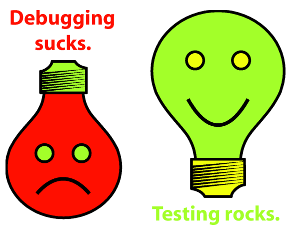

Testing on the Toilet

April 25, 2007

Refactoring Tests in the Red

With a good set of tests in place, refactoring code is much easier, as you can quickly gain a lot of confidence by running the tests again and making sure the code still passes.

As suites of tests grow, it's common to see duplication emerge. Like any code, tests should ideally be kept in a state that's easy to understand and maintain. So, you'll want to **refactor your tests**, too.

However, refactoring tests can be hard because you don't have tests for the tests.

How do you know that your refactoring of the tests was safe and you didn't accidentally remove one of the assertions?

If you **intentionally break the code under test**, the failing test can show you that your assertions are still working. For example, if you were refactoring methods in `CombineHarvesterTest`, you would alter `CombineHarvester`, making it return the wrong results.

Check that the reason the tests are failing is because the assertions are failing as you'd expect them to. You can then (carefully) refactor the failing tests. If at any step they start passing, it immediately lets you know that the test is broken – undo! When you're done, remember to fix the code under test and make sure the tests pass again. (`revert` is your friend, but **don't revert the tests**!)

Let's repeat that important point:

When you're done... remember to fix the code under test!

Summary

- Refactor **production** code with the tests **passing**:
  - This helps you determine that the production code still does what it is meant to.
- Refactor **test** code with the tests **failing**:
  - This helps you determine that the test code still does what it is meant to.

More information, feedback, and discussion:

http://googletesting.blogspot.com