



University of  
Sheffield



COM3529 Software Testing and Analysis

# Unit Testing

Professor Phil McMinn

# Unit Tests – Recap

- 1 Narrow in scope
- 2 Limited to a single class or method
- 3 Small in size

# Why Write Unit Tests?

**To prevent bugs** (obviously!)

But also **to improve developer productivity** since unit tests:

- 1 Help with implementation** – writing tests while coding gives quick feedback on code being written.
- 2 Should be easy to understand** when they fail – each test should be conceptually simple and focussed on a particular part of the system.
- 3 Serve as documentation** and examples to engineers on how to use the part of the system being tested (since written document gets hopelessly out of date very quickly).

**At Google, 80% of tests are unit tests. The ease of writing tests and the speed of running them mean that engineers run several thousand unit tests a day.**

# How to Write Good Unit Tests



The board is a 2D array of the **Piece** enum type

```
public enum Piece { RED, YELLOW; }
```

```
public class Connect4 {
```

```
    private static final int WINNING_SEQUENCE = 4;
```

```
    final Piece[][] board;
    final int cols, rows;
    Piece turn;
    boolean gameOver;
    Piece winner;
```

```
    public Connect4(int cols, int rows) {
        this.cols = cols;
        this.rows = rows;
        board = new Piece[cols][rows];
        turn = Piece.RED;
        gameOver = false;
        winner = null;
    }
```

**“package private” members**

**public  
methods**

```
public boolean isGameOver();  
  
public Piece winner();  
  
public Piece whoseTurn();  
  
public Piece getPieceAt(int col, int row);  
  
public void makeMove(int col);
```

**“package  
private”  
methods**

```
int firstAvailableRow(int col);  
  
boolean isValidCol(int col);  
  
boolean isValidRow(int row);  
  
boolean isBoardFull();  
  
boolean isGameWon();  
  
boolean isGameWon(int col, int row, int dCol, int dRow);
```

```
}
```

# A Testing Problem

To: p.mcminn@sheffield.ac.uk

From: student3529@sheffield.ac.uk

Subject: A Problem with Testing – Please help!!!

Dear Phil

I'm writing some unit tests to the code of my third year dissertation project. It's not written Java, but as you said in the last lecture, all the principles still apply, and equivalent tools exist, so I can follow all of your advice!

But then I added a new feature to my project, many of my tests broke. There wasn't a bug, but all the tests needed to be updated. Also since many of my tests were written yesterday, I couldn't actually remember what most of them were for or did. So I ended up throwing a lot of them away.

You said that automated tests help speed up development, but this took ages to sort out. I'm not sure I want to go through all that again.

What should I do?

Yours,  
Stu



# A Testing Solution

To: student3529@sheffield.ac.uk  
From: p.mcminn@sheffield.ac.uk  
Subject: Re: A Problem with Testing – Please help!!!

Dear Stu,

Have no fear.

Likely your tests made too many assumptions about the internal structure of your code, and how it works. This means you have to update the tests every time the code changes.

I'm going to be covering how **tests should focus on behaviour rather than implementation** in the next lecture, and **how to write clear tests**. Be sure to be there!

Best,  
Phil

# The Importance of Maintainability

**There are two key issues with this scenario:**

- 1 The unit tests were **brittle**. They broke in response to a harmless and unrelated change that introduced no real bugs.
- 2 The unit tests were **unclear**. It was difficult to understand how to fix the tests because it was not clear what the tests were doing in the first place.

This easily happens when there are multiple contributors to the code and its tests (with real life software projects tending to have many people working on them at once).

# How to *Not* Write Brittle Unit Tests

# Connect4

**To demonstrate examples of **good** and **bad** unit testing**, we're going to be looking at tests written for the `Connect4` class in the `uk.ac.shef.ac.uk.connect4` package of the COM3529 GitHub repository.

Instances of the `Connect4` class represent the state of a game of Connect4, including the positions of counters in the grid, whose turn it is etc.

*Everyone know how the game works?*

# Strive for Unchanging Tests

The key strategy for **preventing brittle tests** is to strive to write tests that **will not need to change** unless the project's requirements change:

- Internal refactorings should not change the tests.
- New features should leave existing ones unaffected.
- Bug fixes shouldn't require updates to tests.
- Behaviour changes: *these may require changes to tests.*

The board is a 2D array of the **Piece** enum type

```
public enum Piece { RED, YELLOW; }
```

```
public class Connect4 {  
    private static final int WINNING_SEQUENCE = 4;
```

```
    Piece[][] board;  
    int cols, rows;  
    Piece turn;  
    boolean gameOver;  
    Piece winner;
```

```
    public Connect4(int cols, int rows) {  
        this.cols = cols;  
        this.rows = rows;  
        board = new Piece[cols][rows];  
        turn = Piece.RED;  
        gameOver = false;  
        winner = null;  
    }
```

“package private” members



**public  
methods**

```
}  
  
public boolean isGameOver();  
  
public Piece winner();  
  
public Piece whoseTurn();  
  
public Piece getPieceAt(int col, int row);  
  
public void makeMove(int col);
```

**“package  
private”  
methods**

```
int firstAvailableRow(int col);  
  
boolean isValidCol(int col);  
  
boolean isValidRow(int row);  
  
boolean isBoardFull();  
  
boolean isGameWon();  
  
boolean isGameWon(int col, int row, int dCol, int dRow);  
  
}
```

# Two Different Tests

Our implementation of Connect4 needs to obey gravity. If a player drops a piece into a column, we need to ensure it ends up on the right row.

The first piece will drop to the first row. The second piece in the same column will drop to the second row, etc.

**I'm now going to test this. I'm going to show you two different tests.**

One tests the implementation, one tests behaviour.

**Which one is more likely to have to change in future (i.e., be brittle)?**



# Testing Implementation

Directly sets  
member variable

```
@Test
public void testFirstAvailableRow() {
    Connect4 c4 = new Connect4(7, 6);
    c4.board[0][3] = Piece.RED;
    assertThat(c4.firstAvailableRow(0), equalTo(4));
}
```

Verifies implementation using package-private method

# Testing Behaviour

```
@Test
public void shouldPlaceCounterAboveLast() {
    Connect4 c4 = new Connect4(7, 6);

    c4.makeMove(0); // RED
    assertEquals(c4.getPieceAt(0, 0), Piece.RED);

    c4.makeMove(0); // YELLOW
    assertEquals(c4.getPieceAt(0, 1), Piece.YELLOW);

    c4.makeMove(0); // RED
    assertEquals(c4.getPieceAt(0, 2), Piece.RED);

    c4.makeMove(0);
    assertEquals(c4.getPieceAt(0, 3), Piece.YELLOW);
}
```

This test is using the public API, to the extent of almost playing a game of Connect4.

The resulting behaviour (a change to the board) is checked using a **public** method

# Testing Implementation

Directly sets  
member variable

```
@Test
public void testFirstAvailableRow() {
    Connect4 c4 = new Connect4(7, 6);
    c4.board[0][3] = Piece.RED;
    assertThat(c4.firstAvailableRow(0), equalTo(4));
}
```

Verifies implementation using package-private method

**What happens to this test if we decide to implement the board differently?** (E.g., swap rows and columns in array, refactor the board out into a separate class entirely, etc.)

# Preventing Brittle Tests

Strive for **unchanging tests** by:

- 1 Test calling **public** methods only.
- 2 Verify what results **are**, not **how** they are achieved.

If you concentrate on testing **implementation** as opposed to **behaviour** you will get **brittle tests**.

**So always prefer to test against behaviour.**

# 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 Already Been Using a Different Style of Assertion This Lecture?*

# 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 rows 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**