify main menu appears
6. Type 3 at choice prompt: Verify that the list of student records are shown in correct ordered sequence

Manual use case testing such as in the example above is acceptable for simple applications with a limited number of use cases. For large, complicated applications, the number of potential use cases possible may well run into the hundreds. In that case, automated use case testing is clearly a more viable option; particularly when repeated testing needs to be done.

Use case testing based on command line UI

We will see how this testing can be done using the Mockito and JUnit in `ApplicationCommandLineSystemTests.java`. These tests makes use of a small file containing sample data in the project root folder, `simpletestfile.txt`.

Note that we create a mock of `DisplayUtilities` and pass this to the constructor of `ApplicationCommandLine`.

```
@Before
public void setupForAllTests() {
   duMock = mock(DisplayUtilities.class);
   acl = new ApplicationCommandLine(duMock);
}
```

Examine the `mainMenu()` method in `ApplicationCommandLine.java`. The user input from the console is obtained by the call to `getFromScreen()`, which if executed on a real `DisplayUtilities` object will call the `nextLine` method on the `Scanner` input. This causes the cursor to pause on the screen while waiting for the user to type something and press enter.

Instead of allowing this to happen, we can set up the mock of `DisplayUtilities` instead so that it functions as a stub to return a certain sequence of values every time the `getFromScreen()` method is invoked. This sequence of values would simulate or mimic what a real user would actually be typing on the console view. For example, in `ApplicationCommandLineSystemTests`, we set up `duMock` to mimic the user typing `1`, typing `simpletestfile.txt`, typing `2` and then typing `7`.

```
when(duMock.getFromScreen()).thenReturn("1", "simpletestfile.txt", "2", "7");
```

Notice that in `ApplicationCommandLine`, all output to the console is achieved via calls to the `showToScreen()` method on the `DisplayUtilities` object. If called on a real object, the String argument is passed on the `System.out.print` method, which results in actual output to the console.

Instead of allowing this to happen, we can set up the mock of `DisplayUtilities` so that it also functions as a mock object to record all String arguments passed to its `showToScreen` method; in effect, keeping track of all intended output to the console from within `ApplicationCommandLine`. We can then pass this mock a String array that is to be used in an in order mock verification, which in effect verifies that the output to the screen occurs in a specific sequence.

```
InOrder inOrder = inOrder(duMock);

for (int i = 0; i < expectedResults.length; i++)
   inOrder.verify(duMock).showToScreen(expectedResults[i]);
```

By carefully matching the simulated values returned by the real user and the expected display to the console corresponding to those values, we can effectively implement an automated testing of a use case.

The String arrays `expectedResult1` – `expectedResult7` outline the expected output to the console for different use cases. They are passed as parameters to `commandLineTestRun()`, along with a number that identifies the specific test run associated with that String array.

```
return new Object[] {
   new Object[] {1,expectedResult1},
   new Object[] {2,expectedResult2},
   new Object[] {3,expectedResult3},
   new Object[] {4,expectedResult4},
   new Object[] {5,expectedResult5},
   new Object[] {6,expectedResult6},
   new Object[] {7,expectedResult7}
};
```

Associated with each particular test run number is a particular stub setting for `duMock` to simulate the values entered by the user at the console.

Look at the first test run, which matches with `expectedResult1`. When we call `mainMenu()` on the `ApplicationCommandLine` object, the first thing that happens is a call to `showScreen()` with the String showing the main menu. This matches correctly with the first element in `expectedResult1`.

Next, is a call to `getFromScreen()`, which results in the value `1` being returned from `duMock`. This results in a call to `performRecordInitialization()`, where the next string to be displayed to the screen is sent to `showToScreen()`. Again this matches correctly with the second element in `expectedResult1`. The call to `getFromScreen()` will result in the value `simpletestfile.txt` being returned from `duMock`.

Next, a call to `initializeRecordsFromFile()` with this value is made. Notice that the method `performRecordInitialization()` calls `initializeRecordsFromFile()` on `StudentRecordApplication.java`, which in turn calls `readStringsFromFile()` from `FileUtilities.java`. Thus we have 3 methods interacting with each other.

We can think of system tests as a more extended form of integration tests; where there is a longer sequence in the method call chain. For example, we might be testing method A, which in turn calls method B, which in turn calls method C, etc. In integration tests, we are most likely to have only 2 methods in the chaining sequence; while in system testing, it may end up with 3 or more.

When this call is complete, control returns to the loop in the `mainMenu()` method, where the main menu String is passed to `showToScreen()` again. This corresponds with the 3rd element in `expectedResult1`. Then the subsequent call to `getFromScreen()` will result in the value 2 being returned from `duMock`. This in turn results in a String being obtained from calling `getOriginalRecordsAsString()` and this String is then passed to the `showToScreen()` method. This String corresponds to the 4th element in `expectedResult1`.

Proceeding in this manner, we should be able to see how the matching of values from `expectedResult1` with the corresponding values returned from the stub, allows the simulation of a sequence of user input entries and the verification of console output corresponding to that input sequence. This in effect constitutes an automated test run of a single use case.

Each String array (`expectedResult1` – `expectedResult7`) represents a sequence of expected console outputs that matches with a specific sequence of user entries returned from the various stub. Thus we are able to simulate a total of 7 use cases in this single parameterised test method.

UECS2354 Software Testing

Lab 08: Integration Testing and System Testing

Use case testing based on GUI

The system testing for the Swing GUI version of the application can be found in `ApplicationGUISystemTests.java`. The Swing GUI works quite differently from standard command line I/O. The GUI application (`ApplicationMainGUI.java`) starts off by displaying the main frame and then waiting until an event happens (such as a button being clicked or the enter key being pressed in a text dialog box). An event handler (the private `ButtonHandler` class) is registered with each component for which a potential event can occur on. The `actionPerformed()` method is then invoked when the event happens.

Depending on the particular source of the event, appropriate action will be taken. For example, if the `readButton` button is clicked, a dialog box prompting for a file name is displayed and the entered string is then passed in a call to the `initializeRecordsFromFile()` method.

```
if (e.getSource() == readButton) {

   String fileName = JOptionPane.showInputDialog("Enter file name to read from: ");
   if (fileName == null)
      return;

   try {
      sra.initializeRecordsFromFile(fileName);
      String recordString = sra.getOriginalRecordsAsString();
      originalRecords.setText(recordString);
   }
   catch (IllegalArgumentException iae) {
      JOptionPane.showMessageDialog(null, iae.getMessage(), "File read error",
                                    JOptionPane.ERROR_MESSAGE);
   }
```

Notice that in the remaining `else if` statements in this method, calls are made to various methods on the `StudentRecordApplication` object.

In order to automate the testing of this UI in the similar way that we have accomplished for the command line UI, we would have to somehow simulate the action of the user pressing certain buttons as well as recording the responses from these actions, such as text appearing in the two areas of the application window.

Due to the fact that a Swing application's main execution thread starts up and runs independently of the test class (rather than being started and controlled by the test class), it is impossible to directly control the application's interaction through the use of mocks as was the case in the previous example.

In order to simulate user interaction with a Swing GUI based application for testing purposes, we need a special testing framework designed specifically for that purpose. Although there are several available, such as FEST (http://code.google.com/p/fest/) or UISpec4J (http://www.uispec4j.org/), they only work with older versions of Java (6 and below) and they require a good understanding of the Swing model as well as the features of that particular framework. Such material will not be covered in this subject.

A simpler and more practical way to automate a use case involving a Swing GUI application is to note down all the relevant methods that are invoked corresponding to a specific GUI component event. For example, studying the `actionPerformed()` method of `ApplicationMainGUI` tells us that clicking on the `readButton` will result in the `initializeRecordsFromFile()` method being called, clicking on the `writeButton` results in the `writeSelectedRecordsToFile()` method being called and so on.

With that in mind, we simply simulate a particular event occurring by invoking all the methods associated with that event occurrence. For example, in `ApplicationGUISystemTests` each block of code in the `if-else if` statements correspond to different use case test runs involving the clicking of certain buttons in certain sequences that result in certain method calls. Each of the use case test runs here directly correspond to the ones initially created in `ApplicationCommandLineSystemTests`.

The String arrays returned from `getParamsForGUITestRun()` specify the expected String outputs that are visible in the text area for the original as well as selected records in the Swing GUI window corresponding to the given sequence of button clicks in each particular test run.

Again, note that this testing approach merely simulates the functionality of the GUI. We are still however unable to perform automated testing of the GUI itself: for eexampl, testing that a drop down list appears in the window with certain items in it when a button is clicked. For that we will require the use of a framework such as FEST or UISpec4J.