# G6046 Software Engineering

# Seminar session: Testing

### Types of testing

We shall be concerned with 2 broad types of testing:

- Unit/component level testing (types of “Development Testing”)
- System level testing (a type of “Release Testing”)

In the real world, there is also user acceptance testing and site acceptance testing. These are just further examples of release testing where the user is making a final check that the delivered software meets their needs. However, today we shall focus on the kind of testing that is relevant and appropriate for your coursework project.

**Unit/component level testing**
Most major languages support unit testing. We will look at how to do this in both Java and Python.

As we have seen before in lecture classes, you can define JUnit test classes and in Python you can create `unittest` sub classes that can be used as automated test units to ensure that your code at a class level is functioning correctly. The “unit” here is a class. We ensure that each class functions the way that it should. It is up to us to decide what constitutes an appropriate range of tests. We should think of edge cases and typical cases. We can define one or more tests. In general, we have one unit test class per class, and then add one or more tests to each unit test class. In Java, the tests are annotated using the `@Test` annotation. This annotation “marks” the method to the compiler as one that is used in the automated testing process. In Python, any method in a `unittest` class that starts with the name `test` is a unit test.

The general process involves:

- Creating an instance of the class under test.
- Calling some methods for that instance.
- Using an `assert` style method to check that the return values given by those methods matches out expectations.

However, it is more flexible than this. We can write any code we like in the test methods. In Java, we can use the `fail()` method to indicate to the test mechanism whether a test should be regarded as a “fail”.

Component level testing is just testing where we test two or more classes working together. In this case, our test code can create as many instances of classes as needed and make them work together and then test that the results are what we expect.

In this seminar, we look at a simple example application. All code, including solutions, can be found on Canvas.

## Example application

The example program allows you to create, lookup and rate some words in a Dictionary. Here is a selection of key functional requirements that describe what the application is supposed to do:

| | |
|---|---|
| F1 | The application shall allow words to be added to the list of all words in the dictionary. The default rating for any word is 0. |
| F2 | All word operations shall be case insensitive. |
| F3 | The application shall allow the user to search for a specified word. If the word if present, it shall be displayed on the screen with the current word rating. |
| F4 | A user shall be able to upvote any specified word. An error shall be produced if the user attempts to upvote a word that is not present in the dictionary. |
| F5 | A user shall be able to downvote any specified word. An error shall be produced if the user attempts to downvote a word that is not present in the dictionary. |
| F6 | A word rating shall be an integer value in the range 0 to 10 inclusive. |

Note the proper use of “shall” for mandatory requirements and “should” for desirable requirements.

## Creating and running unit/component tests

Let’s start by looking at the `Word` and `Dictionary` classes. We will not make any effort to unit test the `TextUI` class. Here are some criteria for unit/component testing.

1) Create a unit test class for `Word`. This should:

    a. Create an instance of the `Word` class for a nice word like “crocodile”.
    b. Check that the initial rating is 0.
    c. Upvote the word.
    d. Check that the rating is now 1.
    e. Downvote the word.
    f. Check that the rating is now 0.
    g. Upvote the word 15 times.
    h. Check that the rating has not exceeded the maximum intended value of 10.

2) Create a unit test class for `Dictionary`. This is actually an example of a component test. The reason for this is that it will create/use two classes at the same time. This should:

a. Create an instance of the `Dictionary` class.
b. Create 3 different words and add them to the `Dictionary`. You can do this by calling the `addWord()` method.
c. Upvote the words by some amounts of your choosing. Work out what the actual ratings should be.
d. Try to upvote a word that does not exist in the `Dictionary`.
e. Try to downvote a word that does not exists in the `Dictionary`.

In Java/BlueJ, use the `TestAll` option (right click on the test class) to run all unit tests defined for that class. In Python/PyCharm, run the unit tests (similar to running any other Python program).

So a unit test considers just one class, and a component test considers two or more classes. In applications without any kind of User Interface (UI), component testing can often be used to test the entire application, and so can be used as a means of conducting automated system level testing.

Solutions projects including the completed unit/component tests can be found on Canvas.

**System level testing**

Once we are satisfied that the individual classes or components are working to our satisfaction, we can proceed to the next level of granularity of testing and try the program as a whole. This is system level testing. The aim is to ensure that the program as a whole does what we want. This means that we are validating the program against the original functional requirements. System level testing is particularly relevant where your program has a User Interface (UI) as these are typically difficult to unit test.

We need to produce a schedule of appropriate tests to validate the program as a whole. It's up to us to decide what level of testing is appropriate. That will depend on factors like:

- How complex is the program to test.
- How critical a failure in the program would be when deployed.
- How much time we have to test.
- How easy it would be to update the program "in the field".
- What the impact of propagating an error (e.g. an invalid input) would be on the rest of the program.

For your coursework project, it would be unrealistic to test every possible source of error, so we have to be pragmatic. System level tests should be linked back to functional (and possibly non-functional) requirements. Remember, we are trying to demonstrate that the program is "fit for purpose". By that, we mean that it delivers on the original requirements set out for it. Some requirements may be more critical than others, so you should focus on those things that you believe are most critical.

The most important factor is to document the system level testing. This ensures that:

- You have adequate records
- You can recreate the error if needed (i.e. determine whether the error is systematic or random).
- You can provide other with information that would allow them to fix the problem at a later stage.
- You have a proper measure of the current quality of your program.

The easiest way to do this is to define a set of tests. An appropriate form of documentation would be a table with headings including:

- Record of time and date of testing.
- Record of software versions(s) under test.
- Record of who performed the testing.

The rest of the table would have one row per test. Each row would have:

- A software test reference.
- Link to the requirement that is being tested (this is the point where you realise why SMART requirements are a good thing).
- Description of the test.
- Input(s) given to the system.
- List of expected output(s).
- List of actual output(s)
- Determination of pass or fail.
- Action required in the event of a “fail” condition.

So, for example:

| **Ref** | **Req** | **Description** | **Input(s)** | **Expected output(s)** | **Actual output(s)** | **Pass/Fail** | **Action?** |
|---|---|---|---|---|---|---|---|
| 1 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |

Devise some appropriate system level tests for the application as a whole. Remember we are trying to demonstrate that the application is “fit for purpose”. An example solution can be found on Canvas.

Dr Kingsley Sage
Khs20@sussex.ac.uk