[Handout for L1P3]

Stop Butterflies From Causing Tornadoes: Preventing Regressions

Testing

IEEE (Institute of Electrical and Electronics Engineers) defines testing as 'operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component'.

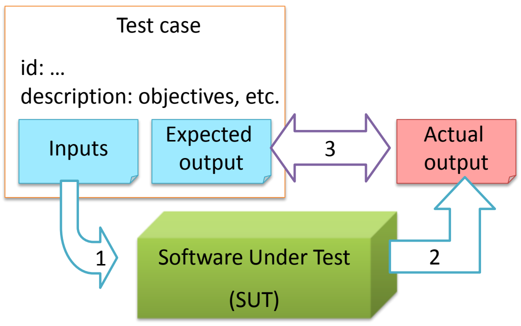

Figure 1. Testing workflow

Testing consists of executing a set of test cases. A *test case* specifies how to perform a test. At the minimum, it specifies what to input to the *software under test (SUT)* and what behavior to expect from it. For example, here is a minimal test case for testing a browser:

- **Input** e.g., First, load a blank file to the browser. This should disable the scroll bar (scroll bar is not required for a blank page). Then, load the longfile.html located in the 'test data' folder.

- **Expected behavior** e.g. The scrollbar should be automatically enabled when the file is loaded.

The above two items can be determined by consulting the specification, or reviewing similar existing systems, or comparing to the past behavior of the SUT.

A more elaborate test case can have other details such as those given below.

- **A unique identifier** e.g. TC0034-a

- **A descriptive name** e.g. Vertical scrollbar activation for long web pages

- **Objectives** e.g., to check whether the vertical scrollbar is correctly activated when a long web page is loaded to the browser

- **Classification information**: e.g., priority - medium, category - UI features

- **Cleanup, if any** e.g. Empty the browser cache

A test case *failure* is a mismatch between the expected behavior and the actual behavior. In the browser example, if the scrollbar remains disabled after loading the long web page, that means the test case failed. A failure is caused by a *defect* (also called a bug). For example, the scrollbar problem could be due to an uninitialized variable.

Scripted vs Exploratory testing

Imagine you are testing a new browser your company has developed. One approach you can take is writing a set of test cases based on the browser's system specification, followed by performing those test cases and reporting any failures to the developers. Alternatively, instead of using a predetermined set of test cases, you could also make up test cases as you go, creating new test cases based on the results of the past test cases. The former approach is called *scripted* testing while the latter is called *exploratory* testing.

Exploratory testing is 'the simultaneous learning, test design, and test execution' [1] where the nature of the next test case is decided based on what happened to the previous test cases we ran. That means running the system while trying out various operations. It is called *exploratory testing* because testing is driven by what we find during testing. Exploratory testing is also known as *reactive testing, error guessing technique, attack-based testing,* and *bug hunting*. Exploratory testing usually starts with areas identified as error-prone, based on tester's past experience with similar systems. One tends to test more for the operations where more faults are found. For example,

> "Hmm... looks like feature *x* is broken. This usually means feature *n* and *k* could be broken too; we need to look at them soon. But before that, let us give a good test run to feature *y* because users can still use the product if feature *y* works, even if *x* doesn't work. Now, if that doesn't work 100%, we have a major problem and the development team needs to know it sooner rather than later..."

Note that the success of exploratory testing depends on the tester's prior experience and intuition. Exploratory testing should be done by experienced testers, using a clear strategy/plan/framework. Ad-hoc exploratory testing by people without testing skills or experience and without a clear strategy is not recommended for real-world non-trivial systems. While exploratory testing often helps us to find some problems in a short time, it is not prudent to use exploratory testing as the sole means of testing a critical system.

In *scripted* (or *proactive*) testing, we use a predetermined set of test cases. Scripted testing is more systematic, and hence, likely to discover more bugs given sufficient time, while exploratory testing would aid in quick error discovery, especially if the tester has strong intuitive skills and sound experience in testing similar systems.

Which one is better? Scripted or exploratory? To quote Bach [1],

> In some contexts, you will achieve your testing mission better through a more scripted approach; in other contexts, your mission will benefit more from the ability to create and improve tests as you execute them. I find that most situations benefit from a mix of scripted and exploratory approaches.

Regression testing

When we modify an already tested system, the modification can result in unintended and undesirable effects on the system. Such an effect is called a *regression*.

Unfortunately, the nature of software is such that even a tiny modification can result in a complete meltdown of the whole system[1]. To detect and correct regressions, we need to retest all related components. Testing with this purpose is called *regression testing*. Regression testing is more effective when it is done frequently, after each small change. However, doing so can be prohibitively expensive if testing is done manually. Hence, regression testing is more practical if it is automated, at least partially.

Test automation (Text UIs)

An automated test case can be run programmatically and the result of the test case (pass or fail) is determined programmatically. Test automation is especially important since manual regression testing is tedious and impractical.

A simple way to test a text UI is using input/output re-direction. For example, let us assume we are testing a program called RouteStore which has a text UI. First, we put the test input in the text file called input.txt. Similarly, we put the output we expect the SUT to produce in another text file called expected.txt. Now, we run the program as:

```
java RouteStore       < input.txt     > output.txt
or
RouteStore.exe        < input.txt     > output.txt
```

Above will direct the text in input.txt as the input to the RouteStore and record the output of RouteStore to a text file called output.txt. Then, all we have to do is to compare output.txt with expected.txt. This can be done using a utility such as Windows FC command or a GUI tool such as Winmerge. For example, we can run the following two commands.

```
java    RouteStore    < input.txt     > output.txt
FC      output.txt    expected.txt
```

Note that this technique is suitable only when we are testing text UIs and only when we can predict the exact output.

Automated testing of program components or GUIs (Graphical User Interfaces) will be addressed in other handouts.

References

[1] Exploratory testing explained, an online article by James Bach (James Bach is an industry thought leader in software testing). Softcopy available at http://www.satisfice.com/articles/et-article.pdf

Worked examples

[Q1]

Given below is some sample output from a text-based program called TriangleDetector that determines whether three numbers given can possibly be the lengths of three sides of a

[1] In nature, it is said that even something minor such as the flapping of a butterfly's wing can trigger off a chain of events that can cause a tornado in another part of the world. Something similar can happen in software too. That's the connection between this lecture content and its title.

triangle. A sample output is shown below. List test cases you would use to test TriangleDetector. Two sample test cases are given below.

```
C:\> java TriangleDetector
Enter side 1:       34
Enter side 2:       34
Enter side 3:       32
Can this be a triangle? :  Yes
Enter side 1:
```

Sample test cases,

```
34,34,34                  : yes
0,any valid, any valid    : no
```

[A1]
In addition to obvious test cases such as below,

- sum of two sides == third,
- sum of two sides < third, …

we can also think of some interesting test cases such as below (note that their applicability depends on the context in which this software is operating).

- Non-integer number, negative numbers, 0, numbers formatted differently (e.g., 13F), very large numbers (e.g., MAX_INT), numbers with many decimal places, empty string,
- Check many triangles one after the other (will the system run out of memory?)
- Backspace, tab, CTRL+C , …
- Give a long delay between entering data (will the program be affected by the screen saver?), minimize and restore window during the operation, hibernate the system in the middle of a calculation, start with invalid inputs (for some weird reason, the system can do error handling differently in the very first test case), …
- Test on different locale.

The main point to note is how difficult it is to test exhaustively even a trivial system.

---End of Document---