

A decorative graphic on the left side of the slide, consisting of a network of thin, light-blue lines and small circles, resembling a circuit board or a neural network, extending from the top to the bottom of the frame.

CS1632, LECTURE 5: AUTOMATED VS MANUAL TESTING AND INTRO TO UNIT TESTING

BILL LABOON

WHAT IS UNIT TESTING?

- A kind of automated testing
- Unit testing involves testing the smallest coherent "units" of code, such as functions, methods, or classes.
- It is white-box; you are looking at and testing the code directly.
- Ensures that the smallest pieces of the code work correctly (NOT that they work correctly with the rest of the system – very localized)

EXAMPLES

1. Testing that a `.sort` method sorts elements
2. Testing that passing a `nil/null` as an argument throws an exception
3. Testing that a `formatNumber` method formats a number properly
4. Checking that passing in a string to a function which expects an integer does not crash
5. Testing that a `.send` and `.receive` method exist on a class

UNIT TESTING

This is usually done by the developer writing the code, another developer (esp. in pair programming), or (very occasionally), a white-box tester.

WHAT'S THE POINT?

1. Problems found earlier
2. Faster turnaround time
3. Developer understands issues with his/her code
4. "Living documentation"
5. Able to tell if your changes caused issues elsewhere by running full test suite

MINITEST - OUR TESTING FRAMEWORK

- <https://github.com/seattlerb/minitest>
- Run “gem install minitest” or (better) add minitest to your Gemfile (see example) and run “bundle install”
- “...a complete suite of testing facilities supporting TDD, BDD, mocking, and benchmarking.
- Why Minitest? Relatively common, easy to learn, very fast, minimal.

MINITEST IS NOT THE ONLY UNIT TEST FRAMEWORK OUT THERE!

- `Test::Unit` (built-in)
- `shoulda`
- `rspec`
- `Cucumber`
- Ideas should apply to other testing frameworks easily

WHAT DO UNIT TESTS CONSIST OF?

- (optional) Set up code
- Preconditions
- Execution Steps
- Postconditions - a/k/a Assertions (a/k/a asserts, shoulds, musts)
- (optional) Tear down code

EXAMPLE (IN NATURAL LANGUAGE, NOT CODE)

I create two Integer objects, 1 and 1.

If I compare them with the equality operator, they **SHOULD** be equal.

(or "they **MUST** be equal.")

(or "I **ASSERT** that they will be equal")

POSTCONDITIONS = ASSERTIONS

- When you think "should" or "must", that is the assertion. It's what you're testing for.
- It's the EXPECTED BEHAVIOR of the unit test.
- When you execute the test, that's when you'll find out the OBSERVED BEHAVIOR.
- If the expected behavior matches the observed behavior, the test passes; otherwise it fails.

MINITEST ASSERTIONS

- Some assertions using MiniTest:
- `assert_true`
- `assert_equals`
- `assert_includes`
- `assert_nil`
- `assert_raises`

MINITEST ASSERTIONS

- You can also do the opposite with “refute” (like “assert not”)
- `refute_true`
- `refute_equals`
- `refute_includes`
- `refute_nil`
- `refute_raises`

TESTS ARE RUN IN RANDOM ORDER

- Make sure your tests are INDEPENDENT and SELF-CONTAINED
- Tests should be focused - one equivalence class, one method call
- Usually one or two assertions - rarely more than that
- Remember you are testing a small bit of code (a unit), not the whole system!

The image features a dark blue gradient background with decorative white circuit-like lines in the corners. These lines consist of straight segments and small circles, resembling a stylized electronic board. The lines are positioned in the top-left, top-right, bottom-left, and bottom-right corners, framing the central text.

EXAMPLES IN SAMPLE_CODE/MINITEST_EXAMPLE