# Test Fixture Setting

## **Typen von Tests und ihre fixtures (Beispiele)**

|  |  |
| --- | --- |
| unit test | Erzeugen neuer Objekte (SUT und test doubles),  Vorbereiten von Eingabedaten |
| integration test | Zurücksetzen der Datenbank zu einem initial state  Kopieren von Dateien, die für den Test gebraucht werden |
| end-to-end test | Installation einer virtual machine die ein runtime environment für die Installation zur Verfügung stellt (oder Zurücksetzen zu einem initial state) für den Web Server und Datenbank welche für die application benutzt wird. |

## **JUnit Execution Model (vorher und nachher)**

### **Übung**

Gegeben:

**ch06.x01JUnitExecutionModel.Address**

**ch06.x01JUnitExecutionModel.Client**

Implementieren Sie die 3 Testfälle

afterCreationShouldHaveNoAddress

shouldAllowToAddAddress

shouldAllowToAddManyAddresses

Lösung: **ch06.x01JUnitExecutionModel\_vorherLösung.JUnitExecutionModel\_vorher**

### **Übung**

1. Bringen Sie die Klasse Tester aus <http://martinfowler.com/bliki/JunitNewInstance.html> zum Laufen.
2. Wandeln Sie sie um im Stil von JUnit 4.x
   1. Dem Test einen Namen zu geben ist hier nicht wichtig. Falls Interesse besteht, so ist es hier beschrieben: <https://blogs.oracle.com/tor/entry/how_to_determine_the_junit>
3. Sorgen Sie durch Umbenennen der Testmethoden dafür, dass die Testmethoden in den beiden möglichen verschiedenen Reihenfolgen ablaufen.
   1. z.B. test1, test2
   2. mit der Annotation @org.junit.FixMethodOrder(MethodSorters.NAME\_ASCENDING);
4. Ziehen Sie Rückschlüsse aus der Tatsache, dass der Test gelingt, egal in welcher Reihenfolge die Testmethoden gerufen werden.
5. Finden Sie Evidenz für Ihre Beobachtung in der Javadoc zu Runner: <http://junit.sourceforge.net/javadoc/org/junit/runner/Runner.html>

Lösung

Zu a) **ch06.x02TesterMartinFowlerBlogLösung.TesterJUnit3**

Zu b) **ch06.x02TesterMartinFowlerBlogLösung.TesterJUnit4**

Zu c) Ausprobieren (und Änderungen ggf. wieder löschen) an obiger Klasse aus Zu b) **ch06.x02TesterMartinFowlerBlogLösung.TesterJUnit4**

Zu d)

Zu e)

### **Übung**

1. Wandeln Sie **ch06.x01JUnitExecutionModel\_vorherLösung.JUnitExecutionModel\_vorher** um, sodass Sie die Erkenntnis der letzten Übung nutzen.

Lösung: **ch06.x03JUnitExecutionModelLösung.JUnitExecutionModelDefaultTest**

### **Übung**

Wandeln Sie **ch06.x01JUnitExecutionModelLösung.JUnitExecutionModelDefaultTest** um, sodass Sie das JUnit-feature aus <http://junit.org/junit4/faq.html#atests_2> nutzen.

Lösung: **ch06.x03JUnitExecutionModelLösung.JUnitExecutionModelBeforeTest**

### **Übung (fortgeschritten;optional)**

Seit JUnit 4.9 gibt es Rules sowie die Möglichkeit, eigene Rules zu (über)schreiben:

|  |
| --- |
| * Rules allow very flexible addition or redefinition of the behavior of each test method in a test class. Testers can **reuse or extend** one of the provided Rules below, or write their own.   <https://github.com/junit-team/junit4/wiki/Rules> |

Eine dieser Rules erlaubt das Initialisieren und nachträgliche Aufräumen von externen Ressourcen und hat überschreibbare Methoden analog zu @Before und @After: org.junit.rules.ExternalResource

Beispiel: <https://github.com/dlee0113/mastering_unit_testing_using_mockito_and_junit-/blob/master/Chapter_1/JUnitTests/test/com/packtpub/junit/recap/rule/ExternalResourceTest.java>

Wandeln Sie **ch06.x03JUnitExecutionModelLösung.JUnitExecutionModelBeforeTest**

um, sodass Sie org.junit.rules.ExternalResource nutzt.

Lösung: **ch06.x03JUnitExecutionModelLösung.JUnitExecutionModelRuleTest**

Hinweis: Natürlich ist diese Lösung komplizierter als eine einfache @Before-Methode. Der Vorteil entsteht erst, wenn man von dem **„**reuse or extend“ aus der Javadoc Gebrauch macht (vgl. auch <http://stackoverflow.com/questions/12237723/externalresource-vs-before-after> :“ You can reuse an ExternalResource implementation in multiple test classes more easily than sharing before/after methods.“

Ein Beispiel hierfür ist z.B. die Verwendung mit einem Framework zur Vereinfachung von JUnit-Test, die auf eine Datenbank zugreifen: <https://theholyjava.wordpress.com/2011/09/04/dry-use-junit-rule-instead-of-repeating-setupbefore-in-each-test/> :“ I was for a long time unhappy that [**DbUnit Express**](https://sourceforge.net/apps/mediawiki/jeeutils/index.php?title=DbUnit_Express) users have to create a @Before method in each test just to get the test database initialized. Fortunately since version 1.3.0 they don’t need to do it anymore thanks to [**JUnit Rules**](http://kentbeck.github.com/junit/javadoc/4.9/org/junit/rules/package-summary.html) (if you are not familiar with them, they are an alternative to @Before/@After and @BeforeClass/@AfterClass, read this [**rules introduction**](http://blog.schauderhaft.de/2009/10/04/junit-rules/)).”

Hinweis 2: Diese Übung soll nicht mehr als einen ersten Berührungspunkt mit JUnit-Rules sein. Das Thema Rules verdient ein eigenes Kapitel.

## **Übung: Wandle einen parametrisierten Test mit JUnits eigenem** org.junit.runners.Parameterized um, sodass junitparams.JUnitParamsRunner (also die JUnitParams-Bibliothek) benutzt wird.

JUnit 4 hat eine weitere Art, test fixtures anzulegen: parametrisiert (wie oben bereits einmal gezeigt)

Anlass für diese Übung ist eine Frage auf stackoverflow.com (<http://stackoverflow.com/questions/14082004/create-multiple-parameter-sets-in-one-parameterized-class-junit> ):“ Currently I have to create a parameterized test class for every method that I want to test with several different inputs. Is there a way to add this together in one file?

Right now there's CalculatorTestAdd.java which has a set of parameters that are used to check if the Add() function works properly. Is there a possbility for me to 'connect' this set to the Add()function and create an additional set meant for the Subtract() method and add this method in the same test class, resulting in one file called CalculatorTest.java?”

Die letzte Antwort auf dieser Seite vergleicht 3 verschiedene Lösungen. Die erste dieser Lösungen benutzt die JUnit 4.x-eigene Möglichkeit dafür (mit dem org.junit.runners.Parameterized-Runner). Diese (um Fehler bereinigte) Lösung ist die oben genannte **ch06.x04JUnitParameterizedVsJUnitParams.CalculatorJUnitParameterizedTest**.

Analysieren Sie diese Klasse einmal und debuggen Sie sie.

Wie man darin sehen kann, müssen die nach und nach an den Konstruktor übergebenen Werte in Fields der Testklasse gespeichert werden, um sie dann in den @Test-Methoden benutzen zu können. Dies macht den dort gezeigten Workaround nötig, im Konstruktor für jede @Test-Methode den dort zu erwartenden Wert mit zu übergeben. Jede weitere @Test-Methode würde einen weiteren Konstruktor-Parameter nötig machen und eine Erweiterung der Daten, die von der @Parameters-Methode zurückgegeben werden. Dies sind unschöne und schwer wartbare Abhängigkeiten.

**Lösen Sie das Problem eleganter mit der JUnitParams-Bibliothek! (vgl ch4 oben)**

Gegeben: SUT: **ch06.x03JUnitParameterized.Calculator**

Test: **ch06.x03JUnitParameterizedVsJUnitParams.CalculatorJUnitParameterizedTest**

Lösung: **ch06.x04JUnitParameterizedVsJUnitParamsLösung.CalculatorJUnitParamsTest**