**Intro to Methods of Software Engineering** Fall 2018

# Lecture 7 — October 11, 2018

*Patrick Lam* *version 1*

## Introduction to Unit Tests

Verification and validation are always part of engineering. In your SE curriculum, the 3A course SE 465 ("Software Testing, Quality Assurance and Maintenance") is where most V&V content lives. There is also some content in CS 247 ("Software Engineering Principles"). But it's important to talk about it earlier than that, which is why we'll be talking about it today.

**Key Concept: Automated testing.** The state of the art in 2018 is to *automate* software testing. Anyone should be able to check out a project and just press a button to run the tests.

Why automate? Testing is boring and people make mistakes. With automation, you can run more tests more quickly. (You need to first put in some work to set up the automated testing. Setting up test suites is engineering, in the broad sense.)

There is also a role for non-automated testing, particularly exploratory testing, but that is beyond the scope of today's lecture.

## Show me the code!

You've already seen tests in your CS 137 assignments. This is a test from Assignment 3 Problem 2, courtesy Prof. Carmen Bruni:

```
int A1[2][3] = {{1,1,1},{1,1,3}};
assert(isDist(2,3,A1) == false);
```

Let's break down what's happening here.

- prepare input data;
- pass inputs to function;
- compare outputs.

The statement

```
  int A1[2][3] = {{1,1,1},{1,1,3}};
```

is preparing array `A`. In general there might be more sophisticated setup code; you might need to read data from a file, for instance.

The second line does two things. It calls the function:

```
isDist(2,3,A1)
```

which executes the System Under Test.

Then it gets a return value and checks it against the expected value.

```
assert(value_from_SUT == false);
```

If the value is indeed false, then the test succeeds. Otherwise it fails.

**Main idea.** Pass your system under test some input and check its output.

**Considerations.** It's easiest to test individual functions as seen here with `isDist`. It's also easiest when functions don't have side-effects. The spaceship code from the Ideas Clinic would be harder to test; for instance, the defence system called the turret controls to fire missiles.

As tests get more complicated, it's good to put tests each in their own function. There are unit testing frameworks as well; JUnit for Java is ubiquitous. C's unit testing frameworks (Check, CUnit) are harder to use, so we're not going to talk about htem.

**What tests should I write?** There is no magic answer. Check all of the obvious "happy paths" and some error paths.

## Continuous Integration / Continuous Deployment

In the spaceship, you saw the concept of integrating your work with others' work. Continuous integration is where everyone merges their work whenever they do it. This requires automated test cases so that you can know that you're not breaking the system with your work. Once you write something, you make sure that it passes the tests.

**Benefits of Automated Test Suites.** In bullet form:

- avoid regressions (things that used to work now being broken);
- enable newcomers to confidently modify your code;
- serve as a specification of the code.

**Test-Driven Development.** You can go even further and write the tests before you write the code. This is what you have in the CS 137 assignments for instance.

**Are tests foolproof?** Maybe you submitted something to Marmoset that you knew wasn't quite right but it passed anyway. Why is that? How would an instructor find such cases?