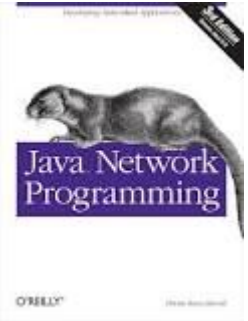
Effective Unit Testing

Elliotte Rusty Harold
TL - Google Cloud Tools for Eclipse

Where do I come from?

- Currently TL of Google Cloud Tools for Eclipse (70% test coverage)
- Developer of XOM (~99% test coverage, almost all test first)
- Jaxen maintainer, Apache Maven committer
- Author of a few books and co-author of https://jlbp.dev



Why do we write unit tests?

- Because we want our code to work
- Because we want it to **keep** working
- Because we want to develop faster with more confidence and fewer regressions
- Because we make mistakes

The Fundamental Principle of Unit Testing

Verify that a known, fixed input produces a known, fixed output.

Start in the Middle of the Road

What if you don’t know the correct output?

- If it’s a deterministic answer, write a characterization test.
- If the problem is fuzzy or not perfectly defined, test a similar problem with less fuzzy answers.

Eliminate everything that makes input or output unclear or contingent.

- No random input. Always fixed values.
- Don’t use named constants from the model code. They may be wrong or change. Prefer literal strings and numbers.
- Don’t access the network and preferably not the file system.
- Control time, the speed of light, or the gravitational constant of the universe.

#2 Write Your Tests First!

- It’s not just about testing; it’s about software development.
- Test first development creates better API because you start with the user, not the used.
- Test first hides implementation and avoids exposing internal implementation details. It avoids brittle, tightly coupled tests.

Test First also makes better tests

```
public class ListTest {
  private List<String> list = new ArrayList<>();

  public void testAdd() {
    list.add("Foo");
    Assert.assertEquals(1, list.size());
  }

}
```

#3 *Unit* Tests

- Unit means ***One***. Each test tests exactly one thing.
- Each test method is one test
- Best practice: one assert per test method
- Share setup in a fixture, not the same method
- You can have multiple test classes per model class. Do not feel compelled to stuff all your tests for `Foo` into `FooTest`.

Unit also means ***Independent***

- Tests can (and do) run in any order.
- Tests can (and do) run in parallel in multiple threads.
- Tests should not interfere with each other.

Avoid Conditional Logic in Tests

```
if (x > 5) {
  assertTrue(y);
}
else {
 assertEquals(1, z);
}
```

Tests and Thread Safety

- **Don’t** use synchronization, semaphores, or special data structures.
- Do not share data between tests:
  - Do not use non-constant static fields in your tests.
  - Be wary of global state in the model code under test.
- Best practice: one assert per test method.
- Share setup in a fixture, not the same method

The Two Least Known Facts of Unit Testing

1. Tests do not share instance data.

```
public class ListTest {
  private List<String> list = new ArrayList<>();

  @Test
  public void testAdd() {
    list.add("Foo");
    Assert.assertEquals(1, list.size());
  }

  @Test
  public void testAdd2Elements() {
    list.add("Baz");
    list.add("Bar");
    Assert.assertEquals(2, list.size());
  }
}
```

2. You can have many test classes per model class.

- Do not feel compelled to stuff all your tests for `Foo` into `FooTest`.
- Every test that needs a slightly different setup can go into a separate test class.

Speed

- A single test should run in a second or less.
- A complete suite should run in a minute or less. (cloud-opensource-java)
- Separate larger tests into additional suites; separate unit and integration tests; e.g. Apache Maven
- This is for ease of development.
- Fail fast. Run slowest tests last.
  - Example

Not All Tests Need to be Unit Tests

The Power of Integration Tests

- Integration tests cover many LOC with very little effort
- The tradeoff is they are harder to debug when something goes wrong
- Integration tests can catch errors in the interfaces between classes that unit tests miss.
- They can even catch changes in the external environment.
- You need both

Passing tests should produce no output

- There should never be any question as to whether a test passed. **Green** or **Red**, **Pass** or **Fail**.
- If necessary, silence loggers in tests.
- Maven gets this badly wrong.
  - Example

Failing tests should produce clear output

- Failing tests should give clear, unambiguous error messages.
- Rotate your test data.
  - Don’t use the same data in every test.
  - E.g. don’t set all ints you test to 3. Use 3, 33, 1117, -98, etc.
  - This makes it much easier to see immediately which test is failing and why.
- Truth helps with collections

Flakiness

- Work really, really hard to avoid
- Sources of flakiness:
  - Time dependence
  - Network availability
  - Explicit Randomness
  - Multithreading
  - Unexpectedly flaky model code
  - Test interdependence and order
  - Bad test infrastructure; e.g. CI servers

System Skew

- Sources of flakiness:
  - Multithreading
  - Assumptions about the underlying operating system
  - Undefined behavior
    - Floating point roundoff
    - Integer width
    - Default character set
    - etc.

Debugging

- Write a failing test before you fix the bug.
- If the test passes, the bug isn’t what you think it is.

Refactoring

- Break the code before you refactor it.
  - Do the tests fail?
  - Example
- Check your code coverage
- If necessary, write additional tests before doing unsafe refactorings.

Development Practices

- Use continuous integration (e.g. Travis, CircleCI, etc.).
- Use a submit queue
- Never, ever check in with a failing test.
- If it does happen, rollback first; ask questions later.
- A **red test** blocks all merges. No further check ins until the build is **green**.

Final Thoughts

- Write your tests first.
- Make all tests unambiguous and reproducible.

Questions?
Comments?
Thoughts?
War Stories?

Thank You

Elliotte Rusty Harold

SWE

NYC

When not laboring in his secret identity of a mild-mannered software developer, Elliotte Rusty Harold lives in a secret mountaintop laboratory on a large island off the East Coast of the United States with his wife Beth and dog Thor. He's an avid birder and insect photographer. His fiction has appeared in Alfred Hitchcock's Mystery Magazine, Crossed Genres, Daily Science Fiction, and numerous anthologies. He's also written over twenty non-fiction books for various publishers including Addison-Wesley, O'Reilly, Wiley, and Prentice Hall. His most recent books are Java Network Programming, 4th edition, and JavaMail API, both from O'Reilly. Find him as @elharo on Twitter or at http://www.elharo.com/blog/