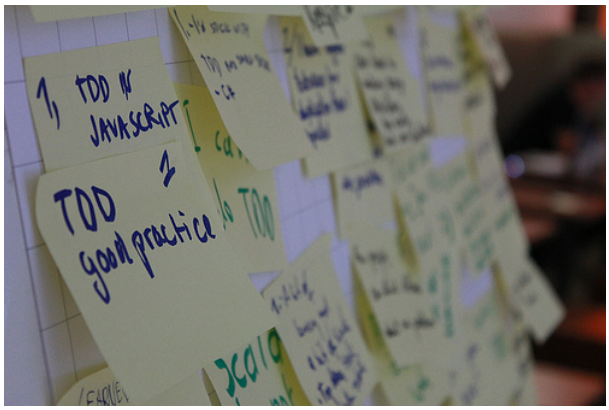
Technology Conversations

Test Driven Development (TDD): Example Walkthrough

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.



The following sequence of steps is generally followed:

- Add a test
- Run all tests and see if the new one fails
- Write some code
- Run tests
- Refactor code
- Repeat

There's a plenty of articles written on TDD and Wikipedia is always a good start. This article will focus on the actual test and implementation using variation of one of the Roy Osherove Katas. Do not click the link until you're finished with this article. This excercise is best done when not all requirements are known in advance.

Below you will find the test code related to each requirement and afterwards the actual implementation. Try to read only one requirement, write the tests and the implementation yourself and compare it with the results from this article. Remember that there are many different ways to write tests and implementation. This article is only one out of many possible solutions.

Let's start!

Requirements

- Create a simple String calculator with a method int Add(string numbers)
- The method can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example "" or "1" or "1,2"
- Allow the Add method to handle an unknown amount of numbers

- Allow the Add method to handle new lines between numbers (instead of commas).
- The following input is ok: “1\n2,3” (will equal 6)
- Support different delimiters
- To change a delimiter, the beginning of the string will contain a separate line that looks like this: “//[delimiter]\n[numbers…]” for example “//;\n1;2” should return three where the default delimiter is ‘;’ .
- The first line is optional. All existing scenarios should still be supported
- Calling Add with a negative number will throw an exception “negatives not allowed” – and the negative that was passed. If there are multiple negatives, show all of them in the exception message stop here if you are a beginner.
- Numbers bigger than 1000 should be ignored, so adding 2 + 1001 = 2
- Delimiters can be of any length with the following format: “//[delimiter]\n” for example: “//[—]\n1—2—3” should return 6
- Allow multiple delimiters like this: “//[delim1][delim2]\n” for example “//[-][%]\n1-2%3” should return 6.
- Make sure you can also handle multiple delimiters with length longer than one char

Even though this is a very simple program, just looking at those requirements can be overwhelming. Let’s take a different approach. Forget what you just read and let us go through the requirements one by one.

Create a simple String calculator

Requirement 1: The method can take 0, 1 or 2 numbers separated by comma (,).

Let’s write our first set of tests.

[JAVA TEST]

```
1  package com.wordpress.technologyconversations.tddtest;
2
3  import org.junit.Test;
4  import com.wordpress.technologyconversations.tdd.StringCalculator;
5
6  public class StringCalculatorTest {
7      @Test(expected = RuntimeException.class)
8      public final void whenMoreThan2NumbersAreUsedThenExceptionIsThrown() {
9          StringCalculator.add("1,2,3");
10     }
11     @Test
12     public final void when2NumbersAreUsedThenNoExceptionIsThrown() {
13         StringCalculator.add("1,2");
14         Assert.assertTrue(true);
15     }
16     @Test(expected = RuntimeException.class)
17     public final void whenNonNumberIsUsedThenExceptionIsThrown() {
18         StringCalculator.add("1,X");
19     }
20 }
```

It’s a good practice to name test methods in a way that it is easy to understand what is being tested. I prefer a variation of BDD with When [ACTION] Then [VERIFICATION]. In this case the name of one of the test methods is whenMoreThan2NumbersAreUsedThenExceptionIsThrown. Our first set of tests verifies that up to two numbers can be passed to the calculator’s add method. If there’s more than two or if one of them is not a number, exception should be thrown. Putting “expected” inside the @Test annotation tells the JUnit runner that the expected outcome is to throw the specified exception. From here on, for brevity reasons, only mod-

ified parts of the code will be displayed. Whole code divided into requirements can be obtained from the GitHub repository (tests and implementation).

[JAVA IMPLEMENTATION]

```
public class StringCalculator {
    public static final void add(final String numbers) {
        String[] numbersArray = numbers.split(",");
        if (numbersArray.length > 2) {
            throw new RuntimeException("Up to 2 numbers separated by comma (,)
        } else {
            for (String number : numbersArray) {
                Integer.parseInt(number); // If it is not a number, parseInt w
            }
        }
    }
}
```

Keep in mind that the idea behind TDD is to do the necessary minimum to make the tests pass and repeat the process until the whole functionality is implemented. At this moment we're only interested in making sure that "the method can take 0, 1 or 2 numbers". Run all the tests again and see them pass.

Requirement 2: For an empty string the method will return 0

[JAVA TEST]

```
@Test
public final void whenEmptyStringIsUsedThenReturnValueIs0() {
    Assert.assertEquals(0, StringCalculator.add(""));
}
```

[JAVA IMPLEMENTATION]

```
public static final int add(final String numbers) { // Changed void to int
    String[] numbersArray = numbers.split(",");
    if (numbersArray.length > 2) {
        throw new RuntimeException("Up to 2 numbers separated by comma (,) are
    } else {
        for (String number : numbersArray) {
            if (!number.isEmpty()) {
                Integer.parseInt(number);
            }
        }
    }
    return 0; // Added return
}
```

All there was to do to make this test pass was to change the return method from void to int and end it with returning zero.

Requirement 3: Method will return their sum of numbers

[JAVA TEST]

```
@Test
public final void whenOneNumberIsUsedThenReturnValueIsThatSameNumber() {
    Assert.assertEquals(3, StringCalculator.add("3"));
}

@Test
```

```
7  public final void whenTwoNumbersAreUsedThenReturnValueIsTheirSum() {
8      Assert.assertEquals(3+6, StringCalculator.add("3,6"));
9  }
```

[JAVA IMPLEMENTATION]

```
1  public static int add(final String numbers) {
2      int returnValue = 0;
3      String[] numbersArray = numbers.split(",");
4      if (numbersArray.length > 2) {
5          throw new RuntimeException("Up to 2 numbers separated by comma (,) are
6      }
7      for (String number : numbersArray) {
8          if (!number.trim().isEmpty()) { // After refactoring
9              returnValue += Integer.parseInt(number);
10         }
11     }
12     return returnValue;
13 }
```

Here we added iteration through all numbers to create a sum.

Requirement 4: Allow the Add method to handle an unknown amount of numbers

[JAVA TEST]

```
1  //  @Test(expected = RuntimeException.class)
2  //  public final void whenMoreThan2NumbersAreUsedThenExceptionIsThrown() {
3  //      StringCalculator.add("1,2,3");
4  //  }
5      @Test
6      public final void whenAnyNumberOfNumbersIsUsedThenReturnValuesAreTheirSums(
7          Assert.assertEquals(3+6+15+18+46+33, StringCalculator.add("3,6,15,18,46
8      }
```

[JAVA IMPLEMENTATION]

```
1  public static int add(final String numbers) {
2      int returnValue = 0;
3      String[] numbersArray = numbers.split(",");
4      // Removed after exception
5      // if (numbersArray.length > 2) {
6      // throw new RuntimeException("Up to 2 numbers separated by comma (,) are
7      // }
8      for (String number : numbersArray) {
9          if (!number.trim().isEmpty()) { // After refactoring
10             returnValue += Integer.parseInt(number);
11         }
12     }
13     return returnValue;
14 }
```

All we had to do to accomplish this requirement was to remove part of the code that throws an exception if there are more than 2 numbers. However, once tests are executed, the first test failed. In order to fulfill this requirement, the test whenMoreThan2NumbersAreUsedThenExceptionIsThrown needed to be removed.

Requirement 5: Allow the Add method to handle new lines between numbers (instead of commas).

[JAVA TEST]

```
@Test
public final void whenNewLineIsUsedBetweenNumbersThenReturnValuesAreTheirSums(
    Assert.assertEquals(3+6+15, StringCalculator.add("3,6n15"));
}
```

[JAVA IMPLEMENTATION]

```
public static int add(final String numbers) {
    int returnValue = 0;
    String[] numbersArray = numbers.split(",|n"); // Added |n to the split reg
    for (String number : numbersArray) {
        if (!number.trim().isEmpty()) {
            returnValue += Integer.parseInt(number.trim());
        }
    }
    return returnValue;
}
```

All we had to do to was to extend the split regex by adding |\n.

Requirement 6: Support different delimiters

To change a delimiter, the beginning of the string will contain a separate line that looks like this: "//[delimiter]\n[numbers...]" for example "//;\n1;2" should take 1 and 2 as parameters and return 3 where the default delimiter is ';' .

[JAVA TEST]

```
@Test
public final void whenDelimiterIsSpecifiedThenItIsUsedToSeparateNumbers() {
    Assert.assertEquals(3+6+15, StringCalculator.add("//;n3;6;15"));
}
```

[JAVA IMPLEMENTATION]

```
public static int add(final String numbers) {
    String delimiter = ",|n";
    String numbersWithoutDelimiter = numbers;
    if (numbers.startsWith("//")) {
        int delimiterIndex = numbers.indexOf("//") + 2;
        delimiter = numbers.substring(delimiterIndex, delimiterIndex + 1);
        numbersWithoutDelimiter = numbers.substring(numbers.indexOf("n") + 1);
    }
    return add(numbersWithoutDelimiter, delimiter);
}

private static int add(final String numbers, final String delimiter) {
    int returnValue = 0;
    String[] numbersArray = numbers.split(delimiter);
    for (String number : numbersArray) {
        if (!number.trim().isEmpty()) {
            returnValue += Integer.parseInt(number.trim());
        }
    }
    return returnValue;
}
```

This time there was quite a lot of refactoring. We split the code into 2 methods. Initial method parses the input looking for the delimiter and later on calls the new one that does the actual sum. Since we already have tests that cover all existing functionality, it was safe to do the refactoring. If anything went wrong, one of the tests would find the problem.

Requirement 7: Negative numbers will throw an exception

Calling Add with a negative number will throw an exception “negatives not allowed” – and the negative that was passed. If there are multiple negatives, show all of them in the exception message.

[JAVA TEST]

```
@Test(expected = RuntimeException.class)
public final void whenNegativeNumberIsUsedThenRuntimeExceptionIsThrown() {
    StringCalculator.add("3,-6,15,18,46,33");
}
@Test
public final void whenNegativeNumbersAreUsedThenRuntimeExceptionIsThrown() {
    RuntimeException exception = null;
    try {
        StringCalculator.add("3,-6,15,-18,46,33");
    } catch (RuntimeException e) {
        exception = e;
    }
    Assert.assertNotNull(exception);
    Assert.assertEquals("Negatives not allowed: [-6, -18]", exception.getMessa
}
```

There are two new tests. First one checks whether exception is thrown when there are negative numbers. The second one verifies whether the exception message is correct.

[JAVA IMPLEMENTATION]

```
private static int add(final String numbers, final String delimiter) {
    int returnValue = 0;
    String[] numbersArray = numbers.split(delimiter);
    List negativeNumbers = new ArrayList();
    for (String number : numbersArray) {
        if (!number.trim().isEmpty()) {
            int numberInt = Integer.parseInt(number.trim());
            if (numberInt < 0) {
                negativeNumbers.add(numberInt);
            }
            returnValue += numberInt;
        }
    }
    if (negativeNumbers.size() > 0) {
        throw new RuntimeException("Negatives not allowed: " + negativeNumbers
    }
    return returnValue;
}
```

This time code was added that collects negative numbers in a List and throws an exception if there was any.

Requirement 8: Numbers bigger than 1000 should be ignored

Example: adding 2 + 1001 = 2

[JAVA TEST]

```
@Test
public final void whenOneOrMoreNumbersAreGreaterThan1000IsUsedThenItIsNotIncluc
    Assert.assertEquals(3+1000+6, StringCalculator8.add("3,1000,1001,6,1234"));
}
```

[JAVA IMPLEMENTATION]

```
private static int add(final String numbers, final String delimiter) {
        int returnValue = 0;
        String[] numbersArray = numbers.split(delimiter);
        List negativeNumbers = new ArrayList();
        for (String number : numbersArray) {
                if (!number.trim().isEmpty()) {
                        int numberInt = Integer.parseInt(number.trim());
                        if (numberInt < 0) {
                                negativeNumbers.add(numberInt);
                        } else if (numberInt <= 1000) {
                                returnValue += numberInt;
                        }
                }
        }
        if (negativeNumbers.size() > 0) {
                throw new RuntimeException("Negatives not allowed: " + negati\
        }
        return returnValue;
}
```

This one was simple. We moved “returnValue += numberInt;” inside an “else if (numberInt <= 1000)”.

There are 3 more requirements left. I encourage you to try them by yourself.

Requirement 9: Delimiters can be of any length

Following format should be used: “//[delimiter]\n”. Example: “//[—]\n1—2—3” should return 6

Requirement 10: Allow multiple delimiters

Following format should be used: “//[delim1][delim2]\n”. Example “//[-][%]\n1-2%3” should return 6.

Requirement 11: Make sure you can also handle multiple delimiters with length longer than one char

Give TDD a chance

This whole process often looks overwhelming to TDD beginners. One of the common complains is that TDD slows down the development process. It is true that at first it takes time to get into speed. However, after a bit of practice development using TDD process saves time, produces better design, allows easy and safe refactoring, increases quality and test coverage and, last but not least, makes sure that software is always tested. Another great benefit of TDD is that tests serve as a living documentation. It is enough to look at tests to know what each software unit should do. That documentation is always up to date as long as all tests are passing. Unit tests produced with TDD should provide “code coverage” for most of the code and they should be used together with Acceptance Test Driven Development (ATDD) or Behavior Driven Development (BDD). Together they are covering both unit and functional tests, serving as full documentation and requirements.

TDD makes you focus on your task, code exactly what you need, think from outside and, ultimately, a better programmer.