COM3529 Software Testing and Analysis

Test Automation

Professor Phil McMinn

A Testing Problem

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

Dear Phil

You asked last lecture whether we liked testing. To be honest, no I don’t, I find manually trying out my software with inputs really dull!

Is this the right module for me? Please help!

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.

I agree, manual testing is really dull! Automated testing however, is much more interesting, and is more like development. In fact, we should be writing tests while developing! Finding problems while developing is much more fun than finding them once the software is deployed. No more late nights spent debugging!

I'm going to be covering **writing automated tests** in the next lecture. Be sure to be there!

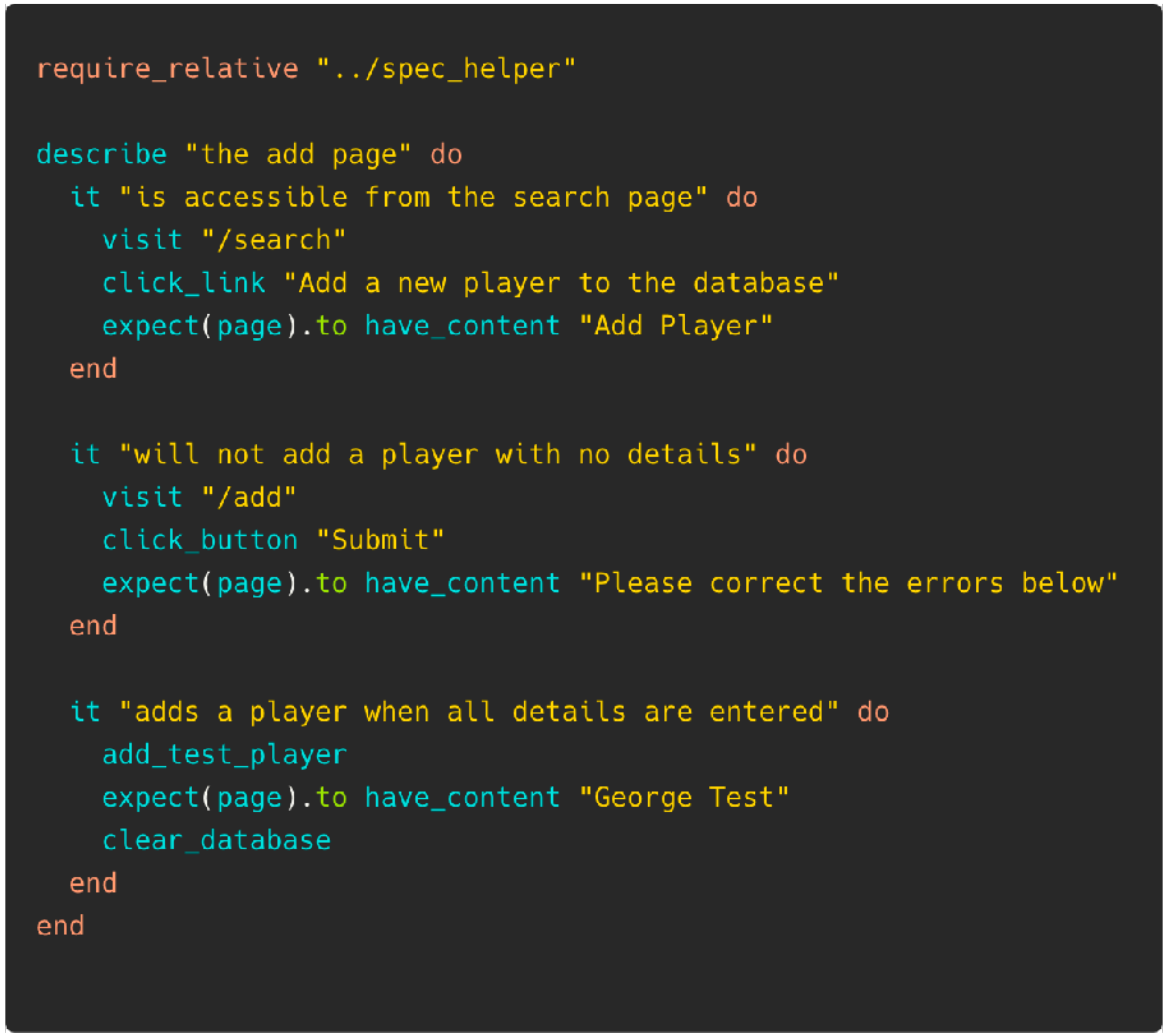
Best,
Phil

What do you understand by the phrase *automated test*?

JUnit example

```java
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertEquals;

public class TriangleTest {

    @Test
    public void shouldClassifyEquilateral() {
        Triangle.Type result = Triangle.classify(10, 10, 10);
        assertEquals(Triangle.Type.EQUILATERAL, result);
    }

    // ...
```

RSpec example

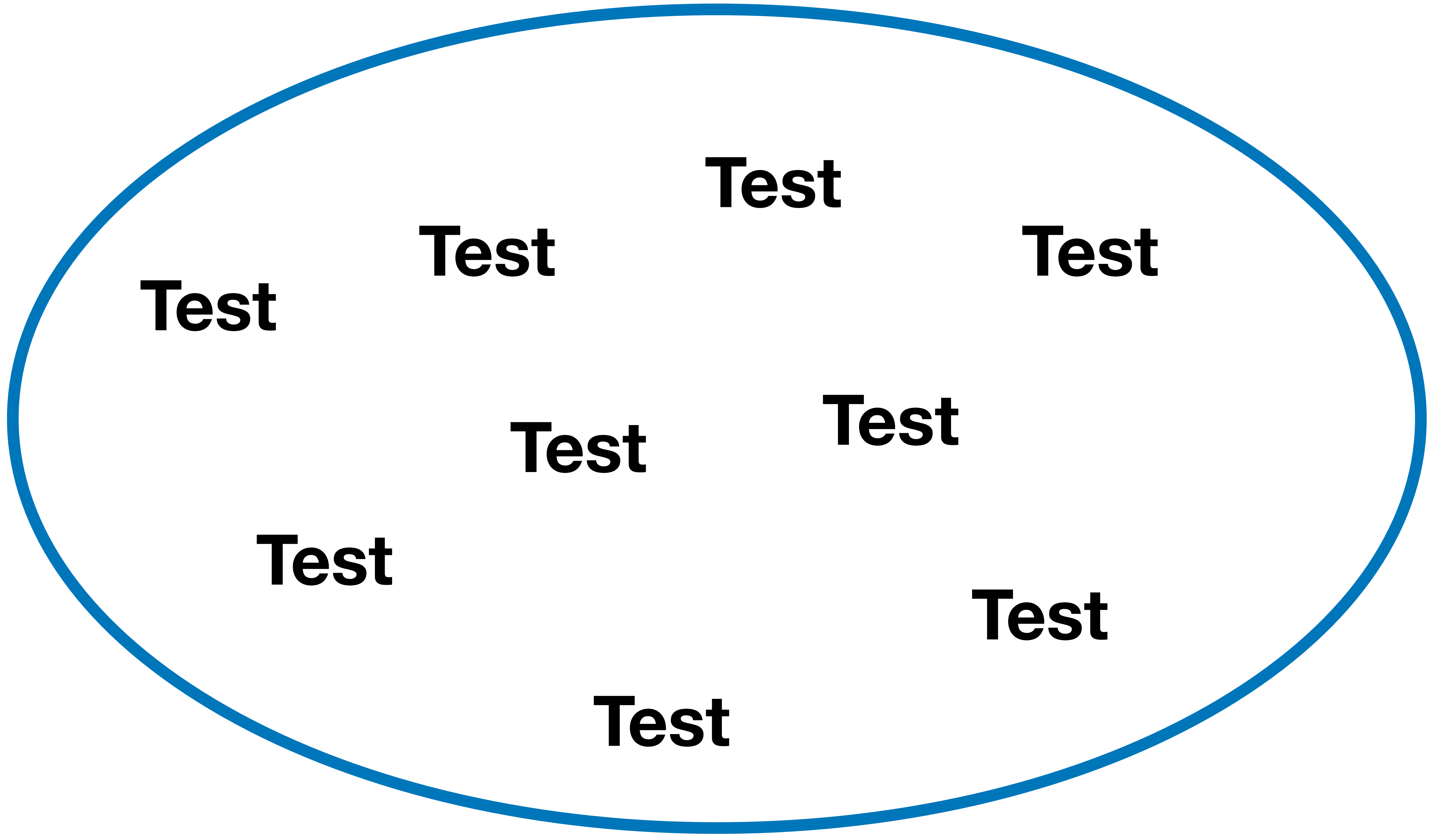


```ruby
require_relative "../spec_helper"

describe "the add page" do
  it "is accessible from the search page" do
    visit "/search"
    click_link "Add a new player to the database"
    expect(page).to have_content "Add Player"
  end

  it "will not add a player with no details" do
    visit "/add"
    click_button "Submit"
    expect(page).to have_content "Please correct the errors below"
  end

  it "adds a player when all details are entered" do
    add_test_player
    expect(page).to have_content "George Test"
    clear_database
  end
end
```

Ingredients of an Automated Test Case

1. The inputs needed to put the software into the right state for the test
2. The actual test case inputs
3. The expected results of the test
4. Reset of the system state

A Test Suite – A Set of Tests



Ideally, the tests can be executed in any order

The Dawn of Test Automation

Testing has always been part of programming

> … when you wrote your first program, you almost certainly tried it out with some sample data
>
> **For a long time, this was the state of the art in industrial practice!**

In the early 2000s, software development practices started to change

> Software systems got too big and too complex for manual testing to remain an effective and efficient way to ensure they were working and **remained working**

Testing at the Speed of Modern Software Development

Software systems are growing larger and evermore complex.

A typical application or service at Google, for example, is made up of thousands or millions of lines of code.

The ability for humans to **manually validate every behaviour in a system** has been **unable to keep pace with the explosion of features and platforms in most software**.

Testing at the Speed of Modern Software Development

Imagine what it would take to manually test the functionality of Google search – every time the code was changed.

> … not just web search, but images, flights, movie times etc.
>
> Then multiply that for every language, country, and device that must be supported.
>
> Then add in factors like accessibility and security.

Manual testing does not scale. We need automation.

Developer-Driven Automated Testing

The idea of coding automated tests (e.g., in JUnit) as a means of improving productivity and velocity may seem antithetical.

After all, the act of writing tests can take just as long (if not longer!) than implementing a feature in the first place … right?

On the contrary!

> In industry, investing in software tests provides several key benefits to developer productivity.

Less Debugging

Tested code has fewer defects when it is submitted.

Crucially, it also has fewer defects throughout its existence – since code tends to be updated during its lifetime.

> … it will be changed by other teams and even automated code maintenance systems.
>
> Changes to code, or its dependencies, can be quickly detected by an automated test and rolled back before the problem reaches production.

Increased Confidence in Changes

Projects with good tests can be modified with confidence since all the important behaviours of their projects are continuously being verified.

These projects *encourage* refactoring.

After a change, we can re-run the automated tests to ensure we didn’t break any of the existing functionality.

Improved Documentation

Software documentation is notoriously unreliable!

Clear, focused tests that exercise one behaviour at a time function as executable documentation.

Thoughtful Design

Writing tests for new code is a practical means of exercising the API design of the code itself.

If new code is difficult to test, it is often because the code being tested has too many responsibilities or difficult-to-manage dependencies.

Well-designed code should be modular, avoiding tight coupling and focusing on specific responsibilities.

Fixing design issues early means less rework later.

Fast, High Quality Releases

With a healthy automated test suite, teams can release new versions of their application with confidence.

Many large projects, involving hundreds of engineers and thousands of code changes submitted every day, involve very short release cycles – often every day.

This would not be possible without automated testing.

Benefits of an Automated Test Suite

1. Less Debugging
2. Increased Confidence in Changes
3. Improved Documentation
4. Thoughtful Design
5. Allows for Fast, High Quality Software Releases

COM3529 Software Testing and Analysis

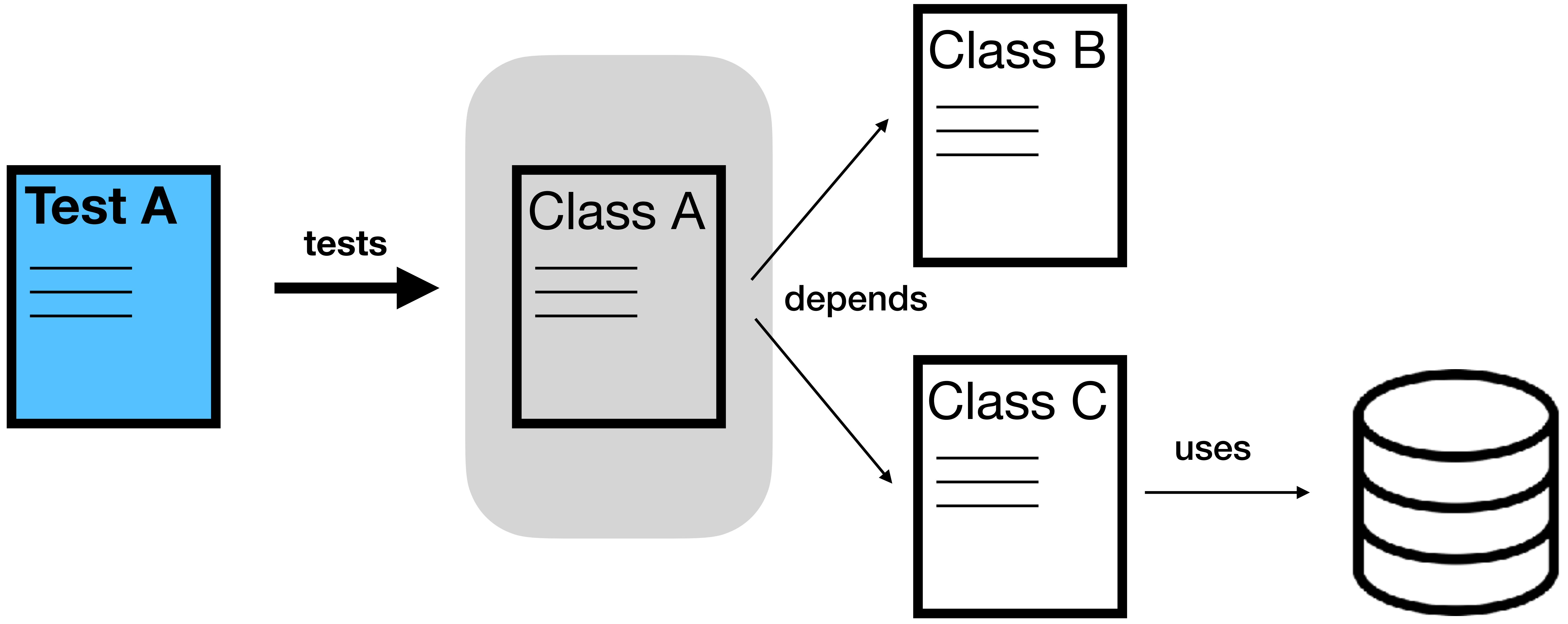
Test Scope

Professor Phil McMinn

Unit Testing

A unit is an individual component of a system, such as a class or an individual method.

Testing units in isolation is called **unit testing.**



Unit Testing

A unit is an individual component of a system, such as a class or an individual method.

Testing units in isolation is called **unit testing.**

- **Fast**
- **Easy to control**
- **Easy to write**

- **Lack reality**
- **Cannot catch all bugs**
  (e.g. interactions with other components or services)

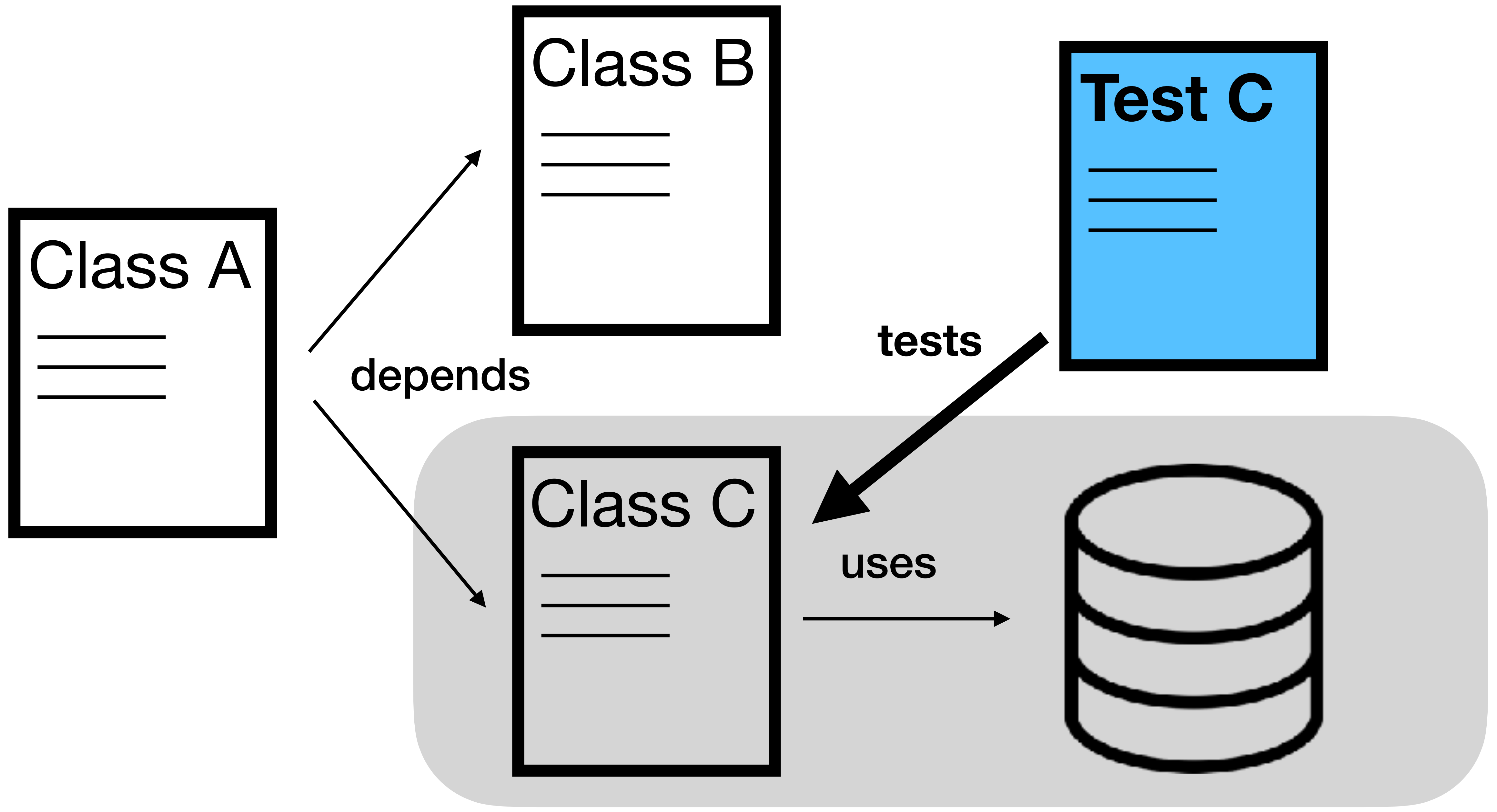
Unit tests are a very useful type of test but are often insufficient on their own.

Integration Tests

Testing in isolation is not enough. Sometimes code goes “beyond” the system’s borders and uses other (often external) components – for example, a database.

Integration tests test the integration between our code and that of external parties.

Example: Testing methods that access a database via SQL queries. Do our methods obtain the right data from the database?



Integration Tests

Testing in isolation is not enough. Sometimes code goes “beyond” the system’s borders and uses other (often external) components – for example, a database.

Integration tests test the integration between our code and that of external parties.

Example: Testing methods that access a database via SQL queries. Do our methods obtain the right data from the database?

- **Can capture integration bugs**
- **Less complex than writing a system test that goes through the entire system, including components we do not care about**

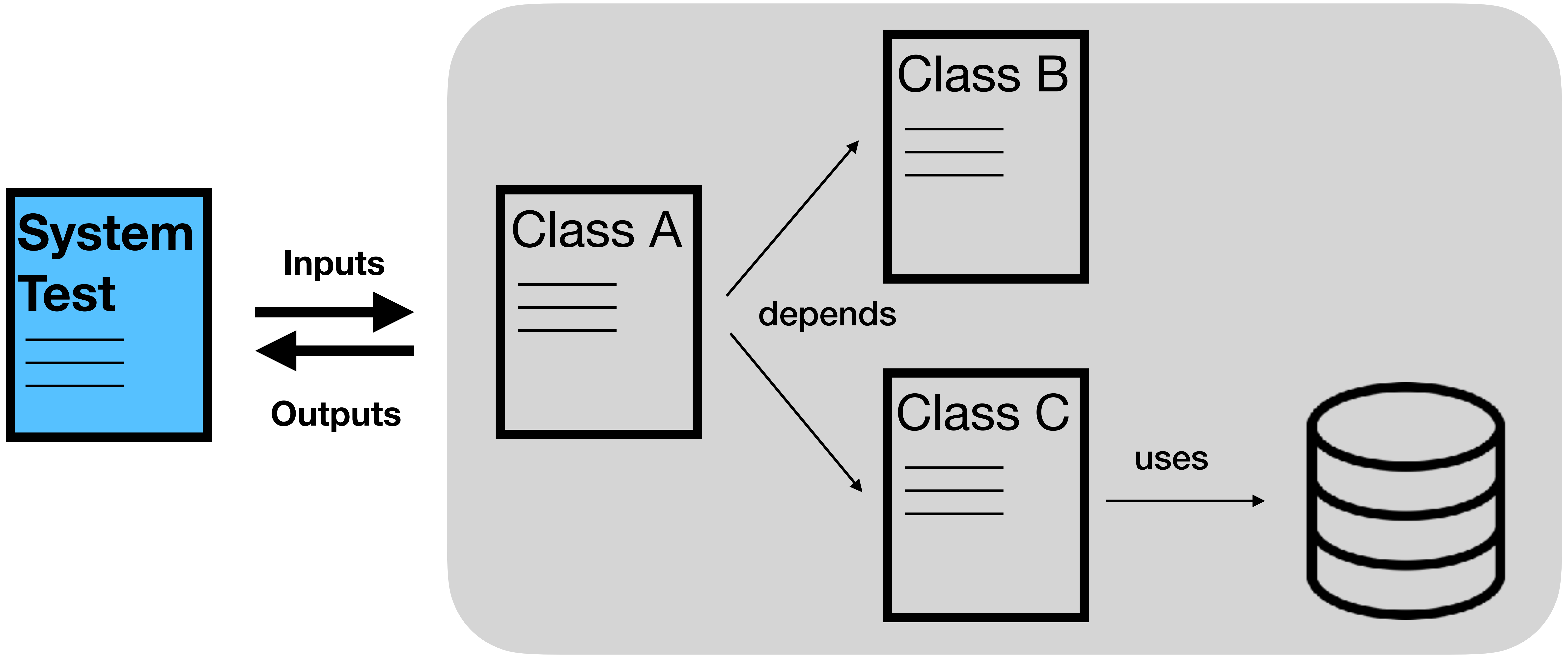
- **Hard to write**, for example:
  - Need to use an isolated instance of the database
  - Put it into a state expected by the test
  - Reset the state afterwards

System Tests

To get a more realistic view of the software we should also perform more realistic tests with it – with all its database, front-end, and other components.

We do not care about how the system works from the inside.

We care that given certain inputs, certain outputs are provided by the system.



System Tests

To get a more realistic view of the software we should also perform more realistic tests with it – with all its database, front-end, and other components.

We do not care about how the system works from the inside.

We care that given certain inputs, certain outputs are provided by the system.

- **Realistic**
  (when the tests perform similarly to the end user, the more confident we can be that the system will work correctly for all end users)

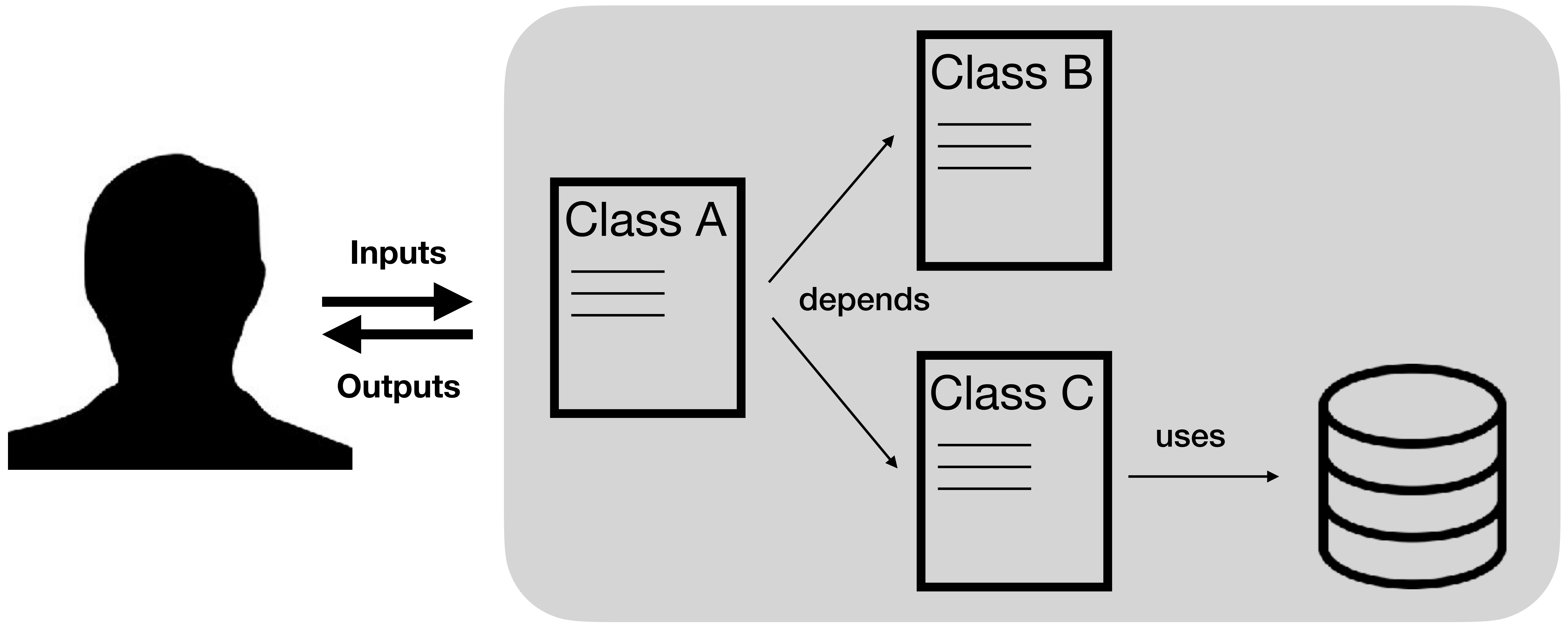
- **Slow!**
- **Hard to write**
  (lots of external services to account for)
- **Prone to Flakiness**

Manual Tests

Not everything can be tested easily in an automated fashion, particularly where there are qualitative judgements (e.g., the quality of a search engine's results).

Furthermore, we may need to explore real system behaviour to know what automated tests to write.

Manual tests are system tests performed manually by a human.



Manual Tests

Not everything can be tested easily in an automated fashion, particularly where there are qualitative judgements (e.g., the quality of a search engine's results).

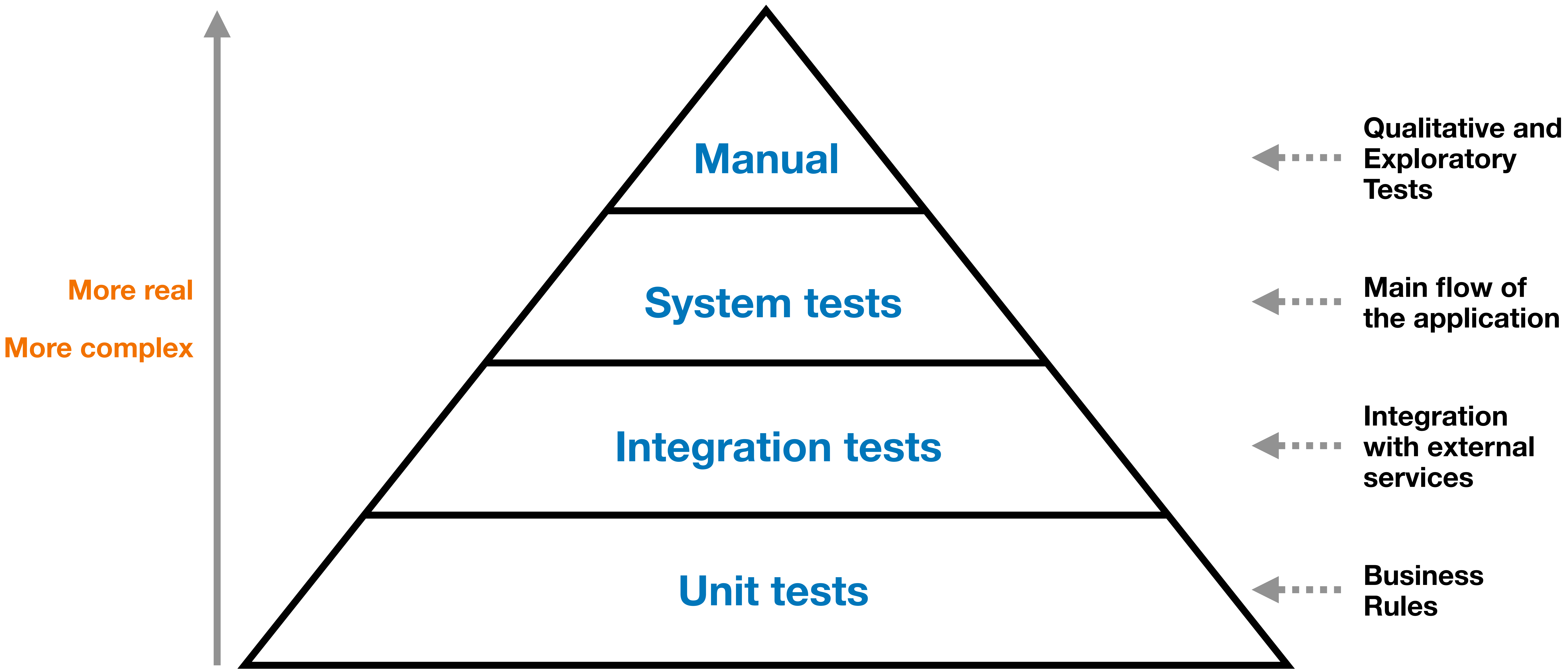
Furthermore, we may need to explore real system behaviour to know what automated tests to write.

Manual tests are system tests performed manually by a human.

- **Real**
  (The tester is acting as an end-user, actually using the system)

- **Time-consuming**
- **Difficult to reproduce**
- **Tedious**

The Test Triangle


Manual
System tests
Integration tests
Unit tests
More real
More complex
Qualitative and Exploratory Tests
Main flow of the application
Integration with external services
Business Rules

University of Sheffield

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.

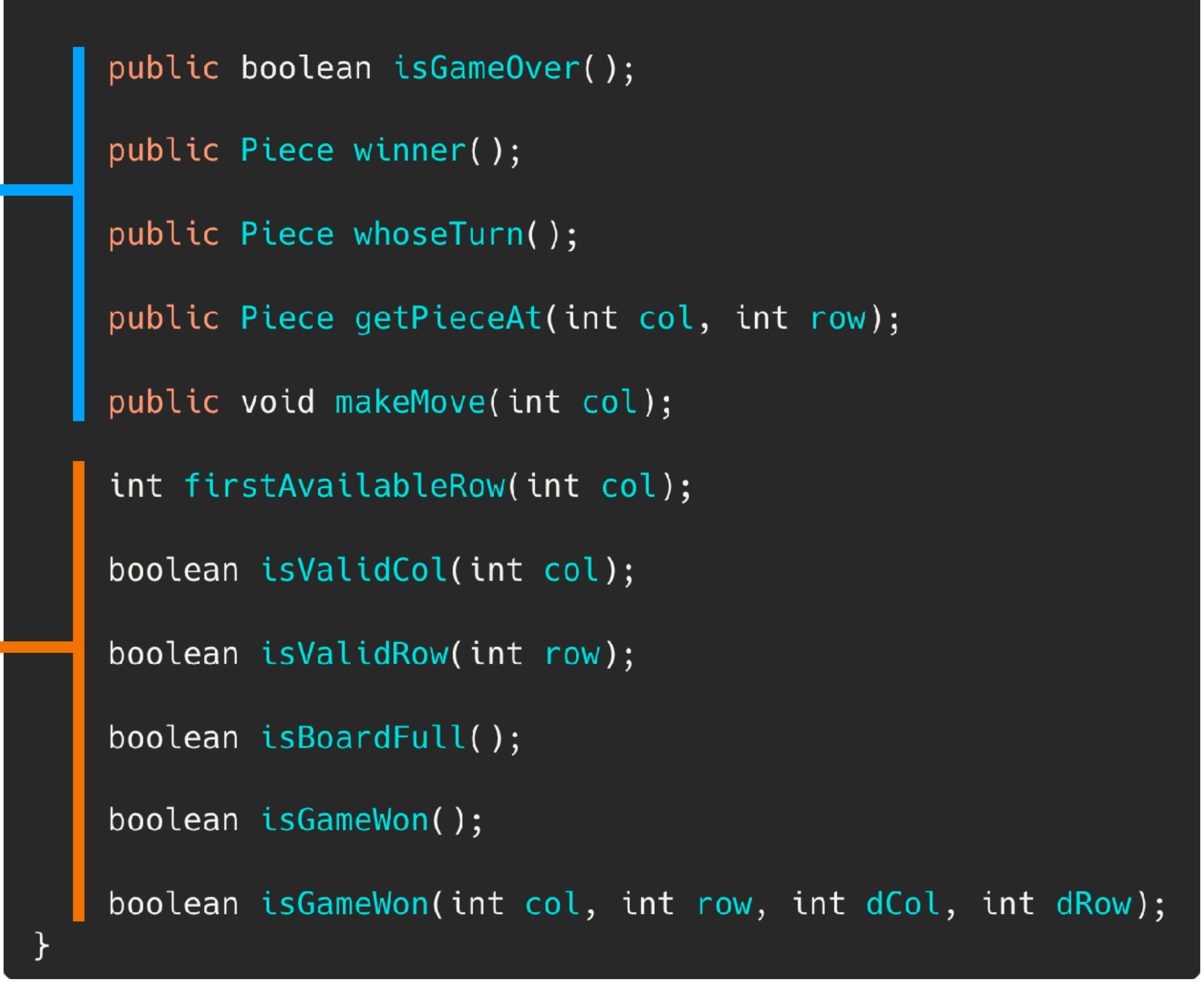
How to Write **Good** Unit Tests

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 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);
}
```

public methods

“package private” methods

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

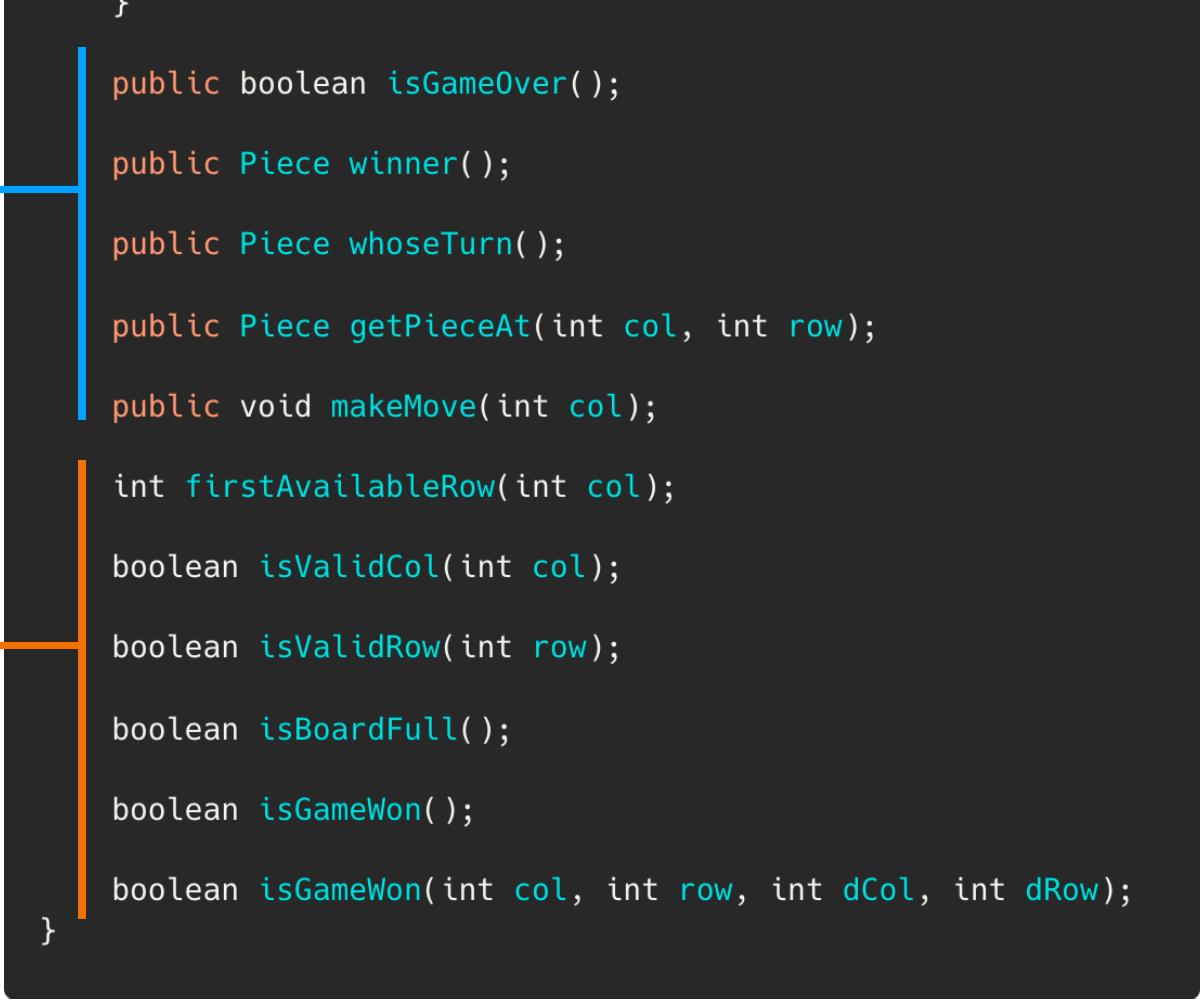
```
public enum Piece { RED, YELLOW; }
```

```
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)?

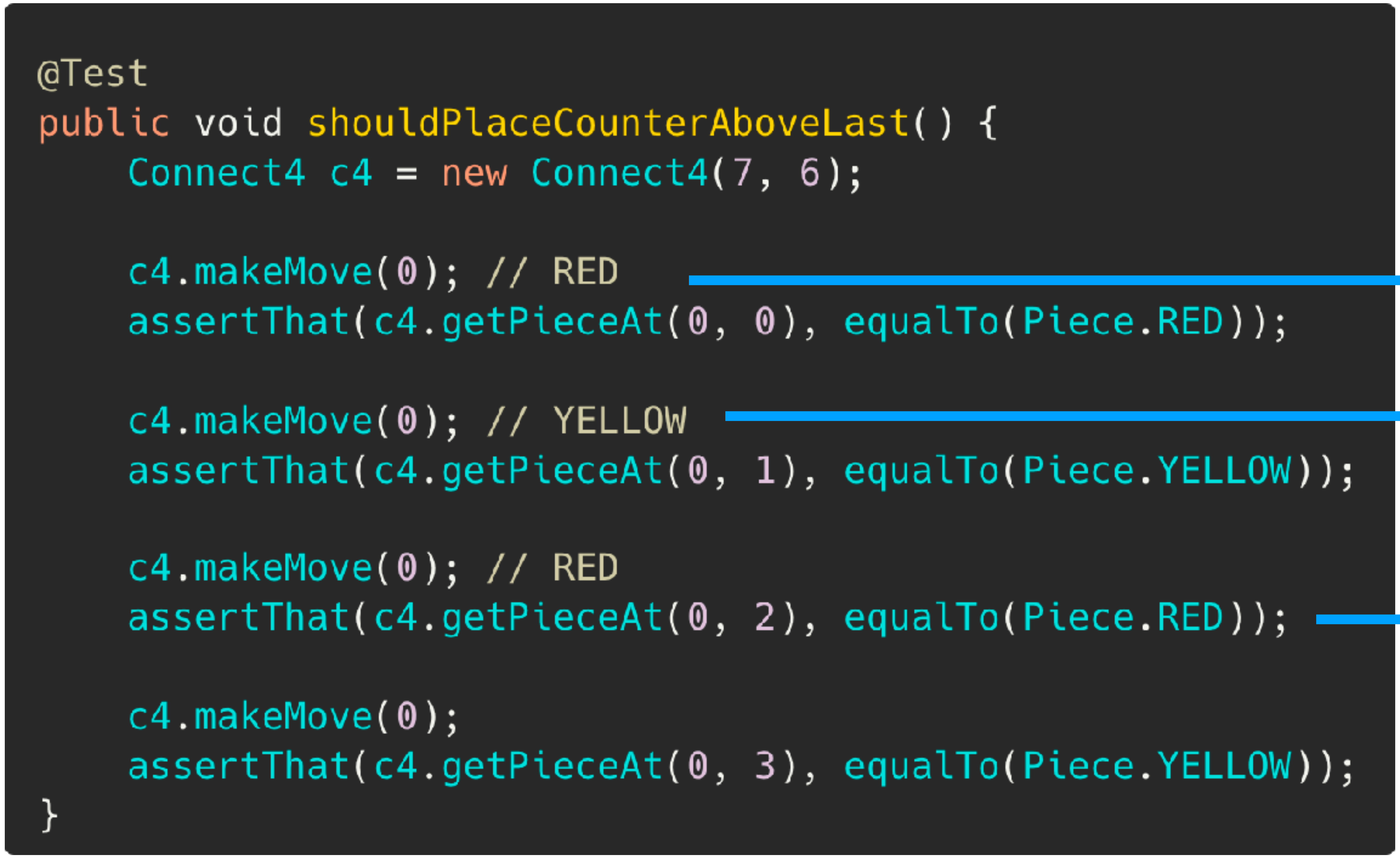
Testing Implementation

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

Directly sets member variable

Verifies implementation using package-private method

Testing Behaviour

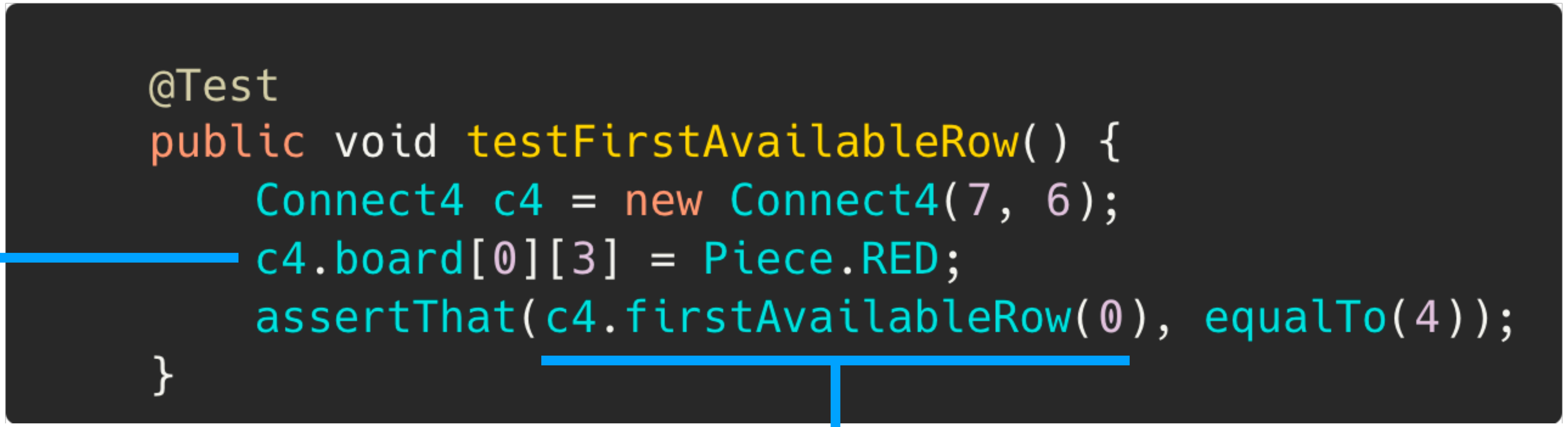


```java
@Test
public void shouldPlaceCounterAboveLast() {
    Connect4 c4 = new Connect4(7, 6);

    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



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

Directly sets member variable

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.