Fluent Assertions

Ein Vergleich von Hamcrest, AssertJ und Truth

Marc Philipp

Ein wesentlicher Bestandteil automatisierter Tests ist die Formulierung des erwarteten Ergebnisses. Erst durch diese Assertions wird ein Test bei Ausführung rot oder grün. Die in JUnit verfügbaren assert-Methoden sind ein Anfang, reichen aber oft nicht aus. Drei Open-Source-Bibliotheken versprechen Abhilfe: Hamcrest, AssertJ und Truth. Dieser Artikel vergleicht und bewertet sie anhand einer Reihe von Kriterien, um herauszufinden, welche Vor- und Nachteile sie bieten.

Probleme herkömmlicher Assertions

JUnit ist ein Open-Source-Framework, das es Entwicklern erleichtert, automatisierte Tests zu schreiben und eine Sammlung solcher Tests auszuführen.

Reihenfolge-Problem

Sehen wir uns zunächst einen einfachen Test an, der mit herkömmlichen JUnit-Assertions arbeitet:

```
@Test
public void firstNameIsEqual() {
  Person bob = new Person("Bob", "Doe");
  assertEquals("Alice", bob.getFirstName());
}
```

Der Test instanziiert eine `Person` mit dem Vornamen „Bob“ und Nachnamen „Doe” und überprüft, ob der Vorname von `bob` „Alice” ist. Die Methode `assertEquals` ist aus der Klasse `org.junit.Assert` statisch importiert. Führt man diesen Test aus, schlägt er fehl und liefert folgende Fehlermeldung:

```
expected:<[Alice]> but was:<[Bob]>
```

Sowohl die Lesbarkeit der Assertion als auch die Aussagekraft der Fehlermeldung sind in diesem Fall angemessen. Ein häufiger Fehler ist allerdings, dass die beiden Parameter von `assertEquals` vertauscht werden:

```
assertEquals(bob.getFirstName(), "Alice");
```

Dies führt zu der folgenden, irritierenden Fehlermeldung:

```
expected:<[Bob]> but was:<[Alice]>
```

In diesem Fall ist der Test einfach und die Ursache schnell gefunden. Wird der Test komplizierter, kann eine solche irritierende Fehlermeldung aber einiges an Zeit kosten, zum Beispiel weil man die Ursache zunächst im Produktivcode vermutet. Daher wäre es schön, wenn uns die Assertion-Bibliothek davor bewahren würde, solche Fehler zu machen. Schließlich wollen wir mit den Tests Fehler im Produktivcode verhindern und nicht weitere Bugs produzieren.

Fehlende Ausdrucksmächtigkeit

Nur `assertEquals` reicht zur Formulierung von Assertions bei Weitem nicht aus. Möchte man zum Beispiel ausdrücken, dass der Vorname von `bob` ein „i” enthält (was natürlich nicht stimmt), muss man auf `assertTrue` zurückgreifen:

```
assertTrue(bob.getFirstName().contains("i"));
```

Die Fehlermeldung ist leider wenig hilfreich:

```
java.lang.AssertionError at …
```

Möchte man im Fehlerfall erkennen, was schiefging, muss man die Fehlermeldung selbst angeben. Etwa so:

```
assertTrue("person's first name must include an 'i'",
  bob.getFirstName().contains("i"));
```

Das erzeugt manuellen Aufwand und Duplikation. Möchte man statt auf „i” doch lieber auf ein „o” prüfen, muss man die Änderung an zwei Stellen durchführen. Diese Fehleranfälligkeit möchte man gerne vermeiden.

Fazit

Eine Assertion-Bibliothek soll uns also ermöglichen, ausdrucksmächtige Assertions zu schreiben, uns vor Leichtsinnsfehlern bewahren und aussagekräftige Fehlermeldungen generieren. Im Folgenden werden die drei Open-Source-Bibliotheken Hamcrest, AssertJ und Truth miteinander verglichen.

Lesbarkeit

Hamcrest

Wir beginnen mit Hamcrest [HaWb], für das JUnit bereits Unterstützung mitbringt. Die Assertion aus dem ersten Test lässt sich mit Hamcrest wie folgt formulieren:

```
assertThat(bob.getFirstName(), is("Alice"));
```

Dieses Statement liest sich wie ein englischer Satz. Es ist nicht möglich, die Parameter aus Versehen zu vertauschen, da das zweite Argument immer ein `Matcher` (Hamcrest ist ein Anagramm von Matchers) sein muss, dessen Typ zum ersten Argument, dem zu prüfenden Objekt, passt.

Die Matcher werden zur besseren Lesbarkeit über statische Imports verwendet. Einstiegspunkt ist die Klasse `org.hamcrest.Matchers`. Dort findet sich für jeden Matcher eine statische Methode.

Die Methode `assertThat` ist statisch aus `org.junit.Assert` importiert. Hamcrest lässt sich auch ohne JUnit, zum Beispiel mit TestNG, verwenden. Dazu muss lediglich der statische Import angepasst werden und stattdessen aus `org.hamcrest.MatcherAssert` erfolgen.

Interessanter ist nun eine komplizierte Assertion, wie die Substring-Bedingung aus dem zweiten Beispiel. Sie lässt sich mit Hamcrest so schreiben:

```
assertThat(bob.getFirstName(), containsString("i"));
```

Die Fehlermeldung, die beim Ausführen entsteht, enthält alle notwendigen Informationen und macht eine von Hand geschriebene Nachricht überflüssig:

```
Expected: a string containing "i"
but: was "Bob"
```

AssertJ

AssertJ [AJWb] ist ein Fork der Bibliothek FEST-Assert. Der Initiator Joel Costigliola erläutert seine Gründe hierfür ausführlich auf der AssertJ-Website [AJFk]. Unser erstes Beispiel lässt sich damit genauso lesbar schreiben wie bei Hamcrest:

```
assertThat(bob.getFirstName()).isEqualTo("Alice");
```

Die Klasse `org.assertj.core.api.Assertions` stellt hierbei den einzigen Einstiegspunkt dar: Die Methode `assertThat` ist statisch von dort importiert. Sie hat nur einen Parameter: das zu überprüfende Objekt. Weitere statische Imports sind nicht erforderlich, weil je nach übergebenem Parameter eine andere, überladen-

de Variante von `assertThat` verwendet wird. In diesem Fall verwenden wir die `String`-Variante, die `String`-spezifische Assertion-Methoden zur Verfügung stellt.

AssertJ verfügt über ein Fluent Interface, das heißt, die Auswahl der Assertion-Methode wird – im Gegensatz zu Hamcrest – durch die Entwicklungsumgebung erleichtert, die einem per *Auto Completion* die möglichen Assertions vorschlagen kann. Auch eine Methode zum Vergleich von Substrings existiert:

```
assertThat(bob.getFirstName()).contains("i");
```

Die Fehlermeldung ist anders formatiert als bei Hamcrest, lässt aber auch keine Wünsche offen:

```
Expecting:
<"Bob">
to contain:
<"i">
```

Truth

Truth [Trth] ist nach eigener Aussage *"very alpha, and subject to change"*. Hinter dem Projekt steht allerdings Google, das die Weiterentwicklung durch sein Java Library Team unterstützt. Auch JUnit-Maintainer David Saff war an der Entwicklung beteiligt. Man kann davon ausgehen, dass die Bibliothek innerhalb von Google verwendet wird und Google daher ein klares Interesse an der Weiterentwicklung und dem Fortbestehen des Projekts hat.

Die Funktionsweise ist sehr ähnlich wie bei AssertJ, daher sehen beide Beispiele (bis auf den statischen Import) identisch aus wie bei AssertJ:

```
assertThat(bob.getFirstName()).isEqualTo("Alice");
assertThat(bob.getFirstName()).contains("i");
```

Einstiegspunkt ist die Klasse `com.google.common.truth.Truth`. Wie AssertJ bietet auch Truth die Vorteile eines Fluent Interface. Die Fehlermeldung der zweiten Assertion sieht erneut anders aus als bei AssertJ und Hamcrest, enthält aber ebenso alle wichtigen Informationen:

```
Not true that <"Bob"> contains <"i">
```

Fazit

Bei der Lesbarkeit der Assertions gibt es keine großen Unterschiede zwischen den drei Bibliotheken. Alle drei helfen, einfache Fehler zu vermeiden, und sie ermöglichen meist besser lesbaren Code als herkömmliche JUnit-Assertions.

Aussagekraft der Fehlermeldungen

Hamcrest

Wir betrachten die Aussagekraft der generierten Fehlermeldung an einem Beispiel, das die Existenz zweier Personen in einer Collection von Personen überprüft:

```
@Test
public void isInCollection() {
  Person alice = new Person("Alice", "Foo");
  Person charlie = new Person("Charlie", "Baz");
  List<Person> friends = Arrays.asList(alice, charlie);
  assertThat(friends, hasItems(alice, bob));
}
```

Da nur `alice` in `friends` enthalten ist, `bob` jedoch nicht, schlägt diese Assertion fehl:

```
Expected: (a collection containing <Alice Foo>
          and a collection containing <Bob Doe>)
but: a collection containing <Bob Doe> was <Alice Foo>, was <Charlie Baz>
```

Die von Hamcrest erzeugte Fehlermeldung ist ziemlich sperrig. Für ihre Deutung benötigt man Erfahrung mit der Funktionsweise von Hamcrest.

AssertJ

In AssertJ lässt sich die gleiche Assertion wie folgt formulieren:

```
assertThat(friends).contains(alice, bob);
```

Die resultierende Fehlermeldung ist deutlich besser strukturiert und dadurch leichter verständlich:

```
Expecting:
<[Alice Foo, Charlie Baz]>
to contain:
<[Alice Foo, Bob Doe]>
but could not find:
<[Bob Doe]>
```

Truth

Die gleiche Eigenschaft lässt sich in Truth wie folgt schreiben:

```
assertThat(friends).containsAllOf(alice, bob);
```

Die Fehlermeldung ist hilfreich und enthält alle notwendigen Informationen.

```
Not true that <[Alice Foo, Charlie Baz]>
 contains all of <[Alice Foo, Bob Doe]>.
 It is missing <[Bob Doe]>
```

Fazit

Die drei Bibliotheken liefern in diesem Fall sehr unterschiedliche Fehlermeldungen. Gerade wenn es um Collections geht, generiert Hamcrest oft sehr sperrige, schwer verständliche Fehlermeldungen. Sowohl AssertJ als auch Truth liefern andererseits auch hier sehr nützliche Meldungen. Im Gegensatz zu AssertJ beschränkt sich Truth auf eine Zeile.

Welche Variante besser gefällt, ist ein Stück weit Geschmackssache. Bei größeren Collections ist die bessere Strukturierung der Fehlermeldung bei AssertJ aber mit Sicherheit von Vorteil.

Erweiterbarkeit

Um Redundanz zu vermeiden und häufige Assertions noch lesbarer zu gestalten, bieten alle drei Bibliotheken die Möglichkeit, Assertions für eigene Klassen zu schreiben.

Hamcrest

Bei Hamcrest benötigt man dazu eine eigene Implementierung von `Matcher<T>`. Hamcrest bietet mit `org.hamcrest.FeatureMatcher` eine nützliche Basisklasse zum Schreiben eigener Matcher:

```
public static Matcher<Person> hasFirstName(String firstName) {
  return new FeatureMatcher<Person, String>(equalTo(firstName),
      "a person with first name", "first name") {
    @Override
    protected String featureValueOf(Person actual) {
      return actual.getFirstName();
    }
  };
}
```

Der Konstruktor von `FeatureMatcher` fordert drei Parameter: einen `Matcher` für den Zieltyp (hier `String`), einen Beschreibungstext und einen kürzeren Text, der für die Fehlerbeschreibung benutzt wird. Die Verwendung dieses neuen Hamcrest Matcher sieht dann so aus:

```
assertThat(bob, hasFirstName("Alice"));
```

Die zugehörige Fehlermeldung ist sehr viel näher an der Fachsprache:

```
Expected: a person with first name "Alice"
but: first name was "Bob"
```

AssertJ

In AssertJ spezifiziert man eigene Assertions über eine eigene Assertion-Klasse:

```
public class PersonAssert extends
    AbstractAssert<PersonAssert, Person> {
  protected PersonAssert(Person actual) {
    super(actual, PersonAssert.class);
  }
  public static PersonAssert assertThat(Person actual) {
    return new PersonAssert(actual);
  }
  public PersonAssert hasFirstName(String expectedFirstName) {
    isNotNull();
    String actualFirstName = actual.getFirstName();
    if (!Objects.equals(actualFirstName, expectedFirstName)) {
      failWithMessage(
        "Expected person's first name to be <%s> but was <%
        s>"
        expectedFirstName, actualFirstName);
    }
    return this;
  }
}
```

Zur Verwendung importiert man zusätzlich die Methode **`assertThat`** aus **`PersonAssert`**:

```
assertThat(bob).hasFirstName("Alice");
```

Bei der Fehlermeldung muss man sich mehr Mühe geben, als beim **`FeatureMatcher`** von Hamcrest, da man sie selbst formulieren muss:

```
Expected person's first name to be <Alice> but was <Bob>
```

Truth

In Truth benötigt man eine Implementierung von **`Subject<S, T>`**, die in unserem Beispiel wie folgt aussieht:

```
public class PersonSubject extends Subject<PersonSubject, Person> {
  public static SubjectFactory<PersonSubject, Person> person() {
    return new SubjectFactory<PersonSubject, Person>() {
      @Override
      public PersonSubject getSubject(FailureStrategy fs,
                                      Person target) {
        return new PersonSubject(fs, target);
      }
    };
  }
  protected PersonSubject(FailureStrategy failureStrategy,
                          Person subject) {
    super(failureStrategy, subject);
  }
  public void hasFirstName(String expectedFirstName) {
    isNotNull();
    String actualFirstName = getSubject().getFirstName();
    if (!Objects.equals(actualFirstName, expectedFirstName)) {
      failWithBadResults("has first name", expectedFirstName,
        "was", actualFirstName);
    }
  }
}
```

Statt **`assertThat`** importieren wir nun **`assertAbout`** aus **`Truth`** und schreiben die Assertion wie folgt:

```
assertAbout(person()).that(bob).hasFirstName("Alice");
```

Fehlermeldung:

```
Not true that <Bob Doe> has first name <Alice>. It was <Bob>
```

Fazit

Sowohl AssertJ als auch Truth benötigen im Gegensatz zu Hamcrest einiges an Boilerplate-Code, um Assertions für eigene Klassen zu ermöglichen. Im Falle von AssertJ gibt es jedoch das Subprojekt *Assertion Generator* [AJGn], das pro Domänenklasse eine Assertion-Klasse generiert und per Kommandozeile oder als Maven-Plug-in verwendet werden kann.

Auffindbarkeit

Hamcrest

Alle mitgelieferten Matcher findet man in der Klasse **`org.hamcrest.Matchers`** (aus *hamcrest-library*). Eigene Matcher sammelt man am besten in einer eigenen Klasse. Beispielsweise enthält **`PersonMatchers`** alle Matcher für die Klasse **`Person`**. Hält man sich an diese Konvention, findet man relativ einfach die zur Verfügung stehenden Matcher.

Noch besser wäre es allerdings, wenn man sich nur einen Einstiegspunkt merken müsste. Dafür gibt es bis Version 1.3 das Subprojekt *hamcrest-generator*, das Methoden, die mit der Annotation **`@Factory`** versehen sind, aufsammelt und Delegatmethoden in einer Klasse ähnlich wie **`org.hamcrest.Matchers`** generiert. Im aktuellen Entwicklungsstand von Hamcrest ist dieses Subprojekt allerdings nicht mehr enthalten.

AssertJ

Mitgelieferte Assertions lassen sich wie schon erwähnt per *Auto Completion* finden. Was eigene Assertions angeht, kann man sich ebenfalls mit Namenskonventionen behelfen, das heißt, Assertion-Klassen wie **`PersonAssert`** zu Domänenklassen wie **`Person`** anzulegen. Zusätzlich bietet AssertJ die Möglichkeit, über das bereits erwähnte Subprojekt *Assertion Generator* einen Einstiegspunkt für alle eigenen Assertions zu generieren.

Truth

Für Truth gilt das Gleiche wie für AssertJ, allerdings gibt es keinen Generator. Möchte man nur einen Einstiegspunkt haben, muss man die statischen Methoden, die eine **`SubjectFactory`** liefern, selbst in einer Klasse sammeln.

Fazit

Hat man erst einmal den Einstiegspunkt gefunden, bieten sowohl Truth als auch AssertJ den Vorteil eines Fluent API: Die Entwicklungsumgebung kann direkt alle Assertions anbieten, die für die zu überprüfende Klasse zur Verfügung stehen. Bei Hamcrest hilft ein *Cheatsheet* [HaCh], den Überblick über die mitgelieferten Matcher zu bekommen.

Kombinierbarkeit

Hinter dem Kriterium der Kombinierbarkeit verbirgt sich die Frage, wie gut sich Assertions miteinander verbinden lassen, um neue Assertions zu erzeugen. Kann man ihre Bedeutung umkehren oder mehrere mit logischem Und/Oder verknüpfen? Kann man Assertions, die auf einer einzelnen Instanz funktionieren, dann auch auf Elemente einer Collection anwenden?

Hamcrest

Das **`Matcher`**-Konzept von Hamcrest bieten intuitive Kombinationsmöglichkeiten, um neue Assertions aus bestehenden Matchern zusammenzubauen. Der **`allOf`**-Matcher verknüpft mit logischem Und, **`anyOf`** mit Oder, **`not`** negiert:

```
assertThat(bob.getFirstName(),
  allOf(startsWith("B"), not(startsWith("A"))));
assertThat(bob.getFirstName(),
  anyOf(startsWith("B"), startsWith("A")));
```

Auch für Collections funktioniert das auf natürliche Art und Weise. Folgende Assertion erwartet zwei Elemente in `friends` und verwendet den selbst definierten Matcher `hasFirstName` aus dem vorherigen Abschnitt:

```
assertThat(friends,
  hasItems(hasFirstName("Alice"), hasFirstName("Charlie")));
```

Darüber hinaus bietet Hamcrest die Möglichkeit, eigene Matcher mit bereits vorhandenen zu kombinieren. Dazu überladen wir die Methode `hasFirstName` wie folgt:

```
public static Matcher<Person> hasFirstName(String firstName) {
  return hasFirstName(equalTo(firstName));
}
public static Matcher<Person> hasFirstName(Matcher<String> matcher) {
  return new FeatureMatcher<Person, String>(matcher,
      "a person with first name", "first name") {
    @Override
    protected String featureValueOf(Person actual) {
      return actual.getFirstName();
    }
  };
}
```

Nun können wir den Substring-Match mit unserem eigenen Matcher kombinieren:

```
assertThat(bob, hasFirstName(containsString("i")));
```

Die resultierende Fehlermeldung liest sich immer noch einigermaßen flüssig:

```
Expected: a person with first name a string containing "i"
but: first name was "Bob"
```

AssertJ

AssertJ unterstützt *Method Chaining* zum Formulieren mehrere Assertions in einem Statement:

```
assertThat(bob.getFirstName()).startsWith("B").
  contains("o").endsWith("b");
```

Die Bedingungen werden dabei alle überprüft, eine nach der anderen. Das entspricht in etwa einer Verknüpfung mit logischem Und. Möchte man Assertions stattdessen mit logischem Oder verknüpfen oder negieren, geht das leider nicht mit normalen Assertions. AssertJ bietet hierfür Unterstützung über sogenannte Conditions. Mit Hilfe von Lambdas lässt sich eine `Condition<T>` kurz und bündig definieren, etwa so:

```
private static Condition<String> startsWith(String prefix) {
  return new Condition<>(s -> s.startsWith(prefix),
    "starts with %s", prefix);
}
```

Das erinnert stark an die Implementierung eines `FeatureMatcher` bei Hamcrest. Hat man die `Condition` erstmal definiert, kann man sie wie folgt verwenden:

```
assertThat(bob.getFirstName()).is(
  allOf(startsWith("B"), not(startsWith("A"))));
assertThat(bob.getFirstName()).is(
  anyOf(startsWith("B"), startsWith("A")));
```

Die selbst geschriebene Assertion `hasFirstName` aus dem vorherigen Abschnitt lässt sich bei AssertJ leider nicht für Collections von `Person` wiederverwenden. Allerdings bietet AssertJ ein Feature, das diese Einschränkung (unter Java 8) fast vergessen lässt:

```
assertThat(friends).extracting(Person::getFirstName)
  .contains("Alice", "Charlie");
```

Die Methode `extracting` erhält eine `Function` und transformiert damit – ähnlich wie `map` aus dem Java-Streams-API – die Elemente der Collection. In diesem Fall wandeln wir die Instanzen von `Person` per Aufruf von `getFirstName` (hier als Methodenreferenz angegeben) um und überprüfen die resultierenden Zeichenketten dann auf Gleichheit.

Truth

Obwohl Truth ebenfalls von FEST inspiriert wurde, unterstützt es im Gegensatz zu AssertJ kein *Method Chaining*. Daher muss man in Truth pro Assertion ein Statement mit `assertThat` formulieren:

```
assertThat(bob.getFirstName()).startsWith("B");
assertThat(bob.getFirstName()).contains("o");
assertThat(bob.getFirstName()).endsWith("b");
```

Truth bietet keine Möglichkeit, Assertions mit logischen Operatoren zu verknüpfen. Es gibt genau eine Möglichkeit, unsere selbst geschriebene Assertion `hasFirstName` auf Elemente einer Collection anzuwenden:

```
assert_().in(friends).thatEach(person()).hasFirstName("Bob");
```

Die Assertion ist genau dann erfolgreich, wenn alle Elemente die Bedingung erfüllen.

Fazit

Das Matcher-Konzept von Hamcrest bietet die größten Kombinationsmöglichkeiten. AssertJ erlaubt mehrere Assertions pro Statement und bietet über Conditions eine sinnvolle Erweiterungsmöglichkeit. Truth bietet in Sachen Kombinierbarkeit das eindeutige Schlusslicht.

Lieferumfang

Hamcrest

Hamcrest bietet Matcher für viele Standardtypen im JDK, etwa für `String`, `Iterable`, `Collection`, `Array` oder `Map`. Basis-Matcher, die auf Gleichheit oder `null` prüfen, sind ebenso enthalten wie Matcher für Zahlen und `Comparable`-Implementierungen. Daneben gibt es eine kleinere Zahl von Erweiterungsbibliotheken [HaEx].

AssertJ

AssertJ ist in mehrere Module aufgeteilt. Das *Core*-Modul liefert neben der Kernfunktionalität auch eine beachtliche Menge von Assertions mit: je nach Version für JDK 6 (AssertJ 1.x), JDK 7 (AssertJ 2.x) oder JDK 8 (AssertJ 3.x). Sehr nützlich sind auch die Module für die weitverbreiteten Bibliotheken *Guava* und *Joda Time*. Des Weiteren gibt es Module für *Neo4J*, *Swing* und Datenbank-Tests (*DB*).

Truth

Im Lieferumfang von Truth sind ähnlich wie bei Hamcrest Assertions für JDK-Klassen enthalten. Zusätzlich – hier merkt man den Einfluss von Google – gibt es Assertions für Guavas `Optional`, `Multimap`, `Multiset` usw.

Fazit

Klarer Sieger beim Lieferumfang ist AssertJ. Hamcrest und Truth liegen etwa gleichauf.

Assumptions

Neben Assertions unterstützt JUnit sogenannte *Assumptions*. Während eine fehlschlagende Assertion den Test fehlschlagen

lässt, führt eine Assumption dazu, dass er ignoriert wird. Mit Assumptions lassen sich Annahmen oder Vorbedingungen definieren, die erfüllt sein müssen, damit ein Test sinnvoll ausgeführt werden kann. Beispielsweise kann man damit bei Integrationstests sicherstellen, dass externe Systeme (z. B. eine Datenbank oder ein Server) erreichbar sind. Technisch wird anhand der verwendeten Exception unterschieden: `AssertionError` beziehungsweise `AssumptionViolatedException`.

Hamcrest

JUnit bietet analog zur Klasse `Assert` für Assertions mit `Assume` einen Einstiegspunkt für Assumptions. Die Parameter von `assumeThat` sind identisch zu `assertThat`:

```
assumeThat(System.getProperty("my.setting"), containsString("run"));
```

Daher lässt sich mit Hamcrest alles, was sich durch Assertions ausdrücken lässt, auch als Assumption formulieren.

AssertJ

AssertJ bietet von Haus aus keine Unterstützung für Assumptions. Möchte man AssertJ verwenden, aber nicht auf Assumptions verzichten, kann man entweder auf die einfacheren Methoden in `Assume` ausweichen oder sich eine kleine Hilfsmethode definieren:

```
public static void assumeSucceeds(Runnable runnable) {
  try {
    runnable.run();
  } catch (AssertionError e) {
    throw new AssumptionViolatedException(e.getMessage(), e.getCause());
  }
}
```

Damit lässt sich zumindest ab Java 8 auch AssertJ einigermaßen lesbar zum Schreiben von Assumptions verwenden:

```
assumeSucceeds(() -> {
  assertThat(System.getProperty("my.setting")).contains("run");
});
```

Truth

Truth unterstützt Assumptions über einen separaten Einstiegspunkt, die Klasse `TruthJUnit`:

```
assume().that(System.getProperty("my.setting")).contains("run");
```

Jede Truth-Assertion lässt sich durch Austausch des sogenannten `TestVerb` (*assume* statt *assert*) in eine Assumption umwandeln. Diese Flexibilität verbirgt sich hinter dem Motto von Truth: *"We've made failure a strategy"*.

Fazit

Sowohl Hamcrest als auch Truth bieten die Möglichkeit, Assumptions zu formulieren. AssertJ unterstützt diese Funktionalität nicht.

Gesamtbewertung

Tabelle 1 liefert eine Übersicht über die Bewertungen der drei Bibliotheken in den einzelnen Kategorien. Einen eindeutigen Sieger, der in jeder Kategorie überzeugt, gibt es nicht.

- Hamcrests Schwächen liegen in der teilweise mangelnden Aussagekraft der erzeugten Fehlermeldungen, der Auffindbarkeit der Matcher und der relativ geringen Anzahl von mitgelieferten Matchern. Andererseits überzeugt es mit einer sehr guten Erweiterbarkeit und Kombinierbarkeit und wird von JUnit standardmäßig unterstützt.
- AssertJ hat leichte Defizite bei der Verwendung eigener Assertions sowie der Kombinierbarkeit und unterstützt keine Assumptions. Anderseits ermöglicht sein Fluent API zusammen mit dem großen Lieferumfang einen schnellen Einstieg. Außerdem hat das Projekt mit Abstand die höchste Entwicklungsgeschwindigkeit der drei Bibliotheken: In diesem Jahr gab es bis jetzt bereits sieben Releases.
- Trotz gemeinsamen Vorbilds hinkt Truth mit Ausnahme von Assumptions und einer besseren Unterstützung selbst geschriebener Assertions in fast allen Belangen hinter Hamcrest und AssertJ her. Der `TestVerb`-Ansatz ist jedoch vielversprechend – es bleibt abzuwarten, wie sich Truth weiterentwickeln wird.

Welche Bibliothek soll man nun einsetzen? Die Antwort ist – wie so häufig – es kommt darauf an. Hat man bereits ein Projekt, das intensiv mit Hamcrest Matchern arbeitet, lohnt sich der Umstieg auf AssertJ oder Truth kaum. Setzt man hingegen ein Projekt neu auf, ist AssertJ eine vielversprechende Wahl. Auch ein paralleler Einsatz beider Bibliotheken ist denkbar. Truth hingegen würde ich momentan noch nicht einsetzen.

Die Code-Beispiele sind auf GitHub verfügbar unter `http://andrena.github.io/assertion-libraries-2015`.

	Hamcrest	AssertJ	Truth
Lesbarkeit	★★★	★★★	★★★
Aussagekraft der Fehlermeldungen	★★☆	★★★	★★☆
Erweiterbarkeit	★★★	★★★	★★☆
Auffindbarkeit	★★☆	★★★	★★★
Kombinierbarkeit	★★★	★★☆	★☆☆
Lieferumfang	★☆☆	★★★	★★☆
Assumptions	★★★	☆☆☆	★★★

Tabelle 1: Bewertung der drei Bibliotheken

Literatur und Links

[AJFk] AssertJ, Why having forked Fest Assert, `http://joel-costigliola.github.io/assertj/assertj-core.html#fork`

[AJGn] AssertJ Assertions Generator, `http://joel-costigliola.github.io/assertj/assertj-assertions-generator.html`

[AJWb] AssertJ Website, `http://joel-costigliola.github.io/assertj/`

[HaCh] Hamcrest Cheatsheet, `http://www.marcphilipp.de/blog/2013/01/02/hamcrest-quick-reference/`

[HaEx] Hamcrest – library of matchers for building test expressions, `https://code.google.com/p/hamcrest/wiki/MatcherLibraries`

[HaWb] Hamcrest Website, `http://hamcrest.org/JavaHamcrest/`

[JUnit] `http://junit.org/`

[Trth] `http://google.github.io/truth/`

Marc Philipp (Twitter: @marcphilipp) ist Softwareentwickler und Entwickler-Coach bei der andrena objects ag. Neben seiner täglichen Arbeit beschäftigt er sich mit Open-Source-Entwicklungswerkzeugen: Er ist einer der Maintainer von JUnit und Project Usus. E-Mail: marc.philipp@andrena.de