COM3529 Software Testing and Analysis

Unit Testing – Part 1

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and examples to engineers on how to use the part of the system being tested (since written document gets hopelessly out of date very quickly).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

“package private” methods

```
    public boolean isGameOver();

    public Piece winner();

    public Piece whoseTurn();

    public Piece getPieceAt(int col, int row);

    public void makeMove(int col);

    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

“package private” methods

```
    }

    public boolean isGameOver();

    public Piece winner();

    public Piece whoseTurn();

    public Piece getPieceAt(int col, int row);

    public void makeMove(int col);

    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
    assertThat(c4.getPieceAt(0, 0), equalTo(Piece.RED));

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

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

    c4.makeMove(0);
    assertThat(c4.getPieceAt(0, 3), equalTo(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 by 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.