

Unit Testing

The Definitive Guide

© Copyright Diffblue Ltd 2021

Table of Contents

Introduction	4
Chapter 1: How to write your first unit test	7
Chapter 2: How to measure coverage	14
Chapter 3: How to build a complete test suite	21
Chapter 4: Mocking in unit tests	25
Chapter 5: Finding the time and motivation to unit test	30
Chapter 6: How to avoid common mistakes	33
Chapter 7: How automated unit tests speed up continuous integration	35
Chapter 8: How to deliver on the promises of DevOps	38
Epilogue: Why imperfect tests are better than no tests	42

Introduction

Unit testing is really important for code quality (and for making your life easier when hunting for bugs), but if you've never done it before, where do you begin? To help out, we're sharing the Definitive guide to Unit Testing in Java. Throughout the course of the tutorial, the complexity of the code will increase, but the tutorials will focus on the unit testing that is used.

Before we start, it's useful to define what exactly a unit test is. There are many different types of testing, and unfortunately, there isn't a clean line between the different definitions. Often, on forums like StackOverflow, there are questions asking something like, "What makes a test a unit test vs integration test?"

The short answer to these questions is usually, "Well, it depends on where exactly you draw the line between different types of testing." The important thing is that within your team or organization, you have a clear definition that avoids any potential confusion or conflict.

With that said, here are the characteristics that generally describe different types of testing.

Unit Testing

This is testing the smallest testable part of the code base. In Java, you could consider a standalone method within a class as a unit, or you could consider the class itself as the unit. There certainly should not be any interaction with services outside of the product (e.g. Databases, Kafka, Web Servers). These would typically be stubbed or mocked.

Unit tests often **have a huge overlap with component testing**.

Component Testing

This is testing a single component of the solution. Typically, in Java, a module would be considered a component. The focus is on whether the component delivers the required functionality for the rest of the solution. It should not rely on other modules, as these would typically be mocked or stubbed.

Integration Testing

Testing the interaction between two or more components in the product. One of the key parts of an integration test is that it is testing, or relying on, the actual behavior of an interface between two components. Essentially, unlike component or unit testing, a change in either component can cause the test to fail.

End-to-end Testing

This is testing the entire solution as the user is expected to use the system. The testing should be done via the user interface. This will most likely involve the use of specific automation tools, such as Selenium, for interacting with a Web UI.

Acceptance Testing

Typically used by a client or stakeholder, acceptance testing is a set of tests that prove the product is suitable for its intended use. Acceptance tests are often also end-to-end tests, but with the specific purpose of preventing the release or handover of a solution if they fail.

Chapter 1: How to write your first unit test

Let's use a web-based Java Tic-Tac-Toe game to demonstrate how to write your first unit test. First, here's the code that checks to see if anyone has won the game:

```
public Player whoHasWon() {  
    ArrayList<Coordinate[]> winningPositions = new  
    ArrayList<>(); // rows  
    winningPositions.add(new Coordinate[] {new  
    Coordinate(0,0), new  
    Coordinate(1,0), new Coordinate(2,0)});  
    winningPositions.add(new Coordinate[] {new  
    Coordinate(0,1), new  
    Coordinate(1,1), new Coordinate(2,1)});  
}
```

```

        winningPositions.add(new Coordinate[] {new
Coordinate(0,2), new
Coordinate(1,2), new Coordinate(2,2)}); // columns
        winningPositions.add(new Coordinate[] {new
Coordinate(0,0), new Coordinate(0,1), new Coordinate(0,2)});
        winningPositions.add(new Coordinate[] {new
Coordinate(1,0), new Coordinate(1,1), new Coordinate(1,2)});
        winningPositions.add(new Coordinate[] {new
Coordinate(2,0), new Coordinate(2,1), new Coordinate(2,2)});
        // diagonals
        winningPositions.add(new Coordinate[] {new
Coordinate(0,0), new Coordinate(1,1), new Coordinate(2,2)});
        winningPositions.add(new Coordinate[] {new
Coordinate(2,0), new Coordinate(1,1), new Coordinate(0,2)});
        for (Coordinate[] winningPosition : winningPositions) {
            if (getCell(winningPosition[0]) ==
getCell(winningPosition[1])
&& getCell(winningPosition[1]) ==
getCell(winningPosition[2])) {
                if (getCell(winningPosition[0]) != null) {
                    return getCell(winningPosition[0]);
                }
            }
        }
        return null;
    }
}

```

Next, we should write some unit tests to ensure the behavior is correct and doesn't break in the future.

We will need to introduce a test framework into the project. For this example, we will use JUnit. To do this in Maven, simply add the following to your dependencies in the pom.xml

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.8.2</version>
  <scope>test</scope>
</dependency>
```

Next, it's time to create a class for the unit tests to exist in. We want the tests to be in the same package as the class that we are testing. In our example, this is package `com.diffblue.javademo.tictactoe`;

Which means that the file should be in `com/diffblue/javademo/tictactoe`. For a Maven project, this needs to be prefixed with `src/test/java`

Therefore, the path from the project root is `src/test/java/com/diffblue/javademo/tictactoe`.

This means that all the test classes are separated nicely from the main source code.

Finally, the class should follow the format `<class under test>Test` which, for this example, means the class will be `BoardTest` and the file will be `BoardTest.java`. Note: when running tests with Maven, the default is to search for classes that are suffixed with `Test`.

With all that in mind, create a class:

```
package com.diffblue.javademo.tictactoe;
public class BoardTest {
}
```

For this example, let's create a test to check that detecting a player winning through the top row works correctly.

Create a method for this test:

```
package com.diffblue.javademo.tictactoe;
import org.junit.Test;
public class BoardTest {
    @Test
    public void playerOTopRow() {
        // Arrange
        // Act
        // Assert
    }
}
```

This is slightly different to the methods that you have written in your source code.

First there is an annotation to say that this is a test:

```
@Test
```

Also, note the relevant import. This will tell the tools that this method is a test. If you are using IntelliJ or Eclipse, you will see that you can now run this method as a test.

This is a good time to point out that unit tests are designed to protect against future mistakes. This means that your test needs to be easy to read and understand both by other developers and your future self. To split up tests to aid in readability, let's add three comments: arrange, act and assert.

Now to start building out the test!

We need to set up an environment where the top row of the board is Naughts:

```
package com.diffblue.javademo.tictactoe;
import org.junit.Test;
public class BoardTest {
    @Test
    public void playerOTopRow() {
        // Arrange
        Board myBoard = new Board();
        myBoard.setCell(new Coordinate(0,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(0,1), Player.CROSS);
        myBoard.setCell(new Coordinate(1,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(1,1), Player.CROSS);
        myBoard.setCell(new Coordinate(2,0), Player.NOUGHT);
        // Act
        // Assert
    }
}
```

Now we have a board that has Naughts winning through the top row. Next, call the method `whoHasWon()` and collect the result.

```
package com.diffblue.javademo.tictactoe;
import org.junit.Test;
public class BoardTest {
    @Test
    public void playerOTopRow() {
        // Arrange
        Board myBoard = new Board();
        myBoard.setCell(new Coordinate(0,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(0,1), Player.CROSS);
        myBoard.setCell(new Coordinate(1,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(1,1), Player.CROSS);
        myBoard.setCell(new Coordinate(2,0), Player.NOUGHT);
        // Act
        Player result = myBoard.whoHasWon();
    }
}
```

```
        // Assert
    }
}
```

Now we have a test that sets up the environment and calls the method under test.

There is one final step to complete the test: add an Assert. The assert is the key part to the test, it is the thing that is checked to say whether the test has passed or failed. Here, we are checking that result is naught.

Here is our complete test:

```
package com.diffblue.javademo.tictactoe;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class BoardTest {
    @Test
    public void playerOTopRow() {
        // Arrange
        Board myBoard = new Board();
        myBoard.setCell(new Coordinate(0,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(0,1), Player.CROSS);
        myBoard.setCell(new Coordinate(1,0), Player.NOUGHT);
        myBoard.setCell(new Coordinate(1,1), Player.CROSS);
        myBoard.setCell(new Coordinate(2,0), Player.NOUGHT);
        // Act
        Player result = myBoard.whoHasWon();
        // Assert
        assertEquals("Player O didn't win in the top row",
            Player.NOUGHT, result);
    }
}
```

Looking at the assert, note the message (first argument to the assert). When a test fails, this message is printed in the results. This can give a clear indication of what has gone wrong to the person debugging the tests.

Another note: the order of the arguments is the expected result first, and then the actual result. Because tools will include the expected and the actual results in the output, it is important to avoid confusion by getting these correct.

Having finished the test, we can now run all the tests using `mvn test` and we will see that our test passes:

```
[INFO] -----  
[INFO] TESTS  
[INFO] -----  
[INFO] Running com.diffblue.javademo.tictactoe.BoardTest  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0,  
      Time elapsed: 0.018 s - in  
      com.diffblue.javademo.tictactoe.BoardTest  
[INFO]  
[INFO] Results:  
[INFO]  
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
[INFO]
```

The full source code for this tutorial is available [here](#).

Congrats! With that, you've written your first test. In the next chapter, we'll discuss how to measure the code coverage of the tests you write.

Chapter 2: How to measure coverage

Now that you've learned the basics of unit testing and have created your first unit test, you're probably wondering how several of these tests can be combined to create a comprehensive test suite (and how many of these tests you'll need to write). In this chapter, we'll talk about code coverage and how to measure it.

Let's begin!

How many unit tests should I write?

This is a common question for new unit testers. There are many schools of thought about the answer, but the rule of thumb is, "enough tests to ensure that serious regressions are not introduced as the code is modified." So how is this measured?

The most commonly used metric is code coverage. **It's not perfect**, but it does tell you how much of the code you've written is covered by tests. It is widely known that high code coverage alone is not necessarily good;

developers can write buggy code and then unit tests that pass with the buggy code. However, low code coverage is definitely bad, because it means your test suite is unlikely to catch issues that are introduced as the code is modified.

Based on this, it makes sense to measure the code coverage for test suites to help us identify code that is currently untested, as well as code that's unused or potentially dead. There are a number of tools available for measuring code coverage, e.g. [Cobertura](#) and [JaCoCo](#).

Getting Started

For the purposes of this tutorial, let's use JaCoCo. Code coverage tools work by instrumenting the class files so they can record which lines are executed. With JaCoCo, there are two types of instrumentation:

1. **Default:** This is where the class files are instrumented on-the-fly as they are called. The advantage of this is that it is the simplest to set up.
2. **Offline instrumentation:** This is where all the class files are instrumented before running any of the tests. The advantage here is that coverage will be recorded in more circumstances. See the documentation for more details.

Given that offline instrumentation can give coverage information in more circumstances than the default, let's set this up.

The first thing we need to do is to add the dependency to our pom.xml:

```
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.3</version>
  <scope>test</scope>
</dependency>
```

Then we need to add the surefire plugin to the 'build, plugins' section of our pom.xml:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M3</version>
  <configuration>
    <systemPropertyVariables>
      <jacoco-agent.destfile>
        ${jacoco.exec}
      </jacoco-agent.destfile>
    </systemPropertyVariables>
  </configuration>
</plugin>
```

Note that we have specified a location for the jacoco.exec file. This is the raw coverage file that is produced by JaCoCo.

Next, we will add the JaCoCo plugin to the build, plugins section of our pom.xml:

```

<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.3</version>
  <executions>
    <execution>
      <id>instrument</id>
      <phase>process-test-classes</phase>
      <goals>
        <goal>instrument</goal>
      </goals>
    </execution>
    <execution>
      <id>restore-instrumented-classes</id>
      <phase>test</phase>
      <goals>
        <goal>restore-instrumented-classes</goal>
      </goals>
    </execution>
    <execution>
      <!-- Ensures that the code coverage report for unit
      tests is created after unit tests have been run. -->
      <id>post-unit-test</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
      <configuration>
        <!-- Sets the output directory for the code coverage
        report. -->
        <outputDirectory>$/jacoco</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>

```


This is the bit that defines the process, essentially split into three parts:

1. Instrument the classes being tested
2. Restore instrumented classes
3. Generate code coverage report

Step one happens before the unit tests are run. Steps two and three are run after the unit tests.

Now we are ready to see our first coverage report. We need to ensure that all the stages are run now that we have added extra steps from the plugin, so it's best to run:

```
mvn clean test
```

From here, we will get a report that looks like this:

Java Demo Project

Java Demo Project

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.diffblue.javademo.tictactoe	<div><div></div></div>	92%	<div><div></div></div>	82%	6	23	5	51	1	9	0	3
Total	36 of 455	92%	5 of 28	82%	6	23	5	51	1	9	0	3

We can find the HTML version of the report under target/site/jacoco (i.e. \$/jacoco). You will notice that you can drill down into the package and the classes to see the coverage, down to individual lines:

Board.java

```

1. package com.diffblue.javademo.tictactoe;
2.
3. import java.util.ArrayList;
4.
5. public class Board {
6.
7.     private Player[][] board = new Player[3][3];
8.
9.     public Player getCell(Coordinate cell) {
10.         return board[cell.col][cell.row];
11.     }
12.
13.     /**
14.      * Place a move.
15.      * @param cell the cell to set
16.      * @param player making the move
17.      */
18.     public void setCell(Coordinate cell, Player player) {
19.         // Ensure that the cell hasn't previously been set
20.         if (board[cell.col][cell.row] != null) {
21.             throw new IllegalArgumentException("Trying to place player in a space already played");
22.         }
23.         // Ensure the game isn't over
24.         if (whoHasWon() != null) {
25.             throw new IllegalArgumentException("Trying to place player once the game has been won");
26.         }
27.         // Ensure that the correct player is being played
28.         if (player != nextPlayer()) {
29.             throw new IllegalArgumentException("Trying to place the wrong player");
30.         }
31.         board[cell.col][cell.row] = player;
32.     }
33.
34.     private Player nextPlayer() {
35.         int countO = 0;
36.         int countX = 0;
37.         for (int col = 0; col < 3; col++) {
38.             for (int row = 0; row < 3; row++) {
39.                 if (board[col][row] == Player.NOUGHT) {
40.                     countO++;
41.                 } else if (board[col][row] == Player.CROSS) {
42.                     countX++;
43.                 }
44.             }
45.         }
46.         if (countO > countX) {
47.             return Player.CROSS;
48.         }
49.         // Noughts always play first
50.         return Player.NOUGHT;
51.     }
52.
53.     public Player whoHasWon() {
54.         ArrayList<Coordinate> winningPositions = new ArrayList<>();
55.         // rows
56.         winningPositions.add(new Coordinate[] {new Coordinate(0,0), new Coordinate(1,0), new Coordinate(2,0)});
57.         winningPositions.add(new Coordinate[] {new Coordinate(0,1), new Coordinate(1,1), new Coordinate(2,1)});
58.         winningPositions.add(new Coordinate[] {new Coordinate(0,2), new Coordinate(1,2), new Coordinate(2,2)});
59.     }
60. }

```

There are three basic colors for the lines:

- Green: the line is completely covered by existing tests.
- Yellow: the line is partially covered. This means that it has been hit, but as in the example above, only one of the possible branches has been executed.
- Red: there is no coverage for this line.

There are more details on the coverage color coding and other info in the [documentation](#). The full source for this tutorial is available [here](#).

Hopefully, this has been helpful in learning how to enable JaCoCo for your maven project. In the next chapter, we'll discuss how to expand your test suite to ensure that you have comprehensive coverage.

Chapter 3: How to build a complete test suite

In Chapter 1, we wrote a single unit test for our example code, which allows people to make a move and check whether someone has won a game of Tic-Tac-Toe. In Chapter 2, we generated a code coverage report to see how much of the code we covered with our single test case.

In this chapter, we will address how to create a complete test suite for this code. The question is, what is a complete test suite? As discussed in Chapter 2, we could say that a test suite that achieves 80% or more code coverage is complete. But instead, let's break that down and think about whether we are accurately testing the functionality required from the code.

Let's dive into the requirements

1. Players take turns to make moves in empty squares until someone wins.
2. Players can check to see who has won.

Looking at the class `board.java`, there are four methods:

1. `setCell`
2. `nextPlayer`
3. `whoHasWon`
4. `getCell`

We won't consider testing `nextPlayer`, because this is a private method used only by the other methods in the class. We expect to gain sufficient coverage by exercising the other methods. We also don't need to test `getCell` directly, because the code for this method is so simple.

For `setCell`, I am going to consider the following test cases:

1. Nought can make a move to an empty cell
2. Cross can make a move to an empty cell
3. Cannot make a move to an occupied cell
4. Same player cannot do two moves in succession
5. Cannot make a move once the game is over

For `whoHasWon` I am going to consider the following tests:

1. Noughts can win through each row

2. Crosses can win through each column
3. Can win through each diagonal

Now is a good time to introduce one of the **JUnit** rules around exception handling. If a player tries to occupy a cell that is already occupied, then an exception will be thrown. Given this is desirable behavior, we want to test this:

```
@Rule
public ExpectedException exception =
    ExpectedException.none();
```

The above line tells JUnit to create a rule that exceptions should not be thrown as the test is executed. If an exception is thrown during a test without this rule, it will not be marked as passed. The important part of defining this rule is to use it in the test case to say that we expect an exception.

```
@Test
public void cannotPlaceMoveInUsedCell() {
    // Arrange
    Board myBoard = new Board();
    Coordinate cell = new Coordinate(0,0);
    myBoard.setCell(cell, Player.NOUGHT);






    // Act
    exception.expect(IllegalArgumentException.class);
    myBoard.setCell(cell, Player.CROSS);
}
```

In this test, we are asking our board to place an X in a cell that's already occupied. This will generate an `IllegalArgumentException`. Therefore, immediately before we try to place an X in an occupied cell, we set the rule to expect an `IllegalArgumentException`; if that exception is not thrown, the test case will fail.

Having written tests for each of the test cases listed earlier, let's validate our coverage using the code coverage that we set up in the last tutorial. Here are the results:

Java Demo Project > com.diffblue.javademo.tictactoe

com.diffblue.javademo.tictactoe

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Ctxy	Missed Lines	Missed Methods	Missed Classes
Coordinate		45%		50%	2 3	1 6	0 1	0 1
Player		91%	n/a	n/a	1 3	1 7	1 3	0 1
Board		100%		100%	0 17	0 38	0 5	0 1
Total	21 of 455	95%	2 of 28	92%	3 23	2 51	1 9	0 3

This result looks good: we have covered all the code in the Board class. However, the Coordinate class has comparatively poor coverage. Let's look at this a little closer.

This class was added so we could have the check for a valid location on the board in a single place. Nowhere in our tests so far have we checked for a case where the cell provided is invalid.

Let's create a file `CoordinateTest.java` and add a test for this. Those of you who have looked at the code carefully will see that we need to add two tests: one with the column greater than 2, and the other with the row greater than 2. For the sake of completeness, let's also add a test that checks the other side of the boundary, i.e. creating a cell with the max values.

Now we can see that we have a complete set of tests. We are covering all the required functionality and we are hitting all the lines. As always, the code for the tutorial is available on [GitLab](https://github.com/diffblue). Next up is Chapter 4, where we'll talk about the basics of mocking!

Chapter 4: Mocking in unit tests

Now that you've created your first unit test, learned how to calculate code coverage with JaCoCo, and started to make a more comprehensive test suite, it's time to learn a slightly more advanced unit testing skill: mocking.

Mocking is where we substitute the behavior of part of the solution with our own definition for the purposes of testing. You're probably thinking that using mocks means you will need to maintain the user-facing code and the mock, so what's the point of making extra work for yourself? Well, you want your test cases to be deterministic, i.e. to generate the same result no matter when or how you run them.

There are a couple of different things that can impact test cases and make them non-deterministic: First is accessing things outside of the program e.g. Network, Files etc. The second is any behavior that is random or based on date or time.

Mocking: Getting Started

Let's have a look at an example where we can use mocking. A method has been added to our Tic-Tac-Toe game which will allow the computer to select the next move for the player. This is based on randomly selecting a cell from the list of those currently available:

```
/**
 * Choose a random free cell for my next move.
 * @param player to place in the cell.
 */
public void randomMove(Player player) {
    // Find all the available cells
    ArrayList<Coordinate> availableCells =
        new ArrayList<>();
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            Coordinate currentCell = new Coordinate(i, j);
            if (getCell(currentCell) == null)

                }
        }
    int numOfCells = availableCells.size();

    if (numOfCells == 0) {
        throw new IllegalArgumentException("Board is full,
cannot place in a move");
    }

    // Pick a random cell
    int index = (int) (Math.random() * numOfCells);

    // Put player in that cell
    setCell(availableCells.get(index), player);
}
```

So how are we going to test this? Essentially, there are two test cases:

1. Player is put into an empty cell
2. Correct error when the build is full

If you've followed Chapters 1 through 3 of this series, you already know how to write a test for the second. But what about the first? We can run the method and ensure that an exception isn't thrown, but how are we going to test that the player has been put in a cell?

The most deterministic approach is to mock the random number generator so that when the test runs, we know which cell is being used. Before we can write the test, we need to include a couple of new dependencies; these are the mocking framework that we are going to use.

Add the following to the dependencies section of our pom.xml:

```
<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-module-junit4</artifactId>
  <version>1.6.5</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.powermock</groupId>
  <artifactId>powermock-api-mockito</artifactId>
  <version>1.6.5</version>
  <scope>test</scope>
</dependency>
```

Now that we have the dependencies set up, let's have a look at this test:

```
@PrepareForTest ({Board.class, Math.class})
@Test
public void
randomMoveInputNoughtOutputIndexOutOfBoundsException() {
    // Setup mocks
    mockStatic(Math.class);
    when(Math.random()).thenReturn(0.5);

    // Arrange
    Board myBoard = new Board();

    // Act
    myBoard.randomMove(Player.NOUGHT);

    // Assert
    assertEquals("Player O not in the middle cell",
        Player.NOUGHT, myBoard.getCell(new Coordinate(1, 1)));
}
```

There are some new things here, including the annotation `PrepareForTest`. With this, we are telling PowerMock (which we are using for mocking) the class that we are about to test and the class that we are going to mock.

In this case, we are going to mock the behavior of `Math.random`. This is a static method, so we need to start off by telling PowerMock to `mockStatic` the class.

Then, we simply choose a return value that we would like to be returned when `Math.random` is called. In this case, `0.5` will provide the index at the middle of the array. This means that the test can check that `O` is placed in the middle cell.

And there you have it!

Try it out yourself! As always, the full code is available on [GitLab](#). Nextup is Chapter 5: How to find the time and motivation to unit test.

Chapter 5: Finding the time and motivation to unit test

If you've made it this far into our unit testing guide, then you probably agree that writing unit tests is a good thing. We've covered a few tricks for writing tests for code that is hard to test in Chapter 4 on Mocking, to hopefully remove some of the barriers that might keep you from writing tests. But often, developers still don't write as many unit tests as they should. Why is that?

Before we address the reasons that we often hear for not wanting to write unit tests, let's pause to remember why we write tests in the first place. We want to ensure that our code works today, and we want to prevent it from breaking in the future, whether a breaking change is made by a teammate or by our future selves.

By writing automated unit tests and running them as part of every build, we can ensure that we aren't inadvertently changing the behavior of the code.

As [Martin Fowler](#) has aptly pointed out, “Imperfect tests, run frequently, are much better than perfect tests that are never written at all.”

Top “Why I’m Not Writing Unit Tests” Excuses

1. “I don’t have time to write all of these unit tests.”

How long does it take to write one unit test? The first unit test in a project is the hardest because it requires the infrastructure to be in place. But after the first one is finished, the simpler unit tests that come after can take under five minutes to write.

Just because you don’t have time to write tons of unit tests doesn’t mean you have a good excuse to not write any. Developers often plan to come back and write unit tests later, but this rarely actually happens. The time pressure making it hard to write a test suite won’t evaporate at a later date, so find a few moments here and now.

2. “It’s ok if I don’t write unit tests; QA will catch the bugs.”

Having a good independent QA team does mean that any bugs left in early on won’t reach the customer. However, the turnaround time for the bug to be found, reported, triaged and returned from QA to the developer to be fixed is definitely going to be measured in days, if not weeks. If the bug gets found as part of the build process, it is much quicker, easier and cheaper to fix.

Let’s also remember that QA will report a bug based on the user behavior. Unit tests help the developer understand where the bug is, and therefore what should be fixed. If QA reports a bug talking in UI terms on a big product with multiple interacting parts, you may spend a significant amount of time finding the root cause of the bug. However, if a unit test fails, you instantly know where the issue is.

3. “Unit tests slow down development due to maintenance.”

It's true that unit tests need to be looked after. They will need updating, adding to, and sometimes replacing as the product grows and changes. It is worth remembering that every time a unit test fails, it means that the behavior of the code has changed. Each time this happens, you should be making a conscious choice about whether you have introduced a bug or a desirable change in behavior. Each change will still give you some work, but knowledge is key: Without the unit test, you might not know that you changed the behavior of the code at all.

4. “Our unit tests are unpredictable. Sometimes they fail for no apparent reason.”

It's painful when this happens, and you can count on getting a new failure just before you want to ship a product. The real question is: is your product unpredictable, or is the test wrong? If it's your product then you're going to have the same challenges debugging customer issues that you do debugging your tests, except you might not have access to the customer environment. Either way, this is something that needs investigating and fixing.

As we'll discuss further, when you're unit testing, perfection is not required. Instead, get on with writing a few test cases and see the rewards they grant you further down the line.

Chapter 6: How to avoid common mistakes

We have other guides on the common mistakes [committed when writing unit tests](#), and it's important to understand and recognize when we're making these mistakes. But even more important is learning how to avoid them in the first place. In this chapter, we'll discuss exactly that!

Use good metrics

In testing, it's easy to get hung up on the wrong metrics. We want a comprehensive unit test suite, but what does comprehensive mean to us and our project? There's not a single answer to this question, but by picking the correct metric(s) to measure yourself and others by, you will be able to provably deliver a valuable unit test suite.

Summarize the purpose and value of each test

The first principle of Test Driven Development (TDD) says to write your test suite before writing your code. Using this approach, we have to think about the complete behavior of the code prior to writing a line of it. Without existing code clouding our judgement, we end up writing tests based on specific expected behaviors, which prevents writing tests for the sake of writing tests or chasing arbitrary coverage targets.

You don't have to be doing full-on TDD to avoid this mistake. When you start writing tests, try adding a couple of lines to the test: one giving you the purpose of the test ("What is it trying to test?") and the other saying what the value is ("Why will the end user care if the test fails?"). If you cannot answer one or both of the questions, then ask yourself: Should this test exist?

Treat test code the same as production code

Test code will cause the build to fail if one of the tests fails. It may not actually be used by the user, but in a modern DevOps environment where you are trying to do multiple releases per day, having poor quality test cases will lead to frustration through intermittent build failures.

This is why it's important to apply the same coding standards to your test code as your production code. The number of tools, such as linters, that ignore test code is astounding. Just like maintaining production code, someone else on the team (or future you) will have to debug and maintain your tests. You want to make this as easy as possible.

'Always code as if the person who ends up maintaining your code is a violent psychopath who knows where you live.'

Code For The Maintainer

There is no magic bullet that will ensure you always write the best test cases possible. However, like anything else in the art of coding, it's important to set a baseline for what is good enough for your needs and ensure that you always stay above this line. With this in mind, and a culture that makes testing as important as writing production code, you won't go wrong.

Chapter 7: How automated unit tests speed up continuous integration

Bugs are big problems for development teams. They're annoying. They pop up at the worst time. And they're expensive, particularly when they're found late in the development process or after the application has already been released. In this chapter, we'll discuss how unit tests can help find bugs as part of continuous integration (CI).

Why are regression bugs so important to prevent?

First, let's get definitions straight: A bug is classified as a regression if it is a new bug that changes the behavior of the system negatively. A new feature bug exists entirely within a new feature.

By now, you might be thinking of a company or product you use that has impacted you through a bug introduced during an upgrade, eroding trust between you and the vendor. For business-to-business relationships, this can

be enough to cause financial penalties in the contract, or even mean that the client looks around for a new vendor.

These are the bugs that convert to severity zero/one customer issues very quickly. When a customer can no longer do their job, they will rightly expect your development and support teams to be sweating while trying to fix their issues.

If regressions are so important, why aren't they tested for more often?

Simple answer: It's boring! By the third release, testing the same feature manually rarely holds much excitement for the engineer doing the testing. Management is only interested if a serious bug is found, and even then, it is an inconvenience to the timetable.

Therefore, most testers will spend the majority of their time working on testing new features. This is the fun and exciting area of working in a QA team: Being able to shape the product before anyone else.

What's the solution?

Automation, automation and a little more automation. Finding regressions is exactly the task that automated testing does well. To stop the pain caused by these bugs, the answer is simple: ensure that your regression coverage is adequate. It's not always reasonable to expect QA/QE to find regression bugs, so it's time for developers to make the time to do more automated testing. But how? And where in the pipeline?

The key to regressions and continuous integration is to do as much as possible, as early as possible. But that can be overwhelming for developers, who are usually creating code and tests during the earliest phases of the pipeline.

For organizations following DevOps practices, introducing automated regression tests in the first phases of software development shortens the

time to delivery by showing the differences between builds right after new commits are made.

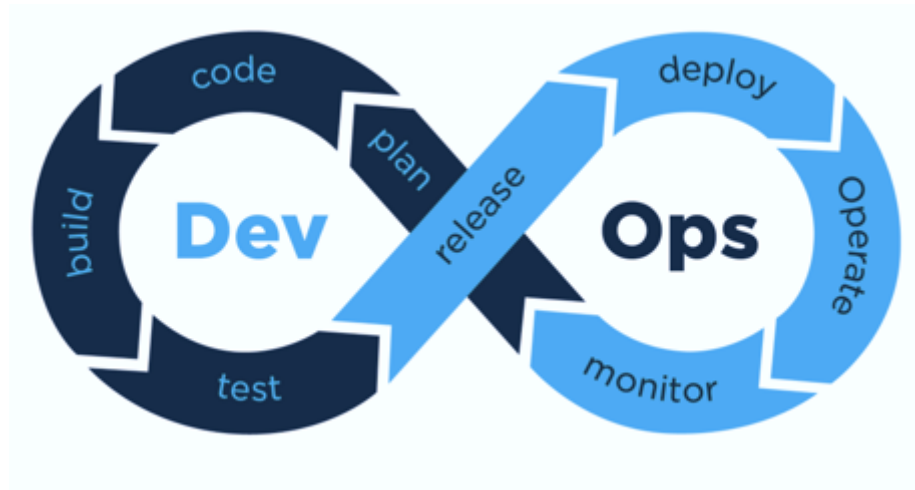


Image Source

These tests are designed to run quickly and perform basic logical validation at the unit level. Without these tests, basic errors advance into the mainline and risk breaking it, and it takes a lot longer to find and fix those bugs later on. This is why, to have a truly continuous CI pipeline, regression tests should be used as early as possible.

In the next chapter, we'll go into more detail about how unit tests can make DevOps goals achievable.

Chapter 8: How to deliver on the promises of DevOps

As discussed in Chapter 7, DevOps was designed to empower individuals and allow teams to reduce the time required to bring features into production. This promise always resonates with leaders. But when it comes to reality, development and operations teams often put a lot of effort into introducing DevOps, only to discover that they don't have the return on the investment that leaders expect.

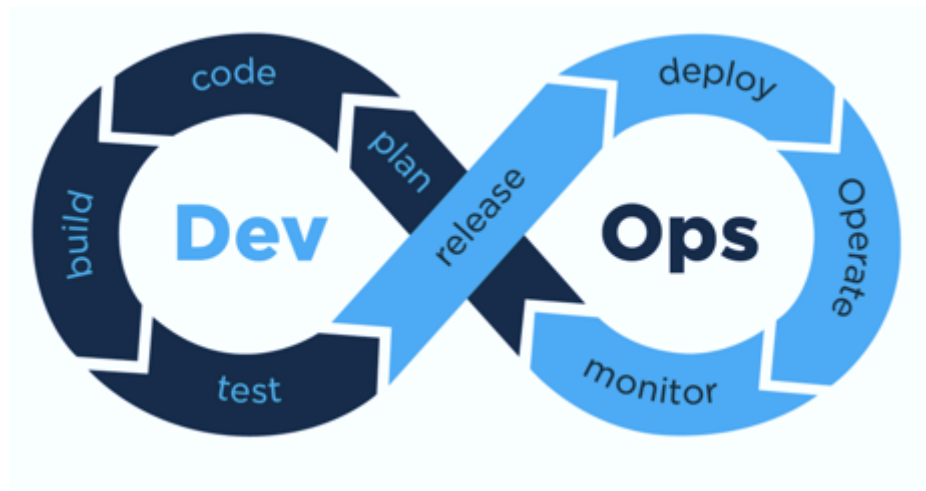


Image Source

At its heart, the DevOps process looks like the image above: the product goes through a cycle of Build, Test, Release, Monitor, Repeat. All the team members are encouraged to get involved at each of the stages and ensure the success of the product. Releases are sped up—to speeds reaching one release per day, rather than one release per year—and **with greater automation comes higher performance**.

But the problem with DevOps is an old one: too many cooks spoil the broth.

Cutting Corners

Given the modern development desire to release as soon as we have a minimum viable product (MVP) it often happens that corners are cut to get each version out. After a few releases, what was meant to be a square has had so many cut corners that it's starting to look a lot like a circle.

Now suppose the product has a few rough edges and the development team is empowered to get involved with solving customer issues. Customer issues trickle in, developers help to resolve those issues, and before long, the developers find themselves spending half their week on customer issues rather than the next piece of work on their list. Suddenly, productivity drops dramatically.

Don't Let Quality Slip

Continuous Integration and Continuous Deployment means that we can get features into the hands of customers quicker than we have ever done before. However, this means that we can quickly force our new bugs upon our customers. While breaking down the barriers between teams and the formal release stages that existed in previous models, developers and operations need to hold themselves to the same or higher standards of quality, and the only way to do this is by **incorporating testing into every stage of the DevOps cycle**, with tests that run automatically (and, as often as possible, are created automatically too):

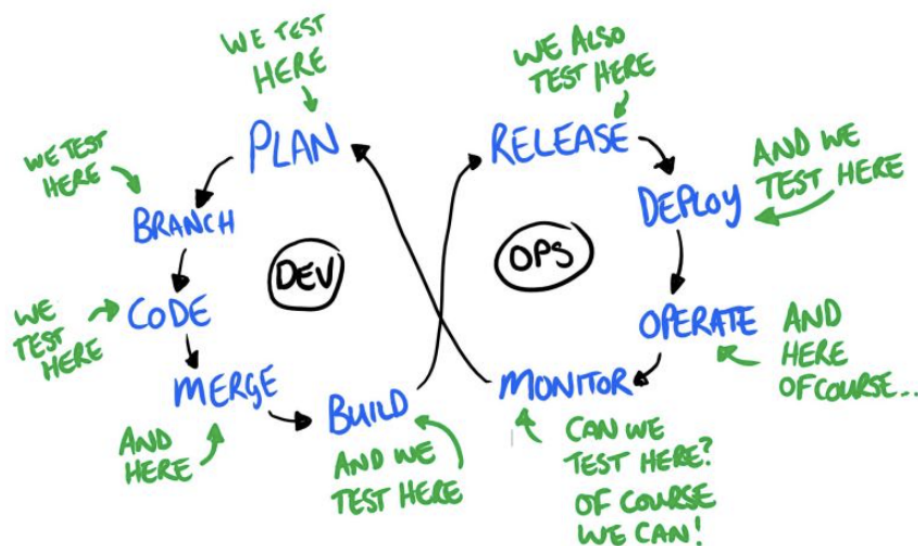


Image Source

DevOps can bring the same benefits to the product lifecycle that SCRUM brings to the development lifecycle. Given all the technology advances that delivered true SCRUM for example continuous integration systems, it's time for the same advancement in the DevOps world. By ensuring software is tested as rigorously as possible, maintenance headaches will be minimized and the promised productivity enhancement can be delivered.

Deliver a Complete Test Suite

To reap the benefits of DevOps, you can't afford to stick with manual processes or risk merging regressions late in the build process. By providing a complete, automated test suite, the development team can ensure that the rigorous testing required is delivered. If there isn't time for the team to write these tests themselves, the right tools can help. **Diffblue Cover** provides a way to quickly generate the tests you need for Continuous Integration, and automatically update the tests as the codebase changes over time. Code quality demands a better solution, and increasingly, that means automating test creation.

Epilogue: Why imperfect tests are better than no tests

Developers want to produce high quality products, but we're often asked to do this with fewer resources and less time than we would like. The balancing act between quality, cost and speed has never rested on the shoulders of developers as much as it does today. Companies assume developers are agile, and therefore can easily introduce CI and CD and shift left product development. Once this small task has been completed, development rates will accelerate and teams will deliver features at 10x the current rate.

If only it were that simple

Behind every buzzwordy new approach is the assumption that it can be implemented perfectly. However, given the pressures that development teams are under, the question we should start thinking about is: what is good enough to deliver a return? Let's turn to our automated test suite, which is integral in accelerating our development processes. What are we trying to

achieve? We want each build to be self-testing, i.e. to give us a clean bill of health (or report an issue). Reporting issues is the easy bit: if a test fails, the build is marked as broken and people can immediately look into it.

Getting a clean bill of health from the build passing is much harder. We are trying to prove the absence of bugs. Therefore, we can consider the perfect test suite as one where, if the build passes, there are no bugs in the software. This is a lofty goal, however—as momentous as trying to prove that aliens don't exist.

Having realized that it is impossible to create the perfect suite of tests, but being under pressure to deliver anyway, often leads to teams simply not writing unit tests and instead raising tickets saying, “We should write unit tests for feature x.” This is a compromise. But is it the best compromise?

Rather than rationalizing the decision not to write tests because of time shortages, how about asking yourself this: Do I have time to write one test? If the answer is yes, then write that one test. And keep going until you run out of time. It's true that this won't lead to a perfect test suite. But as Martin Fowler says: **“Imperfect tests are better than perfect tests that are never written.”**

All too often there is a tug-of-war between teams when tests fail. Teams ask: are the tests wrong, or is the code wrong? If we stop thinking about the tests failing and instead think about tests reporting change in the product, then rather than looking for blame, we can consider whether the change is desirable or not.

Given a spare half an hour each day, we could all write a new test and gradually increase overall code quality. Remember: the test doesn't have to be perfect; it just has to be good enough to highlight an issue when it fails.

If you want to kick start your testing writing process then take a look at **Diffblue Cover**. The tests may not represent all of your business logic, but they will give you a lot more confidence in your code much more quickly than you can get with human test-writing efforts alone.