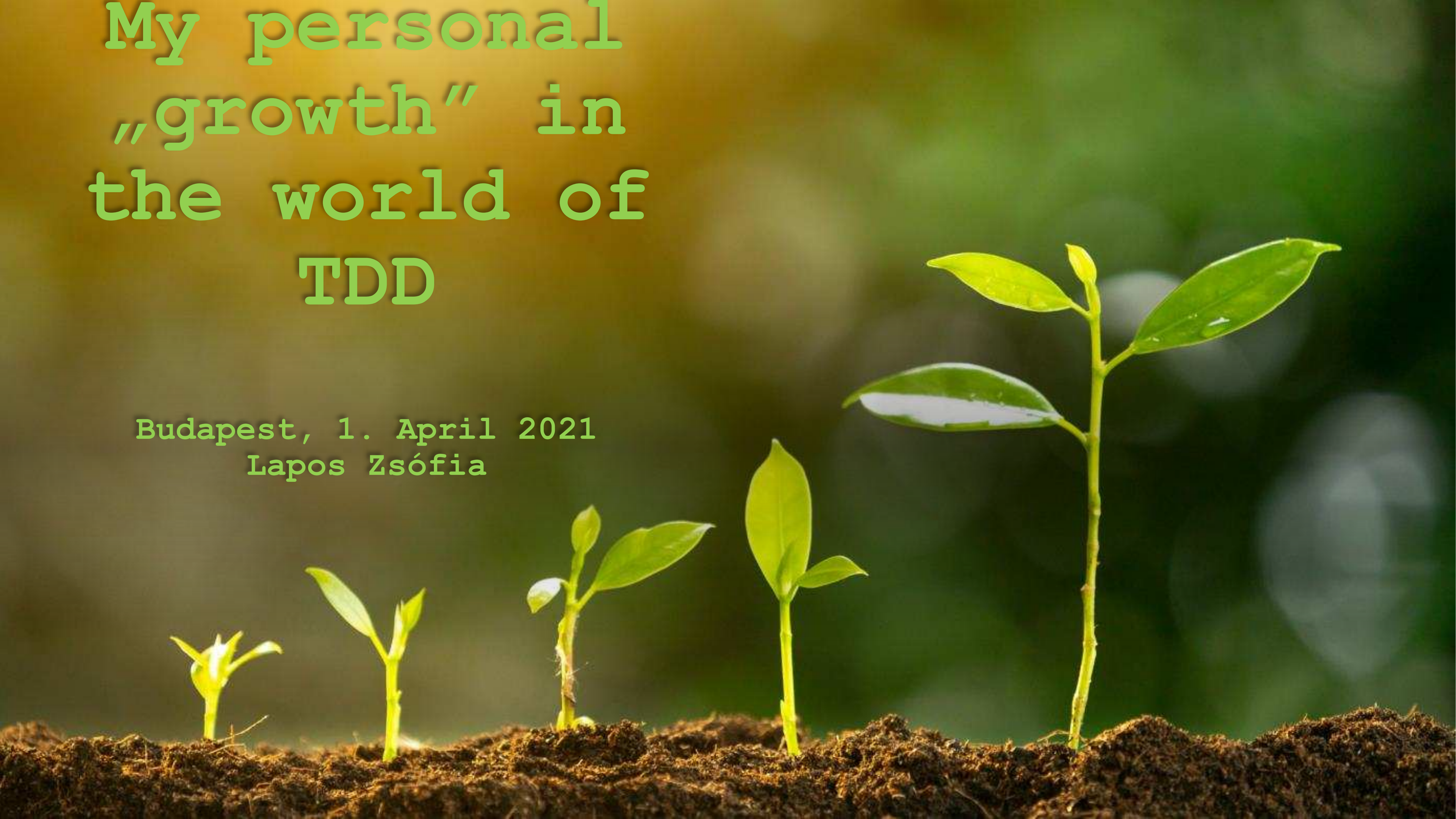


My personal „growth” in the world of TDD

Budapest, 1. April 2021
Lapos Zsófia

Content

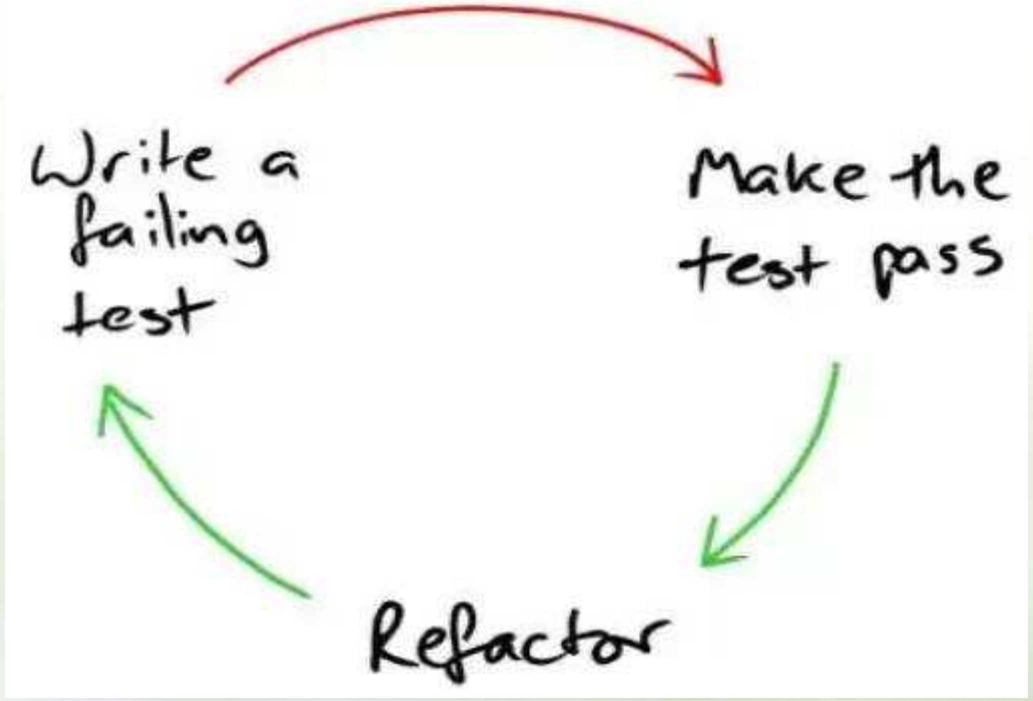
- What is TDD?
- Examples
- Advantages
- Conclusion

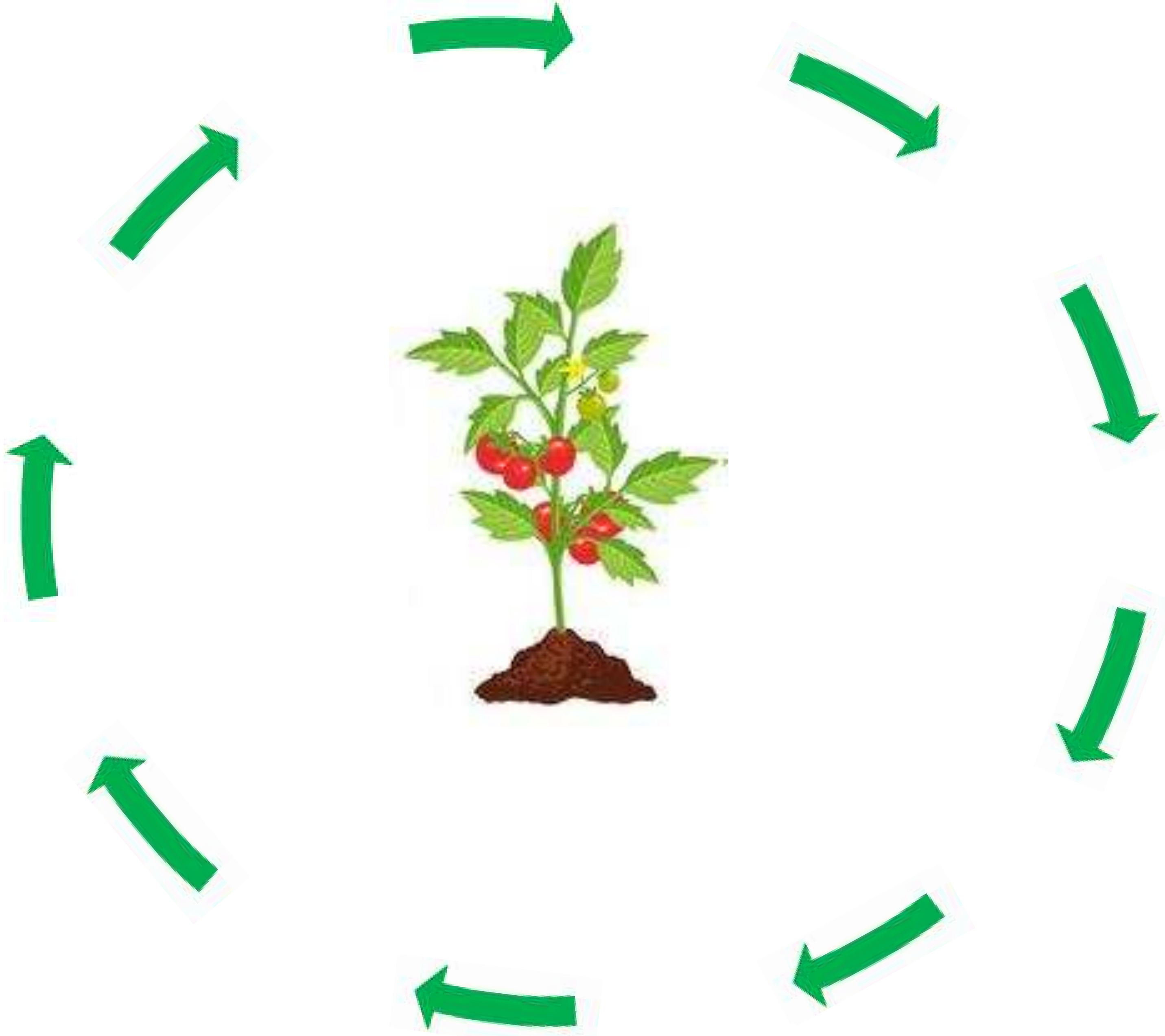
Test-Driven Developmet

Popularized by **Kent Beck,** 2003

- Test-Driven Development (TDD) is a methodology in software development that focuses on an iterative development cycle where the accent is placed on writing test cases **before** the actual feature or function is written. TDD utilizes repetition of **short** development cycles.

- The product requirements are transformed into very specific test cases and then the software is enhanced for the tests to pass.





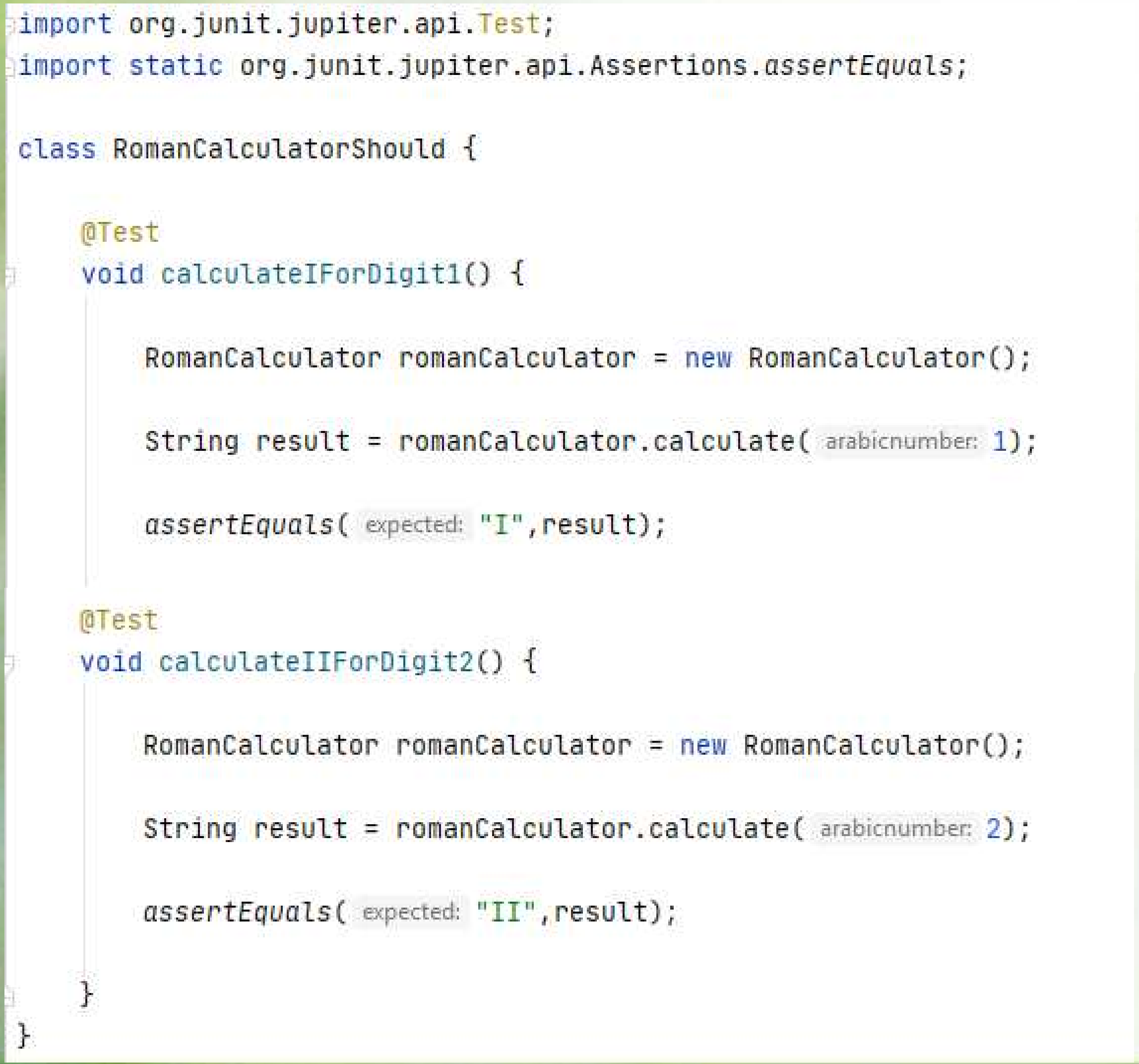
➢ Test written before the code

This doesn't mean writing all of the tests and then writing all of the code.

- We start by writing just one test, a very small failing test.
- Then we write some code, just enough code to get the test to pass.
- Then we write another small test.
- …. and another small test
- The code and the tests are born and grow together.
- The test-ability of the code is „build-in":

→ Testing is built right into the development cycle, not after the fact

Examples: Roman numbers calculator

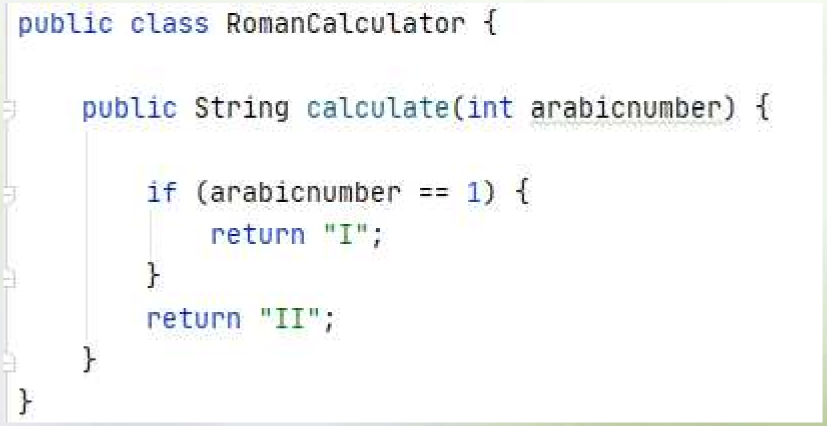


```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class RomanCalculatorShould {

    @Test
    void calculateIForDigit1() {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate( arabicnumber: 1);

        assertEquals( expected: "I",result);

    @Test
    void calculateIIForDigit2() {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate( arabicnumber: 2);

        assertEquals( expected: "II",result);

    }
}
```

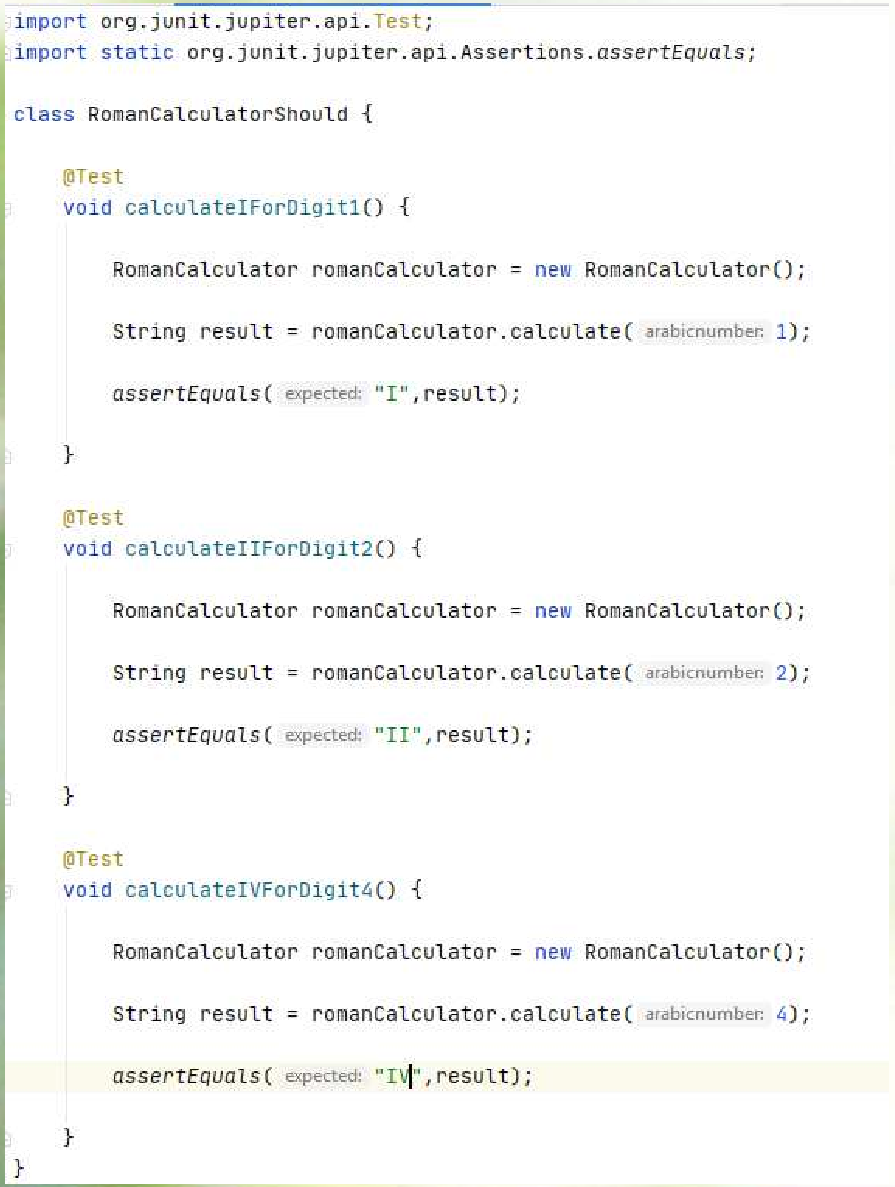
The first small test and a piece of code:

→ the simplest code to pass the test

The next small test and some code
to get the test to pass

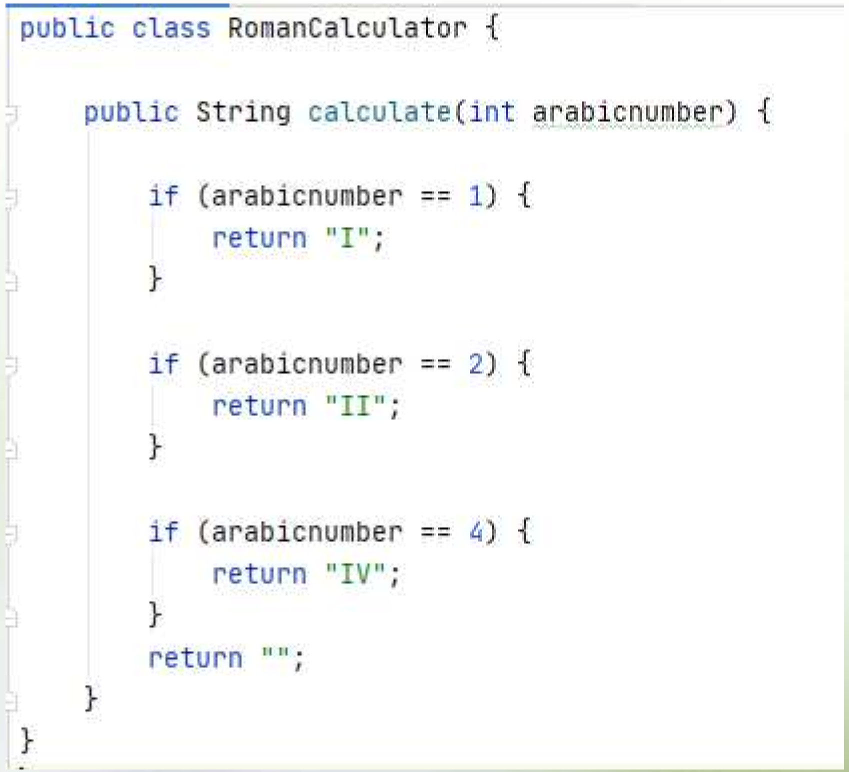


```
public class RomanCalculator {

    public String calculate(int arabicnumber) {

        if (arabicnumber == 1) {
            return "I";
        }
        return "II";
    }
}
```



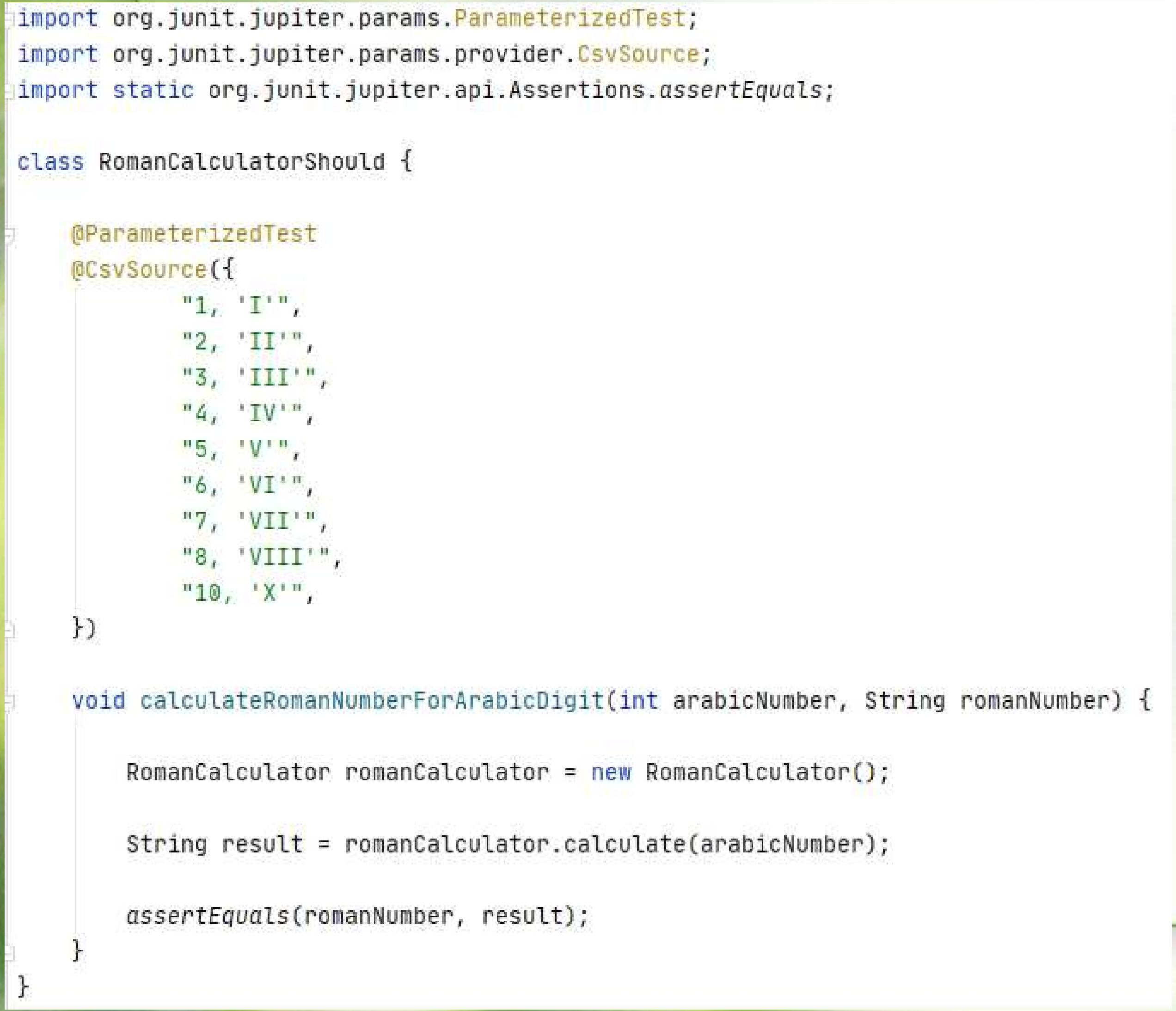
```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class RomanCalculatorShould {

    @Test
    void calculateIForDigit1() {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate( arabicnumber: 1);

        assertEquals( expected: "I",result);

    }

    @Test
    void calculateIIForDigit2() {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate( arabicnumber: 2);

        assertEquals( expected: "II",result);

    }

    @Test
    void calculateIVForDigit4() {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate( arabicnumber: 4);

        assertEquals( expected: "IV",result);

    }
}
```

We moved forward in such tiny steps with each test case that it was impossible to lose the thread.

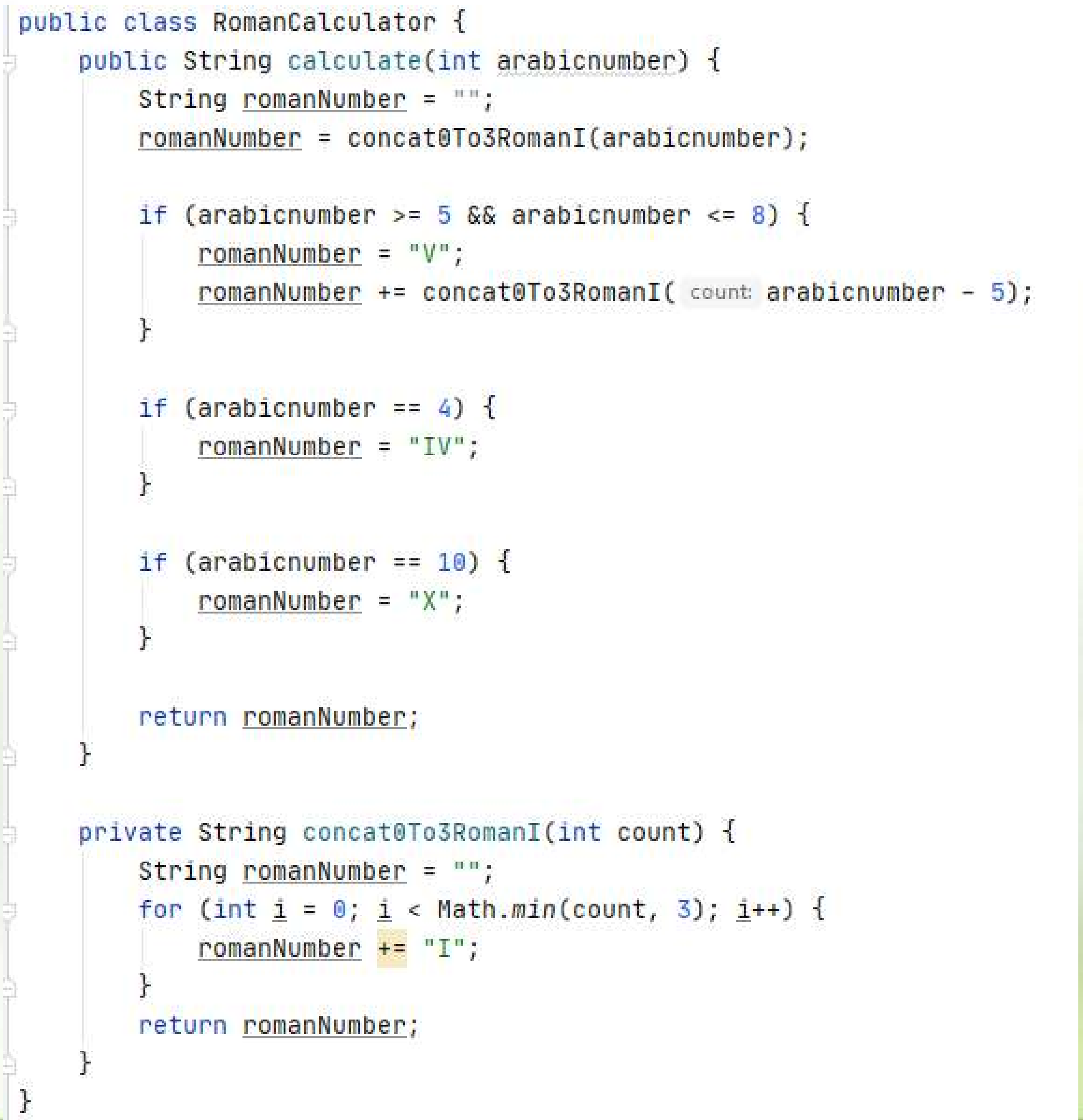


```java
public class RomanCalculator {

    public String calculate(int arabicnumber) {

        if (arabicnumber == 1) {
            return "I";
        }

        if (arabicnumber == 2) {
            return "II";
        }

        if (arabicnumber == 4) {
            return "IV";
        }
        return "";
    }
}
```

The end result (without TPP) was more than I expected.



```java
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
import static org.junit.jupiter.api.Assertions.assertEquals;

class RomanCalculatorShould {

    @ParameterizedTest
    @CsvSource({
            "1, 'I'",
            "2, 'II'",
            "3, 'III'",
            "4, 'IV'",
            "5, 'V'",
            "6, 'VI'",
            "7, 'VII'",
            "8, 'VIII'",
            "10, 'X'",
    })

    void calculateRomanNumberForArabicDigit(int arabicNumber, String romanNumber) {

        RomanCalculator romanCalculator = new RomanCalculator();

        String result = romanCalculator.calculate(arabicNumber);

        assertEquals(romanNumber, result);
    }
}
```



```java
public class RomanCalculator {
    public String calculate(int arabicnumber) {
        String romanNumber = "";
        romanNumber = concat0To3RomanI(arabicnumber);

        if (arabicnumber >= 5 && arabicnumber <= 8) {
            romanNumber = "V";
            romanNumber += concat0To3RomanI( count: arabicnumber - 5);
        }

        if (arabicnumber == 4) {
            romanNumber = "IV";
        }

        if (arabicnumber == 10) {
            romanNumber = "X";
        }

        return romanNumber;
    }

    private String concat0To3RomanI(int count) {
        String romanNumber = "";
        for (int i = 0; i < Math.min(count, 3); i++) {
            romanNumber += "I";
        }
        return romanNumber;
    }
}
```

Probably this solution would be born much later with traditional programming methods.

Advantages

“Preventing bugs is not a TDD goal. It’s more like a *side effect.*”

- **Better program design and higher code quality**
  - When writing tests, we have first to define the goal of what we want to realize with the piece of code.
  - It is important to think about what the code *should not accept*, in addition to what it should accept.
  - All the possible mistakes and errors are already taken into account. Here, we write the necessary tests to avoid all the failures

    → The code appears to give better results.

  - When the code has a clear structure and fits the test requirements, it’s very simple, straightforward and short.

- **Code flexibility and easier maintenance**
  - Implementation of TDD reduces the percentage of bugs by 40 - 80 percent, which consequently means that less time is required for fixing them.
  - The refactoring encourages a pure and attractive code structure, it stands for optimization of existing code to make it more readable and easy to introduce.
- **Each branch is covered**
  - An interesting side effect of using TDD is that it should result in 100% code coverage. If all the application code we write has to pass tests, then in theory we shouldn't have written application code that is untested.
- **We will get a reliable solution**
  - With TDD, we can be sure about the reliability of the developed solution. The tests help us to understand if everything goes right after refactoring or adding a new feature.
  - Without TDD, we go blind with the latest changes, and we are not quite sure how recent development will work with the perfect code built previously. Any new changes can break the solution.

Conclusion

Test-driven development can help us to build software that is:

- ✓ reliable
- ✓ fully usable
- ✓ easy to extend with new features

It comes with a short feedback loop and focuses on what’s important.

A beautifully-structured code is easier to modify, extend, test, and maintain.
The cleaner and simpler the code is, the fewer efforts we’ll have to put into modification or updates.

→ This directly contributes to our project’s success.

My first experience with mob programming:

- Sometimes it can be confusing, but through this programming practice I can learn to develop in a better style

My first goal:

- Get used to and apply the good habits of TDD learned here

okay you can start now!
roger!
brrrummm....
okay nice so test failed... now we can start building the bridge!
ARNO

Thank you for your attention!