ECE459: Programming for Performance Winter 2015

Lecture 36 — April 6, 2015

Patrick Lam *version 0*

Ensuring Testability

This is from Chapter 6 of the Meszaros book.

Retrofitting testability is harder than it needs to be. So is designing testability into a system that doesn't yet exist. The happy medium is in between: incorporate testability as you go along. Build your system for testability. Plus, tests help during development.

Control Points and Observation Points. We previously saw these concepts in Lecture 4. *Control points* are ways to get the system under test (SUT) to do something. It is OK to have test-only control points, although you should probably label them as such. *Observation points* are how you figure out if the system under test is doing the right thing. Direct observation points are clear and mainly consist of return values. But there are also indirect observation points, where you add test doubles and have them check that the SUT is doing the right thing, for instance by observing outbound calls.

Interaction Styles and Testability Patterns. The easiest pattern is the *round-trip test*: your test uses the public interface only. This works great when your system is sufficiently controllable or observable.

An alternative is the *layer-crossing test*: you use the SUT's API, but then you watch what it does using test spies or mock objects. The danger is that you might get overspecified software. Sometimes it doesn't matter that the SUT calls `x()` and then `y()`; it's fine to change the order. Your test should not tie the hands of the implementer.

Dependency Injection is a good way to make mock objects be used by the SUT. More about that shortly. Or, you can use test-specific subclasses, or, as a very last resort, test hooks. They are perilously close to having test code in production, though.

Finally, you might have a *asynchronous test*. You send messages to the system under test and wait for responses. Waiting can be slow and sometimes unreliable, though. This also includes using the UI to test, which is painful and fragile. It's better to interact with the underlying system directly, when possible.

More on Dependency Injection. This has always been confusing to me. So, here's an example. Consider:

```
public void testDisplayCurrentTime_AtMidnight() {
  TimeDisplay sut = new TimeDisplay();
  String result = sut.getCurrentTimeAsHtmFragment();
  String expectedTimeString = "<span class=\"...\">Midnight</span>";
  assertEquals(expectedTimeString, result);
}

public String getCurrentTimeAsHtmlFragment() {
  Calendar currentTime;
  try {
    currentTime = new DefaultTimeProvider.getTime();
  } catch (Exception e) {
    return e.getMessage();
  } // etc.
}
```

This is not going to succeed very often! The dependency on `DefaultTimeProvider` is hard-coded. You can instead introduce a `TimeProvider` argument. There are lots of choices: you can introduce it as a function parameter, or in the constructor, or using a setter method. Or, you can have dependency lookup.