An Introduction into JUnit

Praxis der Software-Entwicklung WS 2012/13

Daniel Bruns Erik Burger | 13.02.2013

INSTITUT FÜR THEORETISCHE INFORMATIK – INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION

KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

Foreword

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> Dijkstra, 1972

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Functional Tests

- Correctness according to specification
- Concurrency/Thread safeness

Non-Functional

- Performance
- Security
- Usability
- Interoperability
- Reliability

Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System

Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System

Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System

Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System

Classification of Tests

Knowledge

- black-box tests
- white-box tests

Structure

- Unit
- Integration
- System

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
    - Lack of modularisation?
    - Bad design?
    - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
  - Lack of modularisation?
  - Bad design?
  - Bad code dependency management
- ⇒ Refactoring

Testing and Design Validation

- Test represents typical usage scenarios
- Less dependencies ⇒ easier to use
- High degree of dependencies
    - Lack of modularisation?
    - Bad design?
    - Bad code dependency management
- ⇒ Refactoring

Test Coverage

Types of coverage

- Statement coverage (Anweisungsüberdeckung)
- Branch coverage (Zweigüberdeckung)
- Path coverage (Pfadüberdeckung)
- . . . several others

Test Coverage

Types of coverage

- Statement coverage (Anweisungsüberdeckung)
- Branch coverage (Zweigüberdeckung)
- Path coverage (Pfadüberdeckung)
- . . . several others

Test Coverage

Types of coverage

- Statement coverage (Anweisungsüberdeckung)
- Branch coverage (Zweigüberdeckung)
- Path coverage (Pfadüberdeckung)
- . . . several others

Test Coverage

Types of coverage

- Statement coverage (Anweisungsüberdeckung)
- Branch coverage (Zweigüberdeckung)
- Path coverage (Pfadüberdeckung)
- . . . several others

Test Coverage

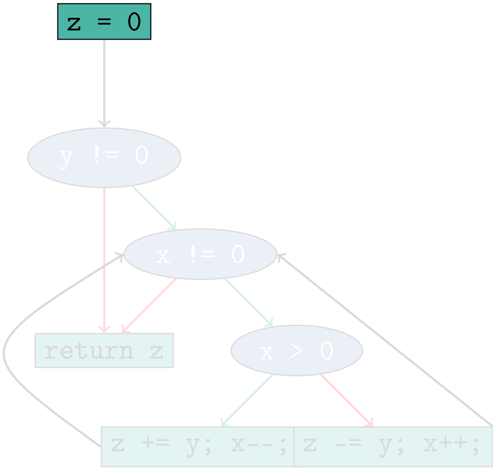
Types of coverage

- Statement coverage (Anweisungsüberdeckung)
- Branch coverage (Zweigüberdeckung)
- Path coverage (Pfadüberdeckung)
- . . . several others

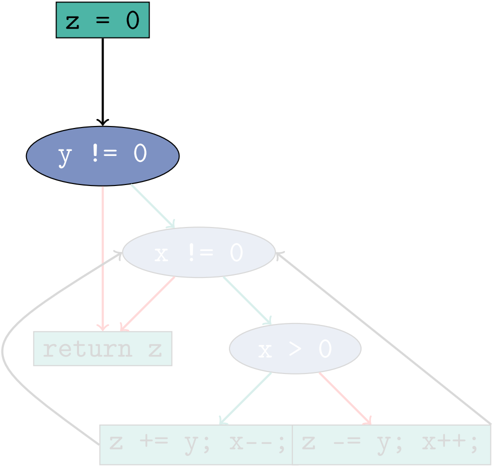
Example Method

```java
public int foo (int x, int y) {
    int z = 0;
    if (y != 0) {
        while (x != 0) {
            if (x > 0) {
                z += y;
                x--;
            } else {
                z -= y;
                x++;
            }
        }
    }
    return z;
}
```

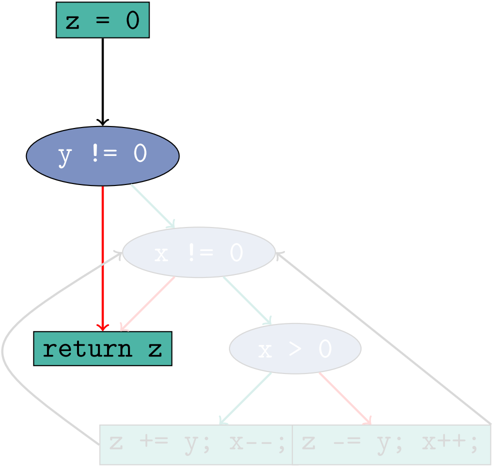
Control Flow Graph



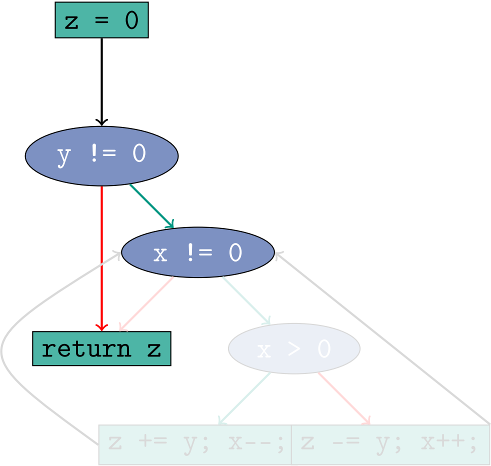
Control Flow Graph



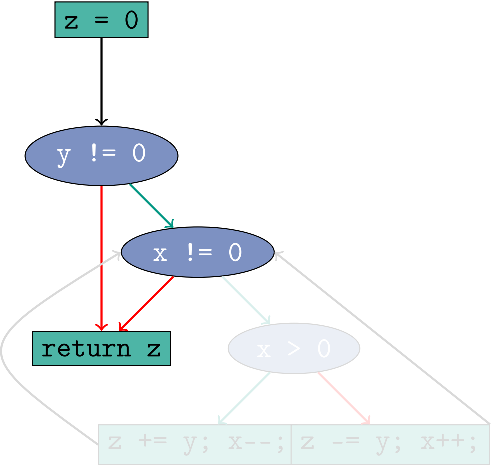
Control Flow Graph



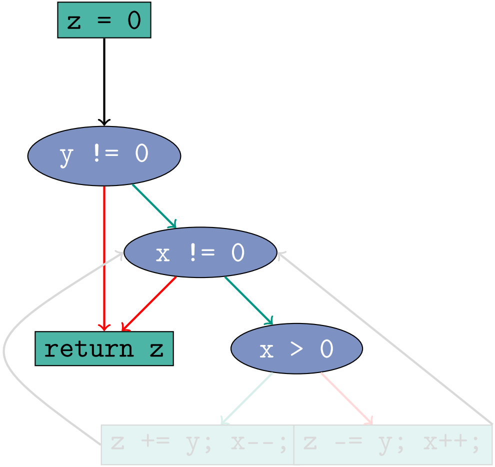
Control Flow Graph



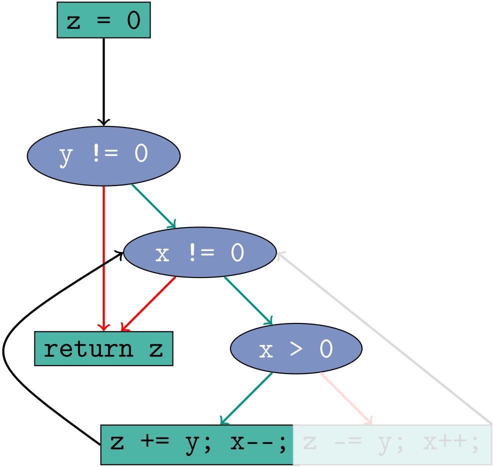
Control Flow Graph



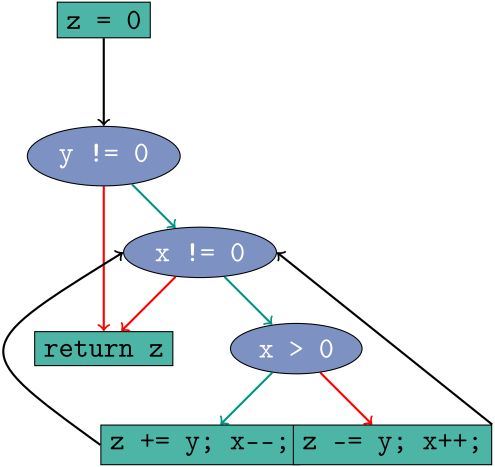
Control Flow Graph



Control Flow Graph



Control Flow Graph



Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Equivalence classes:
- Full path coverage would require $2^{32} + 1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Equivalence classes:
  $\{\{(x, y) \mid y = 0\}, \{(x, y) \mid y \neq 0 \wedge x > 0\}, \{(x, y) \mid y \neq 0 \wedge x \leq 0\}\}$
- Full path coverage would require $2^{32} + 1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Candidate values:
  $\{(0,0),(23,42),(-23,666)\}$
- Full path coverage would require $2^{32}+1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Candidate values:
  $\{(0,0),(23,42),(-23,666)\}$
- Full path coverage would require $2^{32}+1$ tests!

Extreme values

- Variant of equivalence classes approach
- "Off-by-one" most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Candidate values:
  $\{(0,0),(23,42),(-23,666)\}$
- Full path coverage would require $2^{32}+1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Candidate values:
  $\{(0,0),(23,42),(-23,666)\}$
- Full path coverage would require $2^{32}+1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

Test Design

Equivalence classes

- Assumption: similar control flow for similar values
- Last example: only 3 test needed for full branch coverage
- Candidate values:
  $\{(0,0),(23,42),(-23,666)\}$
- Full path coverage would require $2^{32}+1$ tests!

Extreme values

- Variant of equivalence classes approach
- “Off-by-one” most prominent error
- Extreme values for integers: `MIN_VALUE`, -1, 0, 1, `MAX_VALUE`, `someArray.length`
- Extreme values for objects: `null`, empty strings, empty collections

JUnit

3 Overview

4 Assertions

5 Fixtures

- Definition
- Example
- Parameterised Tests
- Test Suites

6 Eclipse Integration

- Test Runners

JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

Karlsruhe Institute of Technology

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

JUnit 3 vs JUnit 4

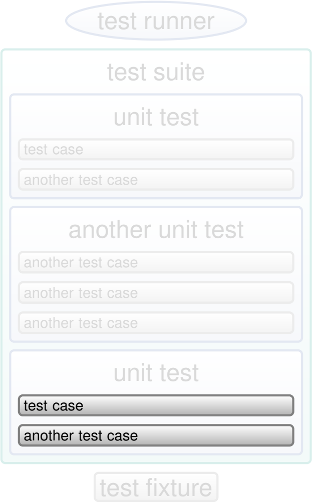
JUnit4

- JUnit4 was a complete redevelopment
- includes ideas from other frameworks and uses features of Java 1.5
- uses Java annotations (like `@Test`)
- This lecture is based on JUnit 4

Be careful

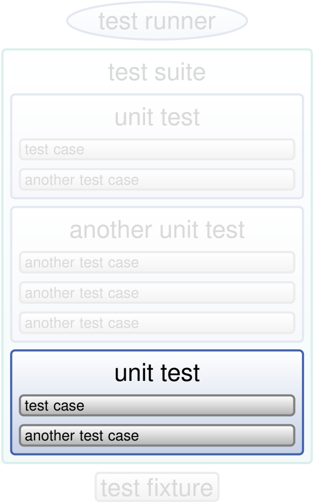
- Many (web) tutorials are still based on JUnit 3
- JUnit 4 is backwards compatible to version 3
- but JUnit 4 is much cleaner

Structure



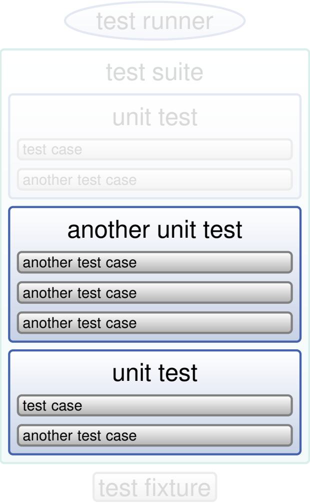
- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



test runner

test suite

unit test

test case

another test case

another unit test

another test case

another test case

another test case

unit test

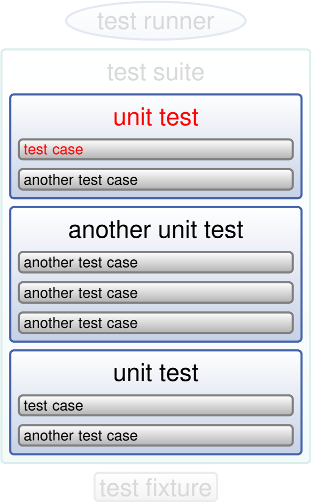
test case

another test case

test fixture

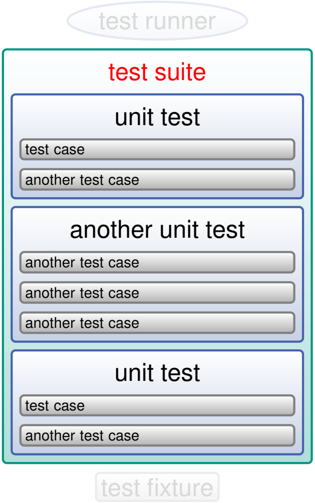
- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



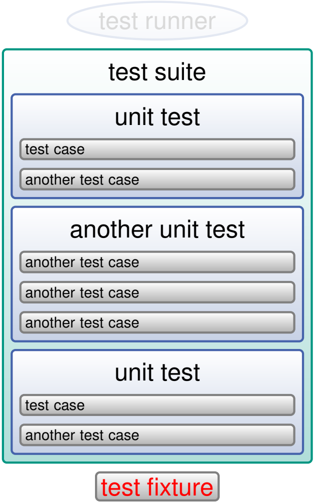
- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



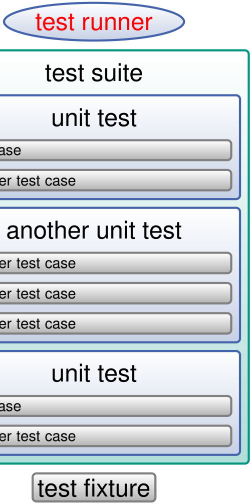
- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Structure



- A unit test is the smallest unit that says pass or fail. It should cover one or more methods of one class.
- A test case is a simple assertion, e.g. `x >= 0`.
- You can have multiple test cases in a single unit test.
- A test suite combines unit tests.
- The test fixture provides software support for all this.
- The test runner runs unit tests or an entire test suite.

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
    - An unexpected event occurred during the test case.
    - The test case could not be set up properly

Test Case Verdicts

- A verdict is the declared result of executing a single test.
- **Pass**: the test case achieved its intended purpose, and the software under test performed as expected.
- **Fail**: the test case achieved its intended purpose, but the software under test did not perform as expected.
- **Error**: the test case did not achieve its intended purpose. Potential reasons:
  - An unexpected event occurred during the test case.
  - The test case could not be set up properly

What is a JUnit Test?

A test “script” is just a collection of Java methods.

General idea is to create a few Java objects, do something interesting with them, and then determine if the objects have the correct properties.

What is added? Assertions.

- A package of methods that checks for various properties:
  - “equality” of objects
  - identical object references
  - null / non-null object references
- The assertions are used to determine the test case verdict.

What is a JUnit Test?

A test “script” is just a collection of Java methods.

General idea is to create a few Java objects, do something interesting with them, and then determine if the objects have the correct properties.

What is added? Assertions.

- A package of methods that checks for various properties:
  - “equality” of objects
  - identical object references
  - null / non-null object references
- The assertions are used to determine the test case verdict.

Organisation of JUnit Tests

- Each method represents a single test case that can independently have a verdict (pass, error, fail).
- Normally, all the tests for one Java class are grouped together into a separate class.
- Naming convention:
  - Class to be tested: `Value`
  - Class containing tests: `ValueTest`

Writing a JUnit test class

Start by importing these JUnit 4 classes

```
import org.junit.*
import static org.junit.Assert.*; // note static import
```

Declare your test class in the usual way

```
public class MyProgramTest {
}
```

Declare an instance of the class being tested

```
public class MyProgramTest {
    MyProgram program;
    int someVariable;
}
```

A simple example

```java
import org.junit.*;
import static org.junit.Assert.*;
public class ArithmeticTest {
    @Test
    public void testMultiply() {
        assertEquals(4, Arithmetic.multiply(2, 2));
        assertEquals(-15, Arithmetic.multiply(3, -5));
    }

    @Test
    public void testIsPositive() {
        assertTrue(Arithmetic.isPositive(5));
        assertFalse(Arithmetic.isPositive(-5));
        assertFalse(Arithmetic.isPositive(0));
    }
}
```

Assertions

Assertions are defined in the JUnit class Assert

- If an assertion is true, the method continues executing.
- If any assertion is false, the method stops executing at that point, and the result for the test case will be fail.
- If any other exception is thrown during the method, the result for the test case will be error.
- If no assertions were violated for the entire method, the test case will pass.

All assertion methods are static methods.

Assertions

Assertions are defined in the JUnit class Assert

- If an assertion is true, the method continues executing.
- If any assertion is false, the method stops executing at that point, and the result for the test case will be **fail**.
- If any other exception is thrown during the method, the result for the test case will be **error**.
- If no assertions were violated for the entire method, the test case will **pass**.

All assertion methods are static methods.

Assertions

Assertions are defined in the JUnit class Assert

- If an assertion is true, the method continues executing.
- If any assertion is false, the method stops executing at that point, and the result for the test case will be **fail**.
- If any other exception is thrown during the method, the result for the test case will be **error**.
- If no assertions were violated for the entire method, the test case will **pass**.

All assertion methods are static methods.

Assertions

Assertions are defined in the JUnit class Assert

- If an assertion is true, the method continues executing.
- If any assertion is false, the method stops executing at that point, and the result for the test case will be **fail**.
- If any other exception is thrown during the method, the result for the test case will be **error**.
- If no assertions were violated for the entire method, the test case will **pass**.

All assertion methods are static methods.

Assertion Methods

Boolean conditions are true or false

```
assertTrue(condition)
assertFalse(condition)
```

Objects are null or non-null

```
assertNull(object)
assertNotNull(object)
```

Objects are identical (i.e. two references to the same object), or not identical.

```
assertSame(expected, actual)
assertNotSame(expected, actual)
```

Assertion Methods

“Equality” of objects

`assertEquals(expected, actual)`

valid if: `expected.equals(actual)`

“Equality” of arrays

`assertArrayEquals(expected, actual)`

- arrays must have same length
- for each valid value for `i`, check as appropriate:

```
assertEquals(expected[i],actual[i])
assertArrayEquals(expected[i],actual[i])
```

There is also an unconditional failure assertion `fail()` that always results in a fail verdict.

Assertion Methods

“Equality” of objects

```
assertEquals(expected, actual)
```

valid if: `expected.equals(actual)`

“Equality” of arrays

```
assertArrayEquals(expected, actual)
```

- arrays must have same length
- for each valid value for `i`, check as appropriate:

```
assertEquals(expected[i],actual[i])
assertArrayEquals(expected[i],actual[i])
```

There is also an unconditional failure assertion `fail()` that always results in a fail verdict.

Assertion Methods

“Equality” of objects

`assertEquals(expected, actual)`

valid if: `expected.equals(actual)`

“Equality” of arrays

`assertArrayEquals(expected, actual)`

- arrays must have same length
- for each valid value for `i`, check as appropriate:

```
assertEquals(expected[i],actual[i])
assertArrayEquals(expected[i],actual[i])
```

There is also an unconditional failure assertion `fail()` that always results in a fail verdict.

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Test Fixture

Test Fixture

- A test fixture is the context in which a test case runs.
- Typically, test fixtures include:
  - Objects or resources that are available for use by any test case.
  - Activities required to make these objects available and/or resource allocation and de-allocation: “setup” and “teardown”.
- Allows multiple tests of the same or similar objects
- Share fixture code for multiple tests

Before/After

- `@Before:` Methods annotated with `@Before` are executed before every test.
- `@After:` Methods annotated with `@After` are executed after every test.
- If there are e.g. 10 test, every `@Before` method is executed 10 times
- More than one `@Before` or `@After` is allowed. But: Call sequence of methods is not defined in JUnit!
- Names of these methods are irrelevant, but must be `public void`

Before/After

- `@Before:` Methods annotated with `@Before` are executed before every test.
- `@After:` Methods annotated with `@After` are executed after every test.
- If there are e.g. 10 test, every `@Before` method is executed 10 times
- More than one `@Before` or `@After` is allowed. But: Call sequence of methods is not defined in JUnit!
- Names of these methods are irrelevant, but must be `public void`

Before/After

- `@Before:` Methods annotated with `@Before` are executed before every test.
- `@After:` Methods annotated with `@After` are executed after every test.
- If there are e.g. 10 test, every `@Before` method is executed 10 times
- More than one `@Before` or `@After` is allowed. But: Call sequence of methods is not defined in JUnit!
- Names of these methods are irrelevant, but must be `public void`

Before/After

- @Before: Methods annotated with @Before are executed before every test.
- @After: Methods annotated with @After are executed after every test.
- If there are e.g. 10 test, every @Before method is executed 10 times
- More than one @Before or @After is allowed. But: Call sequence of methods is not defined in JUnit!
- Names of these methods are irrelevant, but must be public void

Before/After

- `@Before:` Methods annotated with `@Before` are executed before every test.
- `@After:` Methods annotated with `@After` are executed after every test.
- If there are e.g. 10 test, every `@Before` method is executed 10 times
- More than one `@Before` or `@After` is allowed. But: Call sequence of methods is not defined in JUnit!
- Names of these methods are irrelevant, but must be `public void`

Fixture – Example

```
public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    @Before
    public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
}
```

Setup and Teardown

Karlsruhe Institute of Technology

Setup

Use the `@Before` annotation on a method containing code to run before each test case.

Teardown (regardless of the verdict)

Use the `@After` annotation on a method containing code to run after each test case. These methods will run even if exceptions are thrown in the test case or an assertion fails.

It is allowed to have any number of these annotations

All methods annotated with `@Before` will be run before each test case, but they may be run in any order.

Setup and Teardown

Setup

Use the `@Before` annotation on a method containing code to run before each test case.

Teardown (regardless of the verdict)

Use the `@After` annotation on a method containing code to run after each test case. These methods will run even if exceptions are thrown in the test case or an assertion fails.

It is allowed to have any number of these annotations

All methods annotated with `@Before` will be run before each test case, but they may be run in any order.

Setup and Teardown

Setup

Use the `@Before` annotation on a method containing code to run before each test case.

Teardown (regardless of the verdict)

Use the `@After` annotation on a method containing code to run after each test case. These methods will run even if exceptions are thrown in the test case or an assertion fails.

It is allowed to have any number of these annotations

All methods annotated with `@Before` will be run before each test case, but they may be run in any order.

BeforeClass/AfterClass

- `@BeforeClass`: executed once before a test suite
- `@AfterClass`: executed once after a test suite
- Only one `@BeforeClass` and `@AfterClass` allowed
- Methods must be `static`

Fixture – Example

```
public class MoneyTest {
    private static string currency;

    @BeforeClass
    public static void setGlobalCurrency() {
        this.currency = "CHF";
    }

    @Before
    public void setUp() {
        m12= new Money(12, this.currency);
        m14= new Money(14, this.currency);
    }
}
```

Expected Exception

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.

Expected Exception

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.

Expected Exception

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.

Expected Exception

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.

Expected Exception

- Exceptions that are expected on test executing
- Annotation using `@Test`
- `@Test(expected=MyException.class)`
- If no exception is thrown, or an unexpected exception occurs, the test will fail.
- That is, reaching the end of the method with no exception will cause a test case failure.

Expected Exceptions – example

```
new ArrayList<Object>().get(0);
```

- Should throw an `IndexOutOfBoundsException`

```
@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}
```

Ignore/Timeout

Ignore

- Tests annotated using `@Ignore` are not executed
- Test runner reports that test was not run
- `@Ignore("Reason")` allows to specify a reason why a test was not run

Timeout

- Test allows to specify a timeout parameter
- `@Test(timeout=10)` fails if the test takes more than 10 milliseconds

Ignore/Timeout

Ignore

- Tests annotated using `@Ignore` are not executed
- Test runner reports that test was not run
- `@Ignore("Reason")` allows to specify a reason why a test was not run

Timeout

- Test allows to specify a timeout parameter
- `@Test(timeout=10)` fails if the test takes more than 10 milliseconds

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Motivation

If you want a test to run with several parameter values, you'd have to

- loop over a collection of values
  - which means if there was a failure, the loop wouldn't terminate
- write unique test cases for each test data combination
  - which could prove to be a lot of coding

Support in JUnit

With JUnit, you can create highly flexible testing scenarios easily

Parameterised Tests

Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation
3. Create class members for the parameter types required in the generic method defined in Step 1
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation

Parameterised Tests

Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation
3. Create class members for the parameter types required in the generic method defined in Step 1
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation

Parameterised Tests

Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation
3. Create class members for the parameter types required in the generic method defined in Step 1
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation

Parameterised Tests

Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation
3. Create class members for the parameter types required in the generic method defined in Step 1
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation

Parameterised Tests

Creating a parameterised test

1. Create a generic test and decorate it with the `@Test` annotation
2. Create a static feeder method that returns a Collection type and decorate it with the `@Parameters` annotation
3. Create class members for the parameter types required in the generic method defined in Step 1
4. Create a constructor that takes these parameter types and correspondingly links them to the class members defined in Step 3
5. Specify the test case be run with the Parameterized class via the `@RunWith` annotation

Parameterised Test – Example

```
@RunWith(Parameterized.class)
public class ParameterizedTest {
    private int numberToTest;
    private int rest;
    public ParameterizedTest(Integer pValue, Integer rValu
        numberToTest = pValue.intValue();
        rest = rValue.intValue();
    }
    @Parameters
    public static List<Integer[]> testValues() {
        return Arrays.asList(new Integer[][] {
            {1,1}, {3,1}, {6,0}, {7,1}, {9,1}
            });
    }
    @Test
    public void isOdd() {
        assertTrue(numberToTest % 2 == rest);
    }
}
```

Test Suites

Creating a test suite

- Tests can be combined to test suites
- suites can contain other suites
- useful for partitioning your test scenarios
- well supported by Test Runners (see example)

Test Suites

Creating a test suite

- Tests can be combined to test suites
- suites can contain other suites
- useful for partitioning your test scenarios
- well supported by Test Runners (see example)

Test Suites

Creating a test suite

- Tests can be combined to test suites
- suites can contain other suites
- useful for partitioning your test scenarios
- well supported by Test Runners (see example)

Test Suites

Creating a test suite

- Tests can be combined to test suites
- suites can contain other suites
- useful for partitioning your test scenarios
- well supported by Test Runners (see example)

Test Suite – Example

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
    MyTest1.class,
    MyTest2.class,
    MyTest3.class
    }
)
public class AllTests {
}
```

Running JUnit Tests

- The JUnit framework does not provide a graphical test runner. Instead, it provides an API that can be used by IDEs to run test cases and a textual runner than can be used from a command line.
- Eclipse and Netbeans each provide a graphical test runner that is integrated into their respective environments.

Test Runners

With the runner provided by JUnit:

- When a class is selected for execution, all the test case methods in the class will be run.
- The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.

Other Runners

- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).

Test Runners

With the runner provided by JUnit:

- When a class is selected for execution, all the test case methods in the class will be run.
- The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.

Other Runners

- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).

Test Runners

With the runner provided by JUnit:

- When a class is selected for execution, all the test case methods in the class will be run.
- The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.

Other Runners

- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).

Test Runners

With the runner provided by JUnit:

- When a class is selected for execution, all the test case methods in the class will be run.
- The order in which the methods in the class are called (i.e. the order of test case execution) is not predictable.

Other Runners

- Test runners provided by IDEs may allow the user to select particular methods, or to set the order of execution.
- It is good practice to write tests with are independent of execution order, and that are without dependencies on the state any previous test(s).