Creating Tests

Arnav Gupta

April 5, 2024

Contents

1 Creating Test Cases

In order to increase coverage, one needs to generate test cases that exercise certain statements or follow specific paths:

- how to make paths execute: generate input data that satisfies all conditions on the path
- key items: input vector, predicate, path predicate, predicate interpretation, path predicate expression, create test input from path predicate expression

Input vector: collection of all data entities read by the routine whose values must be fixed prior to entering the routine

- members include input arguments, global vars and constants, files, network connections, timers

Predicate: a logical function evaluated at a decision point

Path predicate: the set of predicates associated with a path

- may contain local variables that
  - play no role in selecting inputs that force a path to execute
  - cannot be selected independently of the input variables
  - are eliminated with symbolic execution

1.1 Symbolic Execution

Symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.

Key idea:

- evaluate the program on symbolic input values
- use an automated theorem prover to check whether there are corresponding concrete input values that make the program fail

1.2 Path Predicate Expression

Another name for an interpreted path predicate. Has the following attributes:

- no local variables
- a set of constraints in terms of the input vector and maybe constants
- by solving the constraints, inputs that force each path can be generated
- if a path predicate expression has no solution, the path is infeasible

Can organize the set of path predicates using a decision table.

2 Principles of Maintainable Test Code

Tests should be fast. For slower tests:

- use mocks or stubs
- redesign production code so slower pieces of code can be tested separately from fast pieces
- move slower tests to a different test suite that you can run less often

Tests should:

- be cohesive: test a single behaviour of the system
- be independent: not depend on oher tests to succeed
- be isolated: clean up their mess and not rely on existing data
- have a reason to exist: find bugs or document behaviour
- be repeatable: always give same results no matter how many times they are executed
- have strong assertions: assert that exercised code behaves exactly as expected and break on any slight change in the output
- break if behaviour changes: TDD can help
- have a single reason to fail: test code should help understand what caused the bug
- be easy to write: write good test infrastructure that does all setup and tear down
- be easy to read: all info should be clear enough
- be easy to change and evolve: changing should not be painful, less coupled to production code

3 Test Smells

Code smell indicates symptoms that of deeper problems in source code, hindering comprehensibility and maintainability.

3.1 Eager Test

When a test method checks several methods of the object to be tested, it is hard to read and understand so more difficult to be used as documentation.

Solution: separate test code into test methods that only test one method

3.2 Lazy Test

Occurs when several test methods check the same method using the same fixture.

Solution: join them together using inline method

3.3 Mystery Guest

When a test uses external resources, the test is no longer self-contained so difficult to understand tested functionality. Also this introduces hidden dependencies that can cause tests to fail.

Solution: use refactoring inline resource, or if external resources are needed, apply setup external resource to remove hidden dependencies.

3.3.1 Refactoring: Inline Resource

To remove the dependency between a test and some external resource, setup a fixture that holds the same content as the resource that is used to run the test.

3.3.2 Refactoring: Setup External Resource

If necessary for a test to rely on external resources, make sure the test explicitly creates or allocates these resources before test and releases them when done.

3.4 Resource Optimism

Test code that makes optimistic assumptions can cause non-deterministic behaviour in test outcomes (flaky tests).

Solution: use setup external resource to allocate or initialize all resources used.

3.5 Test Run War

Arise when tests run differently on different devices, caused by resource interference.

Solution: apply make resource unique to overcome interference

3.5.1 Refactoring: Make Resource Unique

Can solve issue of overlapping resource names by using unique IDs for all resources (timestamp).

3.6 General Fixture

When setup fixtures are too general and different tests only access part of the fixture, tests run more slowly.

Solution: use setup only for the part of fixture shared by all tests and put the rest of the fixture in the method that uses it.

3.7 Assertion Roulette

Comes from having a number of assertions in a test method that have no explanation, if one fails, it is unknown why.

Solution: add assertion explanations

3.8 Indirect Testing

A test class should test its counterpart in production code, not other classes.

Solution: move tests to appropriate class

3.9 For Tester Only

When a production class contains methods only used by test methods.

Solution: methods can be removed or are only needed to setup tests in which case they should be moved to test code

3.10 Sensitive Equality

Tests may depend on irrelevant details for equality checks.

Solution: introduce equality method

3.10.1 Refactoring: Introduce Equality Method

If an object structure needs to be checked for equality in tests, add an implementation for the equals method.

3.11 Test Code Duplication

Test code can have duplication of responsibility.

Solution: can be removed by extracting methods

3.12 Other Test Smells

Other examples are redundant assertions, unknown tests (without assertion), empty test, and duplicate assertions.