Developer Testing
Part 1: Microtests and TDD

Johannes Link

Lecture at Friedrich-Alexander University
Erlangen-Nürnberg

@johanneslink

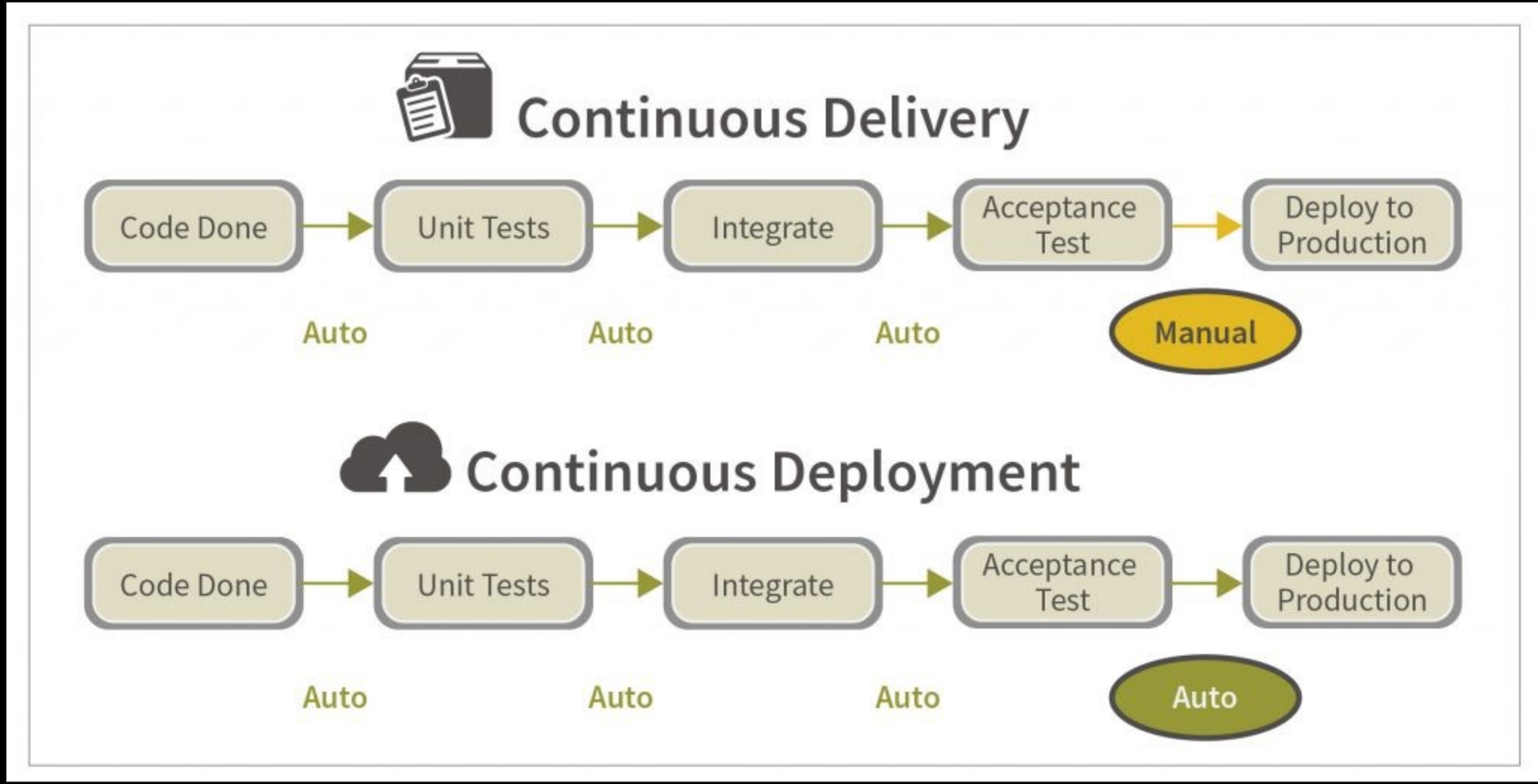
johanneslink.net

Software Therapist

"In Germany the title Therapist by itself or complemented with certain terms is not protected by law. Therefore it does not describe a successfully completed professional training, not even professional expertise."

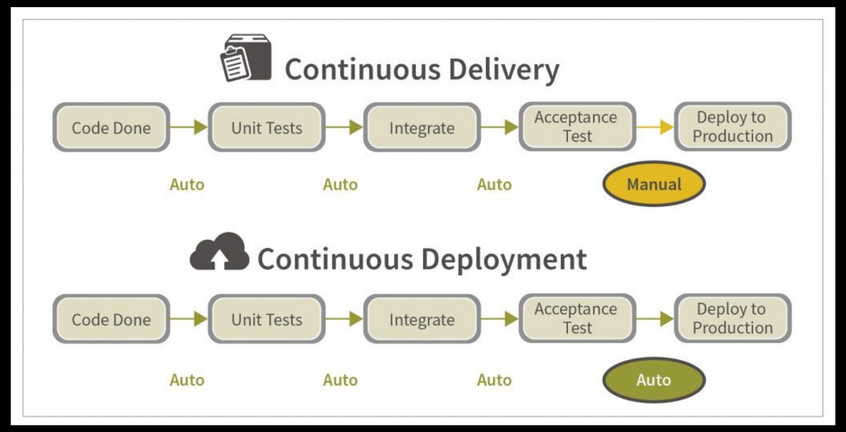
Translated from German Wikipedia "Therapeut"

Why do we need Test Automation?



from: http://www.softcrylic.com/blogs/testing-strategies-continuous-delivery/

Why do we need Test Automation?



We need Feedback:

- Fast
- Reliable
- Up to date

➡ Test Automation must be tightly integrated with development

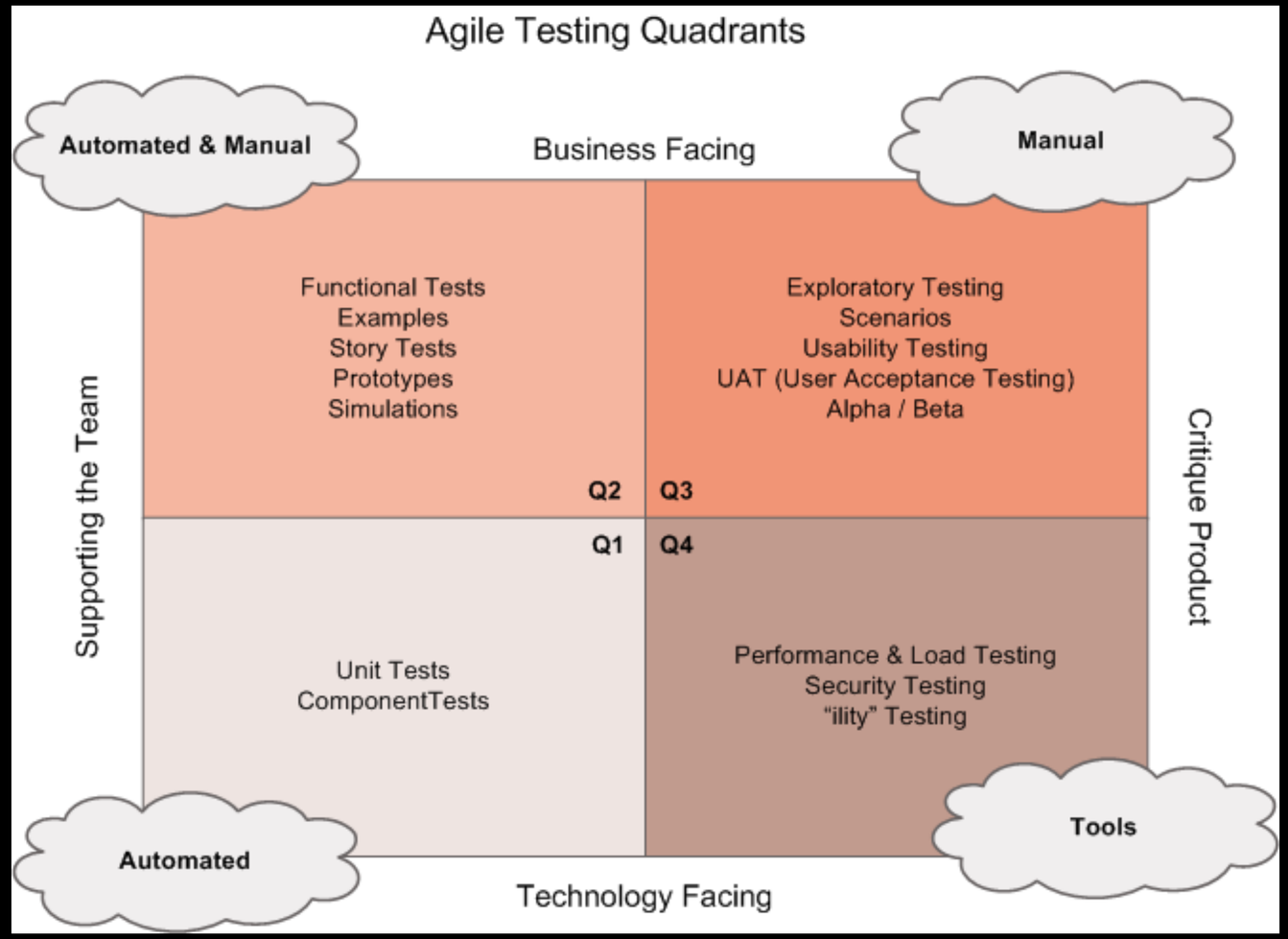
What do we want to learn from our tests?

- Have we developed,
  what we wanted to develop?
- Have we developed,
  what the customer needs?

We need more than one testing approach

- Developers write automated tests to verify their own code
- The customer (analyst, expert, …) specifies acceptance tests to verify their functional requirements and expectations

Agile Testing Quadrants



from: https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/

Goals of Developer Testing

- Validate your expectations
- Find bugs or prevent them

- Absence of bugs cannot be proven with automated tests!

Tools for Developer Testing

- Tight integration with normal development
- Fast turn-around
- Small semantic gap to rest of source code

JUnit

junit.org

- Tests are written in Java
- Tests can be run from IDE
- Tests are supported by build tools
- Current versions
  - JUnit Platform: 1.3.2
  - Jupiter: 5.3.2

JUnit Demo

```java
class CalculatorTests {
    private Calculator calculator;

    @BeforeEach
    void initializeCalculator() {
        calculator = new Calculator();
    }

    @Test
    void resultIsInitiallyZero() {
        assertEquals("0.00", calculator.result());
    }

    @Test
    void addingUpNumbers() {
        calculator.add(2.00);
        assertEquals("2.00", calculator.result());

        calculator.add(42.11);
        assertEquals("44.11", calculator.result());
    }

    @Test
    void upTo6DecimalsAreShown() {
        calculator.add(0.000001);
        assertEquals("0.000001", calculator.result());

        calculator.add(0.0000001);
        assertEquals("0.000001", calculator.result());
    }
}
```

Structure of a Test Case

- Any class can be container for test cases
- Test cases are methods
  `@Test public void myTest()`
- Check expectations with assertion methods
  `Assertions.assert...()` und
  `Assertions.fail()` für Zusicherungen
- Failing assertion will stop the current test case

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

class EuroTest {
   @Test
   void amount() {
      Euro two = new Euro(2.00);
      assertTrue(two.getAmount() == 2.00);
   }
}
```

Test Container Classes

- Use instance variables for common test fixture
- `@BeforeEach public void ...()`
  to setup test fixture and required resources
- `@AfterEach public void ...()`
  to release resources (if necessary)

```
class EuroTest {
   private Euro two;

   @BeforeEach
   void initialize() {
      two = new Euro(2.00);
   }

   @Test
   void adding() {
      Euro sum = two.plus(two);
      assertEquals(new Euro(4.00), sum);
      assertEquals(new Euro(2.00), two);
   }
}
```

Important Methods in *Assertions*

```
assertTrue(boolean condition)

assertFalse(boolean condition)

assertEquals(Object expected, Object actual)

assertEquals(double expected, double actual, double delta)

assertSame(Object expected, Object actual)

assertNull(Object actual)

assertNotNull(Object actual)

assert…(…, String description)
```

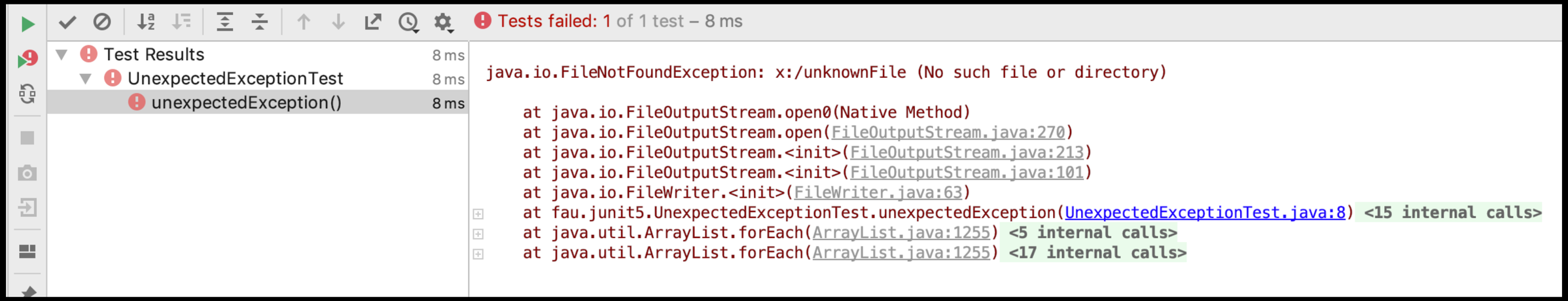
Expected Exceptions

```
@Test
void cannotCreateNegativeEuroAmount() {
   assertThrows(IllegalArgumentException.class, () -> {
      final double NEGATIVE_AMOUNT = -2.00;
      new Euro(NEGATIVE_AMOUNT);
   });
}
```

Check robust behaviour in case of exceptions!

Unexpected Exceptions are Recorded as Test Failure!

```
class UnexpectedExceptionTest {
   @Test
   void unexpectedException() throws Exception {
      new java.io.FileWriter("x:/unknownFile");
   }
}
```



More Jupiter Features

@Disabled("Not yet finished")

@BeforeAll, @AfterAll

@Tag("fast")

@DisplayName("size should return # of elements")

- Parameter Injection
- Extension-Model

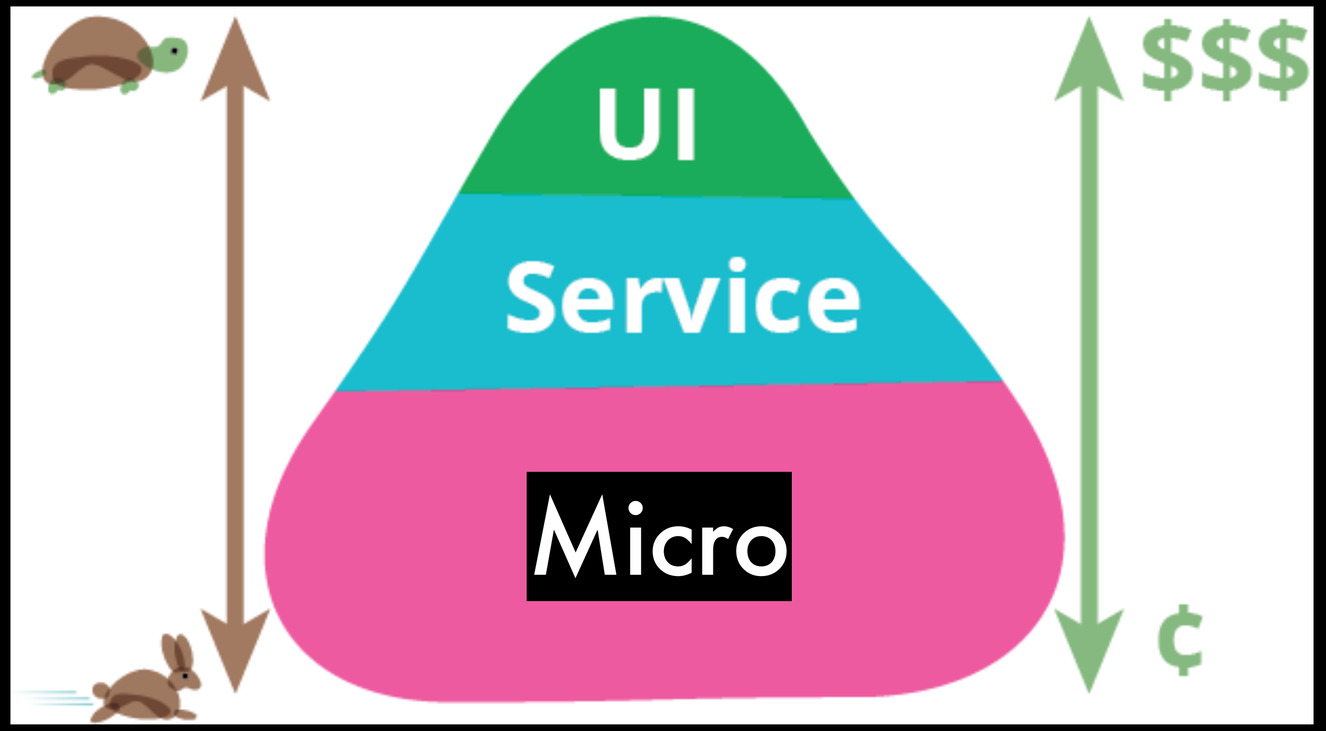
Structure of a Test Case (AAA)

- Arrange:
  Create the object under test and set it up for testing
- Act:
  Invoke the behaviour you want to check
- Assert:
  Verify the expected results

Arrange - Act - Assert

```
class EuroTest…
   private Euro two;

   @BeforeEach
   void initialize() {
      two = new Euro(2.00);  Arrange
   }

   @Test
   void adding() {
      Euro sum = two.plus(two);  Act
      assertEquals(new Euro(4.00), sum);
      assertEquals(new Euro(2.00), two);
   }                                  Assert
```

Test Automation Pyramid



from: https://martinfowler.com/bliki/TestPyramid.html

Video Mike-Hill

Microtests

formerly known as Unit Tests

- fast
- short
- precise
- allow checking of details
- effort scales linearly

Terminology

- Test Case and Test Suite
- White-Box vs Black-box Testing
- Integration Tests vs Integrated Tests

Test-driven Development

TDD

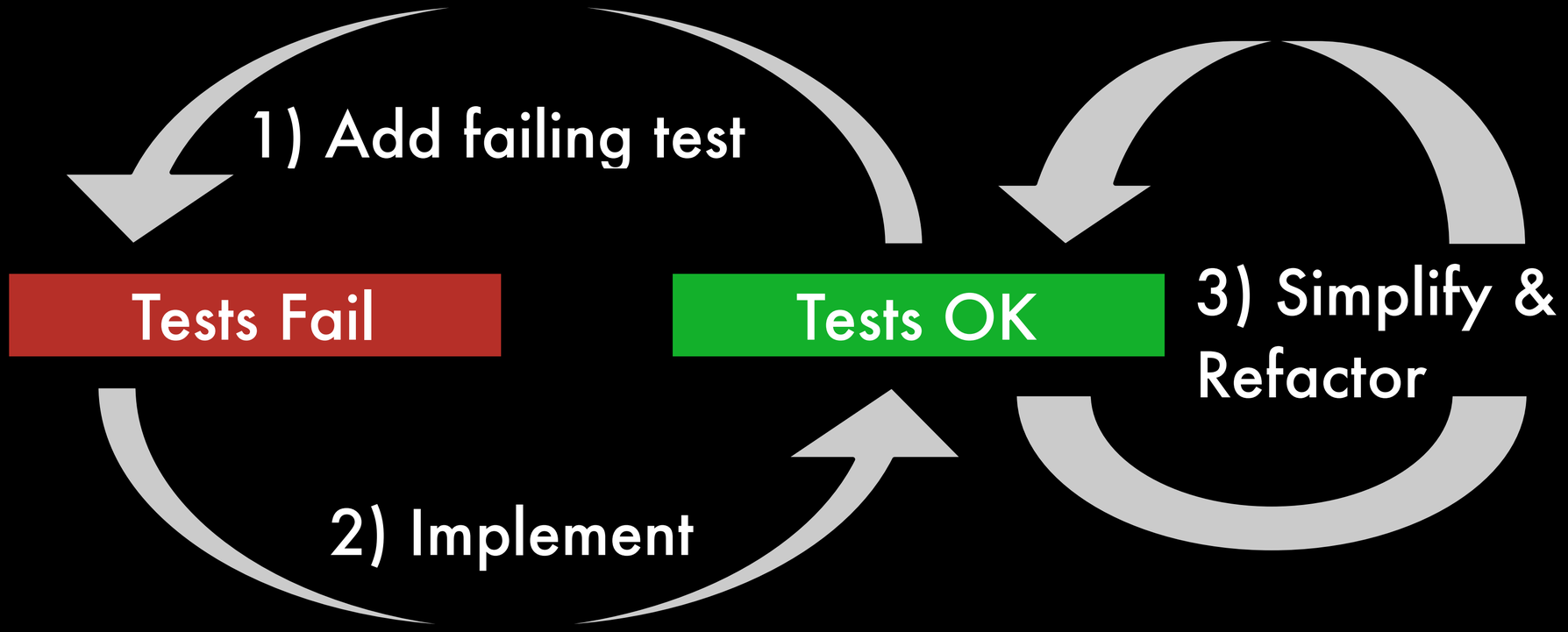
What is TDD?

- Developers write automated tests as they go
- Tests are written in advance of the code
- Design a little at a time

Why do I practice TDD?

- Tests secure the current functionality
- Refactoring prolongs the productive life of our software
- Writing automated tests afterwards can be difficult
  - Testability as basic requirement
  - You never have time at the end
- Small steps enforce continuous progress and regular feedback

Test / Code / Refactor



Test/Code/Refactor – Zyklus

grün-rot: Schreibe einen Test, der zunächst fehlschlagen sollte. Schreibe gerade soviel Code, dass der Test kompiliert.

rot-grün: Schreibe gerade soviel Code, dass alle Tests laufen.

grün-grün: Eliminiere Duplikation und andere üble Codegerüche.

TDD Demo

Prime factorization

How many tests are enough?

- Every missing test means elevated risk
- Every redundant test is a lost investment
- Test code must be maintained!

➡ Test the essential things

➡ Avoid duplicated tests

➡ Choose next test based on risk and learning potential

➡ You're allowed to throw tests away

Heuristics

- Check the main path
- Check the border cases
- Check error behaviour
- Document preconditions

Structuring and Naming Tests and Test Containers

- Meaningful and Navigable
  - What behaviour can I expect?
  - Which test must be changed?
- Robustness during Changes and Refactorings
  - Change implementation of function
  - Change name of class or function
  - Change signature of function
  - Change location of function

Overall Structure

- Every module has its own set of microtests
- Put tests in same package as module under tests
- Integrated Tests live in module(s) of their own

Debatable Conventions

- One test class per domain class, e.g.
  `class MyObject` is tested in `MyObjectTest`
- Name tests like the methods they test, e.g.

```
class Stack { void push(Object element) }
@Test pushTest() {}
```

- Enforce 1:n relation between public methods and test cases

Naming Tests: Basic Style

"Describe the feature under test"

```
Class AccountTests…
   creatingAnAccount()
   withdrawing()
   withdrawingNegativeAmount()
   withdrawingAmountNotCovered()
```

Naming Tests: Advanced Style

"Describe the feature, an optional context, and the expected outcome"

```
Class AccountTests…
   aNewAccount_returnsCustomer()
   aNewAccount_hasZeroBalance()
   withdrawingAmount_reducesBalanceByAmount()
   withdrawingNegativeAmount_failsWithException()
   withdrawingNegativeAmount_doesNotChangeBalance()
```

Naming Tests: Advanced Style

"Describe the feature, an optional context, and the expected outcome"

```
Class AccountTests…
  @Nested class NewAccount…
    returnsCustomer()
    hasZeroBalance()
  @Nested class WithdrawingMoney…
    reducesBalanceByAmount()
    @Nested class WithNegativeAmount
      failsWithException()
      doesNotChangeBalance()
```

Structure and Interpretation of Test Cases

Kevlin Henney: https://vimeo.com/289852238

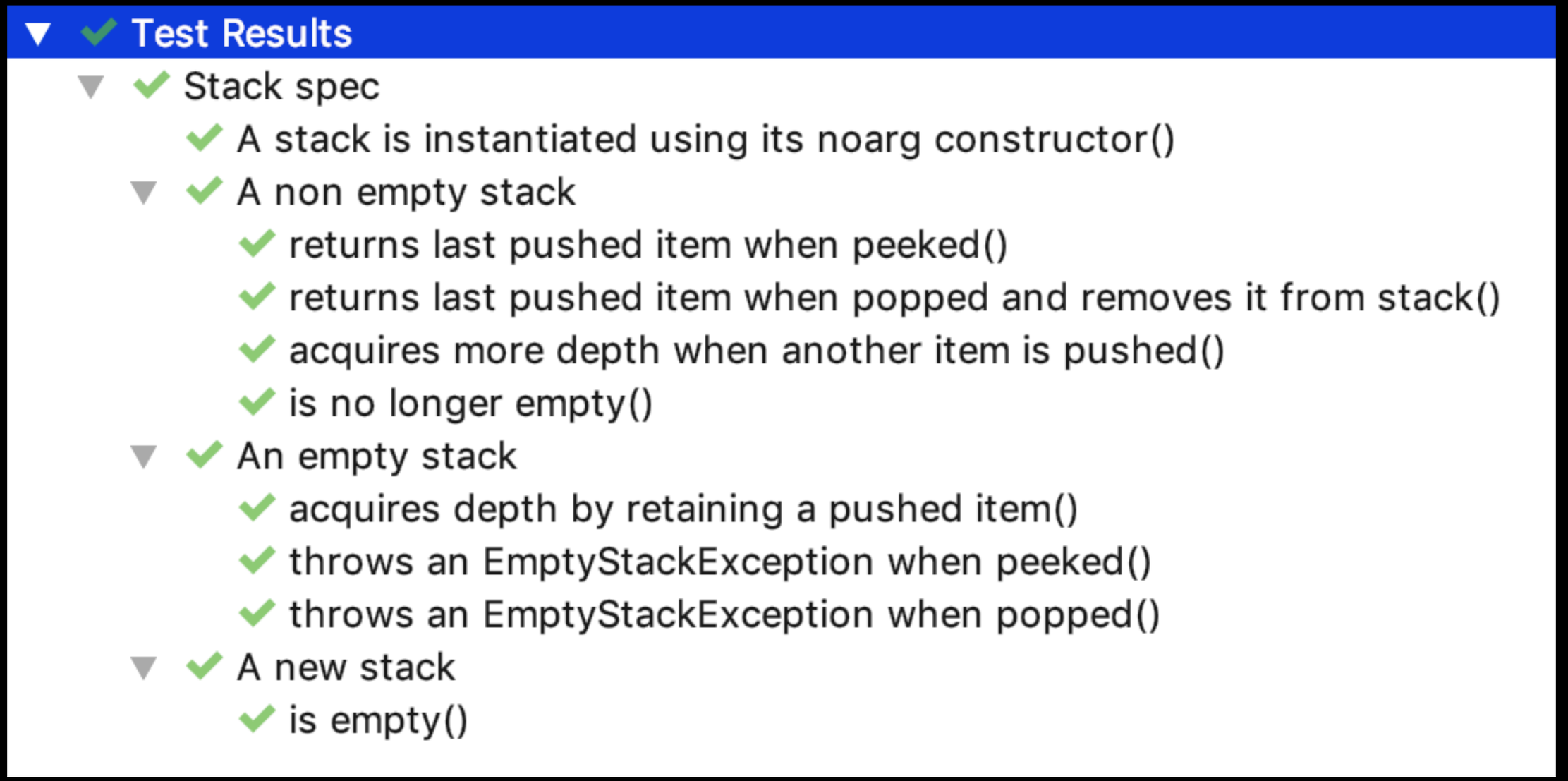
```
class Leap_year_spec {

    @Nested
    class A_year_is_a_leap_year {
        @Test
        void if_it_is_divisible_by_four_but_not_by_100() {}
        @Test
        void if_it_is_divisible_by_400() {}
    }

    @Nested
    class A_year_is_not_a_leap_year {
        @Test
        void if_it_is_not_divisible_by_four() {}
        @Test
        void if_it_is_divisible_by_100_but_not_by_400() {}
    }
}
```

Structure and Interpretation of Test Cases

Kevlin Henney: https://vimeo.com/289852238

```
class Leap_year_spec {
    @Nested
    class A_year_is_a_leap_year {
        @Test
        void if_it_is_divisible_by_four_but_not_by_100() {}
        @Test
        void if_it_is_divisible_by_400() {}
    }

    @Nested
    class A_year_is_not_a_leap_year {
        @Test
        void if_it_is_not_divisible_by_four() {}
        @Test
        void if_it_is_divisible_by_100_but_not_by_400() {}
    }
}
```

```java
@DisplayNameGeneration(DisplayNameGenerator.ReplaceUnderscores.class)
class Stack_spec {
    Stack<Object> stack;

    @Test
    void A_stack_is_instantiated_using_its_noarg_constructor() {
        new Stack<>();
    }

    @Nested
    class A_new_stack {

        @BeforeEach
        void createNewStack() {
            stack = new Stack<>();
        }

        @Test
        void is_empty() {
            assertTrue(stack.isEmpty());
        }
    }

    @Nested
    class An_empty_stack {…}

    @Nested
    class A_non_empty_stack {…}
}
```


Test Results
Stack spec
A stack is instantiated using its noarg constructor()
A non empty stack
returns last pushed item when peeked()
returns last pushed item when popped and removes it from stack()
acquires more depth when another item is pushed()
is no longer empty()
An empty stack
acquires depth by retaining a pushed item()
throws an EmptyStackException when peeked()
throws an EmptyStackException when popped()
A new stack
is empty()

Quality of a Test Suite

- Who is testing the tests?
  - The application code itself
  - Code Coverage Metrics
    - integrated in many IDEs
  - Mutation Testing
    - http://pitest.org/
- Metrics can be very helpful for evaluation and improvement
- Metrics should not be a measured target

Code:

http://github.com/jlink/tdd-fau

Slides:

http://github.com/jlink/tdd-fau/slides