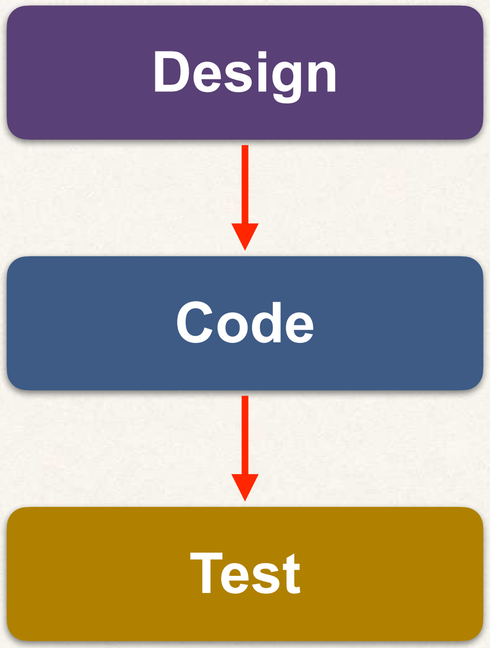
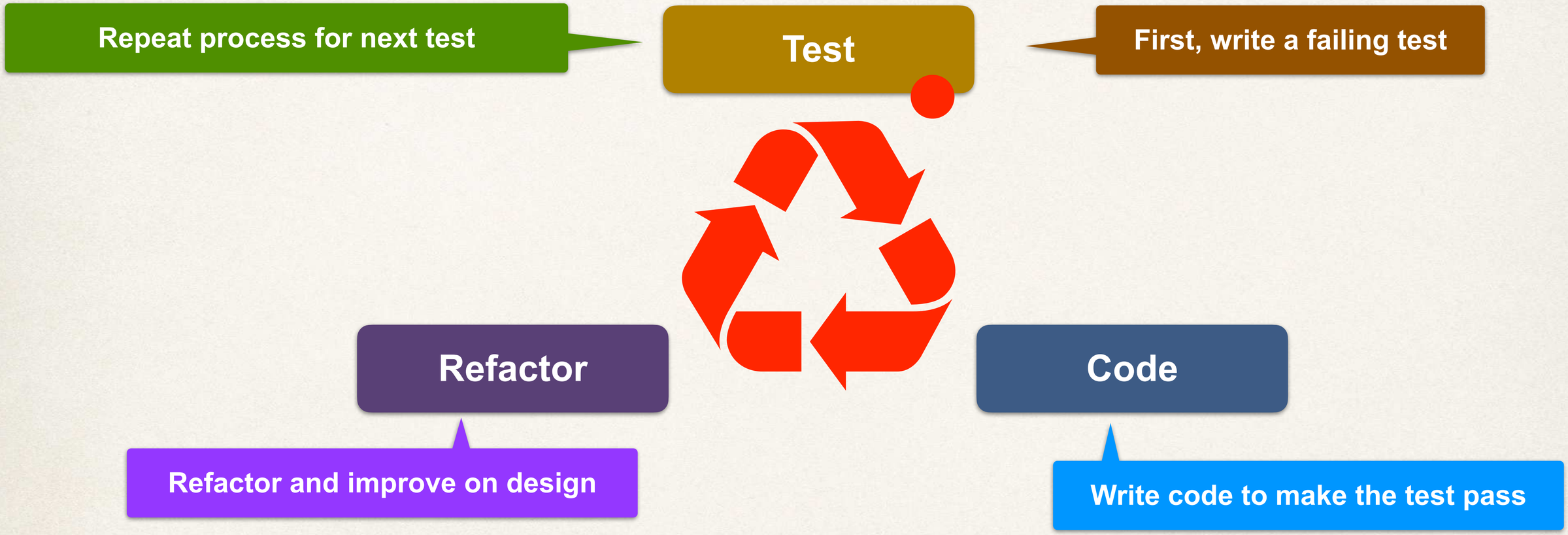
luv2code

Test Driven Development (TDD)

Traditional Development



Test Driven Development (TDD)



Benefits of Test Driven Development (TDD)

- Clear task list of things to test and develop
- Tests will help you identify edge cases
- Develop code in small increments
- Passing tests increases confidence in code
- Gives freedom to refactor ... tests are your safety net ... did I break anything??

Our Project

- We will apply what we've learned so far for a TDD project
- Use the FizzBuzz project as an example

What Is FizzBuzz?

- Coding problem used in technical interviews

- Problem
  - Write a program to print the first 100 FizzBuzz numbers. Start at 1 and end at 100.
    - If number is divisible by 3, print Fizz
    - If number is divisible by 5, print Buzz
    - If number is divisible by 3 and 5, print FizzBuzz
    - If number is not divisible by 3 or 5, then print the number

FizzBuzz Sample Output

- Write a program to print the first 100 FizzBuzz numbers. Start at 1 and end at 100.
  - If number is divisible by 3, print Fizz
  - If number is divisible by 5, print Buzz
  - If number is divisible by 3 and 5, print FizzBuzz
  - If number is not divisible by 3 or 5, then print the number

```
1   1
2   2
3   Fizz
4   4
5   Buzz
6   Fizz
7   7
    ...
```

FizzBuzz ... on the web

- FizzBuzz Wiki

https://wiki.c2.com/?FizzBuzzTest

  - Has solutions in various programming languages
  - Basic solutions and advanced solutions (minimum lines of code)

- FizzBuzz Book

www.fizzbuzzbook.com

  - Yes ... there is a book dedicated to FizzBuzz solutions LOL!

Development Process

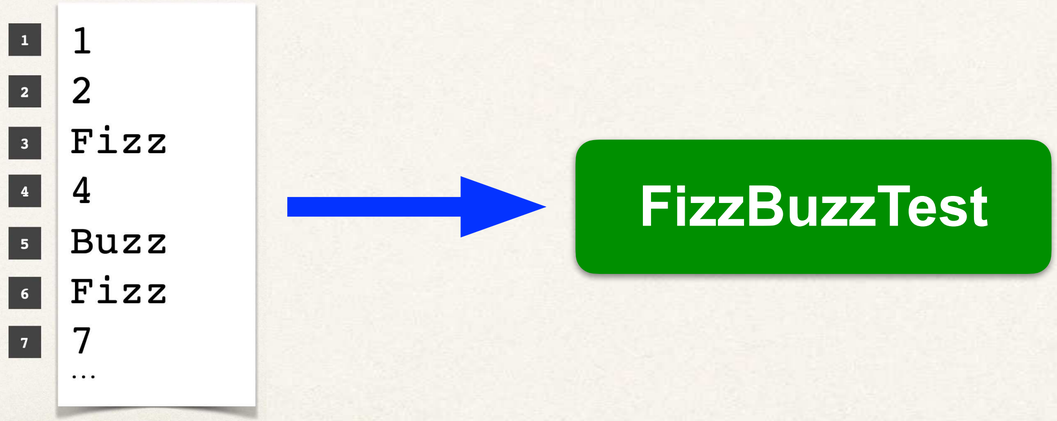
1. Write a failing test
2. Write code to make the test pass
3. Refactor the code
4. Repeat the process

luv2code

Parameterized Tests

Fizz Buzz Input Values

- At the moment we have created tests for specific FizzBuzz input values
- We'd like to pass in a collection of values and expected results
- Run the same test in a loop



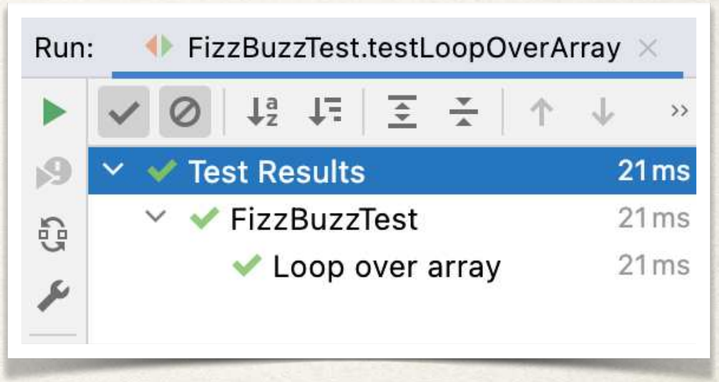
One Possible Solution

```
@DisplayName("Loop over array")
@Test
@Order(5)
void testLoopOverArray() {

    String[][] data = { {"1","1"},
                        {"2","2"},
                        {"3","Fizz"},
                        {"4","4"},
                        {"5","Buzz"},
                        {"6","Fizz"},
                        {"7","7"}

    };

    for (int i=0; i < data.length; i++) {
        String value = data[i][0];
        String expected = data[i][1];

        assertEquals(expected, FizzBuzz.compute(Integer.parseInt(value)));
    }
}
```

value

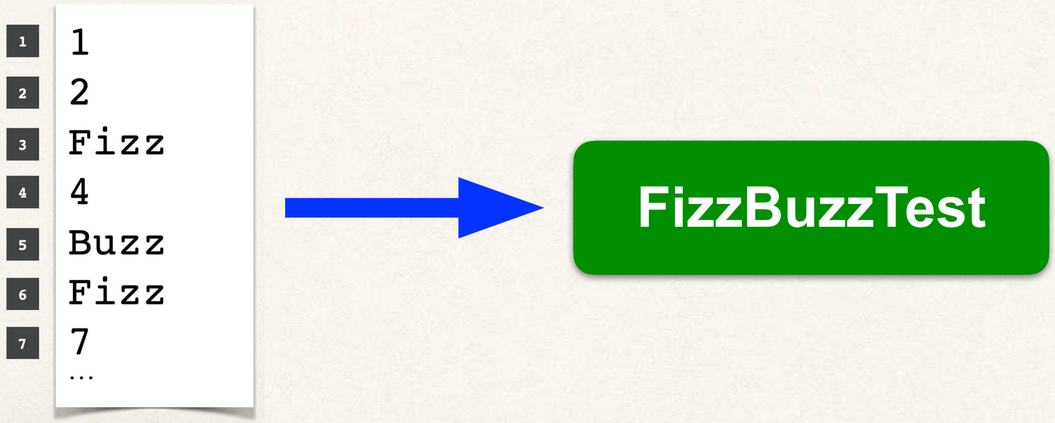
expected

```
1  1
2  2
3  Fizz
4  4
5  Buzz
6  Fizz
7  7
...
```



But wait ... JUnit to the rescue

- JUnit provides @ParameterizedTest
- Run a test multiple times and provide different parameter values



Behind the scenes, JUnit will run the test multiple times and supply the data

JUnit does the looping for you :-)

Source of Values

- When using a @ParameterizedTest, where can we get the values?

Annotation	Description
`@ValueSource`	Array of values: Strings, ints, doubles, floats etc
`@CsvSource`	Array of CSV String values
`@CsvFileSource`	CSV values read from a file
`@EnumSource`	Enum constant values
`@MethodSource`	Custom method for providing values

ParameterizedTest - @CsvSource

```
@DisplayName("Testing with csv data")
@ParameterizedTest
@CsvSource({
            "1,1",
            "2,2",
            "3,Fizz",
            "4,4",
            "5,Buzz",
            "6,Fizz",
            "7,7"
})
@Order(6)
void testCsvData(int value, String expected) {

    assertEquals(expected, FizzBuzz.compute(value));

}
```

expected

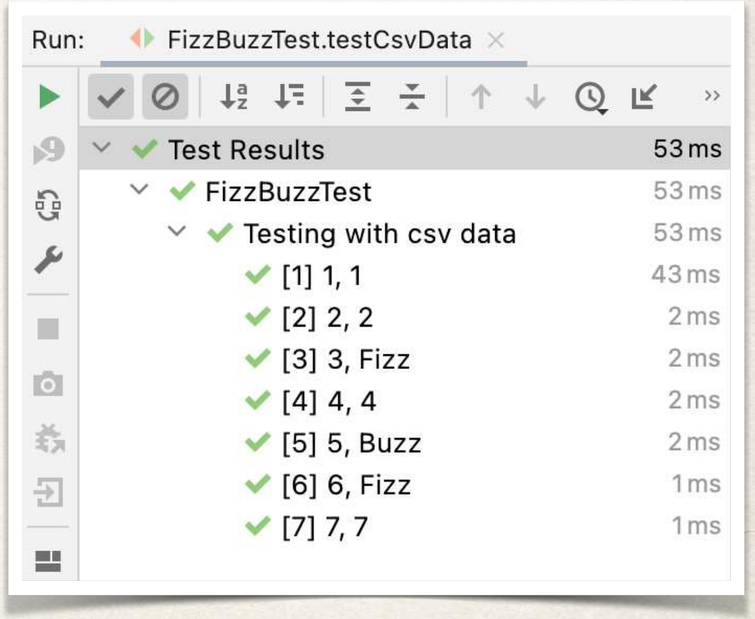
value

Behind the scenes, JUnit will run the test multiple times and supply the data for the parameters

JUnit does the looping for you :-)

Test method now has parameters

```
1
2
Fizz
4
Buzz
Fizz
7
...
```

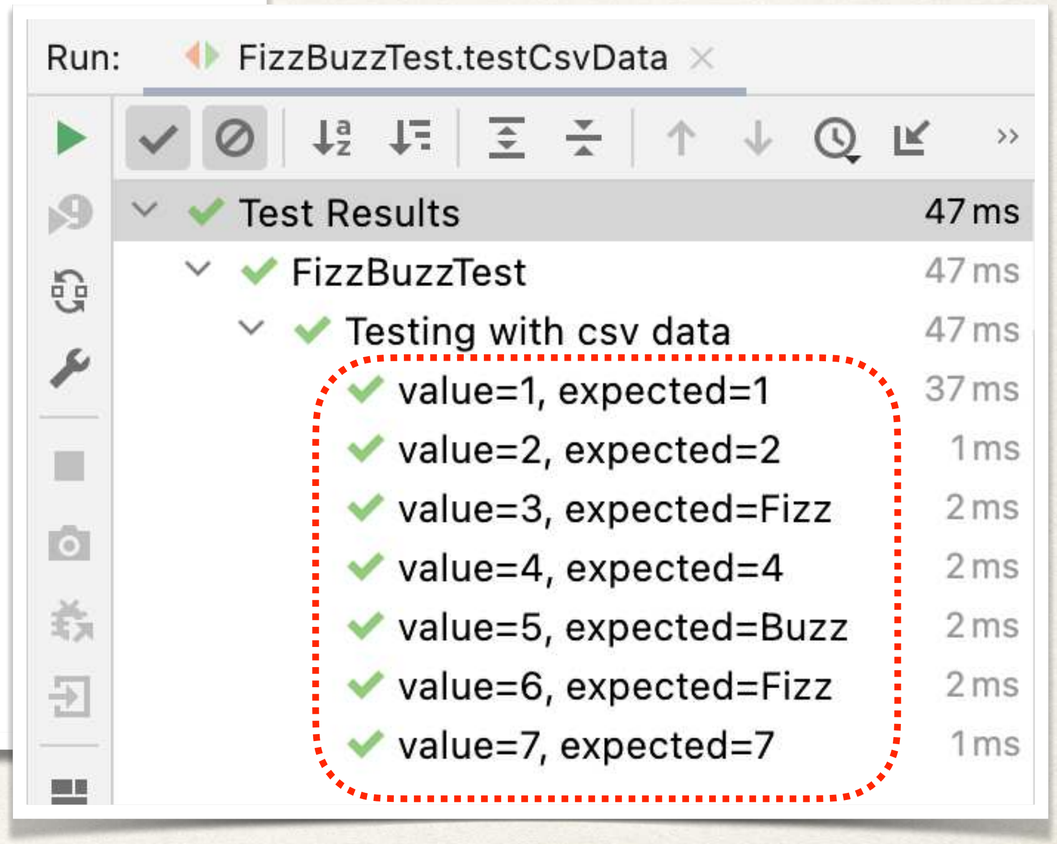


Customize Invocation Names

```
@DisplayName("Testing with csv data")
@ParameterizedTest(name="value={0}, expected={1}")
@CsvSource({
                "1,1",
                "2,2",
                "3,Fizz",
                "4,4",
                "5,Buzz",
                "6,Fizz",
                "7,7"
})
@Order(6)
void testCsvData(int value, String expected) {

    assertEquals(expected, FizzBuzz.compute(value));

}
```

index 0

index 1



Read a CSV file

File: src/test/resources/small-test-data.csv

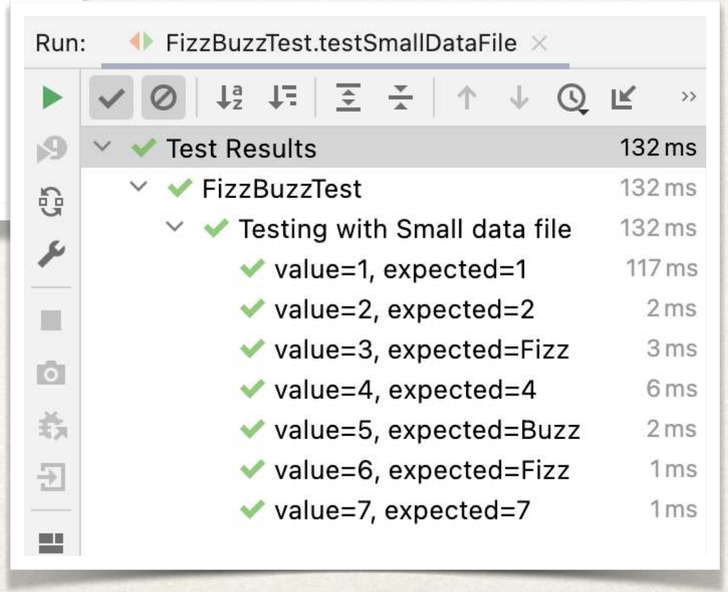
value

expected

```
1,1
2,2
3,Fizz
4,4
5,Buzz
6,Fizz
7,7
```

```
@DisplayName("Testing with Small data file")
@ParameterizedTest(name="value={0}, expected={1}")
@CsvFileSource(resources="/small-test-data.csv")
@Order(7)
void testSmallDataFile(int value, String expected) {

    assertEquals(expected, FizzBuzz.compute(value));

}
```

Reference the CSV file



JUnit User Guide

- Additional features for @ParameterizedTest
  - @MethodSource
  - Argument Aggregation
  - ...

https://junit.org/junit5/docs/current/user-guide

See section on Parameterized Tests