

# Formulierung und Durchführung von Modultests mit KJUnit

Anwendungsdokumentation



## **Inhalt**

<b>1 Überblick</b>	<b>3</b>
<b>2 Vorstellung des Beispiels</b>	<b>6</b>
<b>3 Vorbereitung von Modultests für die Verwendung in KJUnit</b>	<b>7</b>
3.1 Entwicklung eines Moduls zu dem Beispiel	7
3.2 Vorbereitung des Moduls für die Verwendung in KJUnit	9
3.2.1 Parametrisierung der Testfälle	9
3.2.2 Zusammenfassung aller Testressourcen in einem Java-Archiv	11
<b>4 Modifizierung und Ausführung von Modultests mit KJUnit - Wissensträger</b>	<b>14</b>
4.1 Konfiguration der Anwendung	14
4.2 Start der Anwendung und Laden von Testressourcen	15
4.3 Modifikation und Ausführung von Testfällen	16
4.3.1 Reiter Testergebnisse und Neue Konfiguration	16
4.3.2 Reiter Konfigurationsgenerator	19
<b>5 KJUnit - Entwickler</b>	<b>22</b>
5.1 Konfiguration der Anwendung	22
5.2 Start und Aufbau der Anwendung	24
5.3 Erstellen und Laden der Testressourcen	25
5.4 Erstellen von KJUnit Testklassen aus JUnit Testklassen	31
5.5 Erstellen und Entfernen von Testkonfigurationen	34
5.6 Start des Loggers und des Runners	35
5.7 Erstellen von Daten zum Test Coverage	40
<b>Anhang</b>	
<b>A Erlaubte Eingaben für Parameterwerte</b>	<b>43</b>
<b>B Konventionen und Bemerkungen</b>	<b>48</b>

# 1 Überblick

Der Erfolg eines Softwareprojektes ist maßgeblich abhängig von der Einbindung des Auftraggebers beziehungsweise der Auftraggeberin, im folgenden Text beide mit Auftraggeber bezeichnet, in den Entwicklungsprozess. Nur unter seiner Mithilfe können sich ändernde Anforderungen zeitnah im Entwicklungsprozess abgebildet und bisherige Teilentwicklungen überprüft werden. In der Praxis ist es bisher teilweise noch verbreitet, dass der Auftraggeber erst bei Durchführung sogenannter Akzeptanztests die Funktionsfähigkeit eines (Teil-)Programms überprüft. Wünschenswert aber ist eine möglichst frühe Einbindung des Auftraggebers in den Qualitätssicherungsprozess, beispielsweise bei der Durchführung von Modultests. Falls nicht der Auftraggeber selbst für Modultests schon zur Verfügung steht, dann sollten es ein anderer Wissensträger oder eine andere Wissensträgerin wie beispielsweise Softwaretester und Softwaretesterinnen, im folgenden Text mit Wissensträger bezeichnet, sein. Modultests ermöglichen die Identifizierung von Fehlern einzelner Programmteile (z. B. einer Klasse oder Methode) zum Zeitpunkt ihrer Entwicklung. Weit verbreitet für die Realisierung von Modultests sind Tools, mit denen Testfälle in der Programmiersprache des zu prüfenden Moduls codiert werden. Da der Wissensträger in der Regel nicht über Programmierkenntnisse verfügt, fallen Modultests bisher ausschließlich in den Aufgabenbereich der Entwicklungsabteilung. Wenn man sich aber vor Augen führt, dass der Wissensträger die Funktionsweise eines Moduls aufgrund seiner umfassenderen geschäftsspezifischen Kenntnisse besser einschätzen kann als ein Entwickler oder eine Entwicklerin, im folgenden Text beide mit Entwickler bezeichnet, verspielt man eine wesentliche Chance zur Früherkennung von Programmfehlern und den damit verbundenen erfolgreichen Projektabschluss. Um einem Wissensträger ohne Programmierkenntnisse die Möglichkeit von Modultests einzuräumen, wurde die Java-Anwendung KJUnit (*Knowledge Based Unit Testing Application*) entwickelt. Mit dieser kann auf die von einem Entwickler definierten und speziell parametrisierten JUnit Testfälle zugegriffen werden.

Eine Version der in Java geschriebenen Anwendung KJUnit wurde unter anderem auf der Software Engineering 2017, auf dem Software QS-Tag 2018 und auf dem Software QS-Tag 2022 vorgestellt: Oesing, Ursula; Georgiev, Alexander; Langenbrink, Jahn; Jonker, Stefan: *Agiles Testen: Auch Anwender können Unit Tests* in J. Jürjens, K. Schneider (Hrsg.): Software Engineering 2017, LNI – Proceedings Series of the GI, Köllen Druck + Verlag GmbH Bonn, ISBN 978-3-88579-661-9.

Die vorliegende Version der Anwendung KJUnit unterstützt die Schritte 0 bis 5 der Abbildung 1, siehe unten. Sie wurde von Philipp Sprengholz, Alexander Georgiev, Patrick Pete, Yannis Herbig, Gokhan Arslan, Iris Grcic, Siraj Taeb und Ursula Oesing entwickelt. Die vorliegende Anwendungsdokumentation wurde von Philipp Sprengholz und Ursula Oesing entwickelt und von Patrick Pete ergänzt. Die Realisierung von Anwendungsteilen, welche den Schritt 6 unterstützen, sind geplant.

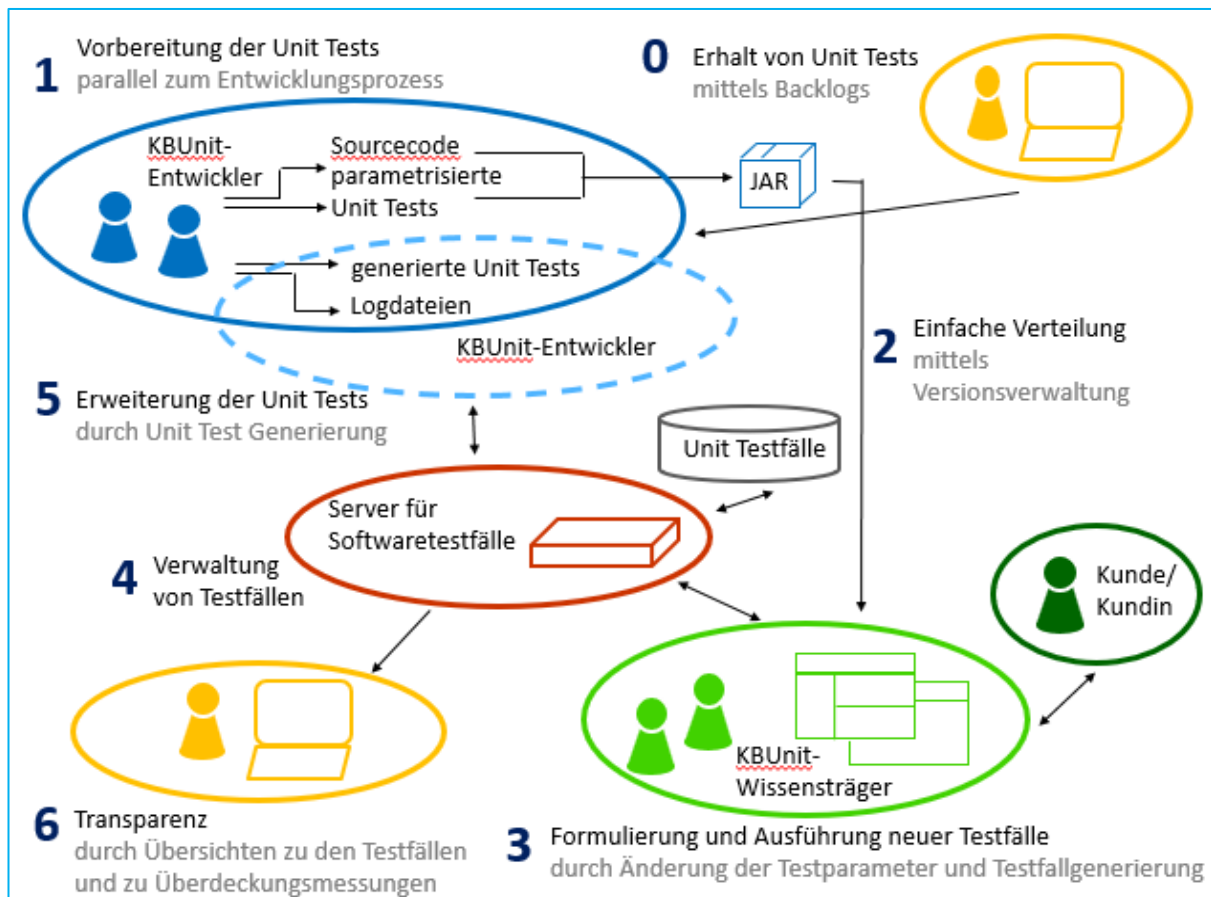


Abbildung 1: Prozess zur Durchführung von Unit Tests

### Schritt 0 – Erhalt von Unit Tests:

Es besteht die Möglichkeit, sich aus den Backlogs agiler Projektmanagementtools entweder JUnit Tests oder parametrisierte JUnit Tests und jeweils zu ergänzenden Sourcecode erzeugen zu lassen.

### Schritt 1 - Vorbereitung der Unit Tests:

Die Entwickler implementieren eine Funktionalität und zu dieser einen JUnit Test, welcher einen Testfall überprüft. Soll dieser von Wissensträgern um Testfälle ergänzt werden, transferieren die Entwickler den JUnit Test in einen parametrisierten JUnit Test. Er enthält diejenigen Parameter, die von Wissensträgern variiert werden sollen.

### Schritt 2 - Einfache Verteilung der Unit Tests:

Der Sourcecode und die parametrisierten JUnit Tests werden den Wissensträgern zugänglich gemacht.

### **Schritt 3 - Formulierung und Ausführung neuer Tests:**

Die Wissensträger greifen auf die von den Entwicklern entworfenen parametrisierten JUnit Tests zu und erstellen in Absprache mit dem Kunden weitere Testfälle, die aus ihrer Sicht geprüft werden müssen, indem sie die Werte der Parameter variieren und Testfälle generieren lassen. Sie können sämtliche Testfälle auch abspielen.

### **Schritt 4 - Verwaltung von Testfällen:**

Die Wissensträger speichern die neuen Testfälle inklusive deren Testergebnis in einer Datenbank. Sie können diese auch verwalten.

### **Schritt 5 - Erweiterung der Unit Tests:**

Die Entwickler haben die Möglichkeit, auf die neuen Testfälle zuzugreifen und diese zu verwalten. Sie können diese in ihrer Entwicklungsumgebung abspielen und entsprechende Logdateien erstellen lassen. Sie können ebenfalls sich zu den neuen Testfällen Unit Tests generieren lassen.

### **Schritt 6 - Transparenz:**

Es besteht zusätzlich die Möglichkeit, sich den Stand zu den Softwaretests anzeigen zu lassen.

## 2 Vorstellung des Beispiels

Bevor Modultests in KJUnit modifiziert und ausgeführt werden können, bedarf es einiger Vorbereitungen, die anhand eines einfachen Beispiels erläutert werden. Die Ausführungen dieses Kapitels sind zum großen Teil übernommen aus: Sprengholz, Philipp: *Lean Software Development – Kundenzentrierte Softwareentwicklung durch Anwendung schlanker Prinzipien*, 1. Auflage: AVM, München 2011, ISBN 978-3-86924-052-7.

Für ein mittelständisches Unternehmen soll eine Software entwickelt werden, mit der alle Finanzierungsaktivitäten überwacht und gesteuert werden können. Die Anwendung soll sowohl Buchhaltung, Kostenrechnung als auch Controlling unterstützen und aus einer Vielzahl von Modulen bestehen. Der Wissensträger hat in Zusammenarbeit mit den Entwicklern einige User Stories erarbeitet, welche die Anforderungen an die zu entwickelnde Software beschreiben. Eine der User Stories enthält folgenden Text:

*Als Mitarbeiter der Investitionsplanung möchte ich die Gesamtbelastung zu einem Tilgungsdarlehen berechnen, um die Konditionen verschiedener Kreditinstitute zu vergleichen.*

Es soll also ein Softwaremodul entwickelt werden, mit dem die durch Aufnahme eines Tilgungsdarlehens entstehende Gesamtschuld berechnet werden kann. An dieser Stelle sei kurz das Wesen des Tilgungsdarlehens erläutert. Es handelt sich um eine Form des Kredits, bei der die Annuitäten über die Laufzeit sinken. Während der Tilgungsbetrag in jeder Periode gleich hoch ist, nehmen die Zinsen kontinuierlich ab, da sie aus der verbleibenden Restschuld berechnet werden. Ein Beispiel zu dem Modul sei das folgende:

*Bei einem Tilgungsdarlehen in Höhe von 100.000,00 Euro, das über zehn Perioden zu 2 % verzinst wird, ergibt sich eine Gesamtschuld von 111.000,00 Euro. Die Tilgung beträgt in jeder Periode 10.000,00 Euro, die Zinsen werden auf die jeweils verbliebene Restschuld vor Tilgung berechnet.*

Es ist ein Modul zu entwickeln, welches vom Wissensträger die Eingabe eines Darlehensbetrages in Cent, einer Laufzeit, welche der Anzahl zurückzuzahlender Raten entspricht, sowie eines Zinssatzes in Prozent erwartet. Aus diesen Informationen soll das Programm die Gesamtschuld berechnen.

## 3 Vorbereitung von Modultests für die Verwendung in KJUnit

### 3.1 Entwicklung eines Moduls zu dem Beispiel

Die in Kapitel 3 vorgestellten Aktivitäten entsprechen den Schritten 1 und 2 der Abbildung 1 und werden durch die Entwickler durchgeführt.

Die Umsetzung der User Story wird mit der Entwicklung eines Modultests begonnen. Ein erster Test soll das vorgestellte Beispiel überprüfen. Es wird von der Administration ein Java-Projekt erstellt mit dem folgenden Aufbau, siehe Abbildung 2 der Administrationsdokumentation. Es wird JUnit 5 benutzt. Falls ein zu testendes Programm einen anderen Aufbau hat, müssen Anpassungen vorgenommen werden.

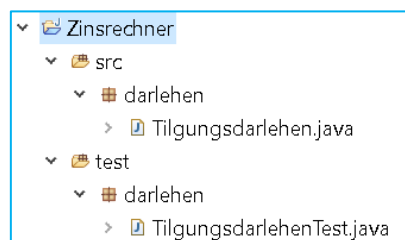


Abbildung 2: Struktur des Beispielprojekts

```
01 package darlehen;
02
03 import static org.junit.jupiter.api.Assertions.assertEquals;
04 import org.junit.jupiter.api.Test;
05
06 class TilgungsdarlehenTest {
07
08     private Tilgungsdarlehen t;
09
10     @Test
11     void testBerechneGesamtschuld()
12         throws Exception{
13         this.t = new Tilgungsdarlehen(10000000, 10, 2);
14         int berechneteGesamtschuld
15             = t.berechneGesamtschuld("Musterperson");
16         int erwarteteGesamtschuld = 11100000;
17         assertEquals(erwarteteGesamtschuld, berechneteGesamtschuld,
18             "Die berechnete Gesamtschuld entspricht nicht"
19             + " der erwarteten.");
20     }
21 }
```

Listing 1: Klasse *TilgungsdarlehenTest* (JUnit 5.x)

Es wird eine Testklasse *TilgungsdarlehenTest* in einem package *darlehen* im source

folder *test* angelegt. Auf imports und auch Kommentare wird in sämtlichen weiteren Listings verzichtet. Diese müssen nach Bedarf ergänzt werden.

Der Testfall wird durch eine Methode repräsentiert, welche mit der Annotation `@Test` versehen ist. In diesem Beispiel wird im Inneren der Testmethode zunächst das Attribut *t* initialisiert, welches vom Typ *Tilgungsdarlehen* ist. Die Klasse *Tilgungsdarlehen* wurde bisher noch nicht definiert. Sie soll die gesamte Rechenfunktionalität enthalten und wird deshalb über einen Konstruktor verfügen, der die notwendigen Berechnungsparameter Darlehenshöhe, Laufzeit und Zinssatz entgegennimmt. Dann wird mittels *t* die Methode *berechneGesamtschuld* des Tilgungsdarlehens mit einem Parameter für die Vorgabe des Users aufgerufen. Der Rückgabewert wird anschließend in der Methode *assertEquals* mit dem Erwartungswert aus der User Story – 111.000,00 Euro – verglichen. Sind beide Werte gleich, so wird JUnit bei Ausführung des Modultests einen erfolgreichen Testlauf ausgeben. Sind die Werte verschieden, so werden eine Fehlermeldung und der im *assertEquals*-Aufruf als Parameter mitgegebene Text ausgegeben. Falls der Rückgabewert vom Typ *boolean* ist, verwenden Sie *assertTrue*, ansonsten *assertEquals*.

Dann wird die Klasse *Tilgungsdarlehen* implementiert.

```
01  package darlehen;
02
03  public class Tilgungsdarlehen{
04
05      private int      darlehen;
06      private int      laufzeit;
07      private double   zinssatz;
08
09      public Tilgungsdarlehen(int darlehen, int laufzeit,
10          double zinssatz){
11          this.darlehen = darlehen;
12          this.laufzeit = laufzeit;
13          this.zinssatz = zinssatz / 100;
14      }
15
16      public int berechneGesamtschuld(String user){
17          int gesamtschuld = 0;
18          int restschuld = darlehen;
19          for(int i = 0; i < laufzeit; i++){
20              gesamtschuld += darlehen / laufzeit
21                  + restschuld * zinssatz;
22              restschuld = restschuld - darlehen / laufzeit;
24          }
25          return gesamtschuld;
26      }
27  }
```

*Listing 2: Klasse Tilgungsdarlehen*



Die drei wichtigen Größen des Tilgungsdarlehens werden als Attribute definiert und über einen Konstruktor gefüllt. Sobald ein neues Objekt der Klasse angelegt werden soll, müssen Darlehensbetrag, Laufzeit und Zinssatz angegeben werden. Neben den Attributen und einem Konstruktor wird zusätzlich die zu testende Methode *berechneGesamtschuld* erstellt, welche den insgesamt an das Kreditinstitut zurückzuzahlenden Betrag, bestehend aus Zins und Tilgung, ermittelt. Der Parameter *user* wurde aus Testzwecken hinzugefügt.

Nachdem die Klasse implementiert wurde, können die in *TilgungsdarlehenTest* definierten Testfälle durch das JUnit-Framework ausgeführt werden.

Eine JUnit Testklasse und die zu testende Klasse können auch aus einem Backlog erzeugt werden, siehe Schritt 0 der Abbildung 1, mehr dazu im Kapitel 5.

Werden die Klasse *Tilgungsdarlehen* und der zugehörige Test wie beschrieben implementiert, so meldet JUnit einen erfolgreichen Testverlauf. Tatsächlich können aber noch immer Fehler bestehen, denn es kann nicht garantiert werden, dass die Tests alle möglichen Probleme und Randbedingungen abdecken. So ist es im vorliegenden Beispiel möglich, bei der Erzeugung eines neuen Tilgungsdarlehens eine Laufzeit von null Perioden oder ein negatives Darlehen anzugeben. Dies macht keinen Sinn und kann beim Nutzer zu Irritationen führen. Die Entwickler sind mit dem Geschäftsbereich des Kunden nur selten vertraut und erkennen die aus seiner Sicht selbstverständlichen Randbedingungen häufig nicht. Im Extremfall werden unzureichende Tests erst im produktiven Einsatz der Software bemerkt, weshalb ein Weg gefunden werden muss, Testfälle bereits frühzeitig durch den Wissensträger überprüfen zu lassen. Hierfür kommt KJUnit zum Einsatz.

## 3.2 Vorbereitung für die Verwendung in KJUnit

Bevor KJUnit zum Einsatz kommen kann, müssen die Testfälle parametrisiert und in einer Archivdatei zusammengefasst werden.

### 3.2.1 Parametrisierung der Testfälle

Innerhalb einer Testklasse müssen diejenigen Parameter gekennzeichnet werden, welche der Wissensträger mittels KJUnit - Wissensträger verändern können soll. Diese müssen in Form öffentlich zugänglicher Klassenvariablen codiert werden.<sup>1</sup> Erstellen Sie ein neues source folder *testForKJUnit* und in diesem das package *darlehen*. In diesem package erstellen Sie die parametrisierte JUnit Testklasse *TilgungsdarlehenKJUnitTest*. Listing 3 zeigt die notwendigen Änderungen an der Beispielklasse *TilgungsdarlehenTest* für JUnit 5.x. In den Zeilen 5 bis 12 werden

---

<sup>1</sup>Dabei müssen sämtliche Parameter von einem Java-Grunddatentyp, dem Datentyp String oder dem Datentyp Enum sein. Parameter anderer Datentypen können in KJUnit bisher nicht modifiziert werden.

zunächst alle Parameter definiert, die durch den Wissensträger vor Aufruf der Testmethode modifiziert werden sollen. Jeder Parameter wird als öffentliche Klassenvariable definiert und beginnt mit dem Namen der Testmethode, in der er verwendet wird. Es folgen ein Unterstrich und anschließend der eigentliche Parametername, wenn es sich um einen Eingangsparameter handelt, der in das Produktivmodul einfließt. Ist der betrachtete Parameter ein Ergebnisparameter, der lediglich zur Prüfung des vom Modul zurückgegebenen Ergebnisses dient, so wird das Schlüsselwort *exp* zwischengeschoben.

Da im vorliegenden Beispiel lediglich ein Testfall definiert wird, beginnen alle Parameter mit dem Testnamen *testBerechneGesamtschuld*. Es wird ersichtlich, dass jeder Parameter nur zu einer Testmethode gehören kann.

```
01 package darlehen;
02
03 class TilgungsdarlehenKJUnitTest {
04
05     public static String testBerechneGesamtschuld_User
06         = "Musterperson";
07     public static int testBerechneGesamtschuld_Darlehen = 10000000;
08     public static int testBerechneGesamtschuld_Laufzeit = 10;
09     public static double testBerechneGesamtschuld_Zinssatz = 2;
10     public static int
11         testBerechneGesamtschuld_exp_ErwarteteGesamtschuld
12         = 11100000;
13
14     private Tilgungsdarlehen t;
15
16     @Test
17     void testBerechneGesamtschuld()
18         throws Exception{
19         this.t = new Tilgungsdarlehen(
20             testBerechneGesamtschuld_Darlehen,
21             testBerechneGesamtschuld_Laufzeit,
22             testBerechneGesamtschuld_Zinssatz);
23         int berechneteGesamtschuld = t.berechneGesamtschuld(
24             testBerechneGesamtschuld_User);
25         int erwarteteGesamtschuld
26             = testBerechneGesamtschuld_exp_ErwarteteGesamtschuld;
27         assertEquals(erwarteteGesamtschuld, berechneteGesamtschuld,
28             "Die berechnete Gesamtschuld entspricht nicht "
29             + "der erwarteten Gesamtschuld");
30     }
31 }
```

*Listing 3: Klasse TilgungsdarlehenKJUnitTest (parametrisiert) (JUnit 5.x)*

In den Testmethoden werden nun nicht mehr die konkreten Werte aus der User Story verwendet, sondern die Parameter eingesetzt. Beispielsweise wird im Testfall

*testBerechneGesamtschuld* der Variablen *erwarteteGesamtschuld* nicht mehr der Wert 11100000 zugewiesen, sondern der Parameter *testBerechneGesamtschuld\_exp\_ErwarteteGesamtschuld*. Damit die Tests nach wie vor funktionieren, wird jeder Klassenvariable der bisherige Standardwert aus der User Story zugewiesen.

## Hinweise

Bei Betrachtung der Listings fällt auf, dass es in den Testklassen keine *setUp*-Methoden oder *tearDown*-Methoden beziehungsweise Methoden, welche mittels *@BeforeEach*, *@AfterEach* und weitere annotiert sind, gibt. Die Initialisierung des Tilgungsdarlehens *t* wird in jeder Testmethode separat vorgenommen. Grundsätzlich ist mit KJUnit die Verwendung von *setUp*-Methoden oder *tearDown*-Methoden natürlich möglich, nur muss eine Testmethode derart implementiert werden, dass sie bei einer Neubelegung der öffentlich zugänglichen Klassenvariablen immer noch korrekt testet. Die Reihenfolge der Ausführung der Testmethoden darf keine Rolle spielen, da der Wissensträger mittels KJUnit – Wissensträger die Testmethoden einzeln und in beliebiger Reihenfolge abspielen und bearbeiten kann. Testmethoden, die mittels KJUnit bearbeitet werden, sollten nicht das Auftreten von Exceptions testen, da KJUnit für diesen Sachverhalt eine eigene Lösung anbietet, siehe unten. Testsuites werden von KJUnit nicht berücksichtigt. Auch hierfür bietet KJUnit eine eigene Lösung an.

Die parametrisierten Testklassen werden im weiteren auch KJUnit Testklassen genannt.

Die Parametrisierung der Testfälle kann auch mittels automatischer Erzeugung vorgenommen werden, mehr dazu im Kapitel 5.

Eine KJUnit Testklasse und die zu testende Klasse können auch aus einem Backlog erzeugt werden, siehe Schritt 0 der Abbildung 1, mehr dazu im Kapitel 5.

### 3.2.2 Zusammenfassung aller Testressourcen in einem Java-Archiv

Um die Testfälle in KJUnit - Wissensträger öffnen und modifizieren zu können, werden Testinformationen benötigt.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <testcases>
04         <testcase>
05             <path> darlehen.TilgungsdarlehenKJUnitTest
06                 .testBerechneGesamtschuld
07             </path>
08             <desc> Dieser Test &#252;berpr&#252;ft die Berechnung der
09                 Gesamtschuld.
10             </desc>
11             <testtype> 5 </testtype>
```

```

12         <parameters>
13             <parameter>
14                 <name> testBerechneGesamtschuld_User </name>
15                 <desc> Benutzer, welcher die Berechnung
16                     durchf&#252;hren darf
17                 </desc>
18             </parameter>
19             <parameter>
20                 <name> testBerechneGesamtschuld_Darlehen </name>
21                 <desc> Darlehen, das zur&#252;ckgezahlt werden soll
22                 </desc>
23             </parameter>
24             <parameter>
25                 <name> testBerechneGesamtschuld_Laufzeit </name>
26                 <desc> Laufzeit des Darlehens in Raten </desc>
27             </parameter>
28             <parameter>
29                 <name> testBerechneGesamtschuld_Zinssatz </name>
30                 <desc> Zinssatz, der vor Zahlung jeder Rate auf die
31                     Restschuld berechnet wird (1% = 0,01)
32                 </desc>
33             </parameter>
34             <parameter>
35                 <name> testBerechneGesamtschuld_exp_ErwarteteGesamtschuld
36                 </name>
37                 <desc> erwartete Gesamtschuld (Summe aller Tilgungen
38                     und Zinsen), die an den Kreditgeber
39                     zur&#252;ckgezahlt werden muss
40                 </desc>
41             </parameter>
42         </parameters>
43     </testcase>
44 </testcases>
45 </root>

```

Listing 4: Inhalt der Datei *CustomerTestCaseInformation.xml*

Die Testinformationen werden als XML-Dokument verwaltet, welches den Namen *CustomerTestCaseInformation.xml* tragen und im source folder *testForkBUnit* liegen muss. Falls ein zu testendes Programm einen anderen Aufbau hat, muss *CustomerTestCaseInformation.xml* derart gelegt werden, dass sie in der zu erstellenden jar-Datei, siehe Abbildung 3, im root-Verzeichnis liegt. Listing 4 zeigt die Testinformationen zu dem Beispiel.

Aus dem Listing wird ersichtlich, dass für jeden Testfall ein eigenes Element *<testcase>* angelegt wird. Dieses enthält wiederum folgende Kindelemente:

- *path*: Pfad der Testmethode innerhalb des JARs. (Entfernen Sie die Zeilenumbrüche!)
- *desc*: Beschreibung des Testfalls
- *testtype*: Angabe, ob ein JUnit 4 - Test oder JUnit 5 – Test vorliegt
- *parameters*: eine Menge von *parameter*-Elementen, die für jeden der Testfallparameter einen Namen *name* (entspricht dem Namen der jeweiligen Klassenvariable in der Testklasse) sowie eine Beschreibung *desc* enthält

Durch diese Informationen wird KJUnit – Wissensträger mitgeteilt, welche der im JAR enthaltenen Testfälle durch den Wissensträger modifiziert und ausgeführt werden dürfen. Weiterhin können die hinterlegten Beschreibungen in KJUnit – Wissensträger angezeigt und damit die Verständlichkeit der Testfälle für den Wissensträger erhöht werden.

Um die Testfälle in KJUnit - Wissensträger öffnen und modifizieren zu können, werden die zu testenden Klassen, die Testfälle und die Testinformationen in einem Java-Archiv zusammengefasst, siehe Abbildung 3.

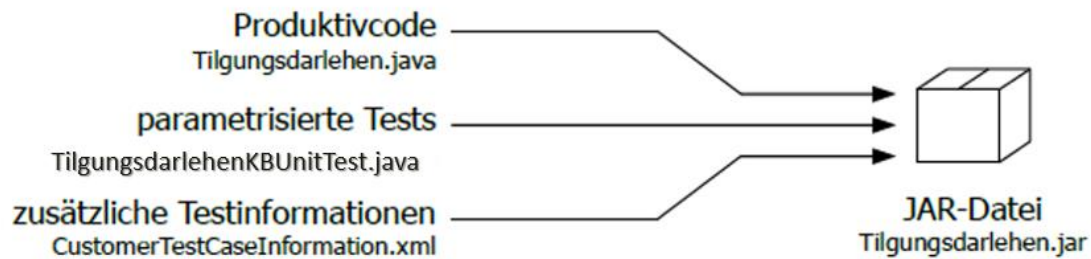


Abbildung 3: Zusammenführung der Testressourcen in einer JAR-Datei

Mit dem Erzeugen des Java-Archivs *Tilgungsdarlehen.jar* sind alle notwendigen Voraussetzungen erfüllt, um den Testfall des Beispiels in KJUnit - Wissensträger modifizieren und ausführen zu können.

### Hinweis

Das Erstellen der Testinformationen kann auch mittels automatischer Erzeugung vorgenommen werden, mehr dazu im Kapitel 5.

## 4 Modifizierung und Ausführung von Modultests mit KJUnit - Wissensträger

Die in Kapitel 4 vorgestellten Aktivitäten entsprechen den Schritten 3 und 4 der Abbildung 1 und werden durch die Wissensträger durchgeführt.

Wurden beliebige Testfälle wie im vorigen Kapitel beschrieben parametrisiert und in einer JAR-Datei zusammengefasst, so können sie in KJUnit – Wissensträger ausgeführt werden. Der Wissensträger hat dabei die Möglichkeit, die Testparameter in einer nutzerfreundlichen Oberfläche zu ändern. Er benötigt keinerlei Programmierkenntnisse und kann das Verhalten von Klassen oder Methoden durch Eingabe der aus seiner Sicht typischen Parameterkombinationen frühzeitig überprüfen. Anhand des Testergebnisses erkennt er, ob sich das getestete Modul wie beabsichtigt verhält oder Änderungen durch die Entwickler vorgenommen werden müssen.

### 4.1 Konfiguration der Anwendung

Es müssen KJUnit - Server und eine zugehörige Datenbank erstellt und jeweils gestartet worden sein. Informationen dazu finden Sie im Kapitel 4 der Administrationsdokumentation. Das Programm unterstützt standardmäßig eine MySQL-Datenbank.

Der komplette Quellcode von KJUnit - Wissensträger steht zur Verfügung. Die Anlage eines Projekts wird im Kapitel 5 der Administrationsdokumentation beschrieben. Sie starten das Projekt mit Hilfe von Eclipse, die *main*-Methode liegt in der Klasse *Main* im package *main*.

Alternativ wird die Anwendung als ausführbares Java-Archiv *KJUnit\_Wissenstraeger.jar* gemeinsam mit der Konfigurationsdatei *config.xml* und einem Verzeichnis mit icons ausgeliefert. Kopieren Sie alle Elemente auf Ihr System in dasselbe Verzeichnis. Bevor die Anwendung gestartet wird, sollten die Konfigurationsparameter in der Datei *config.xml* an die vorhandene Situation angepasst werden. Der Aufbau der Datei ist in Listing 5 dargestellt.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <path>http://localhost:8080/KJUnitServer/rest/kbUnit</path>
05     </rest>
06     <documentation>
07         <path>Anwendungsdokumentation_KJUnit.pdf</path>
08     </documentation >
09 </root>
```

Listing 5: Inhalt der Datei *config.xml*

Die Datei enthält Informationen über KJUnit - Server, ein RESTful Webservice, siehe

Schritt 4 der Abbildung 1. Der RESTful Webservice muss an dem in der Zeile 4 der Datei *config.xml* genannten Ort zur Verfügung stehen. KJUnit – Server wird benötigt, um die vom Wissensträger ausgeführten Tests zu speichern und später erneut ablaufen lassen zu können.

Neben der Anbindung an KJUnit - Server wird in der Datei *config.xml* zusätzlich der Ort genannt, an dem diese Anwendungsdokumentation im pdf-Format zu finden ist. Falls die Einstellung übernommen wird, muss die Anwendungsdokumentation in demselben Verzeichnis liegen wie *config.xml*. Liegt die Anwendungsdokumentation in einem Unterverzeichnis *doc*, so muss als Pfad

*doc\\Anwendungsdokumentation\_KJUnit\_Wissenstraeger.pdf*

angegeben werden. Sie können auch den absoluten Pfad angeben.

Das Programm benötigt weitere externe Bibliotheken, siehe Administrationsdokumentation, die in einem Unterverzeichnis *KJUnit\_Wissenstraeger\_lib* liegen müssen. Weiterhin benutzt das Programm externe Icons, welche im Unterverzeichnis *icons* liegen müssen, siehe Administrationsdokumentation.

## 4.2 Start der Anwendung und Laden von Testressourcen

Nach abgeschlossener Konfiguration kann KJUnit – Wissensträger gestartet werden.<sup>2</sup> Es sollte die in Abbildung 4 dargestellte Programmoberfläche erscheinen. Der Nutzer wird unvermittelt aufgefordert, Testressourcen zum Laden auszuwählen. Durch einen Klick auf das Lupen-Symbol öffnet sich ein Dialogfenster, in dem Sie beispielsweise das in Kapitel 3 erzeugte Java-Archiv *Tilgungsdarlehen.jar* auswählen können. Das Programm ermittelt anschließend, ob sich in der gewählten Datei tatsächlich verwendbare Testressourcen befinden. Ist dies der Fall, so kann der Ladevorgang mit einem Klick auf *OK* gestartet werden.

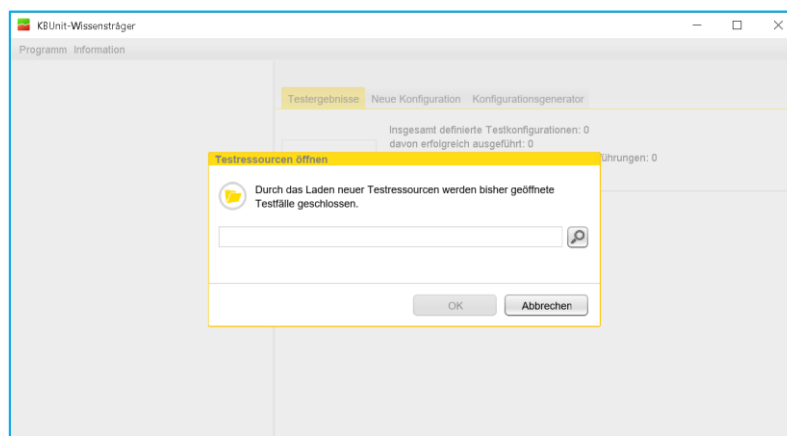


Abbildung 4: Laden von Testressourcen

<sup>2</sup>Insbesondere bei Unix-basierten Betriebssystemen kann es erforderlich sein, vor Start der Anwendung dem System die direkte Ausführung von JAR-Dateien zu erlauben.

Sobald der Ladevorgang abgeschlossen ist, wird der Inhalt des Hauptanwendungsfensters sichtbar, siehe Abbildung 5. Mittels *Programm* → *Beenden* können Sie das Programm beenden. Mittels *Information* → *Information zur Lizenz öffnen* erhalten Sie einen Hinweis zu der zugrundeliegenden Lizenz. Mittels *Information* → *Dokumentation öffnen* können Sie die vorliegende Anwendungsdokumentation öffnen. Auf der linken Seite des Fensters befindet sich eine Baumdarstellung der geöffneten Testfälle. Da im Beispiel nur ein Testfall angelegt wurde, enthält der Baum nur ein Testdokument. Die im Baum ebenfalls dargestellten Ordner entsprechen denjenigen packages, die mittels der zusätzlichen Testinformationen *CustomerTestCase-Information.xml*, siehe Listing 4 hinterlegt sind. Der Testfall *testBerechneGesamtschuld* ist bereits geöffnet. Die vorhandene Testkonfiguration ist diejenige, die der Entwickler mitgegeben hat. Um einen anderen Testfall (, in diesem Beispiel nicht vorhanden,) zu öffnen, wählen Sie im Kontextmenü des Testfalls *Test öffnen*.

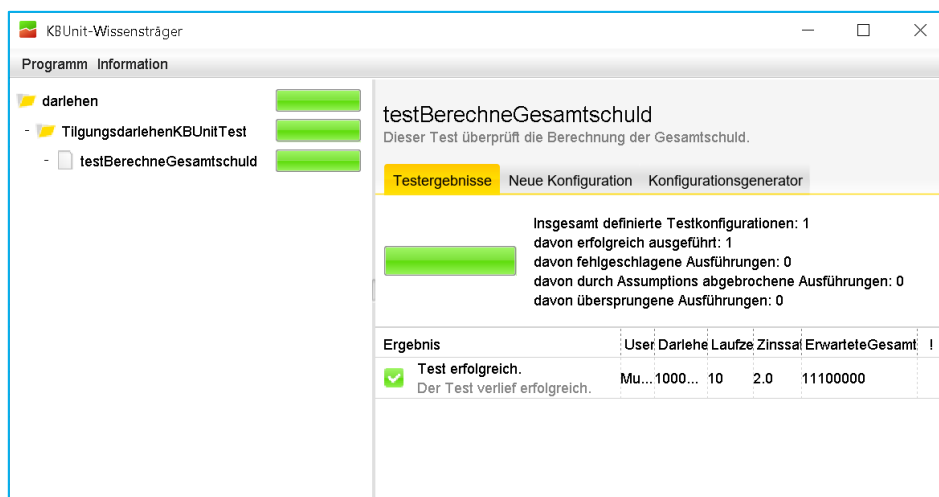


Abbildung 5: Hauptanwendungsfenster

## 4.3 Modifikation und Ausführung von Testfällen

Sie sehen auf der rechten Seite des Anwendungsfensters eine Beschreibung des geöffneten Testfalles, wie sie in Kapitel 3 in der Datei *CustomerTestCase-Information.xml* hinterlegt wurde. Darunter befinden sie drei Reiter, die im Folgenden näher erläutert werden.

### 4.3.1 Reiter Testergebnisse und Neue Konfiguration

Der Reiter *Testergebnisse* zeigt bisher ausgeführte Tests mit ihren Ergebnissen an. Da die in Abbildung 5 dargestellte Anwendung gerade zum ersten Mal gestartet wurde, enthält die Tabelle nur einen Eintrag, die Testkonfiguration vom Entwickler, folgend auch Initialtestkonfiguration genannt.

Wenn Sie auf den Reiter *Neue Konfiguration* klicken, erscheinen die in Abbildung 6 dargestellten Eingabefelder. Hier können Sie für alle Parameter des Testfalles einen



Wert eingeben, mit dem der Test ausgeführt werden soll. Handelt es sich um einen Eingangsparameter (gekennzeichnet durch ein hochgestelltes E), also einen Parameter, der vom zu testenden Modul verarbeitet wird und nicht der Prüfung der Modulrückgabe dient, können sogar mehrere Parameterwerte durch Semikolon getrennt angegeben werden, für die von einem gleichen Testergebnis ausgegangen wird, siehe Anhang. Standardmäßig enthalten alle Eingabefelder die im parametrisierten Modultest hinterlegten Testwerte. Neben den Parameterwerten können Sie zusätzlich angeben, ob Sie bei Ausführung des Tests mit dem Wurf einer Exception rechnen. Mit einem Klick auf *Konfiguration(en) testen* werden die von Ihnen eingegebenen Werte an den Test übertragen, dieser ausgeführt und das Ergebnis unter dem Reiter *Testergebnisse* angezeigt.

The screenshot shows the KJUnit application window. On the left, a tree view shows the project structure: 'darlehen' (expanded), 'TilgungsdarlehenKJUnitTest' (expanded), and 'testBerechneGesamtschuld'. The main area is titled 'testBerechneGesamtschuld' with a description: 'Dieser Test überprüft die Berechnung der Gesamtschuld.' Below this are three tabs: 'Testergebnisse', 'Neue Konfiguration' (selected), and 'Konfigurationsgenerator'. A text block explains: 'Sobald die einzelnen Testparameter mit Werten belegt sind, kann eine neue Konfiguration angelegt und getestet werden. Für Eingangsparameter können mehrere durch Semikolon getrennte Werte angegeben werden, wodurch automatisch mehrere Konfigurationen generiert werden.' Below this is a checkbox 'Exception erwartet' (unchecked). Then, 'Grad der Interaktion' is set to '1' and 'Priorität' is set to 'Kurze Generierungszeit'. The input fields are: 'User<sup>E</sup>' with 'Musterperson', 'Darlehen<sup>E</sup>' with '10000000', 'Laufzeit<sup>E</sup>' with '11', 'Zinssatz<sup>E</sup>' with '2.0', and 'ErwarteteGesamtschuld<sup>R</sup>' with '11199987'. At the bottom is a button 'Konfiguration(en) testen'.

Abbildung 6: Testen einer neuen Konfiguration

Haben Sie für jeden Parameter genau einen Wert eingegeben, so erzeugt KJUnit zum Testfall *testBerechneGesamtschuld* genau eine Konfiguration. Haben Sie dagegen für einige Eingabefelder mehrere Werte eingegeben, so erzeugt KJUnit automatisch mehrere Konfigurationen zu ein und demselben Testfall. Ist der Grad der Interaktion 1, so kommt jeder Wert in mindestens einer Testkonfiguration vor. Ist der Grad der Interaktion 2, so werden sämtliche Werte eines Eingabefeldes mit denjenigen eines weiteren Eingabefeldes kombiniert. Zu jeder Kombination gibt es eine Testkonfiguration. Ist der Grad der Interaktion 3, so wird wie beim Interaktionsgrad 2 verfahren, es werden aber alle Kombinationen von Werten von jeweils 3 Eingabefeldern generiert. Analog wird für höhere Interaktionsgrade verfahren.

Für die Generierung von Testkonfigurationen sind unterschiedliche Algorithmen hinterlegt. Haben Sie als Priorität *Kurze Generierungszeit* ausgesucht, so wird ein Algorithmus ausgewählt, welcher bei der Generierung von Testkonfigurationen möglichst schnell ist. Wird als Priorität *Weniger Testfälle* ausgesucht, so wird ein Algorithmus ausgewählt, welcher unter Einhaltung des angegebenen Interaktionsgrads möglichst wenig Testfälle generiert.

Führen Sie die Testkonfiguration mit den in Abbildung 6 dargestellten Eingaben durch. Anschließend wechselt das Programm automatisch in den ersten Reiter und zeigt Ihnen, gelb hervorgehoben, das Ergebnis der Ausführung, siehe Abbildung 7. Die Konfiguration hat zu einem positiven Testergebnis geführt. Da sich in der Tabelle nur erfolgreiche Testkonfigurationen befinden, sind die Statusbalken in der Baumansicht grün. Die Statusbalken zeigen durch einen grünen, gelben, grauen und / oder roten Balken an, wie sich die Ergebnisse der bisher getesteten Konfigurationen verteilen. Dabei steht gelb für durch Assumptions (Annahmen) nicht ausgeführte Testfälle, grau für übersprungene Testfälle und rot für durchgelaufene Testfälle, die aber nicht erfolgreich waren. (Testfälle können Annahmen enthalten. Sie werden dann nur ausgeführt, wenn die Annahmen zutreffen. Weiterhin können Testfälle Anmerkungen enthalten, dass sie übersprungen werden sollen.)

The screenshot shows the KBUnit-Wissensträger application window. On the left, a tree view shows the project structure: 'darlehen' (green), 'TilgungsdarlehenKBUnitTest' (green), and 'testBerechneGesamtschuld' (green). The main area displays the test results for 'testBerechneGesamtschuld'. A summary box shows: 'Insgesamt definierte Testkonfigurationen: 2', 'davon erfolgreich ausgeführt: 2', 'davon fehlgeschlagene Ausführungen: 0', 'davon durch Assumptions abgebrochene Ausführungen: 0', and 'davon übersprungene Ausführungen: 0'. Below this is a table with the following data:

Ergebnis	User	Darlehen	Laufzeit	Zinssatz	ErwarteteGesamtschuld
Test erfolgreich. Der Test verlief erfolgreich.	Mu...	10000...	11	2.0	11199987
Test erfolgreich. Der Test verlief erfolgreich.	Mu...	10000...	10	2.0	11100000

Abbildung 7: Ergebnis einer getesteten Konfiguration

Die getesteten Konfigurationen werden automatisch in der Datenbank abgelegt und stehen damit auch nach Neustart des Programms wieder zur Verfügung. Innerhalb der Tabelle können Sie einzelne oder mehrere Konfigurationen auswählen und per Rechtsklick ein Popup-Menü aufrufen. Dieses stellt je nach Auswahl folgende

Aktionen zur Verfügung:

- *Test erneut ausführen*: testet die ausgewählte(n) Konfiguration(en) erneut
- *Test löschen*: löscht die ausgewählte(n) Konfiguration(en) aus der Tabelle und der Datenbank

Löschen Sie die Testkonfiguration mit Laufzeit 11. Die Initialtestkonfiguration kann nicht gelöscht werden. Probieren Sie letzteres aus.

Es sei an dieser Stelle darauf hingewiesen, dass ein ähnliches Popup-Menü auch für die Baumstruktur zur Verfügung steht. Hier haben Sie die Möglichkeit, alle zu den von Ihnen im Baum selektierten Testfällen gehörigen, in der Datenbank gespeicherten Konfigurationen erneut auszuführen. Dadurch wird es beispielsweise möglich, sämtliche jemals definierten Konfigurationen bei Programmänderungen mit nur einem Klick wiederholt ablaufen zu lassen.

### 4.3.2 Reiter Konfigurationsgenerator

Der Konfigurationsgenerator generiert auf der Basis von Nutzereingaben mehrere Testkonfigurationen.

The screenshot shows the KJUnit application window with the 'Konfigurationsgenerator' tab selected. On the left, a tree view shows the project structure: 'darlehen' (expanded), 'TilgungsdarlehenKJUnitTest' (expanded), and 'testBerechneGesamtschuld'. The main area displays the test 'testBerechneGesamtschuld' with a description: 'Dieser Test überprüft die Berechnung der Gesamtschuld.' Below this, there are three tabs: 'Testergebnisse', 'Neue Konfiguration', and 'Konfigurationsgenerator' (active). A text block explains: 'Sobald für alle Eingangsparameter gültige Werte und/oder Wertintervalle definiert wurden, ermittelt KJUnit die für eine hohe Testabdeckung erforderlichen Parameterkombinationen und zeigt die generierten Konfigurationen zur Eingabe von Ergebniswerten an.' The configuration parameters are listed below: 'Grad der Interaktion' (4), 'Priorität' (Kurze Generierungszeit), 'Validiere Werte separat' (checked), 'User' (Musterperson), 'Darlehen' (10000000), 'Laufzeit' (10), and 'Zinssatz' ([0,20]). A 'Konfigurationen generieren' button is at the bottom.

Abbildung 8: Parametereingabe im Konfigurationsgenerator

Abbildung 8 zeigt den Aufbau des Generators. Der Nutzer kann für alle Eingangsparameter des Tests mehrere Werte angeben. Das Besondere dabei ist, dass für numerische Datentypen die Eingabe von Intervallen erlaubt ist. KJUnit – Wissens-träger bestimmt automatisch innerhalb und außerhalb dieser Intervalle liegende

Werte sowie Intervallgrenzen. Es gibt in diesem Zusammenhang gültige Werte, Grenzwerte und ungültige Werte. Nähere Informationen sind dem Anhang zu entnehmen. Die zu einem Parameter ermittelten Werte werden anschließend mit Werten anderer Parameter kombiniert. Informationen zu den Angaben von dem *Grad der Interaktion* und der *Priorität* entnehmen Sie aus dem Unterkapitel 4.3.1. Falls Sie *Valide Werte separat* angeben, so werden nicht valide Werte, also ungültige Werte und Grenzwerte, bei der Generierung von Testkonfigurationen gesondert von den validen Werten, also gültigen Werten, behandelt. Falls Sie nicht *Valide Werte separat* angeben, so werden bei der Generierung von Testkonfigurationen gültige Werte, Grenzwerte und ungültige Werte identisch behandelt.

Übernehmen Sie die in Abbildung 8 dargestellten Eingaben. Durch einen Klick auf *Konfigurationen generieren* zeigt KBUit - Wissensträger die ermittelten Parameterkombinationen an, siehe Abbildung 9. Innerhalb der Tabelle werden ungültige Werte rot angezeigt, Grenzwerte gelb und gültige Werte schwarz. Für Kombinationen, die einen ungültigen Parameterwert enthalten, wird vom Programm automatisch der Wurf einer Exception erwartet. Sie können dies ändern, indem Sie den Haken am Ende der Zeile entfernen und anschließend Werte für alle Ergebnisparameter (Zellen, deren Spalte durch ein hochgestelltes R gekennzeichnet ist) der betroffenen Konfiguration eintragen.

Für jede der Konfigurationen müssen Sie entweder die Ergebnisparameter mit Werten belegen oder einen Haken am Ende der Zeile setzen, sollten Sie bei Ausführung der Konfiguration eine Exception erwarten. Sie können auch einzelne Konfigurationen aus der Tabelle entfernen. Klicken Sie hierzu mit der rechten Maustaste auf die zu löschende Kombination und wählen Sie im erscheinenden Menü anschließend *Konfiguration löschen*.

User <sup>E</sup>	Darlehen <sup>E</sup>	Laufzeit <sup>E</sup>	Zinssatz <sup>E</sup>	Erwartete Gesamtschuld <sup>R</sup>	!
null	10000000	10	8.5	11100000	<input checked="" type="checkbox"/>
""	10000000	10	8.5	11100000	<input checked="" type="checkbox"/>
Musterper...	13773635	10	8.5	11100000	<input checked="" type="checkbox"/>
Musterper...	-655	10	8.5	11100000	<input checked="" type="checkbox"/>
Musterper...	10000000	-140	8.5	11100000	<input checked="" type="checkbox"/>
Musterper...	10000000	717	8.5	11100000	<input checked="" type="checkbox"/>
Musterper...	10000000	10	159.1	11100000	<input checked="" type="checkbox"/>
Musterper...	10000000	10	-603.1	11100000	<input checked="" type="checkbox"/>
Musterper...	10000000	10	0.0	11100000	<input type="checkbox"/>
Musterper...	10000000	10	20.0	11100000	<input type="checkbox"/>
Musterper...	10000000	10	8.5	11100000	<input type="checkbox"/>

Abbildung 9: vom Konfigurationsgenerator ermittelte Parameterkombinationen

Durch einen Klick auf *Abbrechen* kommen Sie zurück zur Ausgangsseite des Generators. Setzen Sie für den Testfall mit Zinssatz 0 die erwartete Gesamtschuld auf 0 und einen Haken, da Sie eine Exception erwarten. Ändern Sie für den Testfall mit Zinssatz 20 die erwartete Gesamtschuld ab auf 21000000. Klicken Sie dann auf *Konfiguration(en) testen*, um die einzelnen Konfigurationen auszuführen. Analog zur Anlage einfacher Konfigurationen schaltet das Programm auf den ersten Reiter um und zeigt Ihnen die einzelnen Testergebnisse.

The screenshot shows the KJUnit-Wissensträger application window. On the left, a tree view shows the project structure: 'darlehen' (expanded), 'TilgungsdarlehenKJUnitTest', and 'testBerechneGesamtschuld'. The main area displays the test results for 'testBerechneGesamtschuld'. A summary bar indicates: 'Insgesamt definierte Testkonfigurationen: 13', 'davon erfolgreich ausgeführt: 7', 'davon fehlgeschlagene Ausführungen: 0', 'davon durch Assumptions abgebrochene Ausführungen: 0', and 'davon übersprungene Ausführungen: 0'. Below this is a table of test results.

Ergebnis	User	Darlehen	Laufzeit	Zinssatz	ErwarteteGesamtschuld	
Exception nicht aufgetreten. Die erwartete Exception trat nicht auf.	Mus...	10000000	717	8.5	11100000	✓
Exception nicht aufgetreten. Die erwartete Exception trat nicht auf.	Mus...	10000000	10	159.1	11100000	✓
Exception nicht aufgetreten. Die erwartete Exception trat nicht auf.	Mus...	10000000	10	-603.1	11100000	✓
Exception nicht aufgetreten. Die erwartete Exception trat nicht auf.	Mus...	10000000	10	0.0	0	✓
Test erfolgreich. Der Test verlief erfolgreich.	Mus...	10000000	10	20.0	21000000	
Test fehlgeschlagen. Die berechnete Gesamtschuld entspricht	Mus...	10000000	10	8.5	11100000	
Test erfolgreich. Der Test verlief erfolgreich.	Mus...	10000000	11	2.0	11199987	
Test erfolgreich. Der Test verlief erfolgreich.	Mus...	10000000	10	2.0	11100000	

Abbildung 10: Testergebnisse

Zusammengefasst erhält der Wissensträger dank KJUnit - Wissensträger die Möglichkeit, das Verhalten von Modultests bei bestimmten Parameterbelegungen zu überprüfen. Im Beispiel der Darlehensberechnung kann er beispielsweise untersuchen, wie sich der Testfall bei der Eingabe eines negativen Zinssatzes oder des Zinssatzes 0 verhält. Er erwartet, dass der Test in diesen Fällen Exceptions erzeugt. Diese werden jedoch in beiden Fällen nicht ausgelöst, da in den in Kapitel 3 vorgestellten Beispielmethode keine Überprüfung der Parameterbelegung erfolgt. Die entsprechenden Testfälle sind in der Datenbank gespeichert und stehen dem Entwickler zur Verfügung, siehe Kapitel 4. Durch den Einsatz der Anwendung können also Fehler aufgedeckt und die Entwicklung beschleunigt werden.

Der Übersicht halber werden für die Erläuterungen in den folgenden Kapiteln alle Testkonfigurationen gelöscht bis auf unterste, die erfolgreich durchgelaufene Testkonfiguration mit Darlehen 10000000, Laufzeit 10, Zinssatz 2 und Erwartete Gesamtschuld 11100000 und die nicht erfolgreich durchgelaufene Testkonfiguration mit Zinssatz 0 und Gesamtschuld 0, bei welcher eine Exception erwartet wurde, die nicht aufgetreten ist.

## 5 KJUnit – Entwickler

Die in Kapitel 5 vorgestellten Aktivitäten entsprechen den Schritten 0, 4 und 5 der Abbildung 1 und werden durch die Entwickler durchgeführt. Die Bereitstellung der Daten für den Schritt 6 wird ebenfalls durch die Entwickler durchgeführt und im Kapitel 5 vorgestellt. Weiterhin wird die automatische Erzeugung von parametrisierten JUnit Testklassen und die Generierung der *CustomerTestCaseInformation.xml* vorgestellt, siehe Schritt 1 und die Listings 3 und 4 aus Kapitel 3. Die Listings von Java-Dateien müssen, wie auch in Kapitel 3 um imports und Kommentare ergänzt werden. Gegebenenfalls muss die Anzeige des Projekts nach einer Generierung mittels *Refresh* im Kontextmenü des Projekts aktualisiert werden, damit die neuen Dokumente angezeigt werden.

### 5.1 Konfiguration der Anwendung

Die Konfiguration von *KJUnit – Entwickler* entnehmen Sie der Administrationsdokumentation.

Ein source folder *testKJUnit* wurde bereits erstellt. Es enthält ein package *prlab.kbunit*, welches unter anderem die Datei *PreferencesKJUnitServer.xml* und die Klasse *Main* enthält. Die vom Wissensträger mittels KJUnit - Wissensträger erzeugten Testfälle befinden sich in der zu KJUnit gehörenden Datenbank. In der Datei *PreferencesKJUnitServer.xml*, siehe Listing 6, wird in der Zeile 4 derjenige Ort genannt, an welchem der RESTful Webservice zur Verfügung steht, siehe Schritt 4 der Abbildung 1. Der RESTful Webservice und der Datenbankserver müssen gestartet werden. Der Webservice verwaltet die Datenbankzugriffe für KJUnit. Das Programm unterstützt standardmäßig eine MySQL-Datenbank, siehe Administrationsdokumentation.

Weiterhin werden in der Datei *PreferencesKJUnitServer.xml* ein Datum und eine Uhrzeit festgelegt. Es werden dann vom Runner nur Testfälle berücksichtigt, die nach dem vorgegebenen Datum erstellt worden sind, siehe unten.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <host>http://localhost:8080/KJUnitServer/rest/kbUnit</host>
05     </rest>
12     <update>
13         <!-- Copying test cases performed after [yyyy-MM-dd HH:mm:ss] :-->
14         <date>2020-04-14</date>
15         <time>17:03:35</time>
16     </update>
17 </root>
```

Listing 6: Inhalt der Datei *PreferencesKJUnitServer.xml*

In der Klasse *Main*, siehe Listing 7, wird die Anwendung gestartet.

```
01 package prlab.kbunit;
02
03 public class Main extends Application{
04
05     private Stage primaryStage;
06
07     public static void main(String[] args){
08         launch(args);
09     }
10
11     @Override
12     public void start(Stage primaryStage){
13         this.primaryStage = primaryStage;
14         this.primaryStage.setTitle("KBUnit-Entwickler");
15         this.primaryStage.getIcons().add(new Image(getClass().
16             getResourceAsStream(
17                 "/prlab/kbunit/resources/images/kbunit.png")));
18         shoMainScene();
19     }
20
21     public void showMainScene(){
22         try{
23             FXMLLoader loader = new FXMLLoader();
24             loader.setLocation(getClass().getResource(
25                 "/prlab/kbunit/resources/view/MainFrameScene.fxml"));
26             Parent root = loader.load();
27             Scene scene = new Scene(root);
28             scene.getStylesheets().add(getClass().getResource(
29                 "/prlab/kbunit/resources/css/MainFrameScene.css")
30                 .toExternalForm());
31             primaryStage.setScene(scene);
32             MainFrameController controller = loader.getController();
33             controller.setHostServices(getHostServices());
34             primaryStage.show();
35         }
36         catch(IOException ioExc){
37             System.out.println("Fehler beim Laden der "
38                 + "Oberflächen-Ressourcen.");
39         }
40     }
```

*Listing 7: Klasse Main*

## 5.2 Start und Aufbau der Anwendung

Nach erfolgreicher Konfiguration kann die Hauptanwendung von KJUnit - Entwickler gestartet werden. Mittels *Programm* → *Beenden* können sie das Programm beenden, siehe Abbildung 11. Mittels Informationen können die Lizenzbedingungen, die Autorinnen und Autoren dieser Anwendung und diese Anwendungsdokumentation eingesehen werden.

Die Anwendung KJUnit - Entwickler ist wie folgt aufgebaut, siehe Abbildung 11:

Das Hauptfenster ist horizontal in zwei Bereiche unterteilt. Im unteren Bereich links können Sie aus Dokumenten eines Projektmanagement-Tools, beispielsweise ein Backlog, Daten laden und mit diesen JUnit Testklassen, KJUnit Testklassen und Sourceklassen erzeugen. Im unteren Bereich mittig können Sie im Tabreiter *Transfer* aus einer JUnit Testklasse, welche im source folder *test* liegt, mittels eines Buttons *Parametrisieren* eine KJUnit Testklasse erzeugen, die dann im source folder *testForKJUnit* liegt. Die KJUnit Testklassen werden unterhalb von *Testklasse aussuchen*: in einer Combobox aufgelistet. Sie können mittels Klick auf eine der KJUnit Testklassen aus dieser Combobox diese aussuchen. Im Tabreiter *Customer-TestCaseInformationGenerator* können Sie zu einer ausgesuchten KJUnit Testklasse oder zu allen KJUnit Testklassen die Datei *CustomerTestCase-Information.xml* automatisiert generieren. Sie können weiterhin eine ausgesuchte KJUnit Testklasse mittels Klick auf den Button *KJUnit Testklasse öffnen* öffnen. Dann werden die Testkonfigurationen zu den Testfällen (, also zu den Testmethoden) dieser Testklasse aus der Datenbank gelesen und im oberen Bereich des Hauptfensters in der Tabelle angezeigt.

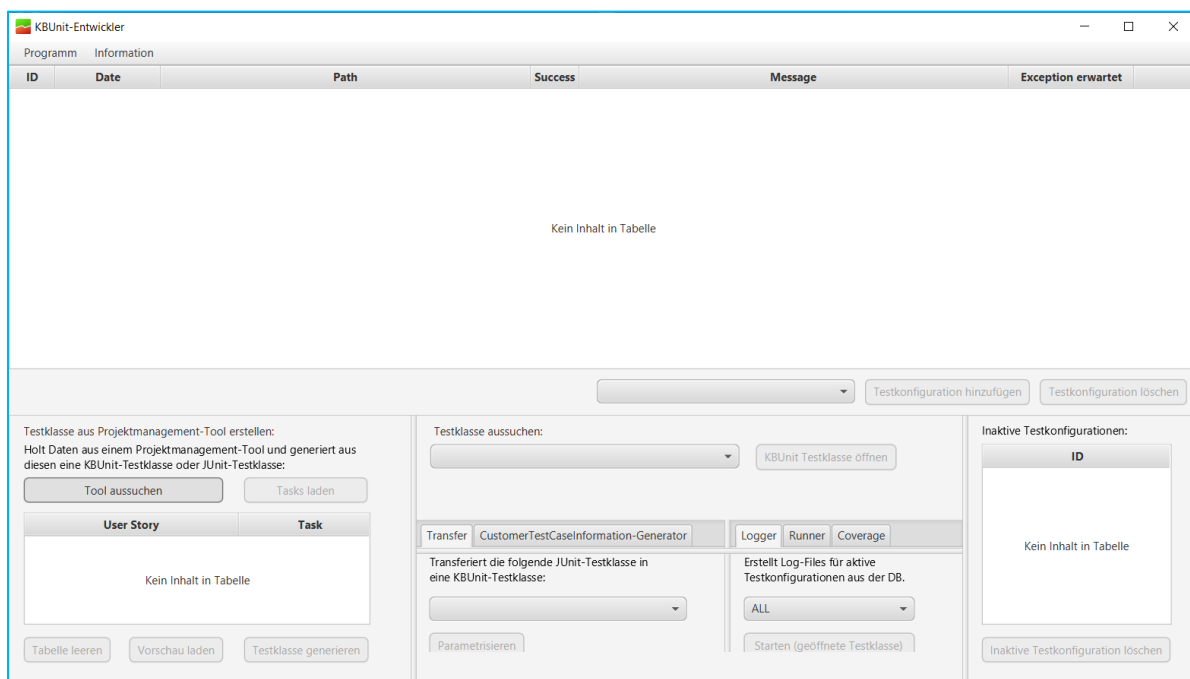


Abbildung 11: KJUnit – Entwickler



Unterhalb der Tabelle mittig werden die Testmethoden der geöffneten KJUnit Testklasse in einer Combobox aufgelistet. Hier können Sie eine andere Testmethode auswählen. Über den Button *Testkonfiguration hinzufügen* ist es dann möglich, zu der ausgesuchten Testmethode eine neue Testkonfiguration zu erstellen. Über den Button *Testkonfiguration löschen* wird die in der Tabelle markierte Testkonfiguration aus der Datenbank entfernt. Wenn KJUnit Testklassen bzw. Testmethoden zu den jeweiligen Testkonfigurationen durch den Runner bereits erzeugt wurden, siehe unten, werden diese ebenfalls gelöscht. Im unteren Bereich gibt es weiterhin noch den Tabreiter *Logger*, den Tabreiter *Runner* und den Tabreiter *Coverage*. Erläuterungen zu diesen Tabreitern finden Sie in den folgenden Unterkapiteln dieser Anwendungsdokumentation. Zudem gibt es im unteren Bereich rechts eine Tabelle mit den IDs derjenigen Testkonfigurationen, die durch den Wissensträger gelöscht wurden und zu denen jedoch KJUnit Testklassen beziehungsweise Testmethoden durch den Runner, siehe unten, bereits generiert worden sind (inaktive Testkonfigurationen). Die inaktiven Testkonfigurationen können hier gelöscht werden.

### 5.3 Erstellen und Laden der Testressourcen

Sie können eine JUnit Testklasse oder KJUnit Testklasse inklusive zugehöriger zu testender Klasse erstellen, deren Daten Sie aus einem Dokument, beispielsweise aus einem Backlog eines Projektmanagement-Tools einlesen. Beim Klick auf *Tool aussuchen* erhalten Sie die folgende Maske zur Auswahl des Tools und zur Angabe des Projektnamens bzw. der Boardnummer.

The screenshot shows a window titled "Projektmanagement-Tool aussuchen". Inside, there are six radio button options, each followed by a text input field:

- ☒ ScrumBO: Projektname
- ☐ ScrumBO Web: Projektname
- ☐ KanbanBO: Projektname
- ☐ BugtrackBO: Projektname
- ☐ Decision TableBO: Projektname
- ☐ Jira: Boardnummer

At the bottom of the form is a button labeled "Projektmanagementtool aussuchen".

Abbildung 12: Auswahl eine Projektmanagement-Tools

## Erstellen von JUnit Testklassen oder KJUnit Testklassen aus Jira

Erstellen Sie ein Projekt *Zinsrechner* mit einem Product Backlog und einem Sprint Backlog, wie es in der *Kurzanleitung\_Jira\_ForkJUnit* beschrieben wird. Zu den User Storys und Tasks des aktuellen Sprint Backlogs können jetzt JUnit Testklassen oder KJUnit Testklassen und jeweils die zugehörigen zu testenden Klassen erstellt werden. Folgende Konventionen werden dabei zugrunde gelegt.

- *epic*: Aus einem epic wird ein package.
- *User Story*: In der User Story wird ein Begriff, in der *Kurzanleitung\_Jira\_ForkJUnit* der Begriff *Kunde*, mit `>...<` notiert, also `>Kunde<`. Dieser Begriff wird zum Namen der KJUnit Testklasse und der zu testenden Klasse, in dem vorliegenden Beispiel *KundeKJUnitTest* und *Kunde*.
- *subtask*: Aus einem subtask wird eine Testmethode innerhalb der KJUnit Testklasse und eine zu testende Methode innerhalb der zu testenden Klasse, in dem vorliegenden Beispiel *setKundeZumKonto* bzw. *testSetKundeZumKonto*. Aus der Beschreibung zum subtask gehen die Parameter, welche zur Testmethode gehören, hervor. Falls Sie eine JUnit Testklasse im source folder *test* anstatt einer KJUnit Testklasse im source folder *testForKJUnit* erzeugen möchten, so markieren Sie den Namen des subtasks mit dem Suffix `>entw<`. Dieser dient nur der genannten Unterscheidung zwischen KJUnit Testklasse und JUnit Testklasse. Für den Namen der Methode bzw. Testmethode bleibt er unberücksichtigt. Geben Sie in diesem Fall keine Parameter an.

Die Verbindungsdaten zu Jira legen Sie in der Datei *PreferencesJira.xml* im package *prlab.kbunit* fest. Dabei entnehmen Sie *URL\_JIRA\_Account* und die Boardnummer, siehe oben, aus dem Kapitel 6.2 der Kurzanleitung, *xxx@hs-bochum.de* aus dem Kapitel 1 und *yyy* aus dem Kapitel 6.1 der Kurzanleitung.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <host>URL_JIRA_Account</host>
05         <username>xxx@hs-bochum.de</username>
06         <api>yyy</api>
07     </rest>
08 </root>
```

*Listing 8: Inhalt der Datei PreferencesJira.xml*

Innerhalb von KJUnit – Entwickler klicken auf *Tasks laden*. Es werden alle Tasks der User Stories des aktiven Sprints geladen und in der Tabelle angezeigt.

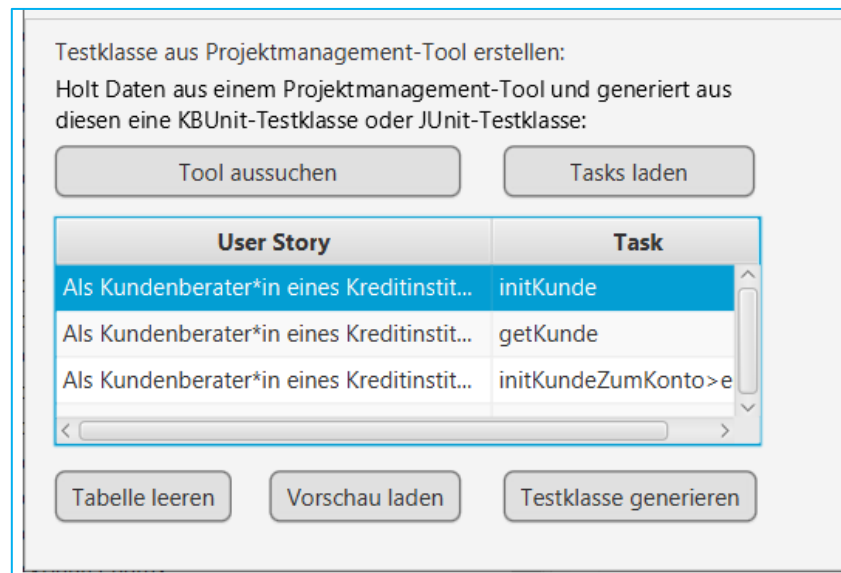


Abbildung 13: Sprint aus Jira laden

Die Tabelle kann mittels Klick auf *Tabelle leeren* wieder geleert werden. Mittels Klick auf *Vorschau laden* wird ein Fenster geöffnet, welches die Daten des in der Tabelle ausgewählten Tasks anzeigt, siehe Abbildung 14.

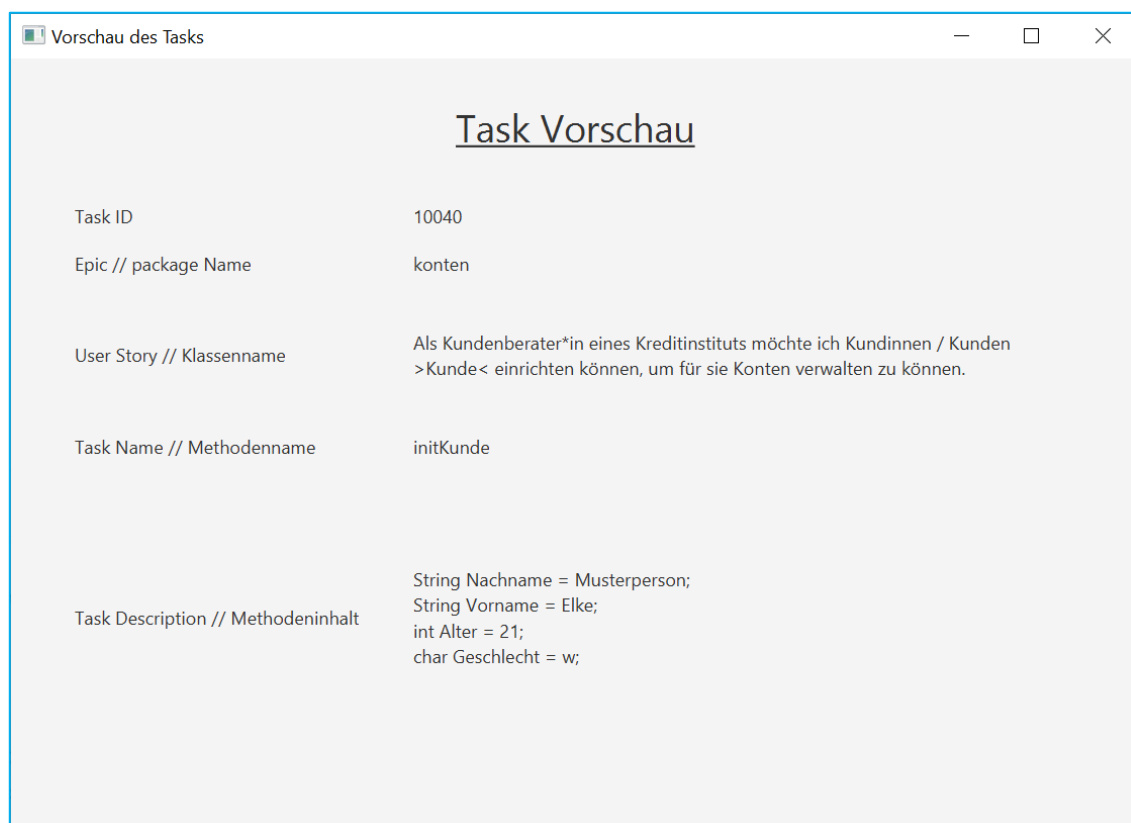


Abbildung 14: Vorschau zu einem Task

Wenn Sie ein Task in der Tabelle auswählen, können Sie mittels Klick auf *Testklasse generieren* eine KJUnit Testklasse inklusive Testmethode zu dem Task erzeugen und eine zugehörige zu testende Klasse inklusive zu testender Methode.

```

01 package konten;
02
03 class KundeKJUnitTest {
04
05     public static String testInitKunde_Nachname = "Musterperson";
06     public static String testInitKunde_Vorname = "Elke";
07     public static int testInitKunde_Alter = 21;
08     public static char testInitKunde_Geschlecht = 'w';
09
10     private Kunde kunde;
11
12     @Test
13     void testInitKunde(){
14         this.kunde = new Kunde();
15         this.kunde.initKunde(
16             testInitKunde_Nachname,
17             testInitKunde_Vorname,
18             testInitKunde_Alter,
19             testInitKunde_Geschlecht);
20         fail("Not yet implemented");
21     }
22 }

```

*Listing 9: Klasse KundeKJUnitTest*

```

01 package konten;
02
03 public class Kunde{
04
05     public void initKunde(
06         String nachname, String vorname, int alter, char geschlecht){
07         // Not yet implemented
08     }
09 }

```

*Listing 10: Klasse Kunde*

Die KJUnit Testklassen werden aber noch nicht erfolgreich durchlaufen. Hier sind Ergänzungen vom Entwickler notwendig.

Falls entsprechende Klassen bereits vorhanden sind, können Sie diese um die jeweiligen Klassenvariablen und Methoden zum ausgewählten Task ergänzen.

Enthält der Task Name die Markierung *>entw<*, so wird anstatt einer KJUnit Testklasse eine JUnit Testklasse generiert, welche dann im source folder *test* zu finden ist.

## Erstellen von JUnit Testklassen oder KJUnit Testklassen aus ScrumBO

Starten Sie den ScrumBO\_Service und ScrumBO\_Client und erstellen Sie ein Projekt *Zinsrechner* mit einem Product Backlog und einem Sprint Backlog, wie es in der *Kurzanleitung\_ScrumBO\_Desktop\_ForKJUnit* beschrieben wird. Zu den User Storys und Tasks des aktuellen Sprint Backlogs können jetzt JUnit Testklassen oder KJUnit Testklassen und jeweils die zugehörigen zu testenden Klassen erstellt werden. Folgende Konventionen werden dabei zugrunde gelegt.

- *Thema*: Aus einem Thema wird ein package.
- *User Story*: In der User Story wird ein Begriff, in der *Kurzanleitung\_ScrumBO\_For-KJUnit* der Begriff *Kunde*, mit >...< notiert, also >Kunde<. Dieser Begriff wird zum Namen der KJUnit Testklasse und der zu testenden Klasse, in dem vorliegenden Beispiel *KundeKJUnitTest* und *Kunde*.
- *task*: Aus einem task wird eine Testmethode innerhalb der KJUnit Testklasse und eine zu testende Methode innerhalb der zu testenden Klasse. Aus der Beschreibung zum task gehen die Parameter, welche zur Testmethode gehören, hervor. Weitere Erläuterungen entnehmen Sie aus dem Abschnitt *KJUnit Testklassen aus Jira erstellen*.

Die Verbindungsdaten zu ScrumBO legen Sie in der Datei *PreferencesScrumBo.xml* im package *prlab.kbunit* fest. Entnehmen Sie die Verbindungsdaten, die URL des RESTful Webservices von ScrumBO xxx aus der Kurzanleitung.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <host>xxx</host>
05         <activeproject>projectService/readProject/</activeproject>
06     </rest>
07 </root>
```

*Listing 11: Inhalt der Datei PreferencesScrumBo.xml*

Innerhalb von KJUnit – Entwickler wählen Sie in der Maske zur Auswahl des Projektmanagement-Tools *ScrumBO* aus und geben Sie den Projektnamen an. Bestätigen Sie Ihre Auswahl mittels *Projektmanagemnttool aussuchen* und schließen Sie das Fenster.

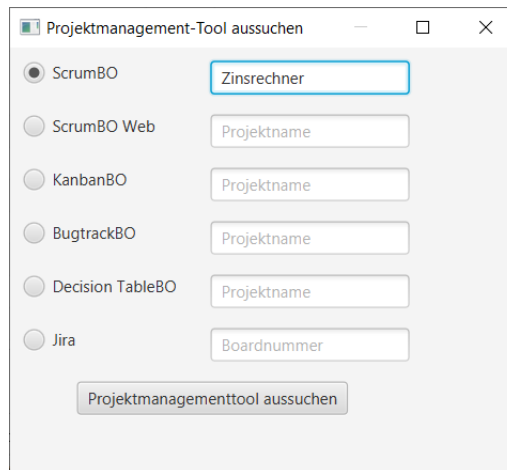


Abbildung 15: Sprint aus ScrumBO laden

Klicken Sie auf *Tasks laden*. Es werden alle Tasks der User Stories des aktiven Sprints geladen und in der Tabelle angezeigt.

Die Tabelle kann mittels Klick auf *Tabelle leeren* wieder geleert werden, siehe oben. Mittels Klick auf *Vorschau* wird ein Fenster geöffnet, welches die Daten des in der Tabelle ausgewählten Tasks anzeigt, siehe oben.

Wenn Sie ein Task in der Tabelle auswählen, können Sie mittels Klick auf *Testklasse generieren* eine KJUnit Testklasse inklusive Testmethode zu dem Task erzeugen und eine zugehörige zu testende Klasse inklusive zu testender Methode. Falls entsprechende Klassen bereits vorhanden sind, können Sie diese um die jeweiligen Klassenvariablen und Methoden zum ausgewählten Task ergänzen, siehe oben.

## Erstellen von JUnit Testklassen oder KJUnit Testklassen aus ScrumBO Web

Starten Sie den ScrumBO\_Server und erstellen Sie ein Projekt *Zinsrechner* mit einem Product Backlog und einem Sprint Backlog, wie es in der *Kurzanleitung\_ScrumBO\_Web\_ForkKJUnit* beschrieben wird. Alles weitere ist analog wie bei ScrumBO.

```

01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <host>xxx</host>
05         <hibernate>yyy</hibernate>
06         <activeproject>scrumprojekt/suche</activeproject>
07     </rest>
08 </root>

```

Listing 12: Inhalt der Datei PreferencesScrumBoWeb.xml

Legen Sie in der Datei *PreferencesScrumBoWeb.xml* im package *prlab.kbunit* die Verbindungsdaten zu ScrumBO Web fest. Entnehmen Sie die Verbindungsdaten, die

URL des RESTful Webservices von ScrumBO Web xxx und den Pfad der Konfigurationsdatei zu Hibernate yyy aus der Kurzanleitung.

## Erstellen von JUnit Testklassen oder KJUnit Testklassen aus KanbanBO

Starten Sie den KanbanBO\_Server und erstellen Sie ein Projekt *Zinsrechner* mit einem Kanban Board mit epics, User Stories und Tasks, wie es in der *Kurzanleitung\_KanbanBO\_Web\_ForKJUnit* beschrieben wird. Alles weitere ist analog wie bei ScrumBO.

Legen Sie in der Datei *PreferencesKanbanBo.xml* im package *prlab.kbunit* die folgenden Verbindungsdaten zu KanbanBO fest.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <root>
03     <rest>
04         <host>xxx</host>
07     </rest>
08 </root>
```

*Listing 13: Inhalt der Datei KanbanBo.xml*

Entnehmen Sie die Verbindungsdaten, die URL des RESTful Webservices von KanbanBO xxx, aus der Kurzanleitung.

## 5.4 Erstellen von KJUnit Testklassen aus JUnit Testklassen

Sie können eine KJUnit Testklasse weiterhin aus einer bereits vorhandenen JUnit Testklasse erstellen. Erstellen sie die folgende Klasse *KontoTest* im source folder *test* und *Konto* im source folder *src*, beide im package *konten*.

```
01 package konten;
02
03 class KontoTest {
04
05     private Konto konto;
06
07     @Test
08     void testInitKonto(){
09         this.konto = new Konto();
10         This.konto.initKonto("DE21123456780123456779", 5);
11         assertEquals(5, this.konto.getKontostand(),
12             "Der berechnete Kontostand entspricht "
13             + "nicht dem erwarteten.");
14     }
15 }
```

*Listing 14: Klasse KontoTest*

```

01 package konten;
02
03 public class Konto{
04
05     private int iban;
06     private int kontostand;
07
08     // get- und set-Methoden
09
10     public void initKonto(String iban, init kontostand){
11         if(iban == null || iban.length() != 22){
12             throw new IllegalArgumentException(
13                 "Die Laenge der Iban des Kontos ist nicht korrekt.");
14             this.iban = iban;
15             this.kontostand = kontostand;
16         }
17     }

```

Listing 15: Klasse Konto

Der JUnit Test verläuft erfolgreich. Jetzt sollen weitere Testkonfigurationen von den Wissensträgern ergänzt werden, da die Entwickler die fachlichen Hintergründe nicht genau kennen. (Beispielsweise unterscheidet sich die Länge der Iban in unterschiedlichen Staaten.) Dazu wird *KontoTest* parametrisiert.

Wählen Sie für die Parametrisierung die entsprechende JUnit Testklasse aus und klicken Sie auf den Button *Parametrisieren*.

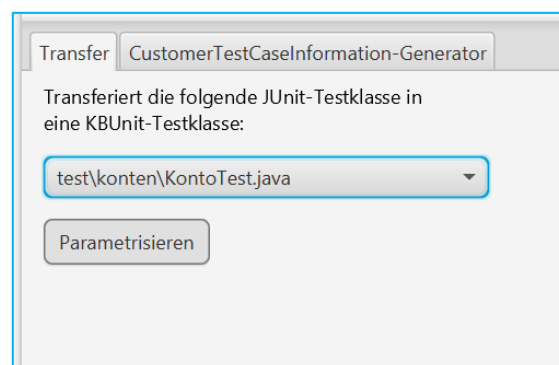


Abbildung 16: Auswahl der JUnit Testklasse zwecks Erzeugen der KJUnit Testklasse

Sie erhalten ein Fenster, ins welchem Sie diejenigen Parameter zu einer Testmethode festlegen können, welche der Wissensträger mittels KJUnit – Wissensträger verändern können soll. Haben Sie alle Parameter angegeben, so starten Sie den Transfervorgang mittels Klick auf *Transferieren*, siehe Abbildung 17.



Datentyp	Attribut	Wert	Beschreibung
String	testInitKonto_Iban	"DE2112345678...	vom Wissenstraeger einstellbare Iban des Kontos
int	testInitKonto_Kontostand	5	vom Wissenstraeger einstellbarer Kontostand des Kontos

Datentyp	Methode	Parameter	Wert
int	testInitKonto	Kontostand	5

Beschreibung  
vom Wissenstraeger einstellbarer Kontostand des Kontos

Eintrag hinzufügen    Eintrag löschen

Transferieren

Abbildung 17: Festlegung der zu ersetzenden Parameter

Sie werden dann zu allen Stellen aus der JUnit Testklasse, die für den Ersatz durch den Parameter relevant sind, gefragt, ob der Ersatz stattfinden soll. Im Anschluss wird die KJUnit Testklasse im source folder *testForKJUnit* erstellt, falls sie noch nicht vorhanden ist, ansonsten ergänzt. Geben sie als Werte von ganzzahligen Zahlen keine Zahl mit einer führenden 0 ein.

## Generieren der CustomerTestCaseInformation

Sie können die Datei *CustomerTestCaseInformation.xml* generieren, indem Sie *Starten (alle Testklassen)* anklicken, siehe Abbildung 18. Soll die Datei nur die Methoden einer bestimmten KJUnit Testklasse erhalten, wählen Sie *Starten (ausgewählte Testklasse)* aus.

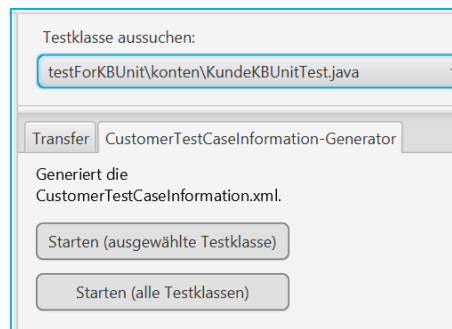


Abbildung 18: Generieren der CustomerTestCaseInformation.xml

Die Datei *CustomerTestCaseInformation.xml* wird generiert. Als Beschreibungen innerhalb der Tags `<desc> ... </desc>`, siehe Listing 4, werden die Kommentare aus der KJUnit Testklasse genommen.

## 5.5 Erstellen und Entfernen von Testkonfigurationen

Ist eine KJUnit Testklasse ausgewählt, hier *TilgungsdarlehenKJUnitTest*, so kann sie über *KJUnit Testklasse öffnen* geladen werden. Im Anschluss werden die Oberflächenelemente des Hauptfensters, siehe oben, mit den vorhandenen Testressourcen gefüllt. Nachdem die Ressourcen erfolgreich geladen wurden, erscheint die in Abbildung 19 dargestellte Programmoberfläche.

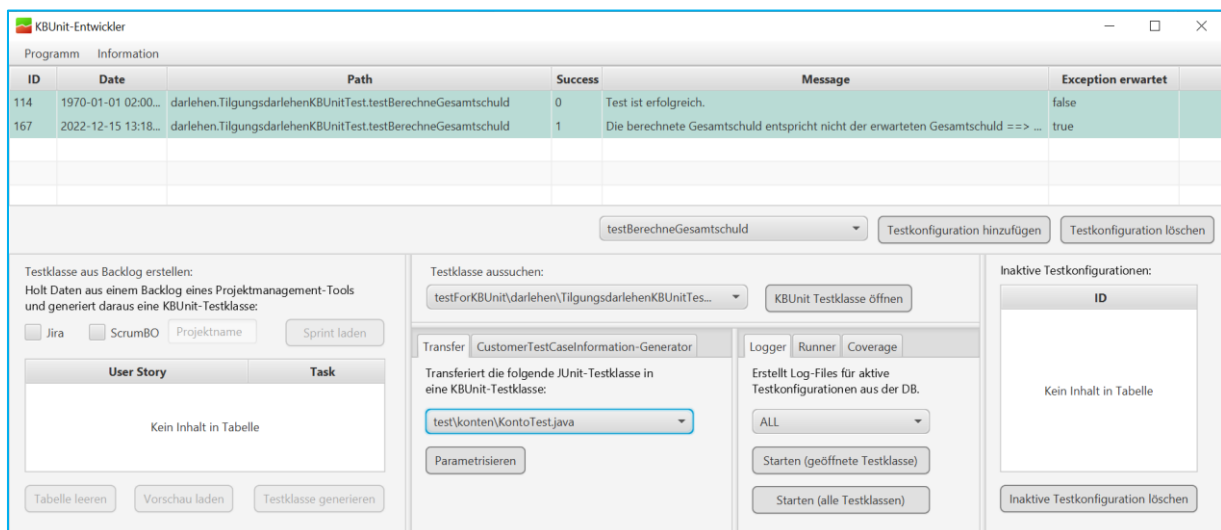


Abbildung 19: KJUnit – Entwickler mit geöffneter KJUnit Testklasse

Soll eine Testkonfiguration gelöscht werden, muss in der Tabelle diejenige Testkonfiguration, die gelöscht werden soll, markiert und über *Testkonfiguration löschen* gelöscht werden.

Um eine neue Testkonfiguration hinzuzufügen, muss erst die Testmethode aus der Combobox ausgewählt werden. Anschließend wird über den Button *Testkonfiguration hinzufügen* ein neues Fenster eingeblendet. Dieses neue Fenster beinhaltet alle zugehörigen Konfigurationsparameter der ausgewählten Methode.

Parameter	Wert
testBerechneGesamtschuld_User	Musterperson
testBerechneGesamtschuld_Darlehen	10000000
testBerechneGesamtschuld_Laufzeit	10
testBerechneGesamtschuld_Zinssatz	2.0
testBerechneGesamtschuld_exp_ErwarteteGesamtschuld	11100000

☐ Exception erwartet

Abbildung 20: Eingabemaske für eine neue Testkonfiguration

In der linken Spalte *Parameter* sind die Parameternamen aufgelistet. In der rechten Spalte *Wert* werden die Parameterwerte eingetragen. Als Platzhalter wird der vorgegebene Wert der Quell-Testklasse dargestellt, siehe Abbildung 20. Um einen Wert zu ändern, klicken Sie doppelt auf die entsprechende Zelle, schreiben den neuen Wert hinein und drücken Sie die Return-Taste. Um anzugeben, dass eine Exception erwartet wird, setzen Sie in der Checkbox zu *Exception erwartet* einen Haken. Bestätigen Sie Ihre Änderungen mit einem Klick auf den Button *Hinzufügen*. Eine neue Testkonfiguration wird dann angelegt und auch in der Tabelle angezeigt.

## 5.6 Start des Loggers und des Runners

### Start des Loggers

Der Logger ermöglicht das Abspielen der aktuell in der Datenbank vorhandenen Testkonfigurationen inklusive Auswertung auf der Entwicklerseite.

Es gibt die Möglichkeiten, den Logger nur für die geöffnete KJUnit Testklasse zu

starten und auch für alle Testklassen, siehe Abbildung 21, linkes Fenster. Des Weiteren haben Sie die Möglichkeit über ein Dropdown-Menü auszuwählen, welche Testfälle beim Generieren der Datei berücksichtigt werden sollen, siehe Abbildung 21, rechtes Fenster. Beispielsweise können Sie sich alle Testkonfiguration aussuchen, deren JUnit Test ein *Failure* ergeben haben. (Das sind diejenigen, die zwar nicht abgebrochen sind, aber deren Ergebnis nicht mit dem erwarteten Ergebnis übereinstimmt.) Es werden alle ausgesuchten Testkonfigurationen mit JUnit ausgeführt. Das Ergebnis wird in XML-Dokumente geschrieben, siehe Abbildung 22. Diese sind in den zugehörigen packages, hier beispielweise *darlehen.log*, zu finden. Sie enthalten das Datum und die Uhrzeit des Testlaufs im Namen. Initialtestkonfigurationen haben das Datum 1970-01-01. Die Information darüber, welche XML-Dokumente erzeugt worden sind, wird in der Konsole ausgegeben.

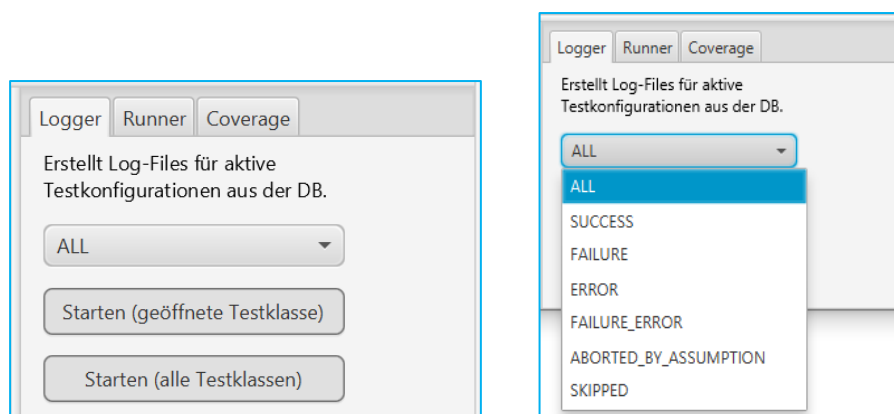


Abbildung 21: Starten des Loggers

TilgungsdarlehenTest_200415_164333.xml	
Node	Content
?? xml	version="1.0" encoding="UTF-8"
darlehen.TilgungsdarlehenTest	
created	2020-04-15 16:43:33
testBerechneGesamtschuld	
ID153_1970-01-01_02.00.01__SUCCESSFUL	
info	
message	This test is successful.
testBerechneGesamtschuld_User	(String) Musterperson
testBerechneGesamtschuld_Darlehen	(int) 10000000
testBerechneGesamtschuld_Laufzeit	(int) 10
testBerechneGesamtschuld_Zinssatz	(double) 2.0
testBerechneGesamtschuld_exp_ErwarteteGesamtschuld	(int) 11100000
ID157_2020-04-15_14.01.17__ERROR	
info	
message	Unexpected exception, expected java.lang.Exception but was null:
testBerechneGesamtschuld_User	(String) Musterperson
testBerechneGesamtschuld_Darlehen	(int) 10000000
testBerechneGesamtschuld_Laufzeit	(int) 10
testBerechneGesamtschuld_Zinssatz	(double) 0.0
testBerechneGesamtschuld_exp_ErwarteteGesamtschuld	(int) 0

Abbildung 22: Ergebnis des Loggers, eine xml-Datei mit Informationen zu den Testfällen

## Start des Runners

Der Runner ermöglicht auf der Entwicklerseite das Generieren von KJUnit Testklassen, die die neuen Testkonfigurationen aus der Datenbank enthalten. Die generierten Testklassen werden anschließend der *TestSuite* hinzugefügt. Die neuen KJUnit Testklassen kann der Entwickler daraufhin ausführen. Es gibt die Möglichkeiten, den Runner nur für die geöffnete KJUnit Testklasse zu starten und auch für alle Testklassen, siehe Abbildung 23.

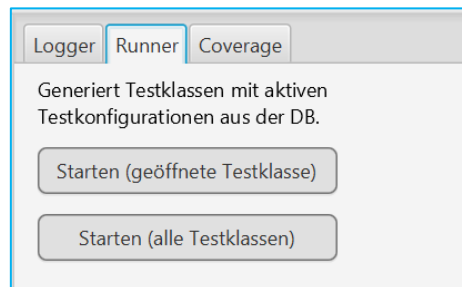


Abbildung 23: Starten des Runners

Der Runner erstellt und modifiziert zu den Testkonfigurationen, die vom Wissens-träger neu erstellt worden sind, neue KJUnit Testklassen. Diese Kopien befinden sich im source folder *testKJUnit*. Der Name der Testklasse wird um das Datum und die Uhrzeit ergänzt.

```
01 package darlehen;
02
03 public class TilgungsdarlehenKJUnitTest_200415_165237 {
04     public static String testBerechneGesamtschuld_User
05         = "Musterperson";
06     public static int testBerechneGesamtschuld_Darlehen = 10000000;
07     public static int testBerechneGesamtschuld_Laufzeit = 10;
08     public static double testBerechneGesamtschuld_Zinssatz = 2;
09     public static int
10         testBerechneGesamtschuld_exp_ErwarteteGesamtschuld = 11100000;
11
12     private Tilgungsdarlehen t;
13
14     @ParameterizedTest
15     @MethodSource("darlehen.TilgungsdarlehenKJUnitTest_200415_165237_"
16         + "Provider#testBerechneGesamtschuld_Provider")
17     void testBerechneGesamtschuld(
18         java.lang.String testBerechneGesamtschuld_User,
19         int testBerechneGesamtschuld_Darlehen,
20         int testBerechneGesamtschuld_Laufzeit,
21         int testBerechneGesamtschuld_Zinssatz,
22         int testBerechneGesamtschuld_exp_ErwarteteGesamtschuld,
23         boolean exceptionExpected) throws Exception{
```

```

24     if(exceptionExpected) {
25         assertThrows(Exception.class, () -> {
26             testBerechneGesamtschuld_Body(
27                 testBerechneGesamtschuld_User,
28                 testBerechneGesamtschuld_Darlehen,
29                 testBerechneGesamtschuld_Laufzeit,
30                 testBerechneGesamtschuld_Zinssatz,
31                 testBerechneGesamtschuld_exp_ErwarteteGesamtschuld);
32             });
33     } else {
34         testBerechneGesamtschuld_Body(
35             testBerechneGesamtschuld_User,
36             testBerechneGesamtschuld_Darlehen,
37             testBerechneGesamtschuld_Laufzeit,
38             testBerechneGesamtschuld_Zinssatz,
39             testBerechneGesamtschuld_exp_ErwarteteGesamtschuld);
40     }
41 }
42
43 private void testBerechneGesamtschuld_Body(
44     java.lang.String testBerechneGesamtschuld_User,
45     int testBerechneGesamtschuld_Darlehen,
46     int testBerechneGesamtschuld_Laufzeit,
47     int testBerechneGesamtschuld_Zinssatz,
48     int testBerechneGesamtschuld_exp_ErwarteteGesamtschuld)
49     throws Exception {
50     this t = new Tilgungsdarlehen(
51         testBerechneGesamtschuld_Darlehen,
52         testBerechneGesamtschuld_Laufzeit,
53         testBerechneGesamtschuld_Zinssatz);
54     int bGesamtschuld = t.berechneGesamtschuld(
55         testBerechneGesamtschuld_User);
56     int erwarteteGesamtschuld
57         = testBerechneGesamtschuld_exp_ErwarteteGesamtschuld;
58     assertEquals(erwarteteGesamtschuld, bGesamtschuld,
59         "Die berechnete Gesamtschuld entspricht "
60         + "nicht der erwarteten");
61 }
62 }
63
64 class TilgungsdarlehenTest_200415_165237_Provider {
65     static Stream<Arguments> testBerechneGesamtschuld_Provider() {
66         return Stream.of(
67             arguments("Musterperson", 100000000, 10, 0.0, 0, true)
68             // ID_157 2020-04-15 14:01:17
69         );
70     }
71 }
72 }

```

*Listing 16: Kopie der parametrisierten Klasse TilgungsdarlehenKJUnitTest (JUnit 5.x)*

Bei der ursprünglichen Testmethode werden die Annotation `@Test` entfernt, die Sichtbarkeit auf `private` gestellt und der Präfix `_Body` ergänzt. Weiterhin werden diejenigen Parameter, die der Wissensträger verändern kann, zu Parametern der Methode. Diese Methode wird von einer neu erstellten Testmethode mit der Annotation `@ParameterizedTest` aufgerufen. Die neue Methode verarbeitet außerdem die Information zu der erwarteten Exception. Die Parameter dieser neuen Methode kommen aus einer Providerklasse, welche die einzelnen neuen Testkonfigurationen auflistet, im Listing 16 in den Zeilen 68 und 69. Diese Abänderung wird vorgenommen, da man davon ausgehen muss, dass viele neue Testkonfigurationen zu einer Testmethode vom Wissensträger erstellt werden. Diese werden dann alle in der Providerklasse aufgelistet.

Bei JUnit 5 - Testmethoden, die eine andere Annotation als `@Test` besitzen, beispielsweise `@TestFactory` oder `@TestTemplate`, wird teilweise anders verfahren, ebenso bei JUnit 4 - Testmethoden, siehe unten.

Die Information darüber, welche Kopien erzeugt worden sind, wird in der Konsole ausgegeben. In der Datei `PreferencesKJUnitServer.xml` werden das Datum und die Uhrzeit aktualisiert. KJUnit – Entwickler erkennt, ob zu einem Testfall und zu einer Test-konfiguration bereits Kopien existieren. Dann werden keine neuen Kopien erstellt. Es werden zu den Initialtestkonfigurationen ebenfalls keine Kopien erstellt.

#### **JUnit 4 -Testmethoden und Spezialfälle von JUnit 5 - Testmethoden**

Der Runner kopiert die Testklasse und nimmt die folgenden Modifikationen vor. Von der ursprünglichen Testmethode wird die Annotation `@Test` entfernt. Zu jeder Testkonfiguration, welche der Wissensträger mittels KJUnit - Wissensträger erstellt hat und die in der Datenbank neu zu finden ist, wird eine Methode ergänzt, welche den Namen der ursprünglichen Testmethode, ergänzt durch die Identnummer der Testkonfiguration, erhält. Die Methode wird weiterhin um `throws Exception` ergänzt. In der Methode werden die Klassenvariablen mit den Werten der Testkonfiguration belegt. Dann wird die ursprüngliche Testmethode aufgerufen. Für den Fall, dass keine Exception erwartet wird, erhält die Methode die Annotation `@Test`. Für den Fall, dass eine Exception erwartet wird, erhält die Methode die Annotation `@Test (expected = Exception.class)`.

Der Runner erzeugt weiterhin zu jeder Testklasse ein XML-Dokument, welches einen Überblick über die generierten Testfälle enthält, hier `TilgungsdarlehenTest.xml`. Es werden zu jeder ursprünglichen Testmethode die zugehörigen Testkonfigurationen aufgelistet. Solange sie in der Datenbank vorhanden sind, sind sie *active*. Hat der Wissensträger mittels KJUnit - Wissensträger eine Testkonfiguration gelöscht, so stellt der Runner fest, dass die Testkonfiguration nicht mehr vorhanden ist. Sie wird dann *inactive*. Der Entwickler hat die Möglichkeit, über KJUnit – Entwickler inaktive Testfkonfigurationen samt KJUnit-Files zu löschen, siehe Abbildung 11, rechts unten. Ein Entwickler hat also mit Hilfe des Runners die Möglichkeit, sämtliche Test-

konfigurationen, die der Wissensträger mit KJUnit erzeugt, als KJUnit Testmethoden innerhalb seines Entwicklungsprojekts anzulegen. Er kann sie dann auch als solche ausführen.

Damit ein Entwickler die JUnit Tests sämtlicher generierter Testklassen durch einen einzigen Befehl ausführen kann, wird durch den Runner das package *allTestsKJUnit* erzeugt und in diesem package eine Testsuite *RunAllTests*. (Für die Testsuite in JUnit 5 ist die alte Bibliothek erforderlich: `junit.platform.runner_1.3.1.`) Die TestSuite muss allerdings mit dem JUnit4-Runner laufen.

```
01  @RunWith(JUnitPlatform.class)
02  @IncludeClassNamePatterns(".*")
03  @SelectClasses({
04      darleben.TilgungsdarlehenKJUnitTest_200415_165237.class
05  })
06  public class RunAllTests {}
```

*Listing 17: Inhalt der Testsuite RunAllTests*

Werden weitere Tests mittels KJUnit – Wissensträger/Entwickler erzeugt und startet ein Entwickler den Runner, so wird die Testklasse, die durch den Runner erstellt wurde, mit dem aktuellen Datum und der aktuellen Uhrzeit erzeugt. Die Testsuite wird um die neue Testklasse ergänzt.

Der Entwickler korrigiert aufgrund der neuen Testkonfigurationen sein Programm. Dann muss er im Anschluss das Java-Archiv *Tilgungsdarlehen.jar*, siehe Kapitel 3, aktualisieren und dem Wissensträger zukommen lassen. Dieser testet die Korrektur mittels KJUnit – Wissensträger, indem er die Testkonfigurationen erneut ausführt. Diese werden in der Datenbank aktualisiert.

## 5.7 Erstellen von Daten zum Test Coverage

Das Covarage ermöglicht auf der Entwicklerseite das Erstellen von Daten zum Test Covarage zu einer zu testenden Klasse. Diese geben Auskunft darüber, inwiefern die vorhandenen Tests die Zeilen und Inhalte der zu testenden Klasse abdecken. Es werden zu einer zu testenden Klasse die zugehörigen JUnit Testklassen, KJUnit Testklassen, und auch die KJUnit Testklassen, die durch den Runner erzeugt wurden, berücksichtigt. Sie können die zu testende Klasse mittels einer Combobox aussuchen und dann den Button *Messen* betätigen, siehe Abbildung 24.



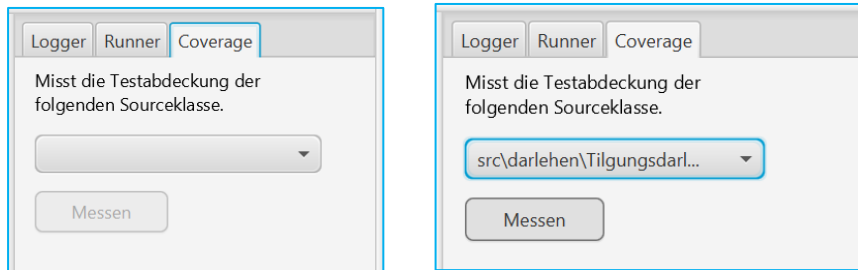


Abbildung 24: Messen der Test Coverage

Sie erhalten den folgenden Hinweis, wenn die JUnit Testklasse beziehungsweise die KJUnit Testklasse keine *initTestobjekt-Methode* enthält.

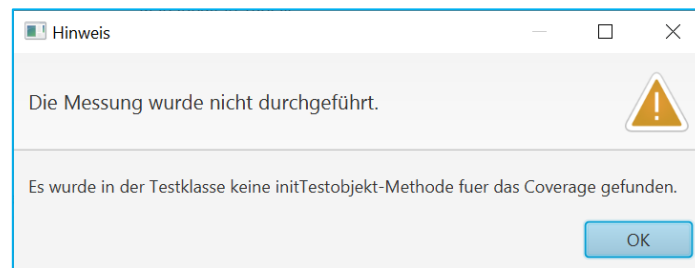


Abbildung 25: Hinweis bei fehlender *initTestobjekt-Methode*

Ergänzen Sie in der KJUnit Testklasse demnach die folgende Methode. Ist keine KJUnit Testklasse vorhanden, aber eine JUnit Testklasse, so nehmen Sie die Ergänzung in der JUnit Testklasse vor.

```

01 // das zu testende Tilgungsdarlehen-Objekt
02 private Tilgungsdarlehen t;
03
04 // Anfang: initTestobjekt for Coverage
05 Tilgungsdarlehen initTestobjekt(){
06     this.t = new Tilgungsdarlehen(10000000, 10, 2);
07     return t;
08 }
09 // Ende: initTestobjekt for Coverage

```

Listing 18: *initTestobjekt-Methode*

Falls kein Attribut der zu testenden Klasse existiert, muss eines ergänzt werden. Beim erneuten Klick auf den Button *Messen* werden Daten zum Test Coverage zu der zu testenden Klasse erstellt und bei Bestätigung der Nachfragen in die Datenbank geschrieben. Diese sind die Grundlage für Schritt 6 der Abbildung 1.

Beachten Sie:

KBUnit Entwickler sucht zuerst in der KBUnit Testklasse nach einer *initTestobjekt*-Methode, dann in der JUnit Testklasse. Die *initTestobjekt*-Methode muss einen Konstruktoraufruf verwenden, der auch in einer der Testmethoden vorkommt., damit das Coverage-Ergebnis nicht verfälscht wird. Ob Sie im Falle der Ergänzung der KBUnit Testklasse beim Aufruf des Konstruktors mit Parametern feste Werte oder Variablen für Parameter wählen, siehe Zeile 6 des Listings 18, ist unerheblich.

Das zu testende Objekt darf nicht bei der Deklaration bereits initialisiert werden.

In der Datenbank erhalten in der Tabelle *linecoverage* Zeilen einer Sourceklasse als Coverage-Ergebnis den Wert *?yellow*, falls nicht bekannt ist, ob die KBUnit Testklassen und die JUnit Testklassen zusammen als Coverage-Ergebnis den Wert *yellow* oder *green* ergeben.

In der Datenbank erhält in der Tabelle *classcoverage* der Name einer Sourceklasse den suffix *>entw<*, falls beim Coverage nur JUnit Testklassen berücksichtigt wurden. Sie erhalten den suffix *>+entw<*, falls KBUnit Testklassen und JUnit Testklassen berücksichtigt wurden. Ansonsten erhalten sie keinen suffix.

Beachten Sie auch die für das Coverage innerhalb von KBUnit geltenden Konventionen, siehe Anhang.

## Anhang: Erlaubte Eingaben für Parameterwerte

Die Eingabe von Parameterwerten unterliegt bestimmten Vorschriften, die im Folgenden näher vorgestellt werden. Dabei wird zwischen den Reitern *Neue Konfiguration* und *Konfigurationsgenerator* unterschieden.

### A1 Reiter Neue Konfiguration

Jeder Parameter erfordert die Eingabe von Werten eines bestimmten Datentyps. Zum gegenwärtigen Zeitpunkt unterstützt KBUit die Eingabe primitiver Java-Datentypen sowie Zeichenketten. Je nachdem ob es sich um einen Eingangsparameter (einen Parameter, dessen Wert während des Tests an das zu testende Modul weitergegeben wird) oder einen Ergebnisparameter (einen Parameter, der nur zur Prüfung des vom zu testenden Modul zurückgelieferten Ergebnisses genutzt wird) handelt, können in den unter dem Reiter *Neue Konfiguration* dargestellten Eingabefeldern je Parameter ein oder mehrere durch Semikolon getrennte Werte eingegeben werden. Während für Ergebnisparameter nur ein Wert zulässig ist, akzeptieren Eingabefelder für Eingangsparameter mehrere Werte. Der Einfachheit halber beziehen sich die folgenden Ausführungen auf Eingangsparameter.

An dieser Stelle sei darauf hingewiesen, dass eine nicht konforme Eingabe durch das betroffene Eingabefeld sofort mittels Warnsymbol quittiert wird. Nur wenn keines der Eingabefelder ein solches Symbol zeigt, können Sie den Test mit den von Ihnen eingegebenen Werten ausführen.

#### A1.1 Eingabe von numerischen Werten

Die Anwendung unterstützt die Eingabe von ganzen Zahlen sowie Gleitkommawerten (Java-Datentypen *byte*, *short*, *long*, *int*, *float* und *double*). Beispielsweise erwartet das im Beispiel gezeigte Parameterfeld *Laufzeit* die Eingabe einer ganzzahligen Kreditlaufzeit. Sie können für den Parameter einen einzelnen Wert angeben oder aber mehrere, durch Semikolon getrennte Werte eintragen. Bei der Eingabe mehrerer Werte werden automatisch mehrere Testkonfigurationen erstellt und ausgeführt.

Beispieleingaben: Laufzeit: 12 → es wird eine Konfiguration angelegt, in welcher der Parameter *Laufzeit* mit dem Wert 12 belegt ist

Laufzeit: 12;17 → es werden zwei Konfigurationen angelegt, wobei der Parameter *Laufzeit* in der ersten Konfiguration mit dem Wert 12, in der zweiten Konfiguration mit dem Wert 17 belegt ist

Bitte beachten Sie, dass ein Eingabefeld für numerische Parameter in keinem Fall frei gelassen werden darf.

### A1.2 Eingabe von Wahrheitswerten

Sie können einen oder zwei Wahrheitswerte eingeben. Der Zustand Wahr oder Richtig wird dabei durch das Schlüsselwort *true*, der Zustand Unwahr oder Falsch durch das Schlüsselwort *false* codiert (Java-Datentyp *boolean*). Sollten Sie beide Zustände als Parameterwerte angeben, so müssen diese durch ein Semikolon getrennt werden.

Beispieleingabe: Solvent: true → es wird eine Konfiguration angelegt, in welcher der Parameter *Solvent* mit dem Wert *true* belegt ist

Eingabefelder für Wahrheitswerte dürfen nicht freigelassen werden. Es ist immer mindestens einer der beiden Werte anzugeben.

### A1.3 Eingabe von Zeichen

Es können einzelne Zeichen (Java-Datentyp *char*) oder mehrere durch Semikolon getrennte Zeichen eingegeben werden. Das entsprechende Eingabefeld muss immer mindestens ein Zeichen enthalten. Sollten Sie das Semikolon selbst als Parameterwert angeben wollen, so tun Sie dies genauso wie bei allen anderen Zeichen.

Beispieleingabe: Zeichen: a;b;;9 → es werden vier Konfiguration angelegt, in welchen dem Parameter *Zeichen* die Werte *a*, *b*, *9* sowie das *Semikolon* zugeordnet werden

### A1.4 Eingabe von Zeichenketten

Zeichenketten stellen einen von KJUnit unterstützten Parametertyp dar, der nicht zu den primitiven Datentypen gehört (Java-Klasse *String*). Wie bei den anderen Datentypen können mehrere Zeichenketten durch Semikolon getrennt eingegeben werden. Ein leeres Eingabefeld ist erlaubt und wird vom Programm als Zeichenkette mit keinem Zeichen interpretiert. Soll das Semikolon Bestandteil einer Zeichenkette sein und nicht als Trennsymbol aufgefasst werden, so ist ihm ein Backslash voranzustellen.

Da Zeichenketten nicht primitiven Datentyps sind, können entsprechende Parameter

auch Nullreferenzen enthalten. Um einem Parameter den Wert *null* zuzuweisen, geben Sie das Schlüsselwort *null* in das Eingabefeld ein. Um dieselbe Wortgruppe innerhalb einer Zeichenkette nutzen zu können, müssen Sie wiederum ein Backslash voranstellen.

Beispieleingabe: Wörter: `hallo;null;wort\;gruppe;a\nullieren`

→ es werden vier Konfiguration angelegt, in welchen dem Parameter *Wörter* die Werte *hallo*, *null*, *wort;gruppe* und *anullieren* zugeordnet werden, wobei *null* nicht als Zeichenkette sondern als Nullreferenz interpretiert wird

## A1.5 Eingabe von Aufzählungstypen

Aufzählungstypen oder auch Enumerations stellen einen von KUnit unterstützten Parametertyp dar, der nicht zu den primitiven Datentypen gehört (*Enum*). Wie bei den anderen Datentypen können mehrere Aufzählungswerte eines Aufzählungstyps durch Semikolon getrennt eingegeben werden. Ein leeres Eingabefeld ist nicht erlaubt. Da Zeichenketten nicht primitiven Datentyps sind, können entsprechende Parameter auch Nullreferenzen enthalten. Diese werden bisher aber nicht von KUnit unterstützt.

## A2 Reiter Konfigurationsgenerator

Im Konfigurationsgenerator werden nur Eingabefelder für Eingangsparameter angezeigt. Folglich können hier mehrere Werte angegeben werden. Im Vergleich zum Reiter *Neue Konfiguration* ergeben sich dennoch einige Unterschiede. Wie in Kapitel 4 erläutert wurde, ermittelt der Konfigurationsgenerator im Sinne einer hohen Testabdeckung optimale Parameterwertkombinationen. Aus den Nutzereingaben werden gültige und ungültige Parameterwerte sowie Grenzwerte extrahiert. Um (im Gegensatz zum Reiter *Neue Konfiguration*) nicht zu viele Konfigurationen durch Realisierung aller möglichen Parameterwertkombinationen zu erhalten, ermittelt KUnit eine minimale Anzahl von Konfigurationen nach folgendem Schema:

- alle gültigen Werte eines Parameters werden mit allen gültigen Werten der anderen Parameter kombiniert
- alle ungültigen Werte eines Parameters werden jeweils mit einem zufällig gewählten gültigen Wert der anderen Parameter kombiniert
- alle Grenzwerte eines Parameters werden jeweils mit einem zufällig gewählten gültigen Wert der anderen Parameter kombiniert

Wie in Kapitel 4 erläutert, werden dem Nutzer die ermittelten Kombinationen anschließend angezeigt. Dieser hat dann die Möglichkeit, die Werte der Ergebnisparameter für jede einzelne Kombination festzulegen oder anzugeben, für welche der

ermittelten Konfigurationen mit dem Wurf einer Exception gerechnet wird.

Es entsteht die Frage, aus welchen Nutzereingaben gültige und ungültige Werte sowie Grenzwerte für die Generierung von Parameterkombinationen extrahiert werden.

## A2.1 Eingabe von numerischen Werten

Numerische Werte werden nicht nur diskret definiert, sondern in Form von Intervallen angegeben. Anhand dieser Intervalle bestimmt KBUit anschließend gültige Werte (liegen innerhalb der Intervallgrenzen), ungültige Werte (liegen außerhalb der Intervallgrenzen) und Grenzwerte (repräsentiert durch die Intervallgrenzen selbst).

Beispieleingabe: `Laufzeit: [1,25]` → es werden fünf Konfigurationen angelegt, in welchen dem Parameter *Laufzeit* die Werte 1, 25 (Grenzwerte) sowie zufällig ermittelt 6 (gültiger Wert) und -31, 89 (ungültige Werte) zugeordnet werden

In Bezug auf die Eingabe von Intervallen sind einige Besonderheiten zu beachten. Hierzu gehören unter anderem:

- Es können mehrere durch Semikolon getrennte Intervalle eingegeben werden. Diese dürfen sich allerdings nicht überschneiden, sondern maximal an den Grenzen berühren. Außerdem müssen die Intervalle aufsteigend geordnet sein.

Beispiel: Die Eingabe von `[-10,0];[-5,10]` oder `[10,100];[-10,0]` führt zu einer Fehlermeldung, während das Programm die Eingaben `[-10,0];[0,10]` sowie `[-10,0];[10,100]` akzeptiert.

- Zwischen der unteren und oberen Intervallgrenze muss immer mindestens ein gültiger Wert passen.

Beispiel: Die Eingabe von `[7,8]` führt bei ganzzahligen Datentypen zu einer Fehlermeldung, während das Programm die Eingabe `[7,9]` akzeptiert, da hier der gültige Wert 8 ermittelt werden kann.

- Es können auch Einzelwerte eingegeben werden. Diese werden vom Programm als Intervall ohne Grenzwerte, bestehend aus einem einzigen gültigen Wert, aufgefasst.

Beispiel: Aus der Eingabe von `5` ermittelt KBUit einen gültigen Wert von 5 sowie zwei zufällig bestimmte ungültige Werte, beispielsweise -12 und 17. Grenzwerte werden bei diesem unechten Intervall nicht identifiziert.

- Statt einer Zahl können auch die Werte *i* und *-i* als Intervallgrenzen angegeben werden. Diese repräsentieren den größt- beziehungsweise kleinstmöglichen Wert des verwendeten Datentyps.

Beispiel: Die Eingabe von `[100,i]` ist bei einem Parameter vom Typ *byte* gleichbedeutend mit der Eingabe `[100,127]`.

## **A2.2 Eingabe von Wahrheitswerten**

Analog zu A1.2 können die Wahrheitswerte *true* und *false* einzeln oder beide gemeinsam durch Semikolon getrennt eingegeben werden. Dabei werden die vom Nutzer explizit eingegebenen Wahrheitswerte als gültige Werte interpretiert und, sofern vorhanden, ein verbleibender, nicht angegebener Wahrheitswert automatisch ermittelt und der Gruppe ungültiger Parameterwerte zugeordnet.

## **A2.3 Eingabe von Zeichen**

Es gelten die Ausführungen aus A1.3. Jedes eingegebene Zeichen wird als gültiger Parameterwert interpretiert.

## **A2.4 Eingabe von Zeichenketten**

Es gelten die Ausführungen aus A1.4. Jede vom Nutzer eingegebene Zeichenkette oder eine explizit angegebene Nullreferenz wird als gültiger Wert aufgefasst. Hat der Nutzer nicht explizit einen Leerstring (Zeichenkette mit keinem Zeichen) sowie die Nullreferenz als Parameterwerte angegeben, so werden diese von KUnit automatisch zur Liste ungültiger Werte hinzugefügt.

## **A2.5 Eingabe von Aufzählungstypen**

Es gelten die Ausführungen aus A1.5. Jeder vom Nutzer eingegebene Aufzählungswert wird als gültiger Wert aufgefasst.

## B Anhang Konventionen und Bemerkungen

Die Einhaltung folgender Konventionen und Bemerkungen werden vorausgesetzt.

### B1 Namen von Klassen

Sei *xxx.java* der Pfad und Name der zu testenden Klasse im source folder *src*, so liegt die JUnit Testklasse im source folder *test* und heißt *xxxText.java*. Die KJUnit Testklasse für KJUnit - Wissensträger liegt im source folder *testForKJUnit* und heißt *xxxKJUnitTest.java*. Die durch den Runner erzeugte KJUnit Testklasse liegt im source folder *testKJUnit* und heißt *xxxKJUnitTest\_JJMMTT\_hhmmss.java*.

Im source folder *test* dürfen keine source folder oder packages liegen, die *testWWW* heißen.

### B2 Namen von Methoden und Parametern

Sei *yyy* der Name einer zu testenden Methode der zu testenden Klasse, so muss die Testmethode *testYyy* heißen und in der zugehörigen JUnit Testklasse liegen. Die zugehörigen Testmethoden in den KJUnit Testklassen heißen ebenfalls *testYyy*. Die Eingabeparameter zu der Testmethode in den KJUnit Testklassen haben den Aufbau *testYyy\_Parametername*, die erwarteten Ergebnisparameter *test\_exp\_Parametername*.

Zwei Testmethoden *testYyy* und *testYyyz* werden nicht korrekt unterschieden. Der Name einer Testmethode darf nicht in dem Namen einer weiteren Testmethode enthalten sein.

### B3 Allgemeine Bemerkungen zu Testmethoden

Überladene Testmethoden kann KJUnit nicht verarbeiten.

KJUnit kann Assumptions in JUnit 4 nicht verarbeiten. In JUnit 5 kann KJUnit Assumptions verarbeiten.

Folgende Zeichenketten dürfen in den Kommentaren von Testklassen nicht enthalten sein:

*@Test*, *@ParameterizedTest*, *@RepeatedTest*, *@TestTemplate*, *@TestFactory*, *void*, *DynamicNode*, *DynamicTest* und *DynamicContainer*

Testmethoden, die geerbt und dann überschrieben worden sind (zum Beispiel von einem Test-Interface), können nicht vom Wissensträger parametrisiert und ausgeführt werden.



Das Ausgrauen oder Ersetzen durch DisplayNames im KJUnit - Wissensträger-Navigationsbaum ist bisher nur für `@Disabled` und `@DisplayName` umgesetzt worden. (Ausgrauen könnte man auch bei anderen Test-Execution-Conditions (`@DisabledOnOs`, `@DisabledForJreRange`, etc.), Display-Names werden in JUnit 5 häufig erst zur Laufzeit sichtbar (zum Beispiel durch `@DisplayNameGenerators`).) Class-level `@DisplayName` ist in KJUnit nicht umgesetzt.

`@NestedTest` sind in KJUnit nicht umgesetzt.

Bei Ordered-Tests ist es in KJUnit notwendig, dass für alle Testfälle gleichviele Testkonfigurationen vorhanden sind.

Wenn in einer KJUnit Testklasse eine Testmethode enthalten ist, die nicht in der *CustomerTestCaseInfo.xml* eingetragen ist, sich aber trotzdem in der verpackten versendeten Klasse befindet, wird diese mit ausgeführt, wenn alle Methoden der Testklasse ausgeführt werden und die ausgeführte Testklasse mit `@TestMethodOrder` annotiert ist.

## B4 Formatierung von Parametern und Testmethoden

Testmethodenparameter einer KJUnit Testklasse müssen alle innerhalb einer Zeile deklariert worden sein. Die Kommentare zu den Testmethodenparametern müssen mit `/**` beginnen und mit `*/` enden. In der Folgezeile stehen dann die Deklaration und Initialisierung des Attributs.

Die Signatur einer Testmethode einer KJUnit Testklasse darf sich maximal über zwei Zeilen aufteilen.

Bei Testmethoden die nicht mit `@Test`, sondern mit `@ParameterizedTest`, `@RepeatedTest`, `@TestFactory` oder `@TestTemplate` annotiert worden sind, dürfen zur Methode zugehörige weitere Annotationen, die oberhalb der jeweiligen Testannotation platziert worden sind, sich nicht über mehr als eine Zeile ausdehnen. (Unterhalb der jeweiligen Testannotation dürfen sie sich über mehr als eine Zeile ausdehnen.)

Umlaute und ß müssen in den Kommentaren zu Testmethodenparametern und Testmethoden von JUnit Testklassen und KJUnit Testklassen in folgender Form notiert werden:

Zeichen	Ä	Ö	Ü	ä	ö	ü	ß
In Javadoc	[.A]	[.O]	[.U]	[.a]	[.o]	[.u]	[sz]

## B5 Konventionen für das Coverage

In einer Testmethode darf nur maximal eine Überprüfung mit *assertXXX* erfolgen. Die zweite wird nicht berücksichtigt.

In einer Testmethode, welche Assumptions enthält, muss in derjenigen Zeile, die *assertXXX* enthält, die Anzahl der geöffneten Klammern identisch sein mit der Anzahl der geschlossenen Klammern.

In einer Testklasse und auch in der zu testenden Klasse dürfen innerhalb von Kommentaren keine geschweiften Klammern oder runde Klammern enthalten sein.

In einer Testklasse und auch in der zu testenden Klasse darf direkt hinter dem Wort *class* kein Zeilenumbruch erfolgen.

Zugriffe auf packageweit sichtbare Methoden einer anderen Klasse ergeben eine Exception.

Falls die testende Klasse von einer Elternklasse abgeleitet ist, muss diese einen Konstruktor ohne Parameter haben.

Das Mocking mittels Mockito kann beim Coverage nicht verarbeitet werden.

Es werden nur die folgenden JUnit-Annotationen berücksichtigt:

*@Test*, *@RepeatedTest* werden uneingeschränkt unterstützt.

*@ParameterizedTest* mit *@ValueSource* wird nur für JUnit Testklassen unterstützt:

Die Argumente zu *@ValueSource* müssen innerhalb einer Zeile stehen und es müssen geschweifte Klammern für die Werte benutzt werden, auch bei 1 Wert.

Beispiel:

```
@ParameterizedTest
@ValueSource(ints = {3, 4, 5})
```

*@ParameterizedTest* mit *MethodSource* ohne innere Klasse wird nur für JUnit Testklassen und KJUnit Testklassen unterstützt, nicht für KJUnit Testklassen, welche mittels Runner generiert worden sind.

Man kann in diesem Fall KJUnit Testklassen mittels Runner generieren lassen, die neuen Testfälle in die vorhandene KJUnit Klasse übernehmen und dann die mittels Runner generierte KJUnit Testklassen löschen.

Beispiel:

```
@ParameterizedTest
@MethodSource("testWithMethodSource_Provider")
void testWithMethodSource(int argument) {
    this.p = new ParameterizingMethodSource(2);
    assertTrue((argument - p.getBias()) > 0
        && argument < 4 + p.getBias());
}
```

```
static Stream<Arguments> testWithMethodSource_Provider() {  
    return Stream.of(  
        arguments(3),  
        arguments(4),  
        arguments(5)  
    );  
}
```

*@ParameterizedTest* mit *MethodSource* mit innerer Klasse wird nur für JUnit Testklassen und mittels Runner generierte KJUnit Testklassen unterstützt.

Impressum

**Prof. Dr. Ursula Oesing**

[ursula.oesing@hs-bochum.de](mailto:ursula.oesing@hs-bochum.de)

Fachbereich Elektrotechnik und Informatik  
Hochschule Bochum  
Am Hochschulcampus 1  
44801 Bochum