



University of
Sheffield



COM3529 Software Testing and Analysis

Unit Testing – Part 2

Professor Phil McMinn

How to Write Clear Unit Tests

JUnit v. Hamcrest Assertions

The default supplied JUnit assertions have some deficiencies:

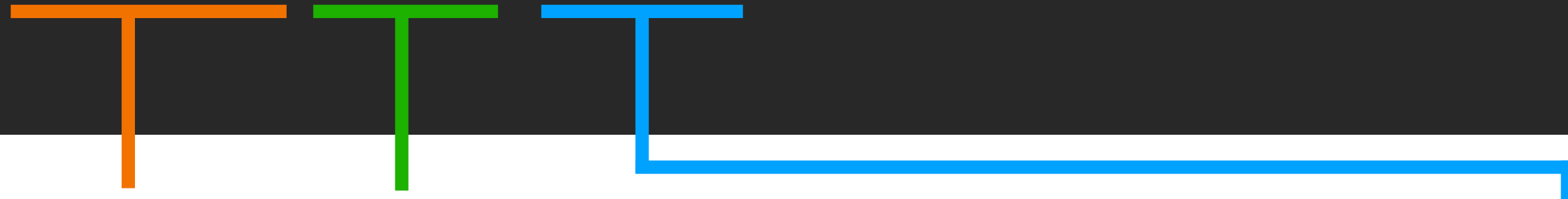
- **It's easy with the assertion method parameters to get expected and actual the wrong way round.** (I do it all the time!) It's not terribly consequential, which is why it's easy to do, but the wrong ordering will confuse other programmers.
- **A different style is required for differing expected-actual relationships – i.e. a different assertion method**
- The different assertion methods available are somewhat limited
- It's difficult to customise error messages

For these reasons, some programmers prefer the Hamcrest style of assertion.

Did You Notice We'd Switched to a Different Style of Assertion?

Hamcrest Assertions

```
@Test
public void isocetesTest() {
    Triangle.Type result = Triangle.classify(5, 10, 10);
    assertThat(result, equalTo(Triangle.Type.ISOSCELES));
}
```



Every assertion uses the generic **assertThat** method

The general assertion format maps more closely to natural language, making it more obvious that it's the **actual result** of the unit under test that goes first in the parameter order.

The relationship between actual and expected results is specified by a **matcher**, in this case, **equalTo**.

Hamcrest Matchers

Hamcrest has a plethora of “matchers”, like `equalTo`.

They can help write assertions involving a variety of types, including:

- Strings – e.g., can check if a string contains a substring, ignore case etc.
- Collections – whether an element is in a collection; what a collection contains, ignoring order etc.
- See <http://hamcrest.org/JavaHamcrest/javadoc/2.2/org/hamcrest/Matchers.html>

If the appropriate matcher is not available it's very easy to write your own.

See <https://www.baeldung.com/java-junit-hamcrest-guide>

Make Your Tests *Complete* and *Concise*

Ensure the test contains all the information needed for a reader to understand how it arrived at its result.

Ensure it contains no other irrelevant and distracting information.

A test case is *complete* when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

What's going on in this helper method?

Why RED? Where does that come from?

Make Your Tests *Complete*

A test case is **complete** when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
// An incomplete test!
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    makeMoves(c4);
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

What's going on in this helper method?

Why **RED**? Where does that come from?

Make Your Tests *Complete*

A test case is **complete** when its body contains all of the information a reader needs to understand how it arrives at its result.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Enumerating all the moves makes the method longer and prevents re-using the move sequence as a helper method, BUT makes it clearer what's going. Two columns of vertical pieces are being added to the board, and **RED** wins in column 0

Don't DRY Tests

DRY – Don't Repeat Yourself: engineer needs to update one piece of code rather than tracking down all instances.

Downside: Can make code less clear, **requires following chains of references.** This might be a small price to pay for making the code easier to work with...

... but the cost/benefit analysis plays out differently with tests:

- **We want tests to break when software changes**
- **Production code has the benefit of a test suite to ensure it keeps working when things get complex. Tests should stand on their own!**
(Something has gone wrong when tests need tests)

DAMP not DRY

DAMP: Descriptive And Meaningful Phrases

DAMP is not a *replacement* for DRY – it is complementary.

“Helper” methods can help make tests clearer by making them more concise – factoring out repetitive steps whose details aren’t relevant to the behaviour being tested.

But the refactoring should be done with an eye for make the tests more readable and descriptive, not solely to reduce repetition.

Don’t comprise clarity and conciseness.

Make Your Tests *Concise*

A test case is **concise** when it contains no other distracting or irrelevant information.

```
@Test
public void shouldEndGameWithWinnerWhenFourInARowVertically() {
    Connect4 c4 = new Connect4(7, 6);
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(2); // RED
    c4.makeMove(2); // YELLOW
    c4.makeMove(3); // RED
    c4.makeMove(3); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    c4.makeMove(1); // YELLOW
    c4.makeMove(0); // RED
    assertThat(c4.isGameOver(), equalTo(true));
    assertThat(c4.winner(), equalTo(Piece.RED));
}
```

Our test now includes a lot of moves that are not needed for the test scenario and make the board even harder to visualise and what the test outcome should be.

Don't Test Methods – Test Behaviours

The first instinct of many engineers is to try to match the structure of their tests to the structure of the code.

```
public Connect4(int cols, int rows) {  
    // ...  
}  
  
public boolean isGameOver() {  
    // ...  
}  
  
public Piece whoseTurn() {  
    // ...  
}  
  
public Piece getPieceAt(int col, int row) {  
    // ...  
}  
  
public void makeMove(int col) {  
    // ...  
}
```

```
@Test  
public void testConstructor() {  
    // ...  
}  
  
@Test  
public void testIsGameOver() {  
    // ...  
}  
  
@Test  
public void testWhoseTurn() {  
    // ...  
}  
  
@Test  
public void testGetPieceAt() {  
    // ...  
}  
  
@Test  
public void testMakeMove() {  
    // ...  
}
```



Don't Test Methods – Test Behaviours

But a single method may have more than one behaviour and/or some tricky corner cases that require more tests.

For example, `makeMove` can have several behaviours, depending on the state of the board.

One test for that method makes no sense, and is likely to not be very clear nor concise.

Better way: Write tests for each behaviour.

Testing Behaviour

A behaviour is a guarantee that a system makes about how it will respond to a series of inputs while in a particular state.

A behaviour can be expressed with “**given X when Y, then Z**”

For example:

Given a Connect4 board, with RED starting first

When RED has played a piece

Then it's YELLOW's turn next.

Writing Behaviour-Driven Tests

Behaviour-Driven tests tend to read more like natural language, so structure them accordingly.

```
@Test
public void shouldChangePieceAfterTurn() {
    // Given a Connect 4 Board, with RED starting first
    Connect4 c4 = new Connect4(7, 6);

    // When RED makes a move
    c4.makeMove(0);

    // Then it's YELLOW's turn next
    assertThat(c4.whoseTurn(), equalTo(Piece.YELLOW));
}
```

Name Tests after the Behaviour Being Tested

A good name describes the actions (the “when”) that are being tested and the expected outcome (the “then”), and sometimes the state to (the “given”).

A good trick is to start the name with “should”, e.g.

`shouldInitializeCorrectly`
`shouldChangePieceAfterTurn`
`shouldEndGameWithWinnerWhenFourInARowHorizontally` etc.

Don't Put Logic in Tests

Don't put conditionals or loops in tests, or logical operations.

Tests should read as simple statements of truth, *not chunks of code that also require tests!*

Here the test is trying to check every position of the board and ensure it is not set to a piece (i.e., it is null)

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(7, 6);

    // Then it's RED's turn
    assertEquals(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    for (int i=0; i < 7; i++) {
        for (int j=0; j < 6; j++) {
            assertEquals(c4.getPieceAt(j, j), nullValue());
        }
    }

    // Then the game is not over
    assertEquals(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertEquals(c4.winner, equalTo(null));
}
```

But oops, the developer made a mistake, checking `(j, j)` instead of `(i, j)`. The test won't fail, so the mistake is hard to spot.

Don't Put Logic in Tests

Don't put conditionals or loops in tests, or logical operations.

Tests should read as simple statements of truth, *not chunks of code that also require tests!*

Better just to initialise a smaller 2x2 board and explicitly check each position

```
@Test
public void shouldInitializeCorrectly() {
    // Given a new Connect 4 Board
    Connect4 c4 = new Connect4(2, 2);

    // Then it's RED's turn
    assertThat(c4.whoseTurn(), equalTo(Piece.RED));

    // Then the board has no piece in every position
    assertThat(c4.getPieceAt(0, 0), nullValue());
    assertThat(c4.getPieceAt(0, 1), nullValue());
    assertThat(c4.getPieceAt(1, 0), nullValue());
    assertThat(c4.getPieceAt(1, 1), nullValue());

    // Then the game is not over
    assertThat(c4.isGameOver(), equalTo(false));

    // Then there is no winner
    assertThat(c4.winner, equalTo(null));
}
```


Making Your Unit Tests Clear

- 1 Make your tests **concise** and **complete** (DAMP and not too DRY!)
- 2 Don't structure tests around methods – instead **structure around behaviours**
- 3 Use the **Given-When-Then** pattern for testing behaviour
- 4 **Name Tests after the Behaviour Being Tested**
- 5 **Don't put logic in tests**