Software Testing, Quality Assurance and Maintenance Winter 2026

Lecture 2 — January 9, 2026

Patrick Lam *version 2*

Why Tests?

Let's start by talking about what test suites can do for you (as a developer).

Reference: Kat Busch. "A beginner's guide to automated testing."
`https://hackernoon.com/treat-yourself-e55a7c522f71`

Back to the situation in Lecture 1. You write some new code, and want some assurance that it works. Passing code review at Dropbox at the time required tests along with the code.

"Lo and behold, I soon needed to fix a small bug." But, of course, it's easy to introduce even more bugs when fixing something. Fortunately, she had some tests.

> I ran the tests. Within a few seconds, I knew that everything still worked! Not just a single code path (as in a manual test), but all code paths for which I'd written tests! It was magical. It was so much faster than my manual testing. And I knew I didn't forget to test any edge cases, since they were all still covered in the automated tests.

Not writing tests is incurring technical debt. You'll pay for it later, when you have to maintain the code. Having tests allows you to move faster later, without worrying about breaking your code.

> If your code is still in the codebase a year (or five) after you've committed it and there are no tests for it, bugs will creep in and nobody will notice for a long time.

Writing tests is like eating your vegetables. It'll enable your code to go big and strong.

> **If it matters that the code works you should write a test for it.** There is no other way you can guarantee it will work.

(We'll look at other ways in this course, but tests are the state of the industry.)

Exploratory Testing

Exploratory testing is usually (but not always) carried out by dedicated testers. In that sense, it's somewhat different from the other testing activities in this course, which are more developer-focussed—our usual goal is learning, as developers, how to deploy better automated test suites for our software. Hallway usability testing, though, is an application of exploratory testing. Furthermore, the dedicated QA function is important, and we should learn about how it works.

Resources. James Bach has an introduction to exploratory testing:

- `https://www.satisfice.com/exploratory-testing`

There is an exhaustive set of notes on exploratory testing by Cem Kaner:

- `https://www.kaner.com/pdfs/QAIExploring.pdf`

> "Exploratory testing is simultaneous learning, test design, and test execution."

Contrast this to scripted testing: test design happens ahead of time and then test execution happens (repeatedly) throughout the product's development cycle. When we think of dedicated QA teams, we think they are manually executing scripted tests. In 2025, that is not an effective use of staff.

There is a continuum between scripted testing and exploratory testing. Good exploratory testing may use prepared scripts for certain tasks.

Scenarios where Exploratory Testing Excels. (from Bach's article)

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process. Exploratory testing should not be randomly bumbling around (we can call that "ad hoc testing")—the random approach finds bugs but isn't the most efficient at giving you an idea of how well the software works.

- Start with a charter for your testing activity, e.g. "Explore and analyze the product elements of the software." These charters should be somewhat ambiguous.
- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Exploratory testing shouldn't produce an exhaustive set of notes. Good testers will be able to reproduce the bugs that they encounter during their testing from brief notes. Taking full notes takes too long.

The output from exploratory testing is at least a set of bug reports. It may also include test notes, which include overall impressions and a summary of the test strategy/thought process. Artifacts such as test data or test materials are also both inputs and outputs from exploratory testing.

Primary vs contributing tasks. One way to classify tasks that software can do (or, in other words, its features) is *primary* vs *contributing*. A *primary* task is core functionality of the system; it's something that you would say "You Had One Job!" about. As examples, text editors must be able to load text files, add text, and save the text files. On the other hand, *contributing* tasks are secondary. A macro system for a text editor would be a contributing task. Being able to read email in your editor is definitely a contributing task. Sometimes it's not black-and-white. Spell-check can go either way.

In-class exercise: Exploratory testing of WaterlooWorks.

We will try out exploratory testing with WaterlooWorks. I believe that all non-exchange students here should have access to the system, although there may be no jobs visible right now.

The charter will be "Explore the overall functionality of WaterlooWorks". Summarize in one or two sentences what the purpose of WaterlooWorks is. Identify the tasks that WaterlooWorks should be able to do and classify them as primary or contributing. Identify areas of potential instability. Test each function and record results (bugs).

Of course, don't do things that have actual effects. Usually, testers would have access to a development server and could test those areas more aggressively. But we are working with production systems here.

Regression Testing

Regression testing refers to any software testing that uncovers errors by retesting the modified program (Wikipedia). This form of testing often refers to comprehensive sets of test cases to detect regressions:

- of bug fixes that a developer has proposed.
- of related and unrelated other features that have been added.

Regression testing usually refers to system level (integration level) testing that runs the entire process.

Attributes of Regression Tests

Regression tests usually have the following attributes:

- **Automated**: no real reason to have manual regression tests.
- **Appropriately Sized**: too small and bugs will be missed. Too large and they will take a long time to run. Optimally, we want to run tests continuously.
- **Up-to-date**: ensure that tests are valid for the version of program being tested.

Automating Regression Tests

Regression tests often have a low yield in terms of finding bugs (and are boring to run). Automation is key.

Input

If the input is from a file, regression tests are easy to run (but should still be automatically triggered on a regular interval). There may still be a problem with validating output. We can also create special mocks that can take input from a file or other sources (e.g. scripting engines).

For UIs, the standard approach is to capture and replay events. This approach can be fragile! For example, tests may fail based on window placement or whitespace. For web applications, there is capture and replay for HTTP using Selenium (see L02-extra notes). Mozilla has a project named Marionette[1] that is used to test Firefox and Thunderbird; it is like Selenium but also works on Chrome elements.

Output

Verifying output can be hard!! Problems can arise from issues such as resolution, whitespace, window placement etc.

Mozilla Case Study[2]

The case study presents an approach to testing Gecko based applications. Gecko is the layout engine for Mozilla applications like Firefox and Thunderbird.

In the past, a frame tree with coordinates for all UI elements was created and manual testers performed tests. Not optimal! The new approach was to capture screenshots after test cases and compare them with the expected screenshot. There were a few problems with the approach due to nondeterminism:

- Animated images: no way to ensure tests would function with animated images.
- Font Hinting: the same character would appear slightly differently after each run of the application.
- Other bugs: resolution problems, minor changes in layout etc.

The problem was "really hellish" and the partial solution was to enable logging in the application. The logs would essentially be compared with expected logs. This became very ugly given 1300 or so test cases; distributing across different computers helped.

Industrial Best Practices

Some more references:

- Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.*
- Kent Beck. *Test Driven Development: By Example.*

[1] `https://developer.mozilla.org/en-US/docs/Mozilla/QA/Marionette`
[2] `http://robert.ocallahan.org/2005/03/visual-regression-tests_04.html`

- Roy Osherove. *The Art of Unit Testing: with examples in C#.*

- **Unit Tests:** Each class has an associated unit test. If you change a class, you must also modify the unit test.
- **Code Reviews:** Each branch in the central version control system has owners. In order to commit code to that branch, you must have your code reviewed and approved by one of the owners. This ensures code quality.
- **Continuous Builds:** There is often a machine that continuously checks out and tests the latest code. All unit and regression tests are run and the status is made public to the team. The status contains information about the whether the code was built successfully, whether all unit and regression tests passed, and a list of the last few commits that were made to the branch. This ensures developers try to submit good code since if you break something, everyone knows about it :)
- **One-button Deploy:** If all tests have passed, one should be able to deploy to production with one command.
- **Back Button:** Systems should be designed so that it's possible to roll back changes.

Unit tests

Unit tests are more low-level than integration tests and focus on one particular "class, module, or function". They should execute quickly. Sometimes you need to create fake inputs (or mocks) for unit tests; we'll talk about that too. You should generally not use an entire real input for a unit test.

There are many unit testing frameworks (e.g. JUnit, NUnit), though I don't know of any dominant ones for C or C++. Here's an NUnit test I found on the Internet[3].

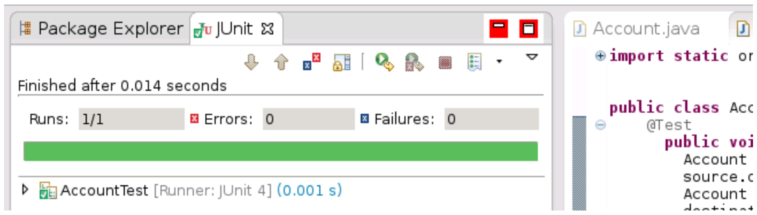
```
[Test]
public void GetMinimum_UnsortedIntegerArray_ReturnsSmallestValue()
{
  var unsortedArray = new int[] {7,4,9,2,5}; // Arrange
  var minimum = Statistics.GetMinimum(unsortedArray); // Act
  Assert.AreEqual(2, minimum); // Assert
}
```

Note that tests have three phases: arrange, act, and assert. I'll say a bit more about good test design, and could say even more, but I don't think I will this term.

> You'll find that writing tests as you go makes your interfaces better and makes your code more testable. If you find yourself writing something hard to test, you'll notice it early on when there's still time to improve the design.

Goal. Well-designed tests are *self-checking.* That means that if the test runs with no errors and no failures (and hence produces a green bar in your IDE), we know that the test was successful.

[3] `https://dzone.com/articles/the-anatomy-of-good-unit-testing`



Finished after 0.014 seconds

Runs: 1/1 Errors: 0 Failures: 0

AccountTest [Runner: JUnit 4] (0.001 s)

```
import static or
public class Acc
    @Test
    public voi
        Account
        source.c
        Account
```

Writing self-checking tests means that the tests automatically report the status of the code. This enables a “keep the bar green” coding style. Implications: 1) you can worry less about introducing bugs (but still take ordinary care); and 2) the tests help document your system’s specs.

How to write self-checking tests. One might think:

> “Isn’t it just calling asserts?”

Sadly, no. That’s not enough.

Two questions about actually deploying asserts:

- Q: what for?
  A: check method call results
- Q: where?
  A: usually after calling SUT (System Under Test)

Counter example. Here’s some example code.

```
public class Counter {
    int count;

    public int getCount() { return count; }
    public void addToCount(int n) { count += n; }
}
```

We can test it with the following JUnit test.

```java
// java -cp /usr/share/java/junit4.jar:. \
//    org.junit.runner.JUnitCore CounterTest
import static org.junit.Assert.*;
import org.junit.Test;

public class CounterTest {
  @org.junit.Test
  public void add10() {
      Counter c = new Counter(); // arrange
      c.addToCount(10); // act
      // after calling SUT, read off results
      int count = c.getCount();
      assertEquals(10, count); // assert
  }
}
```

What kind of test is this? Let's consider two kinds of tests: state-based tests vs. behaviour-based tests.

- **State:** e.g. object field values. Verify by calling accessor methods.
- **Behaviour:** which calls SUT makes. Verify by inserting observation points, monitoring interactions.

Does Counter Test verify state or behaviour?

Flight example. Here's more example code.

```java
// Meszaros, p. 471
// not self-checking
public void testRemoveFlightLogging_NSC() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade facade=new FlightMgmtFacadeImpl();
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 // have not found a way to verify the outcome yet
 //  Log contains record of Flight removal
}
```

Implementing State Verification

We can verify state:

```
// Meszaros, p. 471
// extended state specification
public void testRemoveFlightLogging_NSC() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 assertFalse("flight still exists after removed",
             facade.flightExists(expectedFlightDto,
                                 getFlightNo()));
}
```

Note that we are exercising the SUT, verifying state, and checking return values.

In state-based tests, we inspect only outputs, and only call methods from SUT. We do not instrument the SUT. We do not check interactions.

You have two options for verifying state:

1. procedural (bunch of asserts); or,
2. via expected objects (won't talk about them this year).

Returning to the flight example:

- We do check that the flight got removed.
- We don't check that the removal got logged.
- Hard to check state and observe logging.
- Solution: Spy on SUT behaviour.

Implementing Procedural Behaviour Verification

Or, we can implement behaviour verification. This is one way to do so, behaviourally:

```
// Meszaros, p. 472
// procedural behaviour verification
public void testRemoveFlightLogging_PBV() {
 // arrange:
 FlightDto expectedFlightDto=createRegisteredFlight();
 FlightMgmtFacade=new FlightMgmtFacadeImpl();
 AuditLogSpy logSpy = new AuditLogSpy();
 facade.setAuditLog(logSpy);
 // act:
 facade.removeFlight(expectedFlightDto.getFlightNo());
 // assert:
 assertEquals("number of calls",
              1, logSpy.getNumberOfCalls());
 // ...
 assertEquals("detail",
              expectedFlightDto.getFlightNumber(),
              logSpy.getDetail());
}
```

As an alternative, we can use a mock object framework (e.g. JMock) to define expected behaviour.

Idea. Observe calls to the logger, make sure right calls happen.