Formulierung und Durchführung von Modultests mit KBUnit

Anwendungsdokumentation

Inhalt

1 Überblick 3

2 Vorstellung des Beispiels 6

3 Vorbereitung von Modultests für die Verwendung in KBUnit 7

3.1 Entwicklung eines Moduls zu dem Beispiel 7

3.2 Vorbereitung des Moduls für die Verwendung in KBUnit 9

3.2.1 Parametrisierung der Testfälle 9

3.2.2 Zusammenfassung aller Testressourcen in einem Java-Archiv 11

4 Modifizierung und Ausführung von Modultests mit KBUnit - Wissensträger 14

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

5 KBUnit - Entwickler 22

5.1 Konfiguration der Anwendung 22

5.2 Aufbau der Anwendung 24

5.3 Start der Anwendung und Laden der Testressourcen 25

5.4 Testkonfigurationen erstellen und entfernen 25

5.5 Start des Loggers 26

5.6 Start des Runners 27

Anhang: Erlaubte Eingaben für Parameterwerte 32

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 KBUnit (*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 KBUnit wurde unter anderem auf der Software Engineering 2017 und auf dem Software QS-Tag 2018 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 oft the GI, Köllen Druck + Verlag GmbH Bonn, ISBN 978-3-88579-661-9.

Die vorliegende Version der Anwendung KBUnit unterstützt die Schritte 1 bis 5 der Abbildung 1, siehe unten. Sie wurde von Philipp Sprengholz, Alexander Georgiev, Patrick Pete, Yannis Herbig 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 die Schritte 6 und 7 unterstützen, sind geplant.

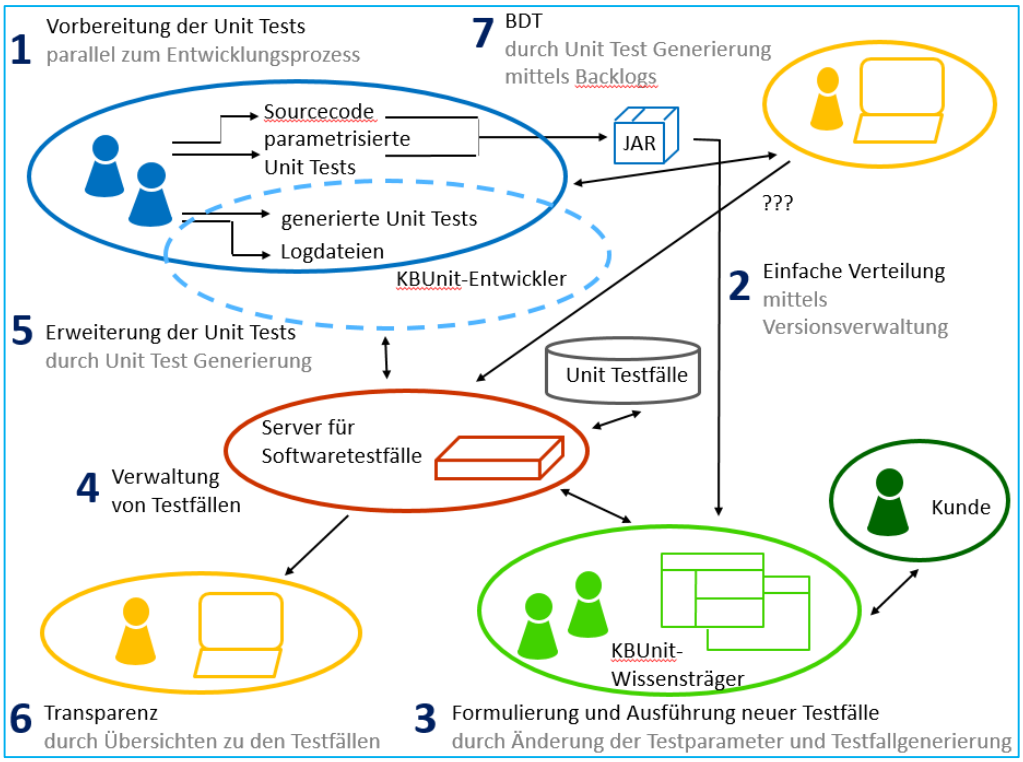

Abbildung 1: Prozess zur Durchführung von Unit Tests

Schritt 1 - Vorbereitung der Unit Tests:
Die Entwickler implementieren eine Funktionalität und zu dieser einen JUnit Test, welcher einen Testfall überprüft. Dieser JUnit Test enthält diejenigen Parameter, die von dem Wissensträger variiert werden sollen. Die Entwickler brauchen zu jeder Funktion beziehungsweise Methode nur einen Unit Test zu schreiben.

Schritt 2 - Einfache Verteilung der Unit Tests:
Der Sourcecode und die Unit Tests werden den Wissensträgern zugänglich gemacht.

Schritt 3 - Formulierung und Ausführung neuer Tests:
Die Wissensträger greifen auf die von den Entwicklern entworfenen Testfälle zu und erstellen weitere Testfälle, die aus ihrer Sicht geprüft werden müssen, indem sie die Werte der Parameter variieren. Sie können auch Testfälle generieren lassen.

Schritt 4 - Verwaltung von Testfällen:
Die Wissensträger speichern die neuen Testfälle inklusive deren Ergebnis in einer Datenbank und können diese auch verwalten, beispielsweise erneut abspielen.

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.

Schritt 7 – BDT (Behavior Driven Testing):
Es besteht zusätzlich die Möglichkeit, sich aus den Backlogs agiler Testmanagement Tools neue Unit Tests generieren zu lassen.

2 Vorstellung des Beispiels

Bevor Modultests in KBUnit 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 KBUnit

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, siehe 3.2.2.

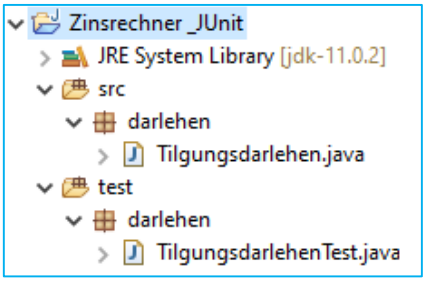

Abbildung 2: Struktur des Beispielprojekts

Es wird eine Testklasse *TilgungsdarlehenTest* in einem package *darlehen* im source folder *test* angelegt. Auf imports und auch Kommentare wird in sämtlichen Listings teilweise verzichtet. Diese müssen nach Bedarf ergänzt werden.

```
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("Die berechnete Gesamtschuld entspricht nicht"
18              + " der erwarteten",
19              erwarteteGesamtschuld, berechneteGesamtschuld);
20      }
21  }
```

Listing 1: Klasse TilgungsdarlehenTest (JUnit 5.x)

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 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.

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.

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 KBUnit zum Einsatz.

3.2 Vorbereitung für die Verwendung in KBUnit

Bevor KBUnit 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 KBUnit - Wissensträger verändern können soll. Diese müssen in Form öffentlich zugängiger Klassenvariablen codiert werden.[1]

[1]Dabei müssen sämtliche Parameter von einem Java-Grunddatentyp oder dem Datentyp String sein. Parameter anderer Datentypen können in KBUnit bisher nicht modifiziert werden.

Listing 3 zeigt die notwendigen Änderungen an der Beispielklasse *Tilgungsdarlehen-Test* für JUnit 5.x. In den Zeilen 5 bis 12 werden zunächst alle Parameter definiert, die durch den Wissensträger vor Aufruf der Testmethode modifiziert werden können 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. 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.

```
01  package darlehen;
02
03  class TilgungsdarlehenTest {
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 TilgungsdarlehenTest (parametrisiert) (JUnit 4.x)

Hinweise

Bei Betrachtung der Listings fällt auf, dass in den Testklassen keine *setUp*-Methoden oder *tearDown*-Methoden vorhanden sind. Die Initialisierung des Tilgungsdarlehens *t* wird in jeder Testmethode separat vorgenommen. Grundsätzlich ist mit KBUnit 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 KBUnit – Wissensträger die Testmethoden einzeln und in beliebiger Reihenfolge abspielen und bearbeiten kann. Testmethoden, die mittels KBUnit bearbeitet werden, sollten nicht das Auftreten von Exceptions testen, da KBUnit für diesen Sachverhalt eine eigene Lösung anbietet, siehe unten. Testsuites werden von KBUnit nicht berücksichtigt. Auch hierfür bietet KBUnit eine eigene Lösung an.

3.2.2 Zusammenfassung aller Testressourcen in einem Java-Archiv

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

```
01  <?xml version="1.0" encoding="UTF-8"?>
02  <root>
03      <date> 2020/04/15 17:41 </date>
04      <testcases>
05          <testcase>
06              <path> darlehen.TilgungsdarlehenTest.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&#228;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 *test* liegen muss. Falls ein zu testendes Programm einen anderen Aufbau hat, muss *Customer-TestCaseInformation.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
- *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 KBUnit – Wissensträger mitgeteilt, welche der im JAR enthaltenen Testfälle durch den Wissensträger modifiziert und ausgeführt werden dürfen. Weiterhin können die hinterlegten Beschreibungen in KBUnit – Wissensträger angezeigt und damit die Verständlichkeit der Testfälle für den Wissensträger erhöht werden.

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

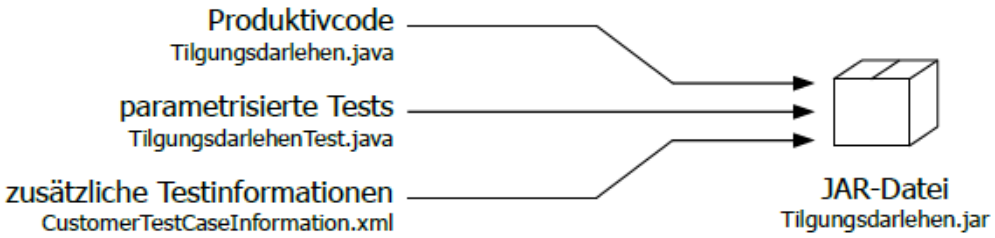

Abbildung 3: Zusammenführung der Testressourcen in einer JAR-Datei

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

4 Modifizierung und Ausführung von Modultests mit KBUnit - 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 KBUnit – 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

Der komplette Quellcode von KBUnit - Wissensträger steht zur Verfügung. Die Anlage eines Projekts wird in 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 *KBUnit_Wissensträger.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 6 dargestellt.

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

Listing 5: Inhalt der Datei config.xml

Die Datei enthält Informationen über den RESTful Webservice von KBUnit, 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. Der RESTful Webservice und der Datenbankserver müssen gestartet worden sein. Sie werden benötigt, um die vom Wissensträger ausgeführten Tests zu speichern und später erneut ablaufen

lassen zu können. Das Programm unterstützt standardmäßig eine MySQL-Datenbank, siehe Administrationsdokumentation.

Neben der Anbindung an den Webservice 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 das Java-Archiv *KBUnit_Wissenstraeger.jar*. Liegt die Anwendungsdokumentation in einem Unterverzeichnis *doc,* so muss als Pfad *doc\\Anwendungsdokumentation_KBUnit.pdf* angegeben werden. Sie können auch den absoluten Pfad angeben.
Das Programm benötigt weitere externe Bibliotheken, siehe Administrations-dokumentation, die in einem Unterverzeichnis *KBUnit_Wissenstraeger_lib* liegen müssen, siehe Administrationsdokumentation. Weiterhin benutzt das Programm externe Icons, welche im Unterverzeichnis *icons* liegen müssen.

4.2 Start der Anwendung und Laden von Testressourcen

Nach abgeschlossener Konfiguration kann *KBUnit_Wissenstraeger.jar* gestartet werden.[2] 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.

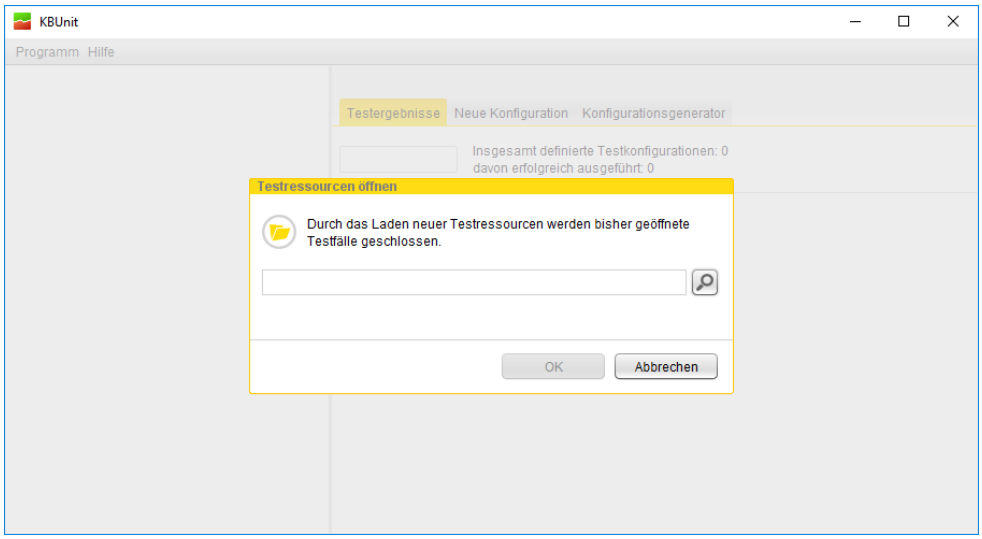

Abbildung 4: Laden von Testressourcen

[2] 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, klicken in der Baumansicht doppelt auf den Testfall.

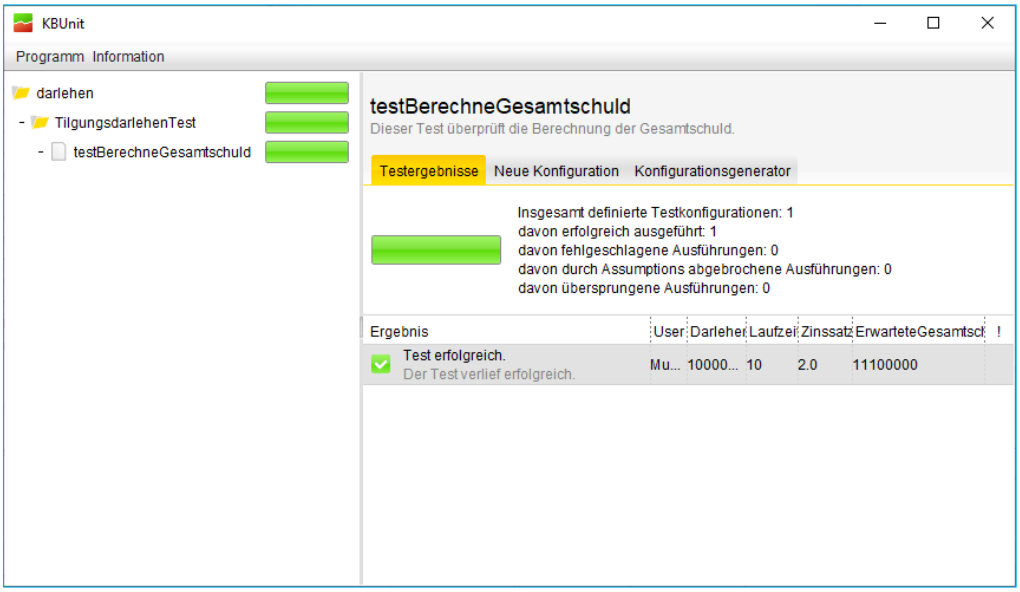

Abbildung 5: Hauptanwendungsfenster

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. Haben Sie für jeden Parameter genau einen Wert eingegeben, so erzeugt KBUnit zum Testfall *testBerechneGesamtschuld* eine Konfiguration zu dem Testfall. Haben Sie dagegen für den Zinssatz zwei durch Semikolon getrennte Werte eingegeben, so erzeugt KBUnit automatisch zwei Konfigurationen zu ein und demselben Testfall.

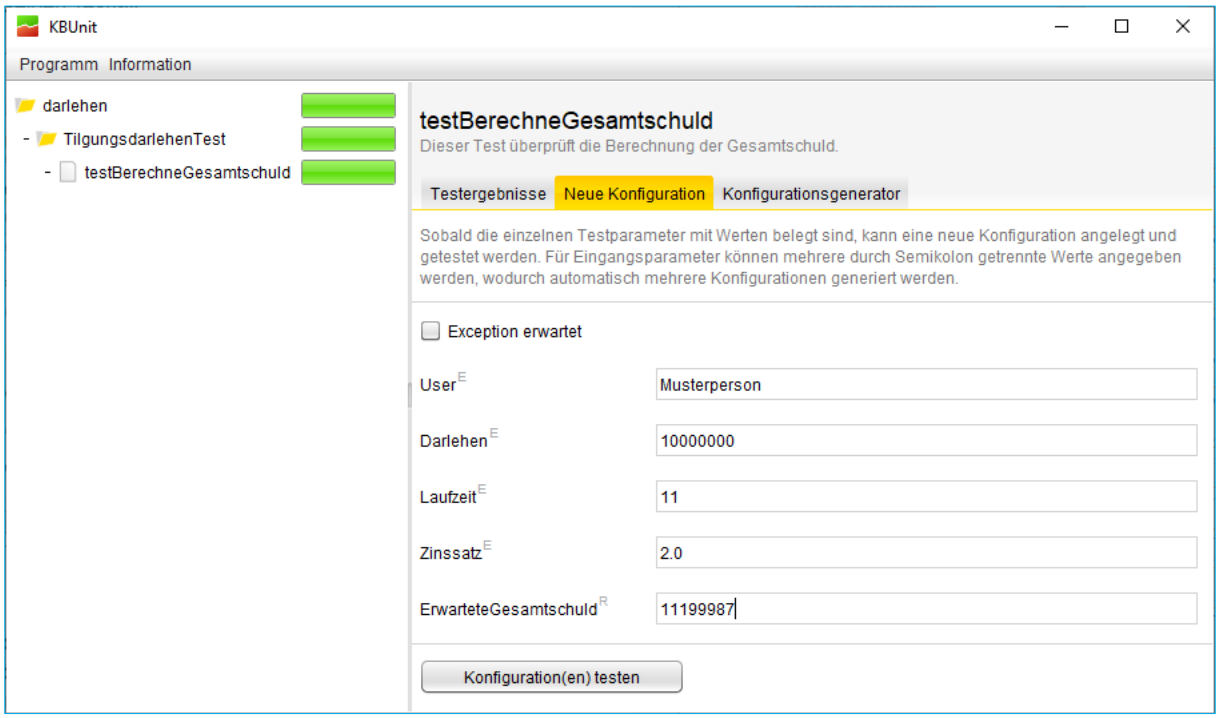

Abbildung 6: Testen einer neuen Konfiguration

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.)

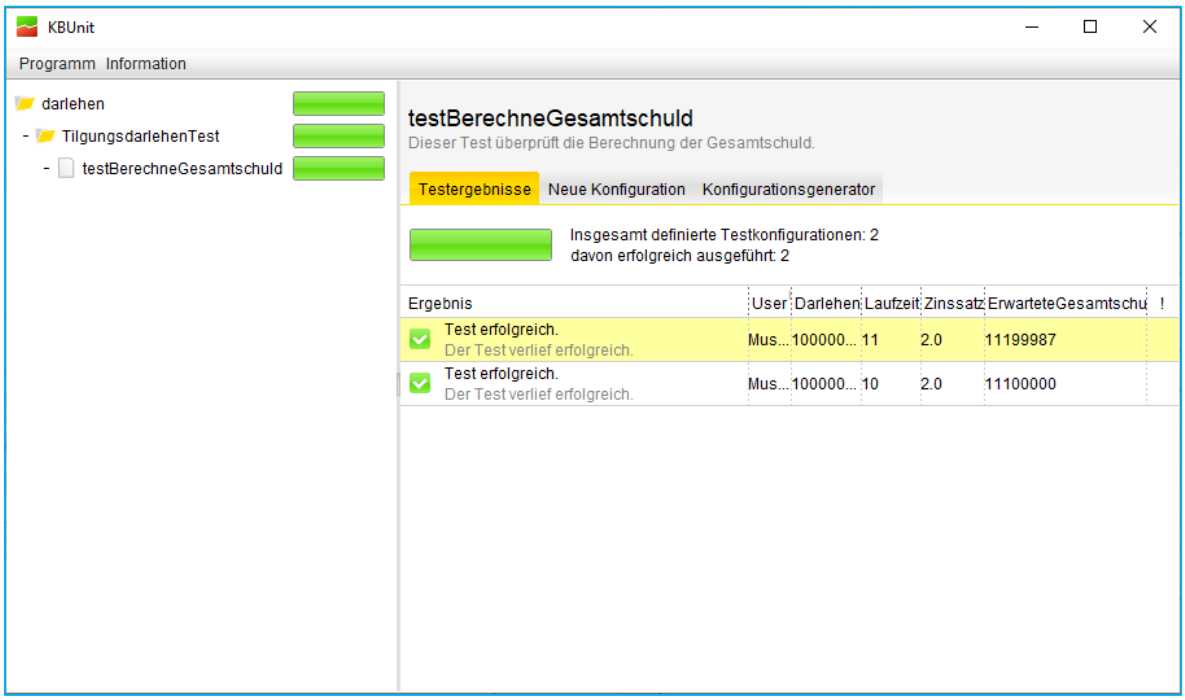

Abbildung 7: Ergebnis einer getesteten Konfiguration

Die getestete Konfiguration wird automatisch in der Datenbank abgelegt und steht 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. Ziel ist es, eine maximale Testabdeckung mit möglichst wenig Konfigurationen zu erreichen.

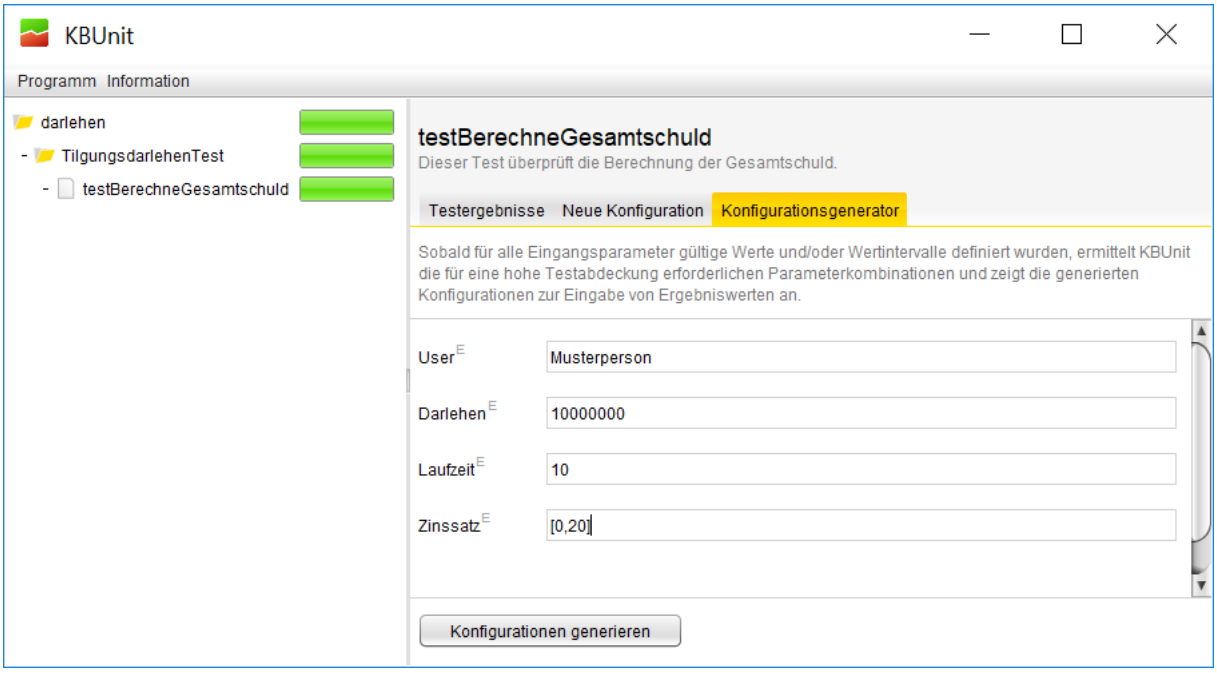

Abbildung 8: Parametereingabe im Konfigurationsgenerator

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. Genauere Ausführungen hierzu sind dem Anhang zu entnehmen. KBUnit - Wissensträger bestimmt automatisch innerhalb und außerhalb dieser Intervalle liegende Werte sowie Intervallgrenzen. Die zu einem Parameter ermittelten Werte werden anschließend mit Werten anderer Parameter kombiniert. Übernehmen Sie die in Abbildung 8 dargestellten Eingaben. Durch einen Klick auf *Konfigurationen generieren* zeigt KBUnit - Wissensträger die ermittelten optimalen Parameterkombinationen an, siehe Abbildung 9. Innerhalb der Tabelle rot angezeigte Werte liegen außerhalb eines von Ihnen angegebenen Intervalls, gelbe Werte stellen Intervallgrenzen dar und schwarze Einträge befinden sich innerhalb eines von Ihnen definierten Intervalls. Für Kombinationen, die einen Parameterwert außerhalb eines Intervalls 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 setzten, 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 erscheinender Menü anschließend *Konfiguration löschen.* Durch einen Klick auf *Abbrechen* kommen Sie zurück zur Ausgangsseite des Generators.

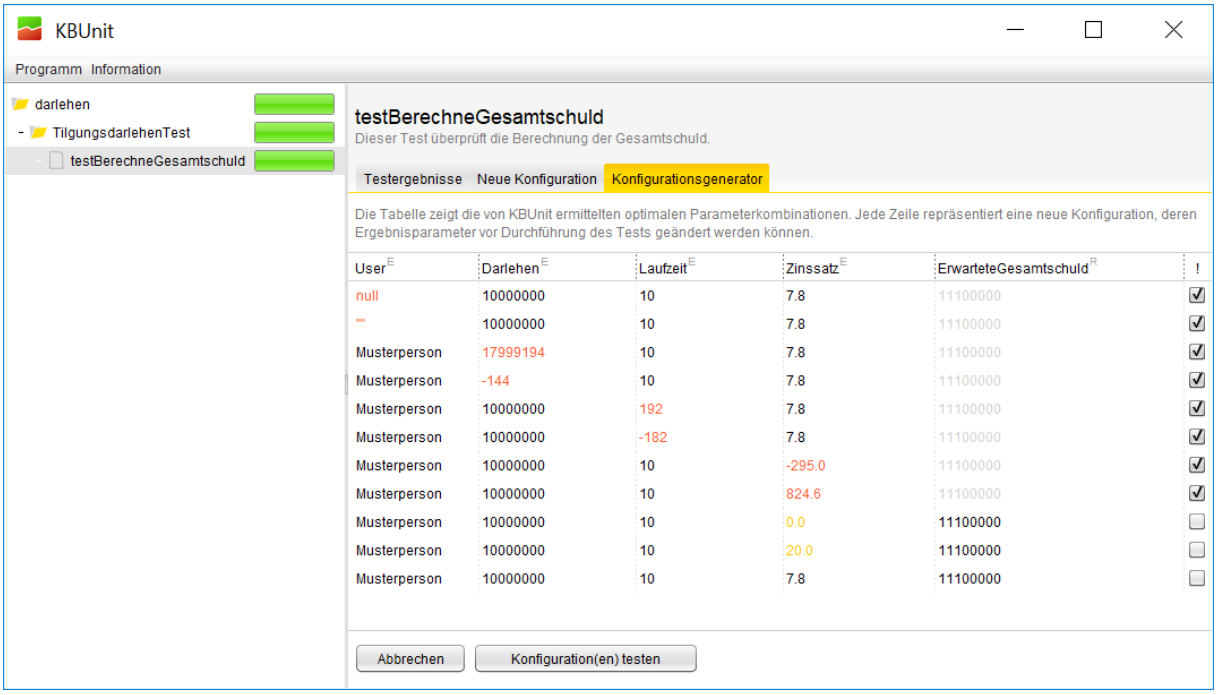

Abbildung 9: vom Konfigurationsgenerator ermittelte Parameterkombinationen

Setzen Sie für den Testfall mit Zinssatz 0 die erwartete Gesamtschuld auf 0 und einen Haken, da Sie eine Exception erwarten. Ändern Sie für die Testfälle mit Zinssatz 20 und 7.8 (beziehungsweise den generierten Zinssatz zwischen 0 und 20) die erwartete Gesamtschuld ab auf 21000000 und 142900000 (beziehungsweise die zu erwartende Gesamtschuld). 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.

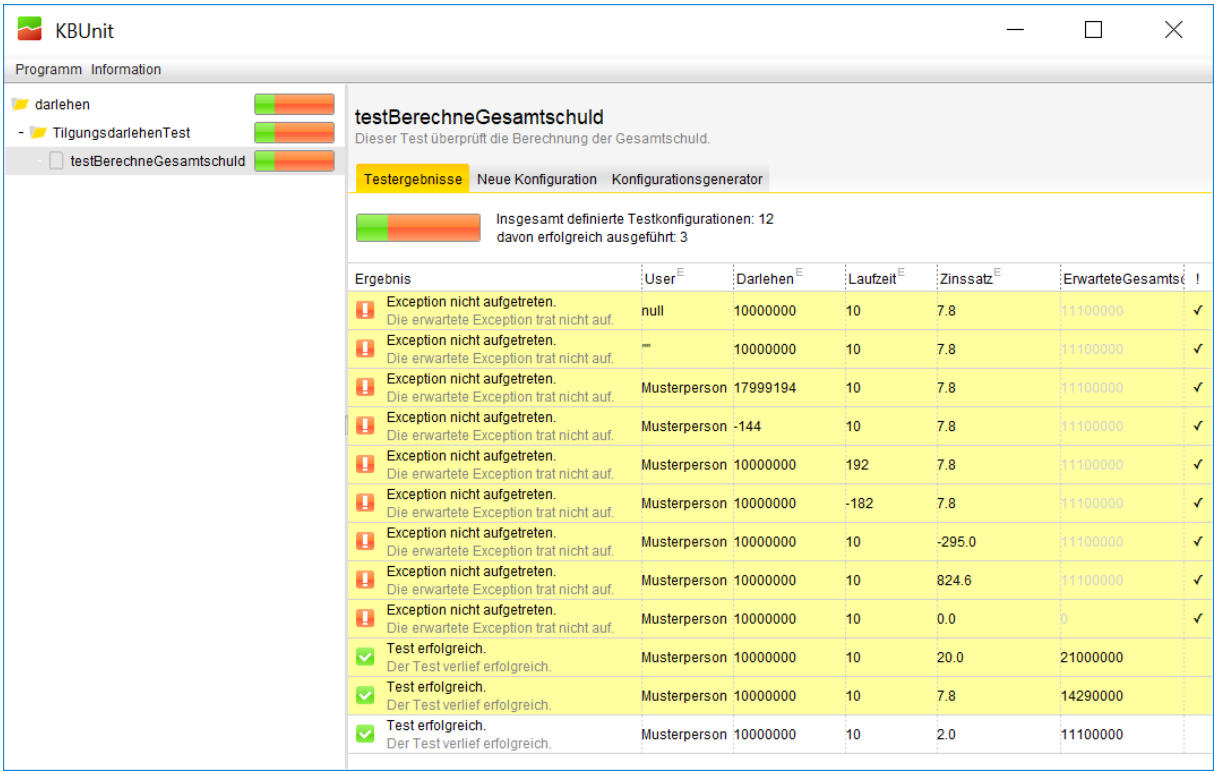

Abbildung 10: Testergebnisse

Zusammengefasst erhält der Wissensträger dank KBUnit - 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 einer negativen Laufzeit oder einer Laufzeit von null Perioden 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 Beispielmethoden 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 Zinssatz 2,) und die viertunterste Konfiguration (, die nicht erfolgreich durchgelaufene Testkonfiguration mit Zinssatz 0).

5 KBUnit – Entwickler

Die in Kapitel 5 vorgestellten Aktivitäten entsprechen den Schritten 4 und 5 der Abbildung 1 und werden durch die Entwickler durchgeführt. Die Listings von Java-Dateien müssen wie auch in Kapitel 3 um die imports und Kommentare ergänzt werden.
KBUnit – Entwickler besitzt Funktionen, um Auswertungen zu generieren und zu den vorhandenen Testfällen aus der Datenbank neue JUnit Testklassen zu erstellen.
Weiterhin erstellt KBUnit – Entwickler dem Entwickler eine Übersicht der vorhandenen Testfällen in der Datenbank sowie der generierten JUnit Testklassen. Zudem ist es möglich, neue Testfälle der Datenbank hinzuzufügen und auch vorhandene zu löschen.

5.1 Konfiguration der Anwendung

Die Konfiguration von *KBUnit – Entwickler* entnehmen Sie der Administrationsdokumentation.
Ein source folder *testKBUnit* wurde bereits erstellt-. Es enthält ein package *prlab.kbunit,* welches wiederum die Datei *PreferencesHome.xml* und die Klasse *Main* enthält. Die vom Wissensträger mittels KBUnit - Wissensträger erzeugten Testfälle befinden sich in der zu KBUnit gehörenden Datenbank. In der Datei *PreferencesHome.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 KBUnit. Das Programm unterstützt standardmäßig eine MySQL-Datenbank, siehe Administrationsdokumentation.
Weiterhin werden in der Datei *PreferencesHome.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/KBUnitServer/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 PreferencesHome.xml

In der Klasse *Main.java,* 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(){
21          try{
22              FXMLLoader loader = new FXMLLoader();
23                  "/prlab/kbunit/resources/view/MainFrameScene.fxml"));
24              Parent root = loader.load();
25              Scene scene = new Scene(root);
26              scene.getStylesheets().add(getClass().getResource(
27                  "/prlab/kbunit/resources/css/MainFrameScene.css")
28                  .toExternalForm());
29              primaryStage.setScene(scene);
30              MainFrameController controller = loader.getController();"
31              controller.setHostServices(getHostServices());
32              primarySTage.show();
33          }
34          catch(IOExcdption ioExc){
35              System.out.println("Fehler beim Laden der "
36                  + "Oberflächen-Resourcen.");
37          }
38      }
39  }
```

Listing 7: Klasse Main

5.2 Aufbau der Anwendung

Die Anwendung KBUnit-Entwickler ist wie folgt aufgebaut (siehe Abbildung 11):
Das Hauptfenster ist horizontal in zwei Bereiche unterteilt. Im unteren Bereich links kann man im Tabreiter *Testklasse* mittels eines Buttons *Öffnen* und einer zugehörigen ComboBox eine Testklasse aussuchen, hier *test\darlehen\TilgungsdarlehenTest*, siehe näheres weiter unten in der Dokumentation. Im oberen Bereich stellt eine Tabelle diejenigen Testkonfigurationen dar, die in der Datenbank zu der ausgewählten Testklasse vorhanden sind und über den RESTful Webservice zur Verfügung gestellt werden. Mittels einer ComboBox kann eine Methode der geöffneten Testklasse ausgesucht werden, hier *testBerechneGesamtschuld*. Über den Button *Testkonfiguration hinzufügen* ist es möglich, zu der ausgesuchten Methode eine neue Testkonfiguration zu erstellen. Über den Button *Testkonfiguration löschen* werden die in der Tabelle markierten Testkonfigurationen aus der Datenbank entfernt. Wenn JUnit Testklassen bzw. JUnit Testmethoden zu den jeweiligen Testkonfigurationen durch den Runner bereits erzeugt wurden, siehe unten, werden diese ebenfalls gelöscht.
Der untere Bereich ist vertikal in drei Bereiche unterteilt. Es gibt in der Registerkarte links drei Kategorien: *Testklasse*, *Logger*, *Runner*. Zudem gibt es rechts eine Tabelle mit den IDs derjenigen Testkonfigurationen, die durch den Wissensträger gelöscht wurden und zu denen jedoch JUnit Testklassen beziehungsweise JUnit Testmethoden durch den Runner, siehe unten, bereits generiert worden sind (inaktive Testkonfigurationen).

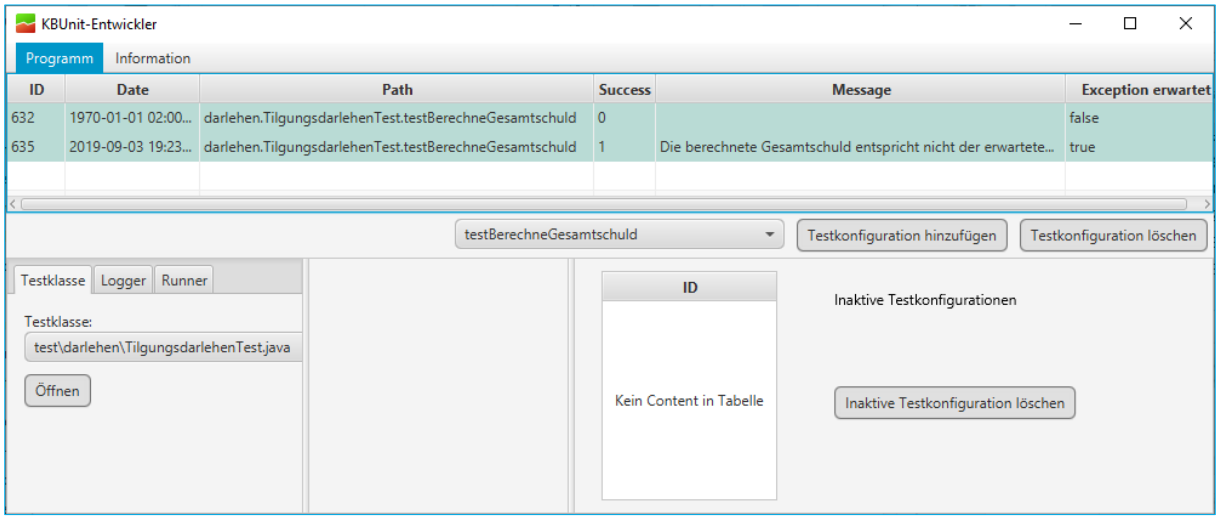

Abbildung 11: KBUnit – Entwickler

5.3 Start der Anwendung und Laden der Testressourcen

Nach erfolgreicher Konfiguration kann nun die Hauptanwendung von KBUnit - Entwickler gestartet werden. Mittels *Programm → Beenden* können sie das Programm beenden. Mittels Informationen können die Lizenzbedingungen und diese Anwenderdokumentation eingesehen werden.

Zu Beginn der Applikation kann eine Testklasse aus dem source folder *test* ausgewählt werden. Unter dem Reiter *Testklasse* werden alle verfügbaren Testklassen in einem Dropdown-Menü aufgelistet (siehe Abbildung 12), die aus dem source folder *test* mit dem Dateisuffix *.java herausgefiltert werden.

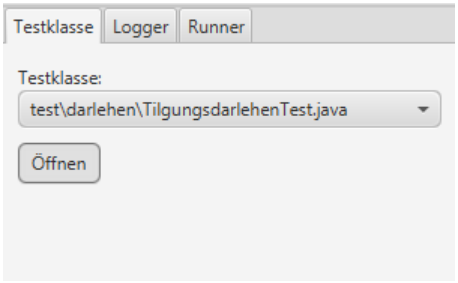

Abbildung 12: Auswahl der Testklasse

Ist die zu öffnende Testklasse ausgewählt, kann diese über den Button *Öffnen* geladen werden. Im Anschluss werden die Oberflächenelemente des Hauptfensters, siehe oben, mit den vorhandenen Testressourcen gefüllt. Nachdem die Ressourcen erfolgreich geladen wurden, sollte die in Abbildung 11 dargestellte Programm-oberfläche erscheinen.

5.4 Testkonfigurationen erstellen und entfernen

Um eine neue Testkonfiguration zu erstellen oder auch löschen zu können, muss eine aktive Testklasse geöffnet werden. Soll eine Testkonfiguration gelöscht werden, muss in der Tabelle diejenige Testkonfiguration, die gelöscht werden soll, markiert und über den Button *Testkonfiguration löschen* gelöscht werden.
Um eine neue Testkonfiguration hinzuzufügen, muss erst die zu testende Methode aus dem Dropdown-Menü ausgewählt werden. Anschließend wird über den Button *Testkonfiguration hinzufügen* ein neues Fenster eingeblendet. Dieses neue Fenster beinhaltet alle zugehörigen Konfigurationsparameter der ausgewählten Methode. 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 13.

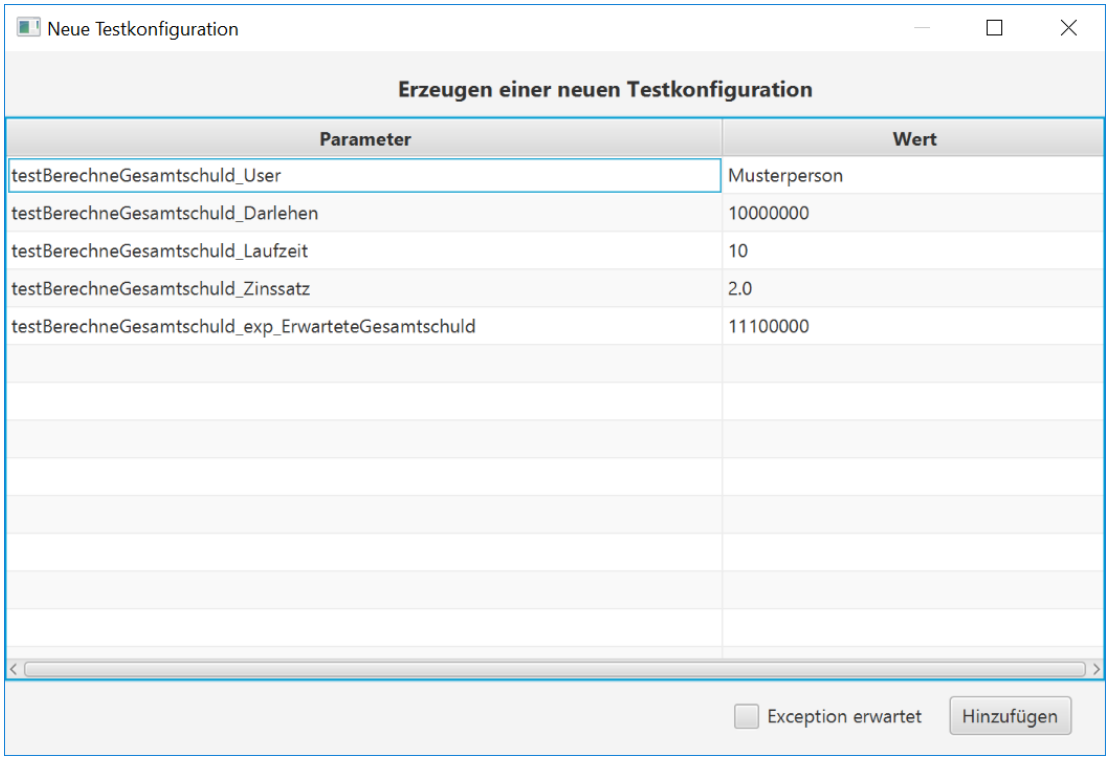

Abbildung 13: Eingabemaske für eine neue Testkonfiguration

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.

5.5 Start des Loggers

Der Logger ermöglicht das Abspielen der aktuell in der Datenbank vorhandenen Testkonfigurationen inklusive Auswertung auf der Entwicklerseite.
Es gibt die Möglichkeit, den Logger nur für die geöffnete Testklasse zu starten und auch für alle Testklassen, siehe Abbildung 14, linkes Fenster. Des Weiteren haben Sie die Möglichkeit über ein Dropdown-Menü auszuwählen, welche Testfälle beim Generieren der Datei berücksichtigt werden sollen, siehe Abbildung 14, 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 15. Diese sind in den zugehörenden packages, hier beispielweise *darlehen.log,* zu finden. Sie enthalten das Datum und die Uhrzeit des Testlaufs im Namen. Gegebenenfalls muss die Anzeige des Projekts mittels *Refresh* im Kontextmenü des Projekts aktualisiert werden, damit die neuen Dokumente angezeigt werden.

Initialtestkonfigurationen haben das Datum 1970-01-01. Die Information darüber, welche XML-Dokumente erzeugt worden sind, wird in der Konsole ausgegeben.

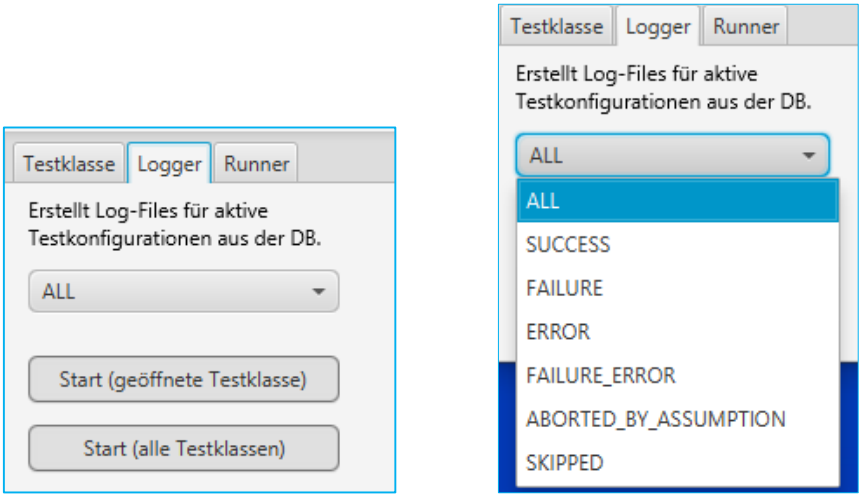

Abbildung 14: Starten des Loggers

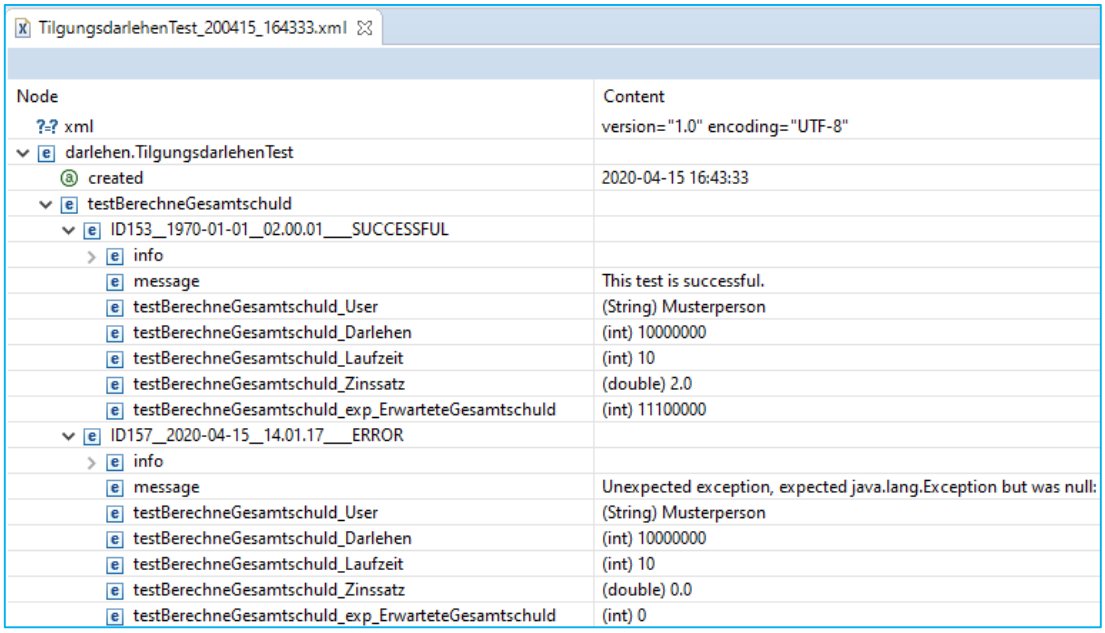

Abbildung 15: Ergebnis des Loggers, eine xml-Datei mit Informationen zu den Testfällen

5.6 Start des Runners

Der Runner ermöglicht auf der Entwicklerseite das Generieren von JUnit Testklassen, die die neuen Testkonfigurationen aus der Datenbank enthalten. Die generierten Testklassen werden anschließend der *TestSuite* hinzugefügt. Die neuen JUnit Testklassen kann der Entwickler daraufhin ausführen. Es gibt die Möglichkeit, den Runner nur für die geöffnete Testklasse zu starten und auch für alle Testklassen, siehe Abbildung 16.

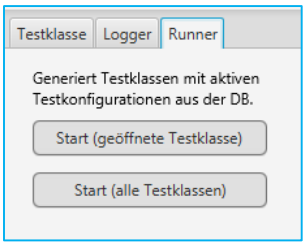

Abbildung 16: Starten des Runners

```
01  package darlehen;
02
03  public class TilgungsdarlehenTest_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.TilgungsdarlehenTest_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
66      static Stream<Arguments> testBerechneGesamtschuld_Provider() {
67          return Stream.of(
68              arguments("Musterperson", 100000000, 10, 0.0, 0, true)
69                  // ID_157 2020-04-15 14:01:17
70          );
71       }
72  }
```

Listing 8: Kopie der parametrisierten Klasse TilgungsdarlehenTest (JUnit 5.x)

Der Runner erstellt und modifiziert zu den Testkonfigurationen, die vom Wissensträger neu erstellt werden sind, neue JUnit Testklassen. Diese Kopien befinden sich im source folder *testKBUnit*. Gegebenenfalls muss mittels *Refresh* im Kontextmenü des Projekts die Anzeige aktualisiert werden. Der Name der Testklasse wird um das Datum und die Uhrzeit ergänzt. 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 8 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 *PreferencesHome.xml* werden das Datum und die Uhrzeit aktualisiert. KBUnit – Entwickler erkennt, ob zu einem Testfall und zu einer Testkonfiguration 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 KBUnit - 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 KBUnit - 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 KBUnit – Entwickler inaktive Testfälle samt JUnit-Files zu löschen, siehe Abbildung 11, rechts unten.

Ein Entwickler hat also mit Hilfe des Runners die Möglichkeit, sämtliche Testkonfigurationen, die der Wissensträger mit KBUnit erzeugt, als JUnit 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 *allTestsKBUnit* erzeugt und in diesem package eine Testsuite *RunAllTests.java*. (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       darlehen.TilgungsdarlehenTest_200415_165237.class
05   })
06   public class RunAllTests {}
```

Listing 9: Inhalt der Testsuite RunAllTests.java

Werden weitere Tests mittels KBUnit – 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 KBUnit – Wissensträger, indem er die Testkonfigurationen erneut ausführt. Diese werden in der Datenbank aktualisiert.

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 KBUnit 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 S*emikolon* zugeordnet werden

A1.4 Eingabe von Zeichenketten

Zeichenketten stellen den einzigen von KBUnit 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

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 KBUnit 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 KBUnit 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 Konfiguration angelegt, in welchen dem Parameter *Laufzeit* die Werte *1, 25 (*Grenzwerte) sowie zufällig ermittelt *6 (*gültiger Wert) und *-31, 89* (gültige 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 KBUnit 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 KBUnit automatisch zur Liste ungültiger Werte hinzugefügt.

Impressum

Prof. Dr. Ursula Oesing

ursula.oesing@hs-bochum.de

Fachbereich Elektrotechnik und Informatik
Hochschule Bochum
Lennershof 140
44801 Bochum