Unit Testing with JUnit

Professorship of Open Source Software

Friedrich-Alexander University Erlangen-Nürnberg

ADAP B04

- Recap
- Unit = Component (in our context)
- Often classes are seen as units

Requirement Specification						Acceptance Testing
	Functional Specification				System Testing	
		Technical Specification		Integration Testing		
			Program Specification	Component Testing		
			Coding			
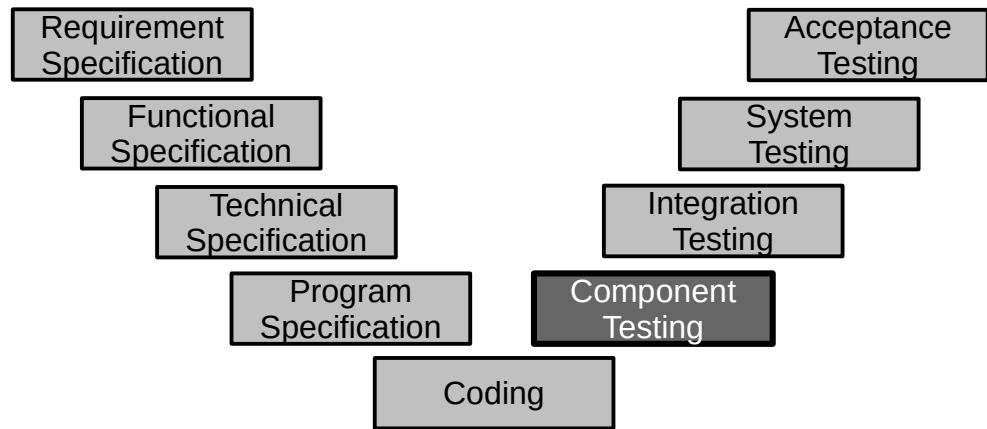
Example of Test Harness

- 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]

Advanced Design and Programming

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
  - 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

JUnit Example: Component Under Test

- Scheduler for tasks that are triggered the first time on a certain point in time and then in a defined fix interval

```java
1 package osrg.adap.testing;
2
3 import java.util.Date;
4
5 public class Scheduler {
6
7     /***
8      * ...... a lof of other Scheduler attributes and methods
9      ***/
10
11     public Date calculateExecutionDate(Date givenExecutionDate, long interval) {
12         Date now = new Date();
13
14         if (givenExecutionDate.after(now)) {
15             return givenExecutionDate;
16         }
17
18         long offset = (now.getTime() - givenExecutionDate.getTime()) % interval;
19         return new Date(now.getTime() + interval - offset);
20     }
21 }
```

JUnit Example: Simple Unit Tests (1/2)

```java
1 package osrg.adap.testing;
2
3 import org.junit.Test;
4 import java.util.Date;
5 import static org.junit.Assert.*;
6
7 public class SchedulerTest {
8
9     @Test
10     public void testCalculateExecutionDateFromFutureDate() {
11         Date futureDate = new Date(new Date().getTime() + 60000);
12
13         Scheduler scheduler = new Scheduler();
14         Date calculatedDate = scheduler.calculateExecutionDate(futureDate, 1000L);
15
16         assertEquals(futureDate, calculatedDate);
17     }
18
19     @Test
20     public void testCalculateExecutionDateFromPastDate() {
21         Date pastDate = new Date(new Date().getTime() - 1000);
22
23         Scheduler scheduler = new Scheduler();
24         Date calculatedDate = scheduler.calculateExecutionDate(pastDate, 60000L);
25
26         assertEquals(pastDate.getTime() + 60000, calculatedDate.getTime());
27     }
28 }
```

- Annotate test method with **@Test**

- Conventions:
  - Containing class name ends with 'Test'
  - Test methods start with 'test'
  - File locations depend on build tool, e.g. Gradle
    - Sources: $project/src/main/java
    - Tests: $project/src/test/java
  - Test package hierarchy should mirror the main hierarchy

JUnit Example: Simple Unit Tests (2/2)

```java
1 package osrg.adap.testing;
2
3 import org.junit.Test;
4 import java.util.Date;
5 import static org.junit.Assert.*;
6
7 public class SchedulerTest {
8
9     @Test
10    public void testCalculateExecutionDateFromFutureDate() {
11        Date futureDate = new Date(new Date().getTime() + 60000);
12
13        Scheduler scheduler = new Scheduler();
14        Date calculatedDate = scheduler.calculateExecutionDate(futureDate, 1000L);
15
16        assertEquals(futureDate, calculatedDate);
17    }
18
19    @Test
20    public void testCalculateExecutionDateFromPastDate() {
21        Date pastDate = new Date(new Date().getTime() - 1000);
22
23        Scheduler scheduler = new Scheduler();
24        Date calculatedDate = scheduler.calculateExecutionDate(pastDate, 60000L);
25
26        assertEquals(pastDate.getTime() + 60000, calculatedDate.getTime());
27    }
28 }
```

- **Assertions** for checking the results
- Explicit assertions / failures in code
  - assert(...)
  - fail(...)
- Annotations with expected results
  - @Test(expected = SomeException.class)
  - @Test(timeout = 500)

Advanced Design and Programming
© 2020 FAU - All Rights Reserved

7

```
1    @Test
2    public void testCalculateExecutionDateFromFutureDate() {
3        Date futureDate = new Date(new Date().getTime() + 60000);
4
5        Scheduler scheduler = new Scheduler();
6        Date calculatedDate = scheduler
7            .calculateExecutionDate(futureDate, 1000L);
8
9        assertEquals(futureDate, calculatedDate);
10   }
```

1. **Arrange**

2. **Act (execute)**

3. **Assert (check)**

1) **Pass**

2) **Fail**
   a) Program is defect

   b) Test is defect

3) Test execution error

```
1 package osrg.adap.testing;
2
3 import org.junit.Test;
4 import static org.junit.Assert.fail;
5
6 public class TestResultTypes {
7
8     @Test
9     public void succeedingTest() {
10        return;
11    }
12
13    @Test
14    public void failingTest() {
15        fail();
16    }
17
18    @Test
19    public void executionError() {
20        throw new IllegalStateException();
21    }
22 }
```

Test Results   3 ms
  osrg.adap.testing.TestResultTypes   3 ms
    executionError   2 ms
    succeedingTest   0 ms
    failingTest   1 ms

JUnit Static Test Setup & Teardown

- Setting up / tearing down test environment **for every** test in a class
  - @Before
  - @After
- Setting up / tearing down test environment **once for all** tests in a class
  - @BeforeClass
  - @AfterClass

```java
 1 package osrg.adap.testing;
 2
 3 import org.junit.After;
 4 import org.junit.Before;
 5 import org.junit.Test;
 6
 7 public class SimpleSetupTearDown {
 8
 9     private Scheduler schedulerUnderTest;
10
11     @Before
12     public void setupScheduler() {
13         schedulerUnderTest = new Scheduler();
14         schedulerUnderTest.setMode("BEST_EFFORT");
15         schedulerUnderTest.start();
16     }
17
18     @After
19     public void teardownScheduler() {
20         schedulerUnderTest.stop();
21     }
22
23     @Test
24     public void testSchedulerDryrun() {
25         // just a dry-run
26     }
27 }
28
```

10

JUnit Dynamic Test Setup & Teardown

- Reusable Setup and Teardown

- TestRule

  - e.g. TmpDir Rule
    - Temporary directory that is cleared after each test case
  - Rule chain supports composition of test rules
    - Rule chain lines up test rules in sequence
    - Fluid programming style chains methods
  - Use @Rule and @ClassRule analogous to @Before and @BeforeClass

- ExternalResource

  - More complex set-ups to be run once or only a few times

 - Applies, for example, to heavyweight database set-up

```java
 1 package osrg.adap.testing;
 2
 3 import org.junit.Rule;
 4 import org.junit.Test;
 5 import org.junit.rules.TemporaryFolder;
 6 import java.io.File;
 7 import java.io.IOException;
 8 import static org.junit.Assert.assertTrue;
 9
10 public class TmpDirTest {
11
12     @Rule
13     public TemporaryFolder folder = new TemporaryFolder();
14
15     @Test
16     public void testTmpFolder() throws IOException {
17         File file = folder.newFile("testfile.txt");
18         assertTrue(file.exists());
19     }
20 }
```

Changes in JUnit Versions

- Tests are implemented in test classes
  - JUnit 3.8 or before
    - Start test method name with "test"
    - End test class name with "Test"
  - 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 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)

Test Drivers and Test Doubles

- Components are part of a dependency graph
- We need to isolate components in order to test them as a single unit
- **Using components** as inspiration for test drivers (calling the component under test)
- **Used components** need to be replaced by test doubles
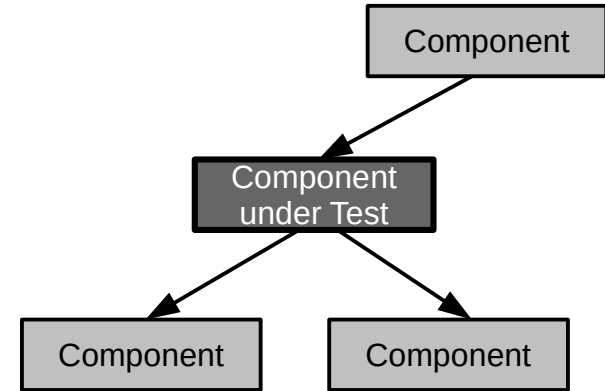
Isolating Units with Test Doubles

- Test Doubles
  - Object or component that we install in place of the real component for a test
- Dummy Object
  - Placeholder object that is passed to the SUT as an argument (or an attribute of an argument) but is never actually used
- Test Stub
  - Replaces component that SUT depends on, configure indirect inputs to the SUT
- Mock Object
  - Test Stub + ability to verify inputs of the SUT by behaviour expectation
- Test Spy
  - Test Stub + ability to verify inputs of the SUT by recording calls to the spy that can be verified
- Fake Object
  - Replaces component with an alternative implementation of the same functionality

Mockito

- Available from https://site.mockito.org/
- Serves a variety of testing double functionality
  - Stubbing
  - Mocking
  - Spying
  - Etc.
- Easy syntax to create test doubles
- Easy syntax to verify test double behaviour
- Interacts very well with JUnit

Mockito Example: Component under Test

```java
1 package osrg.adap.testing;
2
3 public class TodoService {
4
5     private TodoRepository todoRepository;
6     private SlackNotificator notificator;
7
8     public TodoService(TodoRepository todoRepository, SlackNotificator notificator) {
9         this.todoRepository = todoRepository;
10        this.notificator = notificator;
11    }
12
13    public void setDone(long id) {
14        TodoItem todo = this.todoRepository.get(id);
15        todo.setDone(true);
16
17        this.todoRepository.save(todo);
18        this.notificator.notify("Todo " + todo.getId() + " has been settled!");
19    }
20 }
```

Inversion of Control

Mockito Example

```java
package osrg.adap.testing;

import org.junit.Test;
import static org.mockito.Mockito.*;

public class TodoServiceTest {

    @Test
    public void testSetDone() {
        // ARRANGE
        TodoRepository repository = mock(TodoRepository.class);
        SlackNotificator notificator = mock(SlackNotificator.class);

        long todoId = 123L;
        TodoService todoService = new TodoService(repository, notificator);
        when(repository.get(todoId)).thenReturn(new TodoItem(todoId, "test todo"));

        // ACT
        todoService.setDone(todoId);

        // ASSERT
        verify(repository, times(1)).get(todoId);
        verify(repository, times(1)).save(new TodoItem(todoId, "test todo", true));
        verify(notificator, times(1)).notify(anyString());
    }
}
```

- Mock creation with static *mock* method or with *@Mock* annotation
- Behaviour specification with static *when* method
- Behaviour verification with static *verify method*

Inversion of Control

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
  - © 2020 Friedrich-Alexander University Erlangen-Nürnberg and Dirk Riehle, all rights reserved

- Contributions
  - Georg Schwarz (2019)