Unit Testing

Software Engineering

Millersville University
Process

• For each piece of "production" code (e.g. a class or a method):
  • Pair the code with some "unit test" code
  • Only access the public API
  • Call it a few different ways
  • Check the results

• Test code does not need to be exhaustive
  • test code adds a lot of value even just hitting a few fairly obvious cases.

• Unit Tests are an investment
  • effort to build
  • Standard, maintained way to keep tests in parallel with production code
  • improve development for the lifetime of the code
High Quality Code

• We think about building lots of different types of code
  • Throw-away code
  • Minimum working example
  • Proof of Concept
  • Production code

• Code was built to an intuitive "it appears to work" quality level

• With unit tests, we can build code to a much higher quality level
  • We have the tests
  • Infrastructure can run the tests constantly
  • Each component is tested independently of one another
Workflow

• For every class (Wingding), create a test class (WingdingTest)
• For every public function (foo), create a test function (testFoo)

• Write the test code first
• Write the production code and debug it until the tests pass
• Every feature has corresponding unit test code.
Unit Test Types

• **Basic**
  • Cases with small to medium sized inputs
  • So simple they should obviously work.
  • Should not be hard to think of

• **Advanced**
  • harder, more complex cases.
  • Some of these, you only think of later as you get deeper into the algorithm.
  • This is the category that tends to grow over time as you get more insight about the problem and observe more weird cases.

• **Edge**
  • there are also cases that are simple but represent edge conditions
  • the empty string
  • the empty list
Call Every Method A Few Times Differently

• If a class has foo() and bar() methods
  • The test code should call each of those a few different ways

• Don’t just call foo() 5 times
  • Focus on where the calls are very similar

• When testing a equals(x, y) method
  • Don't only give x,y where equals() should return true
  • Call it once or twice where it should return false too!

• If someone has changed the method body to something like return false; the unit tests should at least be able to notice that.
Unit Tests vs. API Design

• API design
  • a class presents a nice interface for use by others -- is vital part of OOP design.
• API design is hard
  • it's difficult for the class designer to understand the class
  • Difficult to understand its API the way they will appear to clients.
• Unit tests have the designer literally act like a client
  • Using the class in a realistic way using only its public API.
  • Unit tests help the designer to see if the public API is awkward for expressing common cases.
  • By writing tests first, this insight about the API appears very early in the life of the class when it's easy to change or tune.
Unit Test Boundary Fun

• Change an important < in the work code to a <= to observe the unit test fail
  • it really is bearing down on that case, then change it back to <.
  • In this way you see that the unit test boundary really is where you think it is.
• Change a comment or something else not scary in the code.
  • If you're bored, run the tests again, just to see the green.