Writing Good Tests

Richard Warburton

@RichardWarburto | www.insightfullogic.com

Good Tests

Why should we care about test code quality?

Tests Are a Trade-Off

- We've seen the benefits

- Costs
  - Maintenance
  - Readability
  - Coupling

Tests are code, treat them like it.

Good Practices

What does a good test look like?

Well Named

Behavior not Implementation

DRY

Diagnostics

Why Name?

Executable Documentation

Maintenance

Readability

Naming Anti-Pattern 1

```
@Test
public void test1() {
```

Naming Anti-Pattern 2

```
@Test
public void espresso_beans() {


 …


@Test
public void restock() {
```

Good Naming

```
@Test
public void brewingEspressoConsumesBeans() {

 …

@Test
public void shouldRestockBeans() {
```

Naming Rules

Use domain terminology

Natural Language

Be descriptive

Behavior not Implementation

Behavior Anti-Pattern

```
assertEquals(5, object.internalState);

assertEquals(5, object.getInternalState());
```

Implementation

Exposing private state results in brittle and hard to maintain tests

Behavior

Assert about the result of an operation

You can change the implementation and the test still passes

Don't Repeat Yourself

Duplication Anti-Pattern

```
Cafe cafe = new Cafe();
cafe.restockBeans(7);
```

Magic Number Anti-Pattern

```
cafe.restockBeans(7);
```

Magic Number Fixed

```
cafe.restockBeans(ESPRESSO_BEANS);
```

Diagnostics

Size of List

```
List<Coffee> order = cafe.brewOrder(...);
assertTrue(order.size() == 1);

java.lang.AssertionError
```

Exposing the Value

```
List<Coffee> order = cafe.brewOrder(...);
assertEquals(1, order.size());

Expected :1
Actual :0
```

Exposing a Reason

```
List<Coffee> order = cafe.brewOrder(...);
assertEquals("Wrong quantity of coffee",
    1, order.size());

Wrong quantity of coffee
Expected :1
Actual :0
```

Before & After

Common code surrounding tests

JUnit Practices

JUnit helps us implement best practices

For example reduced duplication

Before and After tests

Annotations

```
@Before          // Before each test method runs

@After           // After each test method runs

@BeforeClass     // Before all tests in the class

@AfterClass      // After all tests in the class
```

Hamcrest

A compositional matcher library

Helps us avoid repetition and improve diagnostics

Matcher

A simple and general blob of logic used in assertions.

Example: A map contains *this* key

Compositional

Matchers can combine multiple matchers.

Example: a number is 5 or 7.

Summary

Summary

Problems

Naming, Behavior not Implementation, DRY, Diagnostics

Solutions

Use Junit & Hamcrest features to solve these problems