

## Parametrisierte Tests in JUnit

### Einleitung

Wenn man einen Testfall definiert und programmiert, hat man hinterher das Gefühl, dass man den Testfall eigentlich auch unter anderen Bedingungen, unter anderen Parametern durchführen möchte. Das JUnit-Framework hat hierfür eine Lösung: den Testrunner für parametrisierte Tests.

### Stichwörter

JUnit, Parameterized, Test, Zufall, Theory

### Parametrisierte Testfälle

#### Funktionsweise des Testrunners „Parameterized“

Die Parameter werden über eine statische Methode zur Verfügung gestellt, die mit „@Parameters“ annotiert ist. Der Testrunner nimmt die Vektoren vom Typ *Object[]* und nutzt sie als Eingabe für den Konstruktor. Die Vektoren müssen natürlich von den Typen her passend sein. Beispiel:

```
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;

@RunWith(Parameterized.class)
public class ParamTest {
    private Integer a;
    private Double b;
    public ParamTest(Integer a, Double b) {
        System.out.println("" + a + ", " + b); //Ausgabe: 3 3.4
        this.a = a;                          //          -3 -5.1
        this.b = b;
    }

    @Parameters
    public static Collection<Object[]> input() {
        ArrayList<Object[]> res = new ArrayList<Object[]>();
        res.add(new Object[] {3, 3.4});
        res.add(new Object[] {-3, -5.1});
        return res;
    }

    @Test
    public void dummy() {
```

## Parametrisierte Testfälle

```
//nutzt die Variablen a und b für Tests  
}  
}
```

### Randomisierte Testfälle

Nun möchte man nicht für jeden Testfall sich neue Testdaten ausdenken. Zufallsdaten wären ideal, jedoch bietet JUnit keine Lösung für Zufallsdaten an. Ich möchte eine Lösung vorstellen, um schnell Testdaten für parametrisierte Tests und auch für Theorien zu erhalten. Die Lösung besteht darin, dass man das Interface *Iterable* implementiert. Die Testrunner für Theorien und für parametrisierte Tests ermitteln ihre Eingabedaten, indem sie über *Iterable*-Instanzen iterieren.

Das Ziel: In der Variablen *i* in „for (T *i* : randomCollection)“ soll jedes Mal ein neues Zufallsobjekt des Typs *T* sein. Um ein einzelnes Testobjekt zu erstellen, benötigt man zuallererst dieses generische Interface, das nur noch darauf wartet, implementiert zu werden:

```
public interface RandomObjectGenerator {  
    T getRandomObject();  
}
```

Eine abstrakte Klasse, die *Iterable* implementiert, nutzt den *RandomObjectGenerator* als Strategiekategorie. Die Methode *getRandomItem* delegiert den Aufruf nur noch an diese Strategie. Als Iterator-Objekt wird ein spezieller Iterator verwendet, dessen Funktionsweise weiter unten erläutert wird. Die *hasNext*-Methode ist abstrakt, weil ich es offen lassen möchte, wie viele Testobjekte erzeugt werden sollen.

```
import java.util.ArrayList;  
import java.util.Iterator;  
import java.util.List;  
  
public abstract class RandomizedCollection<T> implements Iterable<T> {  
    private final RandomObjectGenerator<T> generator;  
    public RandomizedCollection(final RandomObjectGenerator<T> generator) {  
        this.generator = generator;  
    }  
  
    @Override  
    public Iterator<T> iterator() {  
        return new RandomIterator<T>(this);  
    }  
}
```

## Parametrisierte Tests in JUnit randomisieren

```
abstract boolean hasNext();

T getRandomItem() {
    return generator.getRandomObject();
}

}
```

Der RandomIterator delegiert die Anfragen *next* und *hasNext* an die Collection. Einer Implementierung der *remove*-Methode bedarf es nicht.

```
import java.util.Iterator;

class RandomIterator<T> implements Iterator<T> {
    private final RandomizedCollection<T> collection;
    public RandomIterator(final RandomizedCollection<T> collection) {
        this.collection = collection;
    }
    @Override
    public boolean hasNext() {
        return collection.hasNext();
    }
    @Override
    public T next() {
        return collection.getRandomItem();
    }
    @Override
    public void remove() {
    }
}
```

### Konkrete Implementierung für RandomObjectGenerator

Die Implementierung eines Zufallsgenerator für Integer sähe so aus:

```
public class RandomInteger implements RandomObjectGenerator<Integer> {
    private static final Random random = new Random();
    Integer getRandomObject() {
        return Integer.valueOf(random.nextInt());
    }
}
```

Für Parametrisierte Tests muss `RandomObjectGenerator<Object[]>` implementiert werden:

```
public class RandomTuple implements RandomObjectGenerator<Object[]> {
    private static final Random random = new Random();
```

## Randomisierte Testfälle

```
Integer getRandomObject() {  
    return new Object[] {random.nextInt(), random.nextDouble()};  
}  
}
```

### Konkrete Implementierungen für RandomizedCollection

Wie groß die Menge an Testdaten sein soll, bestimmt die *hasNext*-Methode von *RandomizedCollection*. Zum Beispiel kann man festlegen, dass eine bestimmte Anzahl an Zufallszahlen erzeugt wird (Quelltextbeispiel) oder nur in einer bestimmten Zeitspanne Testdaten erzeugt, dann würde *hasNext()* solange *true* liefern, solange z. B. eine halbe Sekunde nicht vorbei ist. Das hätte den Vorteil, dass der Testfall dann ungefähr eine halbe Sekunde dauert.

```
public class CountBoundedRandomizedCollection<T> extends  
RandomizedCollection<T> {  
    private final int numberOfElements;  
    private int numberOfIterations;  
    public CountBoundedRandomizedCollection(final RandomObjectGenerator<T>  
generator, final int numberOfElements) {  
        super(generator);  
        this.numberOfElements = numberOfElements;  
    }  
    @Override  
    public T next() {  
        numberOfIterations++;  
        return super.next();  
    }  
  
    @Override  
    boolean hasNext() {  
        return numberOfIterations <= numberOfElements;  
    }  
  
    T[] toArray(final T[] array) {  
        ...;  
    }  
}
```

### Nutzung der RandomizedCollection

Die konkrete Nutzung ist nur noch ein Einzeiler. In einem parametrisierten Test könnte die Erzeugung von zwanzig Zufallsvektoren so aussehen:

```
@Parameters  
public static RandomizedCollection<Object[]> data() {  
    return new CountBoundedRandomizedCollection<Object[]>(new  
RandomTuple(), 20);  
}
```

## Parametrisierte Tests in JUnit randomisieren

```
};
```

Natürlich kann man den Mechanismus auch für den Theory-Testrunner<sup>1</sup> verwenden:

```
@DataPoints public static Integer[] numbers = new  
CountBoundedRandomizedCollection<Integer>(new RandomInteger(), 20).toArray(new  
Integer[] {});
```

### Vorteile

Die Notation dieser Collections ist sehr elegant und durch die Strategieobjekte hochgradig wiederverwendbar, was in Hinblick auf komplexere Objekte interessant ist. Man erhält eine Vielzahl von Testdaten, die man sonst manuell eingegeben hätte.

### Weitere Ideen

#### Abstraktion von Dateizugriffen

Natürlich kann man auch eine Collection entwerfen, die beim Iterieren Zeile für Zeile einer CSV-Datei ausliest. So kann man Testdaten extern definieren und dem Testfall zuführen. Einziger Nachteil ist, dass man nur primitive Typen auf diese Weise einlesen kann. Denkbar ist auch ein Datenbankzugriff über JDBC.

#### Deterministische Zahlenreihen und spezielle Strings

Man muss nicht zwangsläufig Zufallszahlen verwenden, sondern kann genauso gut Fibonacci-Zahlen erzeugen, oder Primzahlen in aufsteigender Reihenfolge.

Bei *String* könnte man gezielt Sonderfälle erzeugen: leerer String, Strings mit Umlauten, Strings mit Sonderzeichen und gegebenenfalls durch Zufallsdaten anreichern. Jede Klasse weist bestimmte, spezielle Objekte auf, die unbedingt getestet werden sollten.

#### Generatorobjekt mit Rückkopplung

Anstatt einfach so ein Objekt zurückzugeben, kann man es genauso gut mit einer Methode (*addResult(boolean)*) ausstatten, um das letzte Testergebnis mitzuteilen und die Rückgabe eines neues neuen Objektes davon abhängig zu machen. Zum Beispiel, wenn das Strategieobjekt weiß, dass der Testfall bei -1 fehlschlägt, kann es beim nächsten Mal eine „ähnliche“ (z. B. eine weitere negative) Zahl (z. B. -2) zurückgeben. Schlägt der Testfall auch bei -2 fehl, dann wahrscheinlich bei allen negativen Zahlen und man kann aufhören zu testen; die Hypothese, dass der Test bei allen negativen Zahlen fehlschlägt, würde untermauert. Schlägt der Testfall nicht fehl, wird mit einer ungeraden Zahl weiter probiert. Am Ende könnte man vielleicht schließen, dass alle

---

<sup>1</sup> Erst JUnit 1.5 unterstützt die Möglichkeit, mit @DataPoints eine Vielzahl an Parametern für den Theory-Testrunner zu deklarieren. JUnit 1.4 bietet nur @DataPoint für einen einzelnen Parameter, was die Nutzung des Theory-Testrunners eher verleidet.

## Randomisierte Testfälle

negativen, ungeraden Zahlen als Eingabe zu fehlgeschlagenen Tests führen, was die Fehlerbehebung beschleunigen kann.

### **Abbildung von Vererbungshierarchien auf die Generatormenge**

Angenommen es existiert eine Klasse A und eine Klasse B, die von A erbt. Zufallsdaten mit einer Vielzahl der von Objekten der Klasse A könnten implizit angereichert werden durch Objekte der Klasse B, was den Test qualitativ verbessert. Typische Fehler, die erst durch die Objektorientierung entstanden sind (z. B. die Verletzung des Liskovschen Substitutionsprinzips), würden automatisch getestet.