# TDD & BDD

**Development practices to improve code quality**

Ivan Perl (IoT Saint-Petersburg)

# Agenda

1. What is TDD?
2. TDD: development lifecycle
3. TDD: horribly simple example
4. TDD: benefits
5. What is BDD?
6. BDD for TDD
7. BDD: benefits

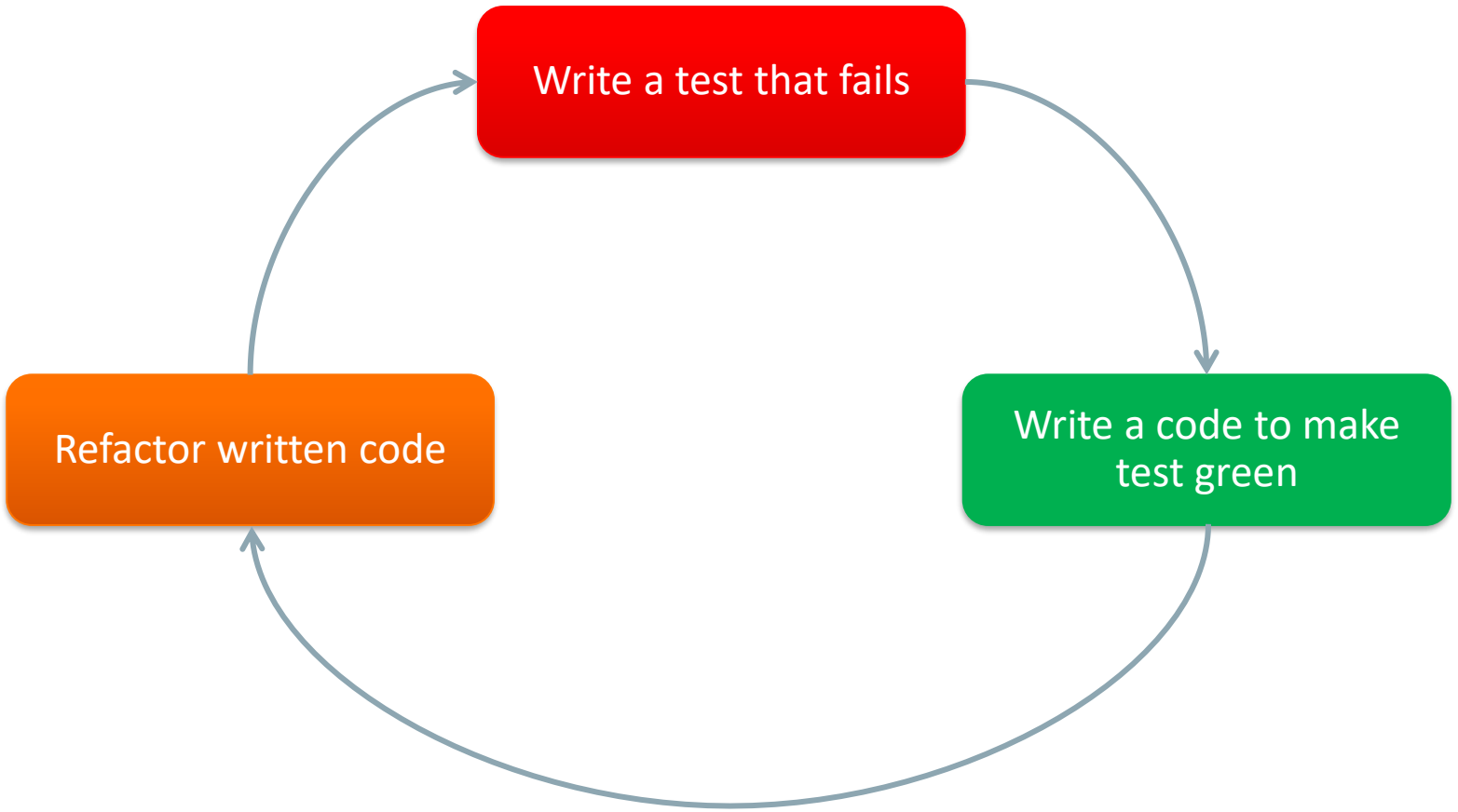
# TDD

# What is TDD?

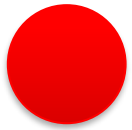
- Formally:
  - TDD = **T**est **D**riven **D**evelopment
- By meaning:
  - TDD = Test First Development + Refactoring
- **Test Driven Development:**
  - A development methodology that promotes rapid feedback of changes to the source code in order to help manage risk.
  - TDD helps a developer focus on solving the problem at hand, and avoid adding unnecessary code to the final product.
  - The result should be a piece of valid logic that is minimal and complete.

# TDD: development lifecycle

- *"Red. Green. Refactor."* is the mantra of a developer working by TDD.
- If this mantra is not mentioned by someone attempting to instruct TDD, then they are most likely not describing it accurately.



# TDD: development lifecycle



- A  test is written for a small non-existent feature, then it is run and fails.

- The feature is implemented – Rerun the tests and it passes.

- Inspect the code, can it be improved?
  - Is all of the functionality implemented?
  - Can the implementation be simplified?

  If not, add the next test.

# TDD: horribly simple example ($C^o$ to $F^o$)

- Let’s say we need to develop a function which will convert temperature from Celsius to Fahrenheit.
- We will do this in TDD way.
- Start with the simplest skeleton of your target class or function that will compile.

```
1.  double celcius_to_farenheit(double temperature)
2.  {
3.    return 0;
4.  }
```

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 1: Let's verify that we will see the correct value, for example at 0

```
1.  void TestCelciusAtZero()
2.  {
3.    ASSERT_EQUAL(32, celcius_to_fahreheit(0));
4.  }
```

This test will obviously fail because our function returns 0 disregarding input values.

`TestCelciusAtZero()`

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 1 Solution: The only thing we need it to make our tests green. So we have to make as simple implementation as possible:

```
1.  double celcius_to_farenheit(double temperature)
2.  {
3.    return 32;
4.  }
```

- Now test 1 will be green.
- Did we add all of the functionality that is required to create the correct solution? Obviously not, Fahrenheit has other temperatures than 32°.

`TestCelciusAtZero()`

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 2: Let's verify that we will see the correct value, at 100

```
1.  void TestCelciusAt100()
2.  {
3.    ASSERT_EQUAL(212, celcius_to_fahreheit(100));
4.  }
```

Now we have two tests, one is green and one is not

TestCelciusAtZero()

TestCelciusAt100()

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 2 Solution: Now we need to have a simplest possible solution which will suit two test cases. Here we go:

```
1.  double celcius_to_farenheit(double temperature)
2.  {
3.    return (temperature == 0) ? 32 : 212;
4.  }
```

- Now tests 1 and 2 are green.
- Are we done? I guess not quite

TestCelciusAtZero()

TestCelciusAt100()

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 3: Let's verify that we will see the correct value, human body temp

```
1.  void TestCelciusAtHumanBodyTemp()
2.  {
3.    ASSERT_EQUAL(98.6f, celcius_to_fahreheit(37.0f));
4.  }
```

Now we have three tests, and the new one will fail

TestCelciusAtZero()

TestCelciusAt100()

TestCelciusAtHumanBodyTemp()

# TDD: horribly simple example ($C^o$ to $F^o$) cont.

- Test 3 Solution: Add an implementation to the function that will allow all of the tests to pass:

```
1.  double celcius_to_farenheit(double temperature)
2.  {
3.    return (temperature * 5.0f / 9.0f) + 32.0f;
4.  }
```

- Now all tests are green.
- And we are done since our implementation covers required cases

TestCelciusAtZero()

TestCelciusAt100()

TestCelciusAtHumanBodyTemp()

# TDD: benefits

- Tests are **NOT** test, or at least they are **not only** tests
  - Tests are validation tool
    - They will tell you that your code is acting according to requirements, since they represent requirements
  - Tests are documentation tool
    - Since tests represents requirements they explain how code under test is working
  - Tests are design tools
    - Writing tests prior to code allow to look at the API from the user perspective, because you are trying to use future API before implementing it.

# TDD: benefits

- Solving problem with test coverage
  - TDD gives 100% code coverage by default, because you have test before the code
- Having outside overview of the API
  - API is designed not from its implementation, but from an attempt of its usage
- Better code composition and complexity
  - Going from test to code helps to reduce strong links between components from insight
- Tests became much simpler
  - Because you don’t need to imagine various hacks to access hidden code parts to test them

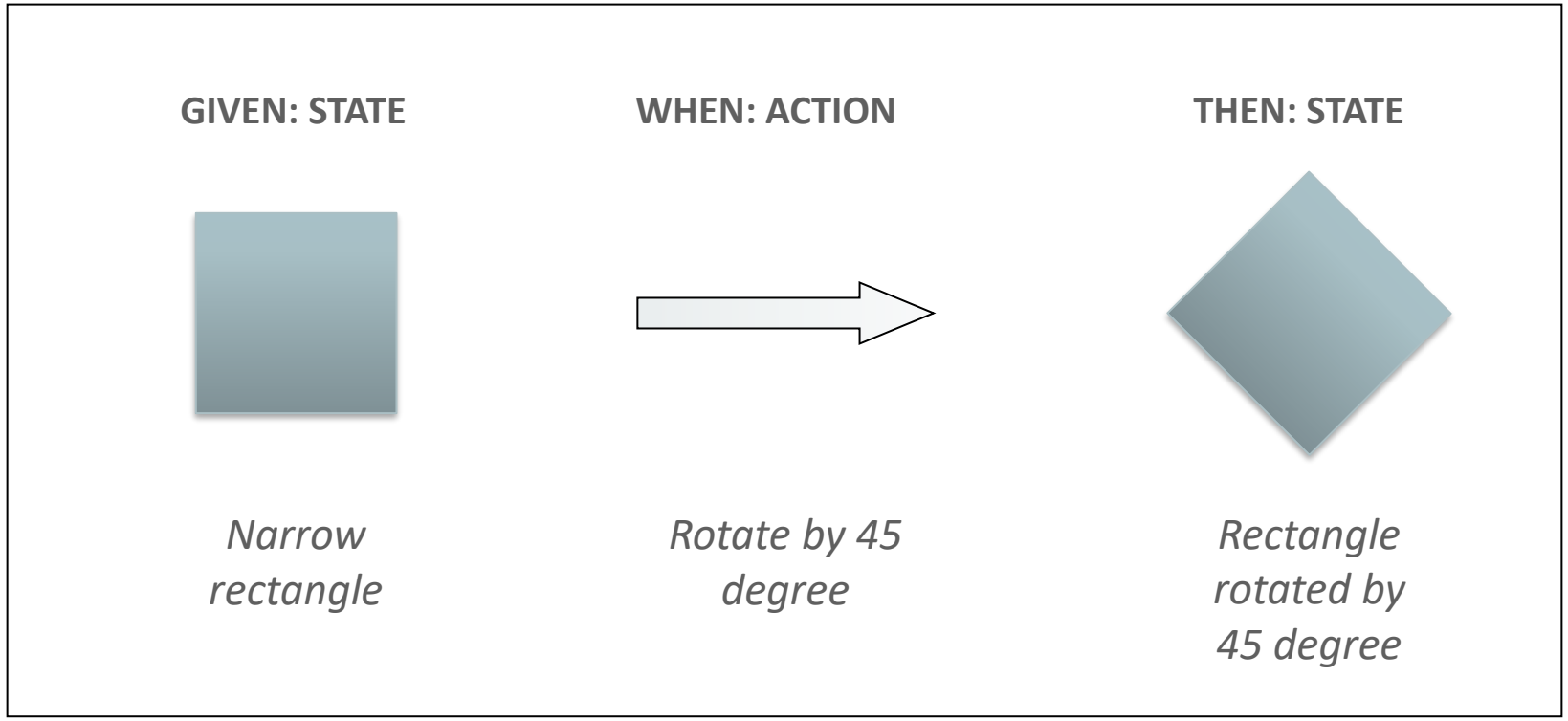
# BDD

# What is BDD?

- Formally:
  - BDD = **B**ehavior **D**riven **D**evelopment
- By meaning:
  - BDD = Performing actions from behavior point of view instead of functional
- **Behavior Driven Development:**
  - A software development process that emerged from test-driven development (TDD) Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.

# What is BDD? cont.

- Key idea – code should represent a state transition:



# BDD for TDD

- BDD is applicable on all levels of testing

Unit tests

- Given – setting mock objects
- When – calling code under test
- Then – verifying what was done

Integration tests

- Given – setting integration environment
- When – calling code under test
- Then – verifying what was done

Acceptance tests

- Given – accessing target view for required action
- When – performing UI actions
- Then – verifying what was done

# BDD for TDD, test example

- Let’s take an existing test case:

```
@Test
public void testEnrollment_BAD_REQUEST_AppAttributes_AppVersion_Empty() {
      AppEnrollmentRequest request = createAppRequestWithAttributes(appName());
      request.setVersion("");
      ClientResponse clientResponse = appEnrollment(request);
      assertFailureHttpStatusOnly(clientResponse, ClientResponse.Status.BAD_REQUEST);
}
```

- Now we will rewrite it in terms of BDD notation:

```
@Test
public void testEnrollmentShouldReturnBadRequetWhenAppVersionIsEmpty() {
        givenEnrollmentRequest();         //This will create a request instance in class scope
        givenRequestAppVersionIsEmpty(); //Setting request app version to "“
        ClientResponse clientResponse = whenEnrolmentRequestSent(); //Performing method call
        thenResponseHasStatusBadRequest(clientResponse);            //Validating result
}
```

# BDD: benefits

- Significant improvement of code readability
  - Test cases written in BDD notation looks pretty much like a normal English text
  - BDD notation much better reflects target requirement which validated by a test
- Increasing level of code reuse
  instead of having this in each test:

```
AppEnrollmentRequest request = createAppRequestWithAttributes(appName());
```

  we will have this:

```
givenEnrollmentRequest();
```

  And if method signature will be changed – only given method will be affected instead of 30 tests

# BDD: benefits cont.

- Better support from any IDE
  - When you'll write a test and type just given – you'll see all the ready to use initial steps. Same for when and then.
- Much better and meaningful stack trace exceptions
  - They will not only tell that some string has value which is not expected, but there will be something like this:
    testcase **`testEnrollmentShouldReturnBadRequetWhenAppVersionIsEmpty`** failed because **`thenResponseHasStatusBadRequest`** failed. So you received not comparison of expected status and received, but explanation of a problem

# References

- Code of the Damned
  - Test Driven Development
  - The Purpose of a Unit Test
  - Unit Test Frameworks
- Books
  - xUnit Test Patterns: Refactoring Test Code
    by Gerard Meszaros
  - Working Effectively with Legacy Code
    by Michael C. Feathers, Author of CppUnit(Lite)
  - Test Driven Development: By Example
    by Kent Beck, Author of first xUnit framework
  - Fowler, Martin. Refactoring. (http://martinfowler.com/refactoring/)
  - Osherove, R. (2012). "Write Maintainable Unit Tests That Will Save You Time and Tears". *MSDN Magazine.* http://msdn.microsoft.com/en-us/magazine/cc163665.aspx
  - (2012) Cunningham & Cunningham. (http://c2.com/cgi/wiki?TestDrivenDevelopment)

Q&A?

Thank you!