

Unit Testing with JUnit

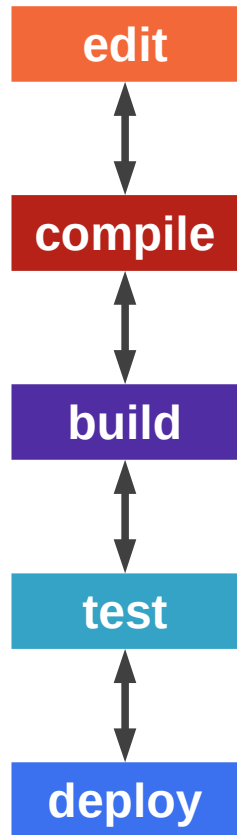
Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP B02

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

Simple Development Cycle



- **Developer**
 - **Edit:** Implements new feature
 - Iterates over the code until it looks right
 - **Compile:** Compiles the code
 - Iterates over the code until it compiles (no syntax error)
 - **Build:** Puts classes, build path together
 - Packages jar, if any, by hand
 - **Test:** Tests the program
 - Keeps going until “behavior looks right” i.e. no bugs
 - **Deploy:** Puts code into production
 - If a student, submits homework

What's Wrong with “Test-by-hand”?

- Manual testing
 - takes time that can be saved by automation
 - is not as reliable as programmed tests
 - tends to be selective, not comprehensive
- But: Human intuition can see problems that computers cannot

Tests and Testing

- Testing is a process
 - that tests some concern (the concern “under test”)
 - for correct and expected operation
 - according to a specification
 - usually as part of quality assurance
- Tests can be manual or automated
- Tests verify against a given specification
- Tests increase confidence in correct functioning
- However, tests can never proof a program correct

Types of Tests [1]

- **Components tests** (a.k.a. unit tests)
 - Focus on testing one component out of context
- **Acceptance tests** (a.k.a. functional tests)
 - Focus on testing one cross-cutting functionality
- **Integration tests** (a.k.a. system tests)
 - Focus on testing end-to-end system integrity

Test Types by other Dimensions

- Inner vs. outer perspective
 - Black-box tests
 - Tests are written to test outside observable behavior
 - White-box tests
 - Tests are written to test the innards of a component
- Static vs. dynamic perspective
 - Static-analysis based
 - Tests are written based on static code analysis
 - Run-time analysis based
 - Tests are written based on simulated behavior
- Other dimensions (see dedicated courses)

Tests and Testing Terminology

- **Test (Case)**
 - A single test for some particular aspect of the software, succeeds or fails
- **Test Suite**
 - A set of related tests that cover a particular domain of the software
- **Test Set-up**
 - The data and preparation necessary to run a test as intended
- **Test Result**
 - The result of running a test, typically succeeds/fails or error
- **Test Harness**
 - A software, like JUnit, that is used to run test suites

Example of Test Harness

- JUnit (Java Unit Testing Framework)
 - A test harness implemented as an object-oriented testing framework
 - Supports tests and test suites, set-ups, tear-downs, etc.
 - Small and simple, easy to learn
 - Well-supported by tools / integrated into IDEs like Eclipse

JUnit popularized unit testing: “Never in the field of software development have so many owed so much to so few lines of code.” [M07]

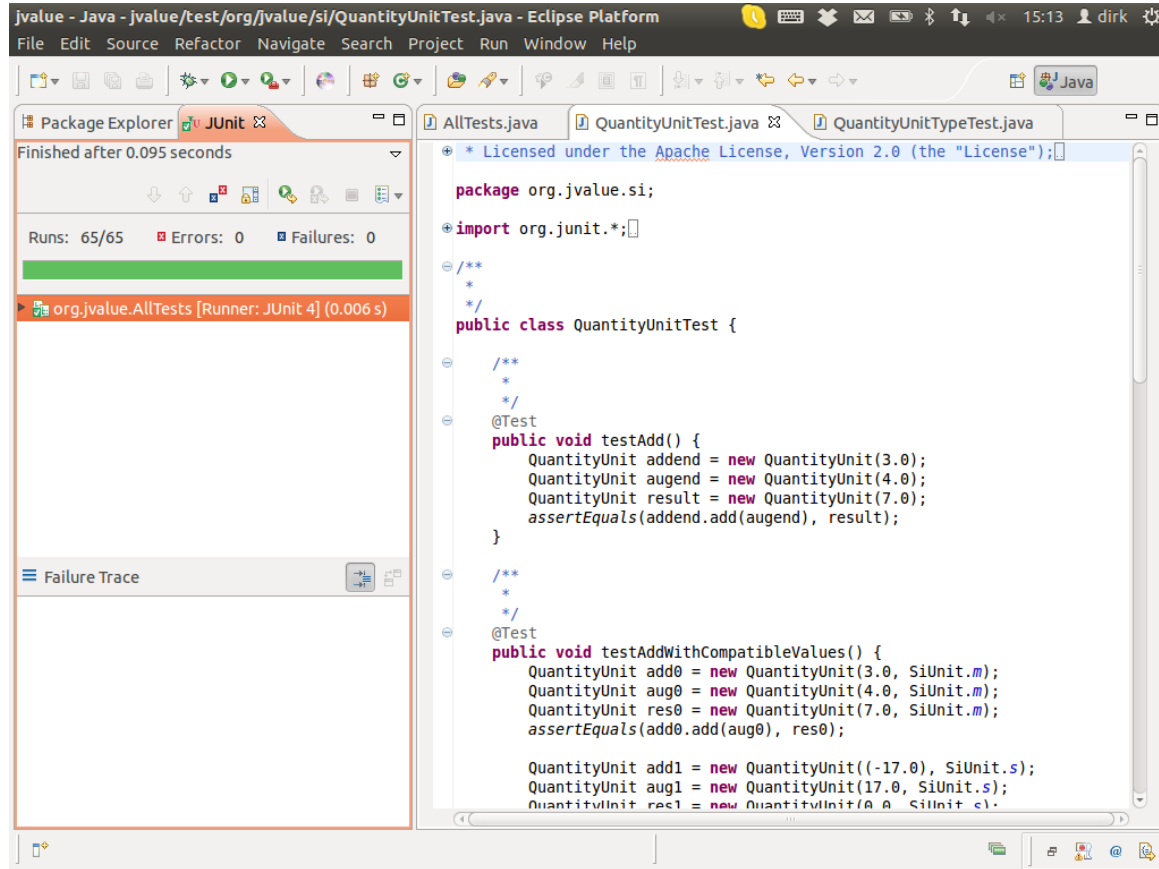
JUnit Information

- Available from <http://junit.org>
 - Comes as pre-installed plug-in with Eclipse and most other IDEs
 - See course literature for an introduction to JUnit
- Version history of JUnit
 - Prior to JUnit 4 conventions rather than annotations
 - Wahlzeit uses JUnit 4
- JUnit 5
 - Is the new major version of the testing framework
 - Is a complete rewrite of JUnit 4 to
 - Provides new foundation for developer-side testing on the JVM
 - Uses Java 8 features, for example, lambdas
 - Has a modular concept, imports only what is needed

Example for Unit Testing

- JValue Value Objects
 - A framework for value objects in Java (more on value objects later)
 - Examples are names, currencies, SI units, etc.
 - Is a small self-contained library → good for unit testing
 - Available at <https://github.com/jvalue/value-objects>

JUnit for JValue using Eclipse Plug-In



The screenshot shows the Eclipse IDE interface. The top toolbar includes icons for file operations, running, and debugging. The Package Explorer on the left shows the project structure with 'JUnit' selected. The central editor displays the source code of 'QuantityUnitTest.java'. The bottom-left pane shows the test results, indicating that all tests passed successfully.

```
package org.jvalue.si;

import org.junit.*;

/**
 *
 */
public class QuantityUnitTest {

    /**
     *
     */
    @Test
    public void testAdd() {
        QuantityUnit addend = new QuantityUnit(3.0);
        QuantityUnit augend = new QuantityUnit(4.0);
        QuantityUnit result = new QuantityUnit(7.0);
        assertEquals(addend.add(augend), result);
    }

    /**
     *
     */
    @Test
    public void testAddWithCompatibleValues() {
        QuantityUnit add0 = new QuantityUnit(3.0, SiUnit.m);
        QuantityUnit aug0 = new QuantityUnit(4.0, SiUnit.m);
        QuantityUnit res0 = new QuantityUnit(7.0, SiUnit.m);
        assertEquals(add0.add(aug0), res0);

        QuantityUnit add1 = new QuantityUnit(-17.0, SiUnit.s);
        QuantityUnit aug1 = new QuantityUnit(17.0, SiUnit.s);
        QuantityUnit res1 = new QuantityUnit(0.0, SiUnit.s);
    }
}
```

JUnit Test Results:

- Finished after 0.095 seconds
- Runs: 65/65
- Errors: 0
- Failures: 0
- org.jvalue.AllTests [Runner: JUnit 4] (0.006 s)

Test Cases in JUnit

- Tests are implemented in test classes
 - JUnit 5
 - Annotate test method with `@Test`
 - **Annotate set-up methods with `@BeforeEach` and `@BeforeAll`**
 - **Annotate tear-down methods with `@AfterEach` and `@AfterAll`**
 - End class name with `Test` (optional)
 - JUnit 4
 - Annotate test method with `@Test`
 - **Annotate set-up methods with `@Before` and `@BeforeClass`**
 - **Annotate tear-down methods with `@After` and `@AfterClass`**
 - End class name with `Test` (optional)
 - JUnit 3.8 or before
 - Start test method name with “test”
 - End test class name with “Test”

Two Example Test Cases

@Test

```
public void testEquality1() {  
    String[] helloWorld = { "hello", "world", "!" };  
    Name defaultName = new DefaultName(helloWorld);  
    Name compactName = new CompactName(helloWorld);  
    Assert.assertEquals(defaultName, compactName);  
}
```

@Test

```
public void testEquality2() {  
    String[] charMalaise = { "\\###\\", "#", "\\\"", "\\\\\\\\\\\\\\\\#", "#" };  
    Name defaultName = new DefaultName(charMalaise);  
    Name compactName = new CompactName(charMalaise);  
    Assert.assertEquals(defaultName, compactName);  
}
```

How to Write a Test (3A Pattern)

1. **Arrange**
2. **Act (execute)**
3. **Assert (check)**

Example of 3A

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

```
public class QuantityUnitTest {
```

```
    @Test
```

```
    public void testAdd() {
```

```
        QuantityUnit addend = new QuantityUnit(3.0, SIUnit.s);
```

```
        QuantityUnit augend = new QuantityUnit(4.0, SIUnit.s);
```

```
        QuantityUnit result = new QuantityUnit(7.0, SIUnit.s);
```

```
        assertEquals(addend.add(augend), result);
```

```
    }
```

```
    ...
```

1. Arrange

3. Assert

2. Act

By-Test-Case Test Set-ups in JUnit

- Annotation `@before` implements set-up method
 - Is executed before each test case (method) in a given class
 - Used to be (naming convention) `setUp`
- Annotation `@after` implements tear-down method
 - Is executed after each test case (method) in a given class
 - Used to be (naming convention) `tearDown`

Example of a Test Set-up

```
protected Name ne; // empty name
protected Name n0; // ("")
protected Name n1; // ("org", '.')
protected Name n2; // ("jvalue.org", '.')
```

@Before

```
public void setUp() {
    ne = Name.EMPTY_NAME;
    n0 = getName("");
    n1 = getName("org", '.');
    n2 = getName("jvalue.org", '.');
}
```

@Test

```
public void testEquals() {
    assertEquals(n0, getName(""));
    assertEquals(n1, getName("org"));
    assertEquals(n2, getName("jvalue#org"));
}
...
```

1. Arrange

2. Act

3. Assert

- 1. Pass / fail**
- 2. Test execution error**

Checking for Pass / Fail

- Explicit assertions / failures in code
 - `assert(...)`
 - `fail(...)`
- Annotations with expected results
 - `@Test(expected = SomeException.class)`
 - `@Test(timeout = 500)`

Another Example of 3A

1. Arrange

```
public class FlagReasonTest {
    private Map<FlagReason, TestEntry> entries;

    @Before
    public void setUp() {
        entries = new HashMap<FlagReason, TestEntry>();
        addEntry(FlagReason.MISMATCH, "mismatch", 0);
        addEntry(FlagReason.OFFENSIVE, "offensive", 1);
        ...
    }

    protected void addEntry(FlagReason reason, String str, int i) {
        entries.put(reason, new TestEntry(str, i));
    }

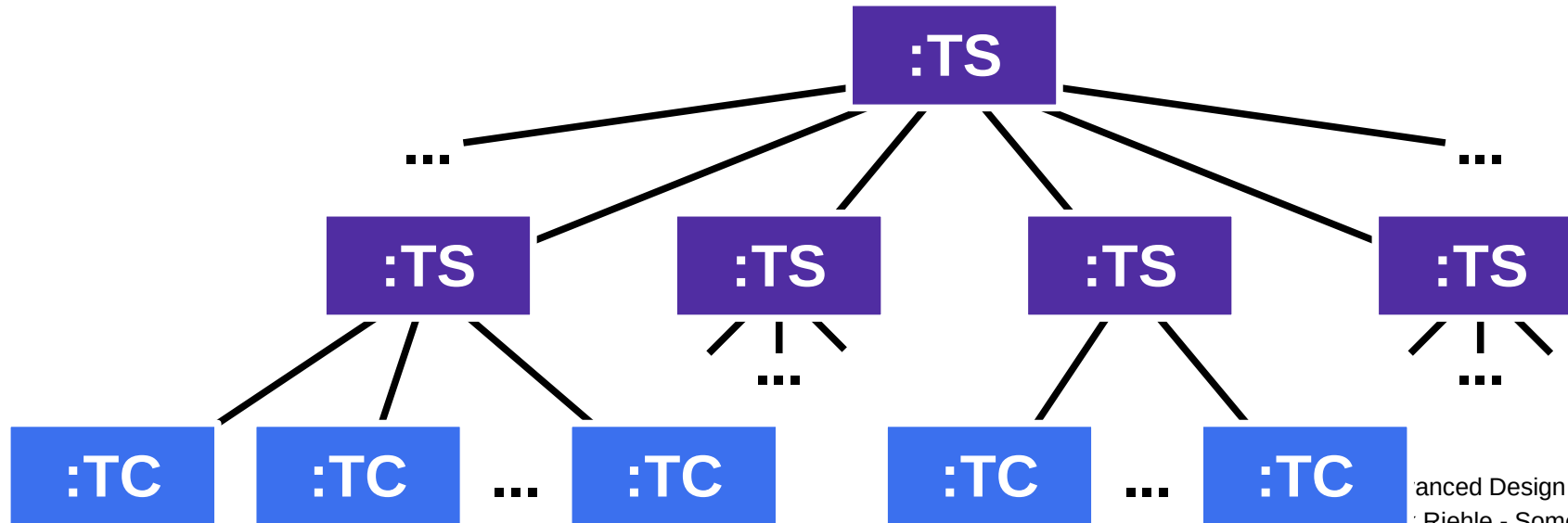
    @Test(expected = IllegalArgumentException.class)
    public void tooBigIndexShouldCauseException() {
        FlagReason.getFromInt(FlagReason.MAX_VALUE + 1);
    }
    ...
}
```

2. Act

3. Assert

Test Suite

- A test suite groups related tests into one group (object)
- This allows the group to share the same set-up, if desired
- Test suites are applied recursively to build full test hierarchy
- Tests can be run starting with any test suite in the hierarchy
- The test suites typically mirror the Java package structure



Test Suites in JUnit

- JUnit realization of Test Suites
 - JUnit 4 or later
 - Have become mostly invisible
 - Can still be used to collect test cases across classes
 - JUnit 3.8 or before
 - Create new TestSuite instance using “new TestSuite()”
 - Collect test suites using “addTestSuite(Test.class)”

Example of (By-Hand) Test Suite

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

public class QuantityUnitTest {
    @Test
    public void testAddWithCompatibleValues() {...}
    @Test(expected = IllegalArgumentException.class)
    public void testAddWithIncompatibleValues() {...}
    ...
}
```

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    org.jvalue.si.QuantityUnitTest.class,
    org.jvalue.si.QuantityUnitTypeTest.class,
    ...
})

public class AllTests { /** do nothing */ }
```

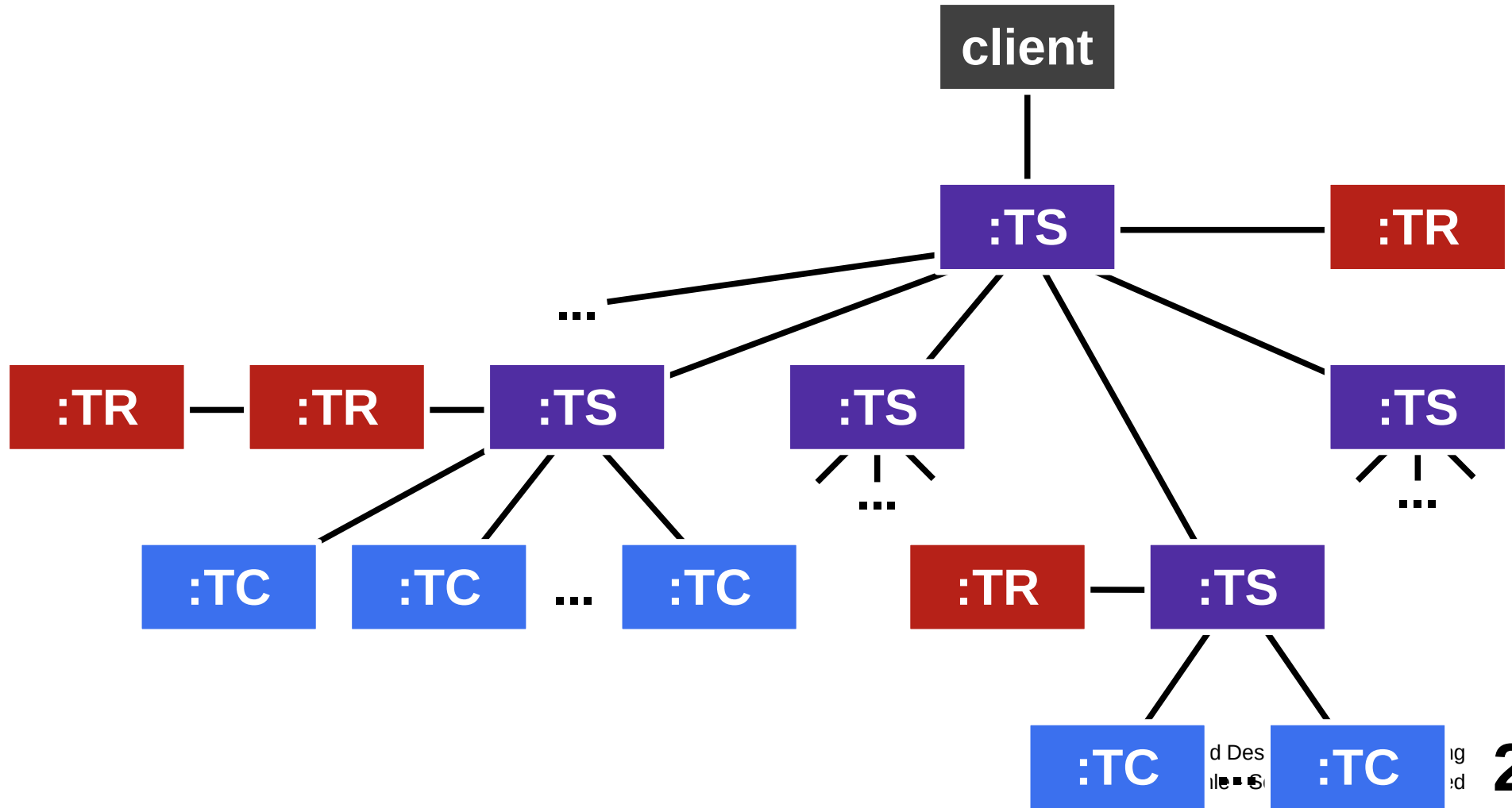
Static By-Test-Suite Test Set-ups

- Annotation `@BeforeClass` implements (static) set-up method
 - Is executed once before any test case in the class is run
 - Used to be `TestSetup` subclass (decorator)
- Annotation `@AfterClass` implements (static) tear-down method
 - Is executed once after all the test cases in the class have run
 - Used to be `TestSetup` subclass (decorator)
- Applies to (heavy-weight) required resources

Dynamic By-Test-Suite Test Set-ups

- TestRule supports dynamic set-up and tear-down
 - Test rules are attached to test suites in the test hierarchy
- Rule chain supports composition of test rules
 - Rule chain lines up test rules in sequence
 - Fluid programming style chains methods
- Supports method and class-level execution
 - Use @Rule and @ClassRule analogous to @Before and @BeforeClass

Rule-based Test Set-ups



Wahlzeit on GAE RuleChain for JUnit

```
@ClassRule
public static RuleChain ruleChain = RuleChain.
    outerRule(new LocalDatastoreServiceTestConfigProvider()).
    around(new RegisteredOfyEnvironmentProvider()).
    around(new SysConfigProvider()).
    around(new UserServiceProvider()).
    around(new UserSessionProvider());
```

- This RuleChain instance initializes most of what you'll need
- It may be an overkill for most situations but gets you going
- Example to be found in `org.wahlzeit.services.LogBuilderTest`

System-Specific Set-up and Tear-down

- More complex set-ups to be run once or only a few times
- Should be implemented in their own class, to be reused
- Applies, for example, to heavyweight database set-up
- In JUnit 4 or later to be implemented as `ExternalResource`

Location and Scope of Test Set-ups

- Test case
 - With test method (directly by calling set-up method)
 - Within test class, using @Before annotation
- Test suite
 - Within test class, using @BeforeClass annotation
 - With rules and rule chain, using separate classes
- System
 - As external resources called from outer test suite

Test Source Code Organization

- Three scopes of organizing tests
 - Within the same class
 - Within the same package
 - Within the same package hierarchy
- File locations depend on build tool
 - In Gradle, test directory captures all tests, branches of src
 - `$project/src/main/java`
 - `$project/src/test/java`
- Test package hierarchy should mirror the main hierarchy

Quiz: Simple Test Set-up

1. Your tests for the Money class all need a few example values to work with. Where should you create them?
 - In each test case method
 - In the test class' setup method
 - In a separate test setup class
2. You are programming tests for a Money.divideBy() method. Should you also test for any possible ArithmeticException?
 - Never
 - Always
 - Depends

Advanced Testing Concepts

- Handling complex system set-ups
 - Mocking, stubbing, nulling
 - Dependency injection
- Testing specific system aspects
 - Concurrency
 - Legacy code
- Test structure and practicality
 - Extent of tests run, run-time

Review / Summary of Session

- General testing
 - What are tests? What is testing?
 - What types of tests are there?
- Testing and JUnit
 - What is JUnit and how can it help you write tests?
 - What are the most basic annotations you'll need?
- Pragmatics of testing with JUnit
 - How are they aggregated into test suites?
 - How do you create test set-ups?

Thank you! Questions?

dirk.riehle@fau.de – <http://osr.cs.fau.de>

dirk@riehle.org – <http://dirkriehle.com> – [@dirkriehle](#)

Credits and License

- Original version
 - © 2012-2019 [Dirk Riehle](#), some rights reserved
 - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
 - Andreas Bauer (2018)

Unit Testing with JUnit

Prof. Dr. Dirk Riehle

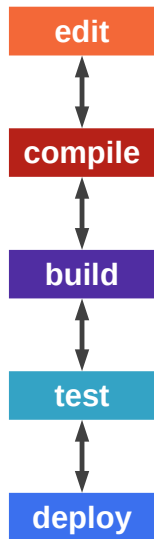
Friedrich-Alexander University Erlangen-Nürnberg

ADAP B02

Licensed under [CC BY 4.0 International](#)

It is Friedrich-Alexander University Erlangen-Nürnberg – FAU, in short.
Corporate identity wants us to say “Friedrich-Alexander University”.

Simple Development Cycle



- **Developer**
 - **Edit:** Implements new feature
 - Iterates over the code until it looks right
 - **Compile:** Compiles the code
 - Iterates over the code until it compiles (no syntax error)
 - **Build:** Puts classes, build path together
 - Packages jar, if any, by hand
 - **Test:** Tests the program
 - Keeps going until "behavior looks right" i.e. no bugs
 - **Deploy:** Puts code into production
 - If a student, submits homework

What's Wrong with "Test-by-hand"?

- Manual testing
 - takes time that can be saved by automation
 - is not as reliable as programmed tests
 - tends to be selective, not comprehensive
- But: Human intuition can see problems that computers cannot

Tests and Testing

- Testing is a process
 - that tests some concern (the concern “under test”)
 - for correct and expected operation
 - according to a specification
 - usually as part of quality assurance
- Tests can be manual or automated
- Tests verify against a given specification
- Tests increase confidence in correct functioning
- However, tests can never proof a program correct

Types of Tests [1]

- **Components tests** (a.k.a. unit tests)
 - Focus on testing one component out of context
- **Acceptance tests** (a.k.a. functional tests)
 - Focus on testing one cross-cutting functionality
- **Integration tests** (a.k.a. system tests)
 - Focus on testing end-to-end system integrity

Test Types by other Dimensions

- Inner vs. outer perspective
 - Black-box tests
 - Tests are written to test outside observable behavior
 - White-box tests
 - Tests are written to test the innards of a component
- Static vs. dynamic perspective
 - Static-analysis based
 - Tests are written based on static code analysis
 - Run-time analysis based
 - Tests are written based on simulated behavior
- Other dimensions (see dedicated courses)

Tests and Testing Terminology

- **Test (Case)**
 - A single test for some particular aspect of the software, succeeds or fails
- **Test Suite**
 - A set of related tests that cover a particular domain of the software
- **Test Set-up**
 - The data and preparation necessary to run a test as intended
- **Test Result**
 - The result of running a test, typically succeeds/fails or error
- **Test Harness**
 - A software, like JUnit, that is used to run test suites

Example of Test Harness

- JUnit (Java Unit Testing Framework)
 - A test harness implemented as an object-oriented testing framework
 - Supports tests and test suites, set-ups, tear-downs, etc.
 - Small and simple, easy to learn
 - Well-supported by tools / integrated into IDEs like Eclipse

JUnit popularized unit testing: “Never in the field of software development have so many owed so much to so few lines of code.” [M07]

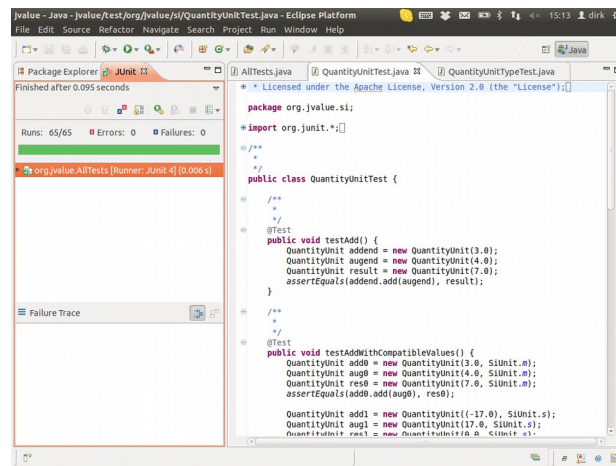
JUnit Information

- Available from <http://junit.org>
 - Comes as pre-installed plug-in with Eclipse and most other IDEs
 - See course literature for an introduction to JUnit
- Version history of JUnit
 - Prior to JUnit 4 conventions rather than annotations
 - Wahlzeit uses JUnit 4
- JUnit 5
 - Is the new major version of the testing framework
 - Is a complete rewrite of JUnit 4 to
 - Provides new foundation for developer-side testing on the JVM
 - Uses Java 8 features, for example, lambdas
 - Has a modular concept, imports only what is needed

Example for Unit Testing

- JValue Value Objects
 - A framework for value objects in Java (more on value objects later)
 - Examples are names, currencies, SI units, etc.
 - Is a small self-contained library → good for unit testing
 - Available at <https://github.com/jvalue/value-objects>

JUnit for JValue using Eclipse Plug-In



Test Cases in JUnit

- Tests are implemented in test classes
 - JUnit 5
 - Annotate test method with `@Test`
 - **Annotate set-up methods with `@BeforeEach` and `@BeforeAll`**
 - **Annotate tear-down methods with `@AfterEach` and `@AfterAll`**
 - End class name with `Test` (optional)
 - JUnit 4
 - Annotate test method with `@Test`
 - **Annotate set-up methods with `@Before` and `@BeforeClass`**
 - **Annotate tear-down methods with `@After` and `@AfterClass`**
 - End class name with `Test` (optional)
 - JUnit 3.8 or before
 - Start test method name with "test"
 - End test class name with "Test"

Two Example Test Cases

```
@Test
public void testEquality1() {
    String[] helloWorld = { "hello", "world", "!" };
    Name defaultName = new DefaultName(helloWorld);
    Name compactName = new CompactName(helloWorld);
    Assert.assertEquals(defaultName, compactName);
}

@Test
public void testEquality2() {
    String[] charMalaise = { "\\###\\", "#", "\\", "\\\\", "#" };
    Name defaultName = new DefaultName(charMalaise);
    Name compactName = new CompactName(charMalaise);
    Assert.assertEquals(defaultName, compactName);
}
```


How to Write a Test (3A Pattern)

1. **Arrange**
2. **Act (execute)**
3. **Assert (check)**

Example of 3A

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

public class QuantityUnitTest {

    @Test
    public void testAdd() {
        QuantityUnit addend = new QuantityUnit(3.0, SIUnit.s);
        QuantityUnit augend = new QuantityUnit(4.0, SIUnit.s);
        QuantityUnit result = new QuantityUnit(7.0, SIUnit.s);
        assertEquals(addend.add(augend), result);
    }
    ...
}
```

1. Arrange

3. Assert

2. Act

By-Test-Case Test Set-ups in JUnit

- Annotation `@before` implements set-up method
 - Is executed before each test case (method) in a given class
 - Used to be (naming convention) `setUp`
- Annotation `@after` implements tear-down method
 - Is executed after each test case (method) in a given class
 - Used to be (naming convention) `tearDown`

Example of a Test Set-up

```
protected Name ne; // empty name
protected Name n0; // ("")
protected Name n1; // ("org", '.')
protected Name n2; // ("jvalue.org", '.')
```

@Before

```
public void setUp() {
    ne = Name.EMPTY_NAME;
    n0 = getName("");
    n1 = getName("org", '.');
    n2 = getName("jvalue.org", '.');
}
```

1. Arrange

@Test

```
public void testEquals() {
    assertEquals(n0, getName(""));
    assertEquals(n1, getName("org"));
    assertEquals(n2, getName("jvalue.org"));
}
...
```

2. Act

3. Assert

1. **Pass / fail**
2. **Test execution error**

Checking for Pass / Fail

- Explicit assertions / failures in code
 - `assert(...)`
 - `fail(...)`
- Annotations with expected results
 - `@Test(expected = SomeException.class)`
 - `@Test(timeout = 500)`

Another Example of 3A

1. Arrange

```
public class FlagReasonTest {
    private Map<FlagReason, TestEntry> entries;

    @Before
    public void setUp() {
        entries = new HashMap<FlagReason, TestEntry>();
        addEntry(FlagReason.MISMATCH, "mismatch", 0);
        addEntry(FlagReason.OFFENSIVE, "offensive", 1);
        ...
    }

    protected void addEntry(FlagReason reason, String str, int i) {
        entries.put(reason, new TestEntry(str, i));
    }

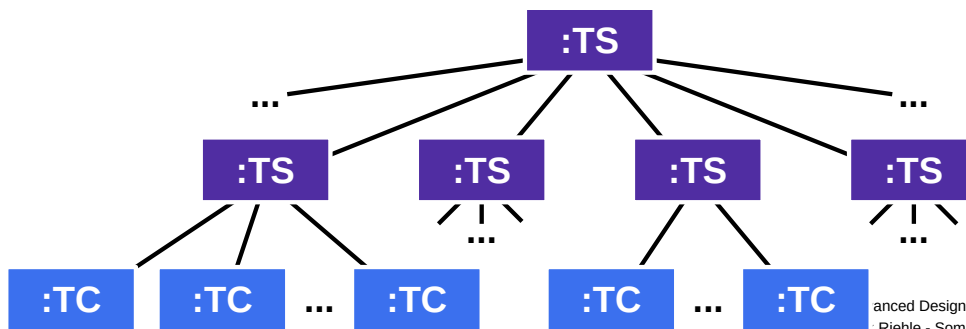
    @Test(expected = IllegalArgumentException.class)
    public void tooBigIndexShouldCauseException() {
        FlagReason.getFromInt(FlagReason.MAX_VALUE + 1);
    }
    ...
}
```

2. Act

3. Assert

Test Suite

- A test suite groups related tests into one group (object)
- This allows the group to share the same set-up, if desired
- Test suites are applied recursively to build full test hierarchy
- Tests can be run starting with any test suite in the hierarchy
- The test suites typically mirror the Java package structure



Test Suites in JUnit

- JUnit realization of Test Suites
 - JUnit 4 or later
 - Have become mostly invisible
 - Can still be used to collect test cases across classes
 - JUnit 3.8 or before
 - Create new TestSuite instance using "new TestSuite()"
 - Collect test suites using "addTestSuite(Test.class)"

Example of (By-Hand) Test Suite

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

public class QuantityUnitTest {
    @Test
    public void testAddWithCompatibleValues() {...}
    @Test(expected = IllegalArgumentException.class)
    public void testAddWithIncompatibleValues() {...}
    ...
}
```

```
import org.junit.runner.*;
import org.junit.runners.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    org.jvalue.si.QuantityUnitTest.class,
    org.jvalue.si.QuantityUnitTypeTest.class,
    ...
})

public class AllTests { /** do nothing */ }
```

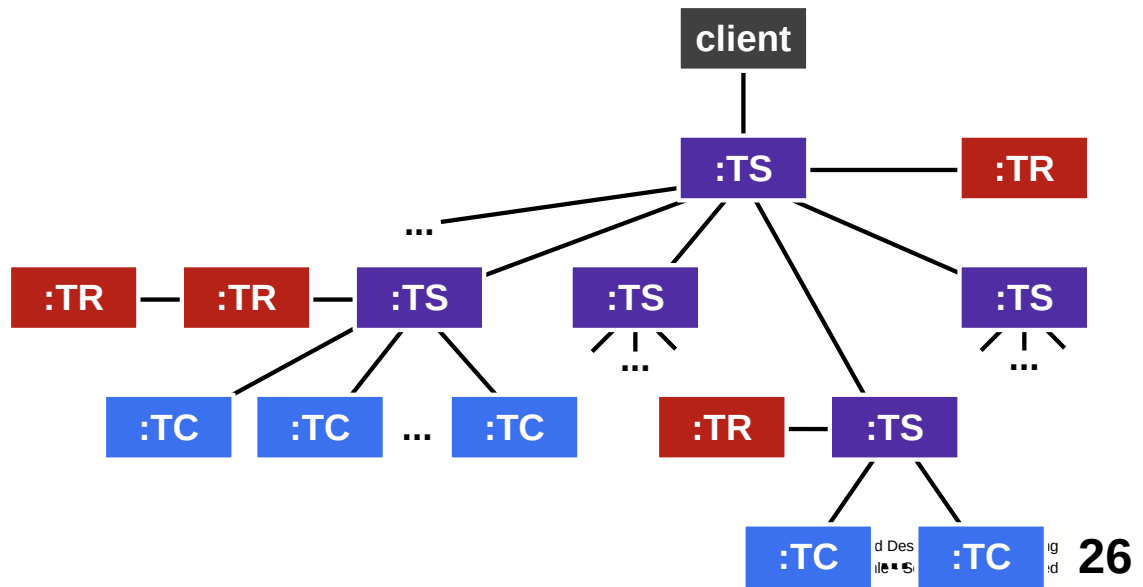
Static By-Test-Suite Test Set-ups

- Annotation `@BeforeClass` implements (static) set-up method
 - Is executed once before any test case in the class is run
 - Used to be `TestSetup` subclass (decorator)
- Annotation `@AfterClass` implements (static) tear-down method
 - Is executed once after all the test cases in the class have run
 - Used to be `TestSetup` subclass (decorator)
- Applies to (heavy-weight) required resources

Dynamic By-Test-Suite Test Set-ups

- TestRule supports dynamic set-up and tear-down
 - Test rules are attached to test suites in the test hierarchy
- Rule chain supports composition of test rules
 - Rule chain lines up test rules in sequence
 - Fluid programming style chains methods
- Supports method and class-level execution
 - Use @Rule and @ClassRule analogous to @Before and @BeforeClass

Rule-based Test Set-ups



Wahlzeit on GAE RuleChain for JUnit

```
@ClassRule
public static RuleChain ruleChain = RuleChain.
    outerRule(new LocalDatastoreServiceTestConfigProvider()).
    around(new RegisteredOfyEnvironmentProvider()).
    around(new SysConfigProvider()).
    around(new UserServiceProvider()).
    around(new UserSessionProvider());
```

- This RuleChain instance initializes most of what you'll need
- It may be an overkill for most situations but gets you going
- Example to be found in `org.wahlzeit.services.LogBuilderTest`

System-Specific Set-up and Tear-down

- More complex set-ups to be run once or only a few times
- Should be implemented in their own class, to be reused
- Applies, for example, to heavyweight database set-up
- In JUnit 4 or later to be implemented as `ExternalResource`

Location and Scope of Test Set-ups

- Test case
 - With test method (directly by calling set-up method)
 - Within test class, using `@Before` annotation
- Test suite
 - Within test class, using `@BeforeClass` annotation
 - With rules and rule chain, using separate classes
- System
 - As external resources called from outer test suite

Test Source Code Organization

- Three scopes of organizing tests
 - Within the same class
 - Within the same package
 - Within the same package hierarchy
- File locations depend on build tool
 - In Gradle, test directory captures all tests, branches of src
 - \$project/src/main/java
 - \$project/src/test/java
- Test package hierarchy should mirror the main hierarchy

Quiz: Simple Test Set-up

1. Your tests for the Money class all need a few example values to work with. Where should you create them?
 - In each test case method
 - In the test class' setup method
 - In a separate test setup class
2. You are programming tests for a Money.divideBy() method. Should you also test for any possible ArithmeticException?
 - Never
 - Always
 - Depends

Advanced Testing Concepts

- Handling complex system set-ups
 - Mocking, stubbing, nulling
 - Dependency injection
- Testing specific system aspects
 - Concurrency
 - Legacy code
- Test structure and practicality
 - Extent of tests run, run-time

Review / Summary of Session

- General testing
 - What are tests? What is testing?
 - What types of tests are there?
- Testing and JUnit
 - What is JUnit and how can it help you write tests?
 - What are the most basic annotations you'll need?
- Pragmatics of testing with JUnit
 - How are they aggregated into test suites?
 - How do you create test set-ups?

Thank you! Questions?

dirk.riehle@fau.de – <http://osr.cs.fau.de>

dirk@riehle.org – <http://dirkriehle.com> – [@dirkriehle](#)

DR

Credits and License

- Original version
 - © 2012-2019 [Dirk Riehle](#), some rights reserved
 - Licensed under [Creative Commons Attribution 4.0 International License](#)
- Contributions
 - Andreas Bauer (2018)