Test-First Programming
Test-First Programming

• The tests should drive you to write the code, the reason you write code is to get a test to succeed, and you should only write the minimal code to do so. Note that test-first-design is more than just unit testing. Unit testing by itself does not change the design of the code. In addition to documenting how code should be used, test-first-design helps you keep the design simple right from the start, and keeps the design easy to change.
Test-First Programming

• As project complexity grows, you may notice that writing automated tests gets harder to do. This is your early warning system of overcomplicated design. Simplify the design until tests become easy to write again, and maintain this simplicity over the course of the project.
Guidelines from Sean Shubin

• The name of the test should describe the requirement of the code
• There should be at least one test for each requirement of the code. Each possible path through of the code is a different requirement
• Only write the simplest possible code to get the test to pass, if you know this code to be incomplete, write another test that demonstrates what else the code needs to do
• A test should be similar to sample code, in that it should be clear to someone unfamiliar with the code as to how the code is intended to be used
• If a test seems too large, see if you can break it down into smaller tests
• If you seem to be writing a lot of code for one little test, see if there are other related tests you could write first, that would not require as much code
• Test the goal of the code, not the implementation
• One test/code/simplify cycle at a time. Do not write a bunch of tests, and try to get them working all at once
Guidelines

• Keep writing tests that could show if your code is broken, until you run out of things that could possibly break
• When choosing an implementation, be sure to choose the simplest implementation that could possibly work
• If you are unsure about a piece of code, add a test you think might break it
• A test is one specific case, for which there is a known answer
• If all of the tests succeed, but the program doesn't work, add a test
• Tests should be as small as possible, before testing a requirement that depends on multiple things working, write a test for each thing it depends
• Tests should not take longer than a day to get working, typical test/code/simplify cycles take around 10 minutes
• Don't test every single combination of inputs. Do test enough combinations of inputs to give you confidence that the any code that passes the test suite will work with every single combination of inputs
• Do not write a single line of code that doesn't help a failing test succeed. (Clarification for GUI's, some aspects of GUI's are impossible to test automatically, so it will have to be an acceptance test that drives you two write some GUI code. Use automated testing whenever possible)
• Do not fix a bug until you have written a test that demonstrates the bug
**What is the simpliest code?**

- All of the tests run
- There is no duplicate code (any given code segment or structural pattern should appear "once and only once")
- Clarity. The code and tests communicate the intent as clearly as possible
- The code is minimal (no classes or methods unnecessary to get the tests to pass)
Test-Code-Simplify Cycle

- Write a single test
- Compile it. It shouldn't compile, because you haven't written the implementation code it calls
- Implement just enough code to get the test to compile
- Run the test and see it fail
- Implement just enough code to get the test to pass
- Run the test and see it pass
- Refactor for clarity and "once and only once"
- Repeat