



HAMCREST API UND TESTFRAMEWORK ASSERTJ

Seminar 1919 Moderne Programmiertechniken und -methoden – Sommer 2019

Thiemo Belmega

Inhalt

1.	Einleitung.....	3
1.1	Assertions	3
1.2	Hamcrest	3
1.3	AssertJ.....	4
1.4	Code-Beispiele	4
1.5	Zur Sprache.....	5
1.6	Begriffsbestimmungen	5
	Manuelle und automatisierte Tests	5
	Unit Tests.....	5
	Integration Tests, Component Tests	5
	End-to-End Tests, System Tests, Functional Tests, Use-Case-Tests	5
2.	Software Testing.....	7
2.1	Ziele des Testens	7
2.2	Nutzen eines Tests.....	8
2.3	Kosten eines Tests	9
2.4	Vergleichskriterien für Assertion-Bibliotheken	10
3.	Einfache Tests mit JUnit, Hamcrest und AssertJ	11
3.1	Aufbau eines Tests	11
3.2	Einfacher Testfall: Lesbarkeit der Assertions	13
3.3	Einfacher Testfall: Nutzerfreundlichkeit und Lernkurve	14
3.4	Einfacher Testfall: Verständlichkeit und Informationsgehalt der Fehlerausgaben.....	15
3.5	Schlussfolgerung.....	16
4.	Komplexe Tests mit JUnit, Hamcrest und AssertJ	17
4.1	Tests mit erwarteter Collection.....	18
4.2	Vergleich der Lesbarkeit von komplexen Assertions	20
4.3	Fehlernachrichten bei komplexen Assertions	22
4.4	Erwartete Exceptions	23
4.5	Nutzerfreundlichkeit und Lernkurve	24
4.6	Verfügbare Assertions	25
5.	Auswertung	26
	Literaturverzeichnis	29

Abbildung 1: Testfälle für ShoppingCart	11
Abbildung 2: Aufbau eines Tests mit given, when, then	11
Abbildung 3: Einfacher Test mit JUnit	13
Abbildung 4: Einfacher Test mit Hamcrest.....	13
Abbildung 5: Einfacher Test mit AssertJ	14
Abbildung 6: Fehlerausgabe von assertEquals (JUnit)	15
Abbildung 7: Fehlerausgabe von equalTo (Hamcrest)	15
Abbildung 8: Fehlerausgabe von assertThat.isEqualTo (AssertJ).....	16
Abbildung 9: Repository-Test mit Spring Boot	17
Abbildung 10: Vergleich zweier Listen mit equals.....	18
Abbildung 11: Ausgabe der erwarteten und tatsächlichen Liste in Kotlin	19
Abbildung 12: Zusicherung der einzelnen Elemente und der Länge.....	19
Abbildung 13: Eigene Fehlerausgaben für Assertions.....	19
Abbildung 14: Ausgabe eigener Fehlernachrichten	19
Abbildung 15: Assertion mit eigener Vergleichsmethode	20
Abbildung 16: containsInAnyOrder mit Hamcrest	21
Abbildung 17: containsExactlyInAnyOrder mit AssertJ	21
Abbildung 18: Fehlerausgabe von containsInAnyOrder (Hamcrest).....	22
Abbildung 19: Fehlerausgabe von containsExectlyInAnyOrder (AssertJ)	22
Abbildung 20: Erwartete Exception mit try-catch.....	23
Abbildung 21: Erwartete Exception per Annotation	23
Abbildung 22: Erwartete Exception in AssertJ	24
Abbildung 23: IntelliJ schlägt passende AssertJ-Methoden vor	25
Abbildung 24: SpringBootTest mit TestRestTemplate, Mockito, DI-Container.....	27

1. Einleitung

Hamcrest und AssertJ sind Klassenbibliotheken, die zur Verbesserung der Lesbarkeit und Ausdrucksstärke von Testklassen verwendet werden können (Vogel 2016; Bushnev 2017). Bei beiden handelt es sich nicht um eigenständige Testing-Frameworks, sondern um mögliche Ergänzungen zu JUnit oder TestNG. Wie auch JUnit sind die Bibliotheken mit Java, Kotlin, Scala und Groovy kompatibel.

AssertJ wurde zu einem Zeitpunkt veröffentlicht, als Hamcrest schon sechs Jahre lang breite Verwendung gefunden hat. Die vorliegende Arbeit soll untersuchen, inwiefern die jüngere Bibliothek eine brauchbare oder sogar vorteilhafte Alternative zum etablierteren Hamcrest bietet.

1.1 Assertions

Das wesentliche Element von Softwaretests ist der Vergleich des erwarteten Verhaltens mit dem beobachteten Verhalten des Programms. In automatisierten Tests erfolgt dieser Vergleich über sogenannte Assertions (dt. Zusicherungen): Der Assertion wird ein Wert übergeben, der aus der Programmausführung resultiert; stimmt der Wert mit der Erwartung überein, so tut das Assertion-Statement nichts, andernfalls löst es eine Exception (dt. Ausnahme) aus, die dem Programmierer als Fehlschlag des Tests angezeigt wird.

JUnit und TestNG enthalten von sich aus eine Anzahl grundlegender Assertions: `assertEquals` vergleicht den eingegebenen zu prüfenden Wert mit dem ebenfalls eingegebenen erwarteten Wert. `assertTrue`, `assertFalse` und `assertNull` vergleichen den eingegebenen Wert mit `True`, `False` oder `Null`. `assertSame` überprüft die beiden Eingabewerte auf referentielle Identität, stellt also fest, ob es sich um zwei Referenzen auf dasselbe Objekt handelt und nicht nur um ein gleichwertiges Objekt im Sinne der jeweiligen `equals`-Implementierung.

Um komplexere Bedingungen prüfen zu können, z.B. ob der beobachtete Wert einen erwarteten Substring enthält, muss der benötigte Vergleich programmatisch durchgeführt und das Ergebnis des Vergleichs über `assertTrue` bzw. `assertFalse` zugesichert werden. Auf diese Weise lässt sich jede erwünschte Bedingung prüfen; deshalb erhöhen Hamcrest und AssertJ nicht die Mächtigkeit der JUnit-Assertions, sondern verbessern nur das Nutzungserlebnis für den Entwickler, wie im Verlauf der vorliegenden Arbeit gezeigt werden soll.

1.2 Hamcrest

Die Hamcrest-Bibliothek ist seit 2007 im öffentlichen Maven Repository verfügbar (<https://mvnrepository.com/artifact/org.hamcrest/hamcrest-core>). Sie wurde 2012 als Bestandteil in JUnit 4.4 aufgenommen; JUnit 5 enthält Hamcrest nicht mehr, weist aber auf die Möglichkeit der Verwendung von Drittanbieter-Bibliotheken wie AssertJ, Hamcrest oder Truth¹ ausdrücklich hin (Bechtold et al.).

Der Quellcode von Hamcrest wurde 2012 auf GitHub (<https://github.com/hamcrest/JavaHamcrest>) veröffentlicht und wurde seitdem für lange Zeit nicht gewartet. Erst 2018 wurde die Wartung des Projekts wiederaufgenommen und mit einigen Bugfixes und technischen Neuerungen als Version 2.0 publiziert.

Hamcrest enthält eine einzige zusätzliche Assertion-Methode: `assertThat` (Baeldung SRL. 2018). Diese Methode nimmt neben dem zu prüfenden Wert einen Matcher entgegen („to match“ dt. entsprechen, passen, übereinstimmen). Matcher machen den größten Teil der Hamcrest-Bibliothek aus; es handelt

¹ <https://mvnrepository.com/artifact/com.google.truth/truth>

sich um Methoden, mit denen der Programmierer die beschriebenen komplexeren Erwartungen formulieren kann. Bei der Testausführung überprüft `assertThat` also, ob der übergebene Wert den durch den Matcher formulierten Kriterien genügt und wirft andernfalls eine Exception.

1.3 AssertJ

AssertJ wurde 2013 auf GitHub und Maven Repository veröffentlicht, seitdem durchgehend gewartet und erscheint monatlich als neue Version (<https://mvnrepository.com/artifact/org.assertj/assertj-core>, <https://github.com/joel-costigliola/assertj-core/releases>). AssertJ war niemals Bestandteil von JUnit, konnte aber aufgrund der mittlerweile weit verbreiteten Dependency-Management-Lösungen wie Maven², Ivy³ und Gradle⁴ leicht in Java-Projekte eingebunden werden.

Laut Google Trends wird nach „AssertJ“ in Deutschland inzwischen häufiger gesucht als nach „Hamcrest“, weltweit liegt Hamcrest mit schwindendem Abstand vorne (Google Trends 2019). An dieser Stelle sei darauf hingewiesen, dass die Häufigkeit als Suchbegriff nicht mit der Verwendung in Softwareprojekten korrelieren muss. So kann eine Bibliothek bei gleicher Nutzung dennoch häufiger Suchbegriff sein, weil sie schwieriger zu verstehen ist; oder die ältere Bibliothek wird wegen ihrer Verwendung in Legacy-Projekten gesucht, während neue Projekte häufiger die jüngere Bibliothek verwenden.

Aus Nutzersicht bringt auch AssertJ nur eine einzelne Assertion-Methode namens `assertThat` mit; technisch handelt es sich um eine Reihe überladener Methoden mit unterschiedlicher Signatur. Der Typ des Eingabewerts bestimmt also, welche der gleichnamigen `assertThat`-Methoden tatsächlich aufgerufen wird. Dieses `assertThat` nimmt nur den zu prüfenden Wert entgegen, keinen Vergleichswert. Stattdessen gibt der Aufruf von `assertThat` ein Assertion-Objekt zurück, auf dem dann eine weitere Methode zum Spezifizieren der Erwartung aufgerufen werden kann, die dann wiederum ein Assertion-Objekt zurückgibt. Auf diese Weise lassen sich mehrere Erwartungen aneinanderhängen, die man flüssig hintereinanderweg lesen kann. Dieser Programmierstil heißt „Fluent Interface“ (Fowler 2005) und ist in moderneren Frameworks weit verbreitet. Die flüssigen Methodenaufrufe sollen eine domänenspezifische Sprache (domain specific language, DSL) bilden, die intuitiv zu lesen und zu schreiben ist.

Wie Hamcrest dient AssertJ dem nutzerfreundlichen Formulieren komplexerer Erwartungen für Assertions. In diesem Fall werden keine Matcher-Objekte erzeugt, sondern Methoden auf den typspezifischen Assertion-Objekten aufgerufen.

1.4 Code-Beispiele

Die aufgeführten Code-Beispiele sind vom Autor dieser Arbeit vollständig selbst erstellt. Um die Beispiele realistisch wirken zu lassen, wurde ein fiktiver Onlineshop für Computerzubehör als leitendes Thema gewählt. Das Projekt ist auf GitHub verfügbar; die Tests sind ausführbar, das Projekt enthält aber nur Beispielcode und keine fertige Shop-Software: <https://github.com/tbelmega/pcshop>

Das Projekt ist in Kotlin implementiert und verwendet Spring Boot⁵ mit einer H2-InMemory-Datenbank⁶. Die Tests laufen mit JUnit 4, das wie Hamcrest und AssertJ in der Gradle-Dependency `spring-boot-starter-test` enthalten ist.

² <https://maven.apache.org/>

³ <https://ant.apache.org/ivy/>

⁴ <https://gradle.org/>

⁵ <https://spring.io/>

⁶ <https://www.h2database.com>

Der Code kann heruntergeladen, mit dem Befehl „gradlew clean build“ gebaut und die Tests ausgeführt werden. Erforderlich ist dafür lediglich eine Installation von Java 8, der Gradle-Wrapper liegt im Projekt und löst sämtliche Abhängigkeiten automatisch auf. Ein Teil der Tests schlägt absichtlich fehl, um die Fehlerausgaben mit JUnit, Hamcrest und AssertJ zu demonstrieren.

1.5 Zur Sprache

In einer deutschsprachigen Arbeit über Softwareentwicklung steht der Autor vor einem Dilemma: Lässt er englische Fachbegriffe wie Deployment, Framework oder Refactoring in den Text einfließen, empfindet das so mancher Leser als schlechten Stil. Übersetzt er dagegen solche Begriffe konsequent ins Deutsche, so täuscht er eine künstliche Fachsprache vor, die in der Praxis niemand verwendet.

In Ermangelung einer besseren Alternative entscheidet sich der Autor dieser Arbeit für die Verwendung englischer Fachbegriffe, sofern die deutsche Entsprechung in der Softwareindustrie nicht tatsächlich verwendet wird. Er bemüht sich dabei, die englischen Worte den Lesefluss nicht zu sehr stören zu lassen.

1.6 Begriffsbestimmungen

Manuelle und automatisierte Tests

Wird Software durch händische Dateneingabe einer Person getestet, die dann das Verhalten der Software beobachtet und mit einer Erwartung vergleicht, nennt man diesen Vorgang einen manuellen Test. Erfolgt die Dateneingabe und der Vergleich dagegen durch ein Programm, so heißt dieses Programm automatisierter Test. Automatisierte Tests müssen von einer Person programmiert und im Falle eines Fehlschlags ebenfalls von einer Person ausgewertet werden.

Ist im Rahmen dieser Arbeit von Tests oder Testing als Aktivität die Rede, sind in der Regel automatisierte Tests gemeint, solange nicht ausdrücklich manuelle Tests erwähnt werden.

Unit Tests

Ein Unit Test ist ein automatisierter Test, der einen kleinen Teil („Unit“, dt. Einheit) eines Softwaresystems testet und nur Bruchteile einer Sekunde zur Ausführung benötigt. Das Verständnis für die Größe einer Unit variiert dabei. Unter Softwareentwicklern ist die Ansicht verbreitet, ein Unit Test müsse die kleinstmögliche isolierbare Einheit Code testen, also eine einzelne öffentliche Methode. Andere Entwickler betrachten eine Klasse als Unit, wieder andere eine Gruppe eng zusammenhängender Klassen (Fowler 2014).

Integration Tests, Component Tests

Ein Integrationstest im ursprünglichen Sinne ist ein Test, der die Integration zweier Komponenten in einem Softwaresystem testet. Getestet wird also nur, ob die Interfaces der Komponenten wie erwartet zusammenspielen, nicht die Funktion der aufgerufenen Komponente.

Wer allerdings die Definition von Unit Tests auf das Testen einzelner Methoden verengt, neigt dazu, das funktionale Testen einer Gruppe eng zusammenhängender Klassen als Integrationstest zu bezeichnen (Hauer 2019). Für diese Art von Test bietet sich eigentlich die Bezeichnung Komponententest an (Fowler 2013), sofern der Begriff Komponente im Kontext des getesteten Systems nicht bereits anderweitig belegt ist.

End-to-End Tests, System Tests, Functional Tests, Use-Case-Tests

Über Integrationstests hinaus kann auch das gesamte System automatisiert getestet werden, inklusive aller Komponenten und der Datenbank. Die Begriffe End-to-End, System-, Funktions- oder Use-Case-Test werden hier überwiegend synonym verwendet. Aufgrund der verschwommenen Begrifflichkeiten bleibt die Frage offen, ob das Testen eines Backend-Webservices über seine Http-API noch ein

Integrationstest oder bereits ein End-to-End-Test/Systemtest ist, gerade wenn eine technisch getrennte Frontend-Anwendung existiert, die aus Nutzersicht Teil des Systems ist.

Das Fehlen einer einzigen, allseits anerkannten Taxonomie für Testarten ist aus wissenschaftlicher Sicht unbefriedigend. Im Rahmen dieser Arbeit wird der Einfachheit halber von Unit Tests oder einfach nur von Tests gesprochen, die meisten Aussagen lassen sich aber auf alle Formen des automatisierten Testens übertragen, falls nicht anders erläutert.

2. Software Testing

Um die beiden vorgestellten Bibliotheken im wissenschaftlichen Sinne vergleichen zu können, ist erforderlich zunächst Vergleichskriterien zu entwickeln. Zu diesem Zweck beginnt dieses Kapitel mit der Motivation für Software Testing, um daraus abzuleiten, unter welchen Gesichtspunkten Hamcrest und AssertJ zu vergleichen sind.

2.1 Ziele des Testens

Eine vielzitierte Aussage über Software Testing stammt von Edsger W. Dijkstra auf der NATO-Konferenz zu Software Engineering von 1969: „Testing shows the presence of bugs, not the absence.“ (Buxton und Randell 1970, S. 16) Dijkstra äußert also sinngemäß, Testen könne nur das Vorhandensein von Fehlern zeigen, aber nicht die Abwesenheit von Fehlern beweisen.

Diese Feststellung ist zwar logisch wahr und im Kontext der zitierten Diskussion über formale Programmkorrektheit durchaus angebracht; Schlussfolgerungen für die Praxis, insbesondere unter Berücksichtigung heutige verbreiteter Methoden und Technologien, dürfen aber nur äußerst vorsichtig gezogen werden. Die Aussage unterstellt, die Intention von Software Testing wäre der Beweis der Fehlerlosigkeit eines Programms, und weist darauf hin, dass dieses Ziel nicht erreicht werden kann. Tatsächlich sind automatisierte Tests aber ein wesentlicher Bestandteil jedes modernen Softwareentwicklungsprojekts (Ammann und Offutt 2008, XV), weil Testing eben nicht auf formale Korrektheit abzielt, sondern andere Aufgaben erfüllt.

Zwei davon führt Kent Beck, Schöpfer von JUnit und Extreme Programming, in seinem Standardwerk „Test-Driven Development by Example“ an: Reduzierung der enthaltenen Fehler und Zutrauen in den Code. („We hope to reduce our defects enough to move forward with confidence.“ (Beck 2002, S. 20)) Auf diese und weitere Definitionen soll im Folgenden noch näher eingegangen werden.

Die meisten Beschreibungen der Ziele und Vorzüge des Testens kreisen um dieselben vier Kernpunkte, wie im nächsten Abschnitt gezeigt wird. Eine wichtige Feststellung wird jedoch in keiner Quelle an die erste Stelle der Betrachtung gestellt, die für den Vergleich von Testwerkzeugen unabdingbar ist. Daher formuliert der Autor an dieser Stelle eine eigene Aussage über Software Testing, die den bekannten Zielen nicht widerspricht, sondern ihnen vorangestellt werden soll:

Testen als Maßnahme der Qualitätssicherung ist mit Kosten und Nutzen verbunden. Kosten-Nutzen-Optimierung ist das Ziel allen wirtschaftlichen Handelns.

Daraus folgt, dass für eine Entscheidung wie die zwischen Hamcrest und AssertJ betrachtet werden muss, inwiefern sie den Nutzen des Testens erhöhen oder die Kosten senken.

Thema dieser Arbeit ist der Vergleich von zwei Testwerkzeugen; vor einer solchen Detailfrage muss aber unbedingt darauf verwiesen werden, dass Kosten und Nutzen eines Tests überwiegend von den Fähigkeit des Entwicklers abhängen, der den Test schreibt (Qusef et al. 2011). Typischerweise werden Informatikstudenten an der Universität nicht mit Codebasen konfrontiert, die vom Umfang, Komplexität und Technologie mit produktiven Softwaresystemen vergleichbar sind (in Ermangelung eines treffenden deutschen Begriffs: Die Rede ist von „enterprise-level systems“). Insbesondere erlernen die meisten Studenten daher nicht die erforderlichen Ideen und Werkzeuge zum Testen derartiger Software, sondern müssen sich diese Fähigkeiten im Beruf aneignen (Samarthyam et al. 2017).

Aus eigener Erfahrung kann der Autor dieser Arbeit aus Projekten berichten, in denen hunderte bis tausende von Testfällen lediglich die Zusicherung `assertNotNull` enthielten – es wurde also nur

getestet, dass der Aufruf einer Methode überhaupt irgendetwas zurückgibt ohne Exceptions zu werfen. Der tatsächliche Nutzen solcher Tests ist nahe Null, aber sie tragen in Kundenprojekten zum Erreichen einer vereinbarten Kennzahl bei („Testabdeckung“), die lediglich den Anteil der durch Tests ausgeführten Zeilen oder Zweige im Code misst, ohne die Qualität der Tests zu berücksichtigen.

Auf der Kostenseite fallen schlecht strukturierte, unverständliche Tests ins Gewicht. Wenn sich im Falle eines Fehlschlags durch jemand anderen als den Autor des Tests nicht nachvollziehen lässt, was er eigentlich zu testen versucht, so ist nicht klar, ob der Test zurecht einen neuen Fehler im Programm anzeigt, oder ob der Test stattdessen an das veränderte Verhalten des Programms angepasst werden muss. Dadurch wird aufwendiges Reverse Engineering des Tests notwendig; in der Praxis wird der fehlschlagende Test dann mit hoher Wahrscheinlichkeit schlicht gelöscht (Engel 2018), sodass der Aufwand ihn zu schreiben umsonst war.

In der Summe tragen von unerfahrenen Entwicklern geschriebene Tests oft weniger Nutzen bei und sind schwerer zu warten. Daher haben die Fähigkeiten der Entwickler, ihre Motivation und die Unternehmenskultur in Bezug auf Testing einen weit größeren Einfluss auf Kosten und Nutzen von Tests als die Auswahl bestimmter Werkzeuge. Nachdem diese Tatsache angemessen betont wurde, kann im Folgenden vom Fähigkeitsaspekt abstrahiert werden, um die Kriterien für eine Technologieentscheidung zu erarbeiten.

2.2 Nutzen eines Tests

Der offensichtliche Nutzen eines Tests besteht darin, Fehler möglichst früh zu finden und zu beheben (Massol 2004, S. 66). Schon Programmieranfänger nutzen Testing zu diesem Zweck: Sie schreiben Code, von dem sie denken, dass er das erwünschte Verhalten zeigt, und führen ihn dann aus, um Fehler zu finden und zu beheben. Kommt Testen bei Anfängern noch Debugging gleich (Ammann und Offutt 2008, S. 8–9), wird es bei komplexeren Programmen schnell langwierig, eingefügten Code durch Ausführen des gesamten Programms zu testen. Deshalb schreiben Softwareentwickler automatische Tests, die den hinzugefügten Code direkt aufrufen und in Isolation vom Rest des Systems ausführen. Das Finden und Beheben von Fehlern auf diese Weise führt nicht zu Fehlerfreiheit der Software, aber es verringert das Fehlerrisiko (Ammann und Offutt 2008, S. 9).

Abgesehen vom Testen neu hinzugefügten Codes dienen automatisierte Tests aber auch als sogenannte Regression Tests. Als Regression Testing nach Änderungen an einem Programm wird das Ausführen von bestehenden Tests bezeichnet, die zeigen sollen, dass nicht versehentlich Fehler in den bereits bestehenden Code eingebaut wurden (Mishra und Tripathi 2017, S. 151). Regression Tests können nur automatische, nicht aber manuelle Tests sein, weil der Aufwand der wiederholten Ausführung nahezu null sein sollte (Ammann und Offutt 2008, S. 215). Tests, die ursprünglich für neuen Code geschrieben wurden, können anschließend ihre Bestimmung ändern und zu Regression Tests werden. Tatsächlich ist dies die Regel, aber Entwickler sollten auch bereit sein geschriebene Tests zu löschen, wenn sie sich als redundant erweisen (Beck 2002, S. 37). Software Testing ist ein kontinuierlicher Prozess der Qualitätssicherung (Moiz 2017, S. 68), der nicht nur im Hinzufügen von Tests besteht, sondern auch in ihrer Wartung oder Entfernung.

Eine weitere wichtige Aufgabe, die Testing heute erfüllt, nennt Charles. A. R. Hoare bereits auf der zuvor zitierten NATO-Konferenz von 1969: „You should convince yourself, or other people, [...] it will always work on any data.“ Die Beobachtung, dass ein Programm für eine Reihe sorgfältig ausgewählte Eingaben (Testfälle) reproduzierbar die erwarteten Ausgaben erzeugt, soll dem Entwickler das Zutrauen geben, dass das Programm immer und mit beliebigen Eingaben funktioniert.

Dieser Aspekt des Testens mutet wenig wissenschaftlich an, da er auf ein Gefühl des Entwicklers gegenüber seinem Programm abzielt, und nicht auf die vermeintlichen Absolutheit eines formalen

Beweises. Dennoch macht er die heutige Produktivität der Softwareentwicklung erst möglich: Das Vorhandensein einer umfassenden Suite hochwertiger Tests gibt dem Entwickler das nötige Vertrauen, bestehenden Code ändern und erweitern zu können (im Sinne von Refactoring und neuen Features); er führt nach jedem kleinen Änderungsschritt die Tests erneut aus um sicherzugehen, dass der bestehende Code wie bisher funktioniert (Massol 2004, S. 67). Wenn die gewünschte Änderung oder Erweiterung erfolgt ist, übergibt der Entwickler den Code der Continuous-Integration-Pipeline, wo die Änderungen aller beteiligten Entwickler zusammengeführt und wiederum mit denselben Tests überprüft werden. In vielen modernen Projekten erfolgt daraufhin eine automatische Auslieferung in die Produktivumgebung, sodass die Änderungen den Nutzern noch am selben Tag zur Verfügung stehen, ohne dass zuvor langwierig manuell getestet wird. Dieses Prinzip von Continuous Deployment / Continuous Delivery ist nur möglich, weil die Softwarehersteller auf die Aussagekraft der Tests vertrauen.

Zu guter Letzt sei erwähnt, dass programmierte Testfälle auch als Dokumentation des Codes dienen (Langr 2015, XI). Diagramme und Textdokumente können zwar die großen Zusammenhänge und Muster eines Softwaresystems anschaulich beschreiben. Je detaillierter aber versucht wird die Implementierung zu erläutern, desto weniger hilfreich sind diese klassische Formen der Dokumentation; zum Beispiel weil natürliche Sprache zur Ungenauigkeit neigt, der Autor der Dokumentation seine eigenen Missverständnisse einfließen lässt oder Teile vergisst, und weil die Beschreibung schnell veraltet und mit zunehmender Menge nicht mehr zuverlässig auf dem neuesten Stand gehalten werden kann. Ausführbare Testfälle dagegen sind immer aktuell, da sie andernfalls fehlschlagen und angepasst werden müssen, und sie beschreiben präzise das tatsächliche Verhalten des Codes. Sofern sie übersichtlich und verständlich geschrieben sind, erfüllen sie die Dokumentationsfunktion für Implementierungsdetails besser als jede andere Form.

2.3 Kosten eines Tests

Entwickler interagieren aus unterschiedlichen Anlässen mit einem Test; jede Interaktion bedeutet Aufwand, also Kosten. Um die verschiedenen Kostenfaktoren eines Tests und dadurch die mögliche Auswirkung der Toolauswahl zu verstehen, soll daher sein Lebenszyklus nachvollzogen werden:

Der Test wird zunächst geschrieben, was selbstverständlich Aufwand bedeutet. Je weniger Code geschrieben werden muss um denselben Nutzen zu erzeugen, desto geringer sind die Kosten (Engel 2018). Der Zeitaufwand zum Schreiben eines Tests hängt aber nicht nur von der Codemenge ab, sondern auch davon, ob die Testwerkzeuge intuitiv benutzbar und einprägsam sind. Andernfalls muss der Entwickler zusätzliche Zeit aufwenden, um die Benutzung der Testwerkzeuge (z.B. die Syntax und verfügbaren Funktionen der Assertion-Bibliothek) erneut nachzulesen und auszuprobieren.

Wird das getestete Modul später einem Refactoring unterzogen (also die interne Implementierung geändert unter Beibehaltung des Verhaltens und der Schnittstelle), führt der Entwickler anschließend sämtliche Tests aus, um zu verifizieren, dass sich das Verhalten des Codes tatsächlich nicht geändert hat. Schlägt dabei ein Test fehl, muss der Entwickler anhand der Ausgabe des Tests verstehen, welcher Teil des Codes sich nun nicht wie erwartet verhält, um den Fehler zu beheben. Daher ist eine leicht verständliche Fehlerausgabe anzustreben; fehlt diese, muss der Entwickler mehr Aufwand treiben, um das Problem zu identifizieren.

Im Gegensatz zum Refactoring ist bei einer Änderung des getesteten Moduls zu erwarten, dass Tests fehlschlagen. Im Falle einer Änderung der Schnittstelle treten Kompilierfehler auf, im Falle einer Verhaltensänderung werden Assertions fehlschlagen. So oder so müssen die Tests an die Änderung des Moduls angepasst werden (Langr 2015, S. 115). Wichtiger als die Fehlerausgabe ist hier, dass der Entwickler den Testcode versteht; der Test sollte möglichst klar aussagen, was er testet (Engel 2018). Gute Lesbarkeit des Testcodes reduziert daher die Wartungskosten des Tests.

In modernen Softwareprojekten werden sämtliche Tests vom Continuous Integration Server nach jedem Commit im Versionskontrollsystem ausgeführt; schlägt dann ein Test fehl, der gar nicht mit dem modifizierten Modul zusammen hängt, ist der Aufwand der Fehlersuche in der Regel am höchsten. In diesem Fall sind Aussagekraft der Fehlernachricht und Lesbarkeit des Tests besonders wichtig, weil der Entwickler zu diesem Zeitpunkt wahrscheinlich wenig über den Kontext des fehlschlagenden Tests weiß.

2.4 Vergleichskriterien für Assertion-Bibliotheken

Die vorangegangene Betrachtung ergibt sechs Kriterien für Vorteilhaftigkeit einer Testbibliothek:

1. Die Bibliothek ermöglicht, Fehler im Code zu finden.
2. Die Bibliothek erhöht das Vertrauen des Entwicklers in den Code.
3. Die Bibliothek hilft dem Entwickler, seine Absicht zu dokumentieren.
4. Die Bibliothek regt dazu an, gut lesbaren Testcode zu schreiben.
5. Die Bibliothek stellt verständliche Fehlerausgaben zur Verfügung.
6. Die Bibliothek ist leicht benutzbar und einprägsam.

Wie eingangs erwähnt, erhöhen weder Hamcrest noch AssertJ die Mächtigkeit von JUnit. Jeder Fehler, der sich mit einer der Bibliotheken finden lässt, kann auch mit den anderen gefunden werden. Aus diesem Grund sind die Punkte 1 und 2 als Vergleichskriterium hinfällig.

Bei genauerer Betrachtung stellt sich heraus, dass Punkt 3 und 4 synonym sind: Eine klare Dokumentation der Absicht durch Testcode ist dann gegeben, wenn der Testcode gut lesbar ist.

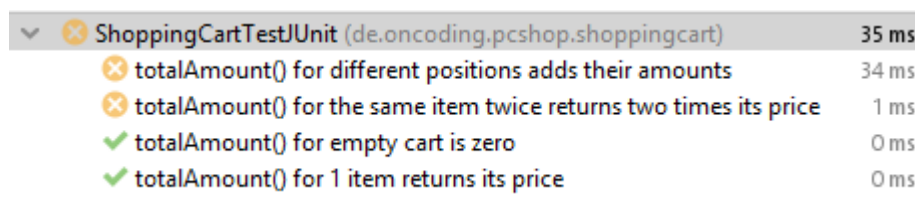
Daher reduziert sich die Liste der Vergleichskriterien auf die folgenden drei:

- Lesbarkeit
- Verständliche Fehlerausgaben
- Nutzerfreundlich / einfach zu erlernen

Anhand dieser Kriterien sollen im nächsten Kapitel Hamcrest und AssertJ bewertet werden.

3. Einfache Tests mit JUnit, Hamcrest und AssertJ

Die Codebeispiele für die vorliegende Arbeit sind der Codebasis eines fiktiven Onlineshops entnommen. Die Beispiele in diesem Kapitel sind dabei einfach gewählt, in dem Sinne, dass die zu testende Klasse Einkaufswagen („ShoppingCart“) keine weiteren Klassen aufruft. Ein Einkaufswagen wird leer initialisiert und kann mehrere Positionen aufnehmen, die aus Produktname, Menge und Stückpreis bestehen. Die Methode `totalAmount` rechnet daraufhin den Gesamtwert der im Wagen befindlichen Positionen aus. Das Verhalten des Einkaufswagens wird über vier Testfälle abgedeckt (Abbildung 1: Testfälle für ShoppingCart Abbildung 1), von denen im Beispielprojekt jeder jeweils einmal mit JUnit-, Hamcrest- und AssertJ-Assertion implementiert ist.



The screenshot shows the test results for the ShoppingCartTestUnit class. The table lists four test cases with their status and execution time.

Test Case	Time
totalAmount() for different positions adds their amounts	34 ms
totalAmount() for the same item twice returns two times its price	1 ms
totalAmount() for empty cart is zero	0 ms
totalAmount() for 1 item returns its price	0 ms

Abbildung 1: Testfälle für ShoppingCart

Diese Art von Klasse eignet sich gut zur Erläuterung grundlegender Konzepte des Testings, da sie von Natur aus in Isolation getestet werden kann. Die Mehrzahl von Klassen in Enterprise-Softwaresystemen ruft aber andere Klassen auf, verwendet ein Framework oder nutzt externe Ressourcen wie Dateien, Datenbanken oder Webservices, wodurch Testen anspruchsvoller wird. Die Beispiele im nachfolgenden Kapitel werden deshalb eine Spring-Data-Repository-Klasse testen, die Daten aus einer Datenbank lädt.

3.1 Aufbau eines Tests

Der erste Testfall (Abbildung 2) prüft das denkbar einfachste Verhalten des Einkaufswagens: Wenn er initialisiert wird und noch keine Waren hinzugefügt wurden, soll die `totalAmount`-Methode den Wert Null zurückgeben.

```
@Test
fun `totalAmount() for empty cart is zero`() {
    // given
    val cart = ShoppingCart()

    // when
    val totalAmount = cart.totalAmount()

    // then
    assertEquals(BigDecimal.ZERO, totalAmount)
}
```

Abbildung 2: Aufbau eines Tests mit given, when, then

Dieses Beispiel demonstriert die Struktur eines guten Tests: Der Test hat zunächst einen beschreibenden Namen, der deutlich macht, welche Funktionalität unter welchen Bedingungen hier getestet wird. Anfänger machen dabei häufig den Fehler, den Test nur mit dem Namen der getesteten Methode und dem Präfix `test` zu benennen, im gegebenen Beispiel also `testTotalAmount`. Dieser

Name sagt nicht nur wenig aus, er suggeriert auch noch, dass es nur einen einzigen Testfall für die Methode `totalAmount` gibt oder geben sollte. Ein guter Name dagegen charakterisiert den Fall („empty cart“) und die Erwartung („zero“) in Kürze.

Kotlin erlaubt die Verwendung von Leerzeichen in Methodennamen, solange der Methodenname in Backticks gesetzt wird. In der Kotlin-Community ist Konvention, diese Art von Methodennamen ausschließlich für Tests zu verwenden, da Testmethoden nur vom Testframework und nicht von normalem Code aus aufgerufen werden. Für Testnamen ist diese Schreibweise aber umso beliebter, da sich so gut lesbare Testnamen formulieren lassen. Java bietet diese Möglichkeit nicht, Testnamen können aber mit Hilfe von Unterstrichen gebildet werden, z.B. `totalAmount_forEmptyCart_isZero`. Unterstriche in Methodennamen verletzen die übliche Konvention von Methodennamen in Java, können aber für Testmethoden verwendet werden um lange Methodennamen optisch zu gliedern und lesbar zu machen.

Der Test selbst sollte dann drei deutlich getrennte Abschnitte vorweisen, die dem Muster „given – when – then“ folgen (Vance 2013, S. 48). Ebenfalls verbreitet ist die Benennung „arrange – act – assert“, wegen der markanten Alliteration als „Triple-A“ bekannt (Langr 2015, S. 35; Beck 2002, S. 97–98; Pinnegar). Inhaltlich bedeutet diese Variante dasselbe.

Im „when“-Teil des Tests erfolgt der Aufruf des zu testenden Produktivcodes. Anschließend wird im „then“-Teil ausgewertet, ob der Aufruf das erwartete Ergebnis geliefert hat. Der „given“-Teil bereitet alle nötigen Objekte vor, die für den „when“-Teil benötigt werden, also Eingabeparameter und ggf. Instanzen der zu testenden Klassen mit einem wohldefinierten Anfangszustand. Obwohl „given“ der erste Teil des Tests ist und zuerst ausgeführt wird, bietet es sich beim Schreiben des Tests an mit „when“ zu beginnen: Hat der Programmierer den zu testenden Aufruf und die beabsichtigten Assertions erst niedergeschrieben, kann er leicht rückwärts folgern, welches Setup dazu nötig ist.

Das Setup der Testdaten („given“) kann wahlweise im Testfall selbst oder in einer gesonderten Setup-Methode erfolgen, die mit `@Before` oder `@BeforeClass` annotiert sind, um das Setup für mehrere Testfälle wiederzuverwenden. Manchmal ist eine vierte Testphase („tear down“) erwünscht, um Seiteneffekte des Tests zu bereinigen, z.B. um gespeicherte Daten aus der Datenbank zu entfernen (Vance 2013, S. 48).

Die beschriebene Struktur wird deshalb als „gut“ bewertet, weil sie zwei wichtige positive Eigenschaften aufweist:

Ein Test in dieser Form sagt präzise aus, welches Verhalten des Produktivcodes er testet; also welche Methode aufgerufen wird und unter welchen Bedingungen. (Hier sei darauf hingewiesen, dass die aufgerufene Methode intern durchaus weitere Methoden und Klassen aufrufen darf. Getestet wird nicht eine bestimmte Methode, sondern ein Verhalten, für das die aufgerufene Methode die Schnittstelle nach außen verkörpert.) Dies ist eine Voraussetzung um im Falle eines fehlschlagenden Tests zu wissen, welches Verhalten des Produktivcodes gerade fehlerhaft ist, und um zu überprüfen ob der Test korrekt implementiert ist, d.h. ob der Test tatsächlich prüft, was der Name des Tests behauptet.

Außerdem ist ein Test in dieser Form lesbar und verständlich. Der Leser kann mit dem Auge unmittelbar den „when“-Teil des Tests ausfindig machen und daran sehen, welche Methode des Produktivcodes hier getestet wird. Von diesem Punkt aus kann der Leser abwärts schauen, um die Erwartungen an das Ergebnis zu verstehen, und aufwärts, um das vorgenommene Test-Setup zu sehen.

Wer Wert auf sauberen, lesbaren Code legt, versucht Kommentare in der Regel zu vermeiden und stattdessen mit Hilfe von sprechenden Variablennamen und extrahierten Methoden selbsterklärenden

Code zu schreiben (Martin 2009). Die Kommentarzeilen „given“, „when“ und „then“ in Tests werden aber auch von Verfechtern sparsamer Kommentare genutzt, um den wichtigen „when“-Teil auch in längeren Tests von 20 bis 30 Zeilen sofort ins Auge springen zu lassen.

3.2 Einfacher Testfall: Lesbarkeit der Assertions

Die nächsten Codebeispiele zeigen denselben einfachen Testfall, einmal implementiert mit einer nativen JUnit-Assertion (Abbildung 3), dann mit Hamcrest (Abbildung 4) und mit AssertJ (Abbildung 5).

```
@Test
fun `totalAmount() for 1 item returns its price`() {
    // given
    val cart = ShoppingCart()
    cart.add(ShoppingCartPosition(
        productName = "Ryzen 7 2700X",
        amount = 1,
        priceInEuros = BigDecimal( val: 300)
    ))

    // when
    val totalAmount : BigDecimal = cart.totalAmount()

    // then
    assertEquals(BigDecimal( val: 300), totalAmount)
}
```

Abbildung 3: Einfacher Test mit JUnit

```
@Test
fun `totalAmount() for 1 item returns its price`() {
    // given
    val cart = ShoppingCart()
    cart.add(ShoppingCartPosition(
        productName = "Ryzen 7 2700X",
        amount = 1,
        priceInEuros = BigDecimal( val: 300)))

    // when
    val totalAmount : BigDecimal = cart.totalAmount()

    // then
    assertThat(totalAmount, equalTo(BigDecimal( val: 300)))
}
```

Abbildung 4: Einfacher Test mit Hamcrest

```

@Test
fun `totalAmount() for 1 item returns its price`() {
    // given
    val cart = ShoppingCart()
    cart.add(ShoppingCartPosition(
        productName = "Ryzen 7 2700X",
        amount = 1,
        priceInEuros = BigDecimal( val: 300)))

    // when
    val totalAmount : BigDecimal = cart.totalAmount()

    // then
    assertThat(totalAmount).isEqualTo(BigDecimal( val: 300))
}

```

Abbildung 5: Einfacher Test mit AssertJ

Die drei Assertions vergleichen jeweils den ausgerechneten Wert des Einkaufswagens mit dem erwarteten Wert von 300. Die native JUnit-Assertion erhält dazu die zu vergleichenden Werte als Parameter, während die Vergleichsoperation in der Assertion `assertEquals` selbst codiert ist. Die Hamcrest-Assertion erhält den zu prüfenden Wert zusammen mit dem Matcher `equalTo`, der AssertJ-Assertion wird nur der zu prüfende Wert übergeben und anschließend die `isEqualTo`-Methode darauf aufgerufen.

Hier ist zu sehen, dass sich die Hamcrest- und AssertJ-Varianten jeweils fließend wie ein englischer Satz in natürlicher Sprache lesen lassen: „Assert that totalAmount is equal to BigDecimal(300)“. Der Unterschied zwischen Hamcrest und AssertJ ist minimal. In der Hamcrest-Variante lässt sich das `equalTo` auf Wunsch in einen weiteren Matcher namens `is` einfügen, der keine Funktion hat, aber eben dieselbe Lesbarkeit herstellt wie in AssertJ.

Die native JUnit-Assertion klingt vorgelesen etwas holpriger, ist aber in diesem trivialen Fall ebenfalls gut genug lesbar.

3.3 Einfacher Testfall: Nutzerfreundlichkeit und Lernkurve

An der JUnit-Assertion fällt zunächst die Reihenfolge der Argumente auf: Als erstes soll der erwartete Wert und als zweites der tatsächlich errechnete Wert übergeben werden. Diese Reihenfolge widerspricht der menschlichen Gewohnheit beim Sprechen („Ich erwarte, dass der Gesamtwert gleich 300 ist“ und nicht „dass 300 gleich dem Gesamtwert ist“), erfordert also Kenntnis und etwas Gewöhnung. Dazu kommt, dass beide Argumente vom selben Typ (hier String) sind; der Compiler kann also eine Vertauschung nicht erkennen, und bei erfolgreichem Test spielt die Vertauschung keine Rolle, weshalb sie nicht auffällt. Erst wenn der Test fehlschlägt, stellt die Fehlermeldung den erwarteten und den tatsächlichen Wert vertauscht dar, was den Entwickler beim Finden des Problems irreführt.

Mit Hamcrest und AssertJ ist eine versehentliche Vertauschung nicht möglich, da nicht zwei Argumente mit selbem Typ erwartet werden. In Hamcrest muss die Erwartung in Form eines Matchers ausgedrückt werden, der vom Typ `Matcher` ist, auch wenn er dann einen String umschließt. Die Vertauschung des

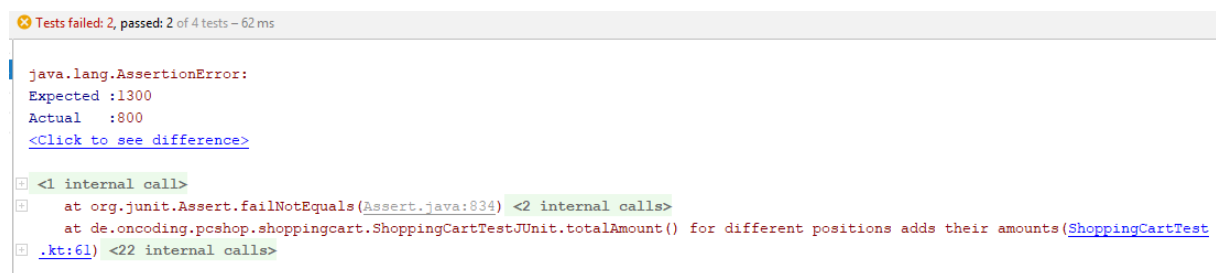
ersten und zweiten Parameters hätte einen Kompilierfehler zur Folge. Der AssertJ-Assertion wird dagegen überhaupt nur ein Parameter übergeben, und die Erwartung durch Aufruf einer Methode auf dem Assertion-Objekt formuliert. Bei beiden Bibliotheken wäre es zwar technisch möglich, die eigentlichen String-Werte zu vertauschen; der erwartete String wird aber jeweils in ein Konstrukt eingeschlossen, das namentlich erklärt die Erwartung zu beinhalten.

3.4 Einfacher Testfall: Verständlichkeit und Informationsgehalt der Fehlerausgaben

Um die Fehlerausgaben in allen drei Varianten vergleichen zu können, ist die `totalAmount`-Methode im Beispielcode nicht ganz richtig implementiert: Die Multiplikation des Stückpreises mit der Anzahl fehlt, deshalb schlagen Testfälle mit mehreren Stück desselben Produkts im Beispielprojekt fehl.

Die Fehlerausgabe von JUnit soll hier als Ausgangspunkt dienen (Abbildung 6): Dem Leser wird angezeigt, dass ein `AssertionError` aufgetreten ist, wobei der erwartete Wert 1300 beträgt und der tatsächliche Wert 800.

In allen drei Fällen gibt die Assertion nicht nur eine Nachricht aus, sondern wirft technisch gesehen einen Error, sodass in der Ausgabe auch ein Stacktrace zu sehen ist. An diesem lässt sich ablesen, dass der Fehler in einer bestimmten Zeile und Klasse aufgetreten ist; in der Entwicklungsumgebung kann der Entwicklung durch Klick in den Stacktrace direkt zur fehlgeschlagenen Assertion springen und beginnen das Problem zu untersuchen.



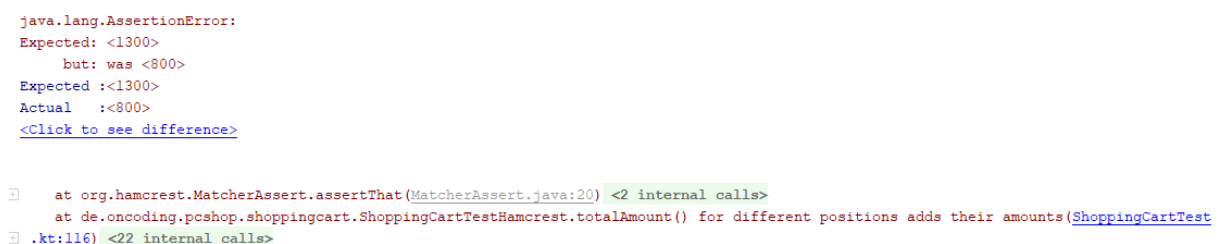
```
Tests failed: 2, passed: 2 of 4 tests - 62 ms

java.lang.AssertionError:
Expected :1300
Actual   :800
<Click to see difference>

<1 internal call>
  at org.junit.Assert.failNotEquals(Assert.java:834) <2 internal calls>
  at de.oncoding.pcshop.shoppingcart.ShoppingCartTestJUnit.totalAmount() for different positions adds their amounts(ShoppingCartTest.kt:61) <22 internal calls>
```

Abbildung 6: Fehlerausgabe von `assertEquals` (JUnit)

Die Hamcrest-Assertion (Abbildung 7) enthält dieselben Informationen wie die JUnit-Assertion, allerdings zweifach. Zusätzlich zu der Ausgabe des erwarteten und tatsächlichen Werts baut hier ein Textgenerator einen einfachen Satz: „Expected 1300, but was 800.“ Diese Ausgabe ist flüssig zu lesen, in dem vorliegenden trivialen Fall aber eher unnötig.



```
java.lang.AssertionError:
Expected: <1300>
but: was <800>
Expected :<1300>
Actual   :<800>
<Click to see difference>

  at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20) <2 internal calls>
  at de.oncoding.pcshop.shoppingcart.ShoppingCartTestHamcrest.totalAmount() for different positions adds their amounts(ShoppingCartTest.kt:116) <22 internal calls>
```

Abbildung 7: Fehlerausgabe von `equalTo` (Hamcrest)

AssertJ gibt in diesem Fall Erwartung und tatsächlichen Wert in derselben Form aus wie JUnit, auch wenn technisch eine andere Fehlerklasse geworfen wird (Abbildung 8).


```
org.junit.ComparisonFailure:
Expected :1300
Actual   :800
<Click to see difference>

<3 internal calls>
    at de.oncoding.poshop.shoppingcart.ShoppingCartTestAssertJ.totalAmount() for different positions adds their amounts(ShoppingCartTest
    .kt:171) <22 internal calls>
```

Abbildung 8: Fehlerausgabe von `assertThat.isEqualTo` (`AssertJ`)

3.5 Schlussfolgerung

Die Betrachtung der ersten Beispiele hat gezeigt, dass Hamcrest und AssertJ für einen simplen `equals`-Vergleich keine wesentlichen Unterschiede aufweisen. Die native JUnit-Assertion ist zwar nicht Teil des angestrebten Vergleichs dieser Arbeit, da JUnit keine komplexen Assertions zur Verfügung stellt. Im Interesse des Erkenntnisgewinns sei jedoch festgehalten, dass die Gefahr der Vertauschung sowie die etwas schlechtere Lesbarkeit bereits in diesem einfachen Fall dafürsprechen, eine der Assertion-Bibliotheken gegenüber `assertEquals` zu bevorzugen.

4. Komplexe Tests mit JUnit, Hamcrest und AssertJ

Nachdem für das vorherige Kapitel ein Lehrbuchbeispiel einer zu testenden Klasse konstruiert wurde, soll nun die Verwendung von Assertions unter realistischen Bedingungen demonstriert werden. Getestet werden soll die Klasse `ProductRepository`, die als Teil einer SpringBoot-Applikation Zugriffsmethoden für die Datenbanktabelle „product“ zur Verfügung stellt.

```
@ActiveProfiles( ...value: "test")
@RunWith(SpringJUnit4ClassRunner::class)
@DataJpaTest
class ProductRepositoryTestJUnit {

    @Autowired
    private lateinit var productRepository: ProductRepository

    @Test
    fun `saves product in database and loads by id`() {
        // given
        val productToSave = Product(
            id = "1",
            productName = "Ryzen 7 2700X",
            manufacturer = "AMD"
        )

        // when
        productRepository.save(productToSave)

        // then
        val loadedProduct : Product = productRepository.getOne( id: "1")
        assertEquals(productToSave, loadedProduct)
    }
}
```

Abbildung 9: Repository-Test mit Spring Boot

Die Testklasse (Abbildung 9) wird anstatt des normalen JUnit-Runners mit dem `SpringJUnit4ClassRunner` ausgeführt. Diese Annotation übergibt die Kontrolle über den Test an das Spring-Framework, das den Spring-Container startet und Instanzen von Komponenten-Klassen erzeugt, sogenannte Beans. Das `ProductRepository` ist eine solche Bean, die von Spring eine Datenbankverbindung erhält und zur Dependency Injection über `@Autowired` zur Verfügung gestellt wird. So muss in der Testklasse selbst keine Datenbankverbindung und keine Instanz des Repositories erzeugt werden.

Die Spring-Anwendung startet im Profil „test“, sodass die Konfiguration der Datenbankverbindung aus der Datei `application-test.yml` statt aus der primären `application.yml` gelesen wird. Der Test ist außerdem als `@DataJpaTest` annotiert, was zur Folge hat, dass während des Tests geschriebene Daten nach dem Test automatisch zurückgerollt und nicht in die Datenbank committed werden.

Der dargestellte Test selbst ist übersichtlich, da das Spring-Framework das anspruchsvolle Setup hinter den Kulissen vornimmt. Der Test übergibt dem Repository eine Produkt-Instanz, um diese in der Datenbank zu speichern. Die Methoden `save` und `findOne` sind Teil des Spring-Frameworks, daher beabsichtigt der Test nicht Fehler in der Implementierung der Methoden aufzudecken; er stellt aber sicher, dass die Entitätsklasse `Produkt` so implementiert ist, dass sie von der Java Persistence Api (JPA) auf die bestehende Datenbanktabelle „`product`“ abgebildet werden kann. Stimmen Spaltennamen oder Datentypen nicht überein, wird der Test fehlschlagen und damit auf kritische Fehler hinweisen.

Dieses erste Beispiel zeigt, dass Tests für echten Produktivcode in der Regel komplizierter sind als im vorigen Kapitel gezeigt, aber durchaus mit denselben einfachen Assertions auskommen können. Die Komplexität eines Tests und die Komplexität der verwendeten Assertions sind voneinander unabhängig. Das vorliegende Kapitel hat in erster Linie die Absicht, komplexere Assertions von Hamcrest und AssertJ zu vergleichen. Der gleichzeitig komplexer gewählte Testkontext ist zu diesem Zweck nicht notwendig, dieselben Assertions könnten auch weiterhin mit lehrbuchartigen Beispieltests genutzt werden; die realistischen Tests sollen aber verdeutlichen, wie das gezeigte in der Praxis verwendet werden kann.

4.1 Tests mit erwarteter Collection

In nächsten Testfall (Abbildung 10) ist der erwartete Rückgabewert eine Liste von Elementen. Die naive Implementierung einer Assertion konstruiert eine erwartete Liste und vergleicht diese mit dem Rückgabewert mittels `assertEquals`.

```
@Test
fun `loads all products from database`() {
    // given
    val product1 : Product = productRepository.save(Product(
        id = "1",
        productName = "Ryzen 7 2700X",
        manufacturer = "AMD"
    ))
    val product2 : Product = productRepository.save(Product(
        id = "2",
        productName = "Core i9 9900K",
        manufacturer = "Intel"
    ))

    // when
    val products : List<Product> = productRepository.findAll()

    // then
    assertEquals(listOf(product1, product2), products)
}
```

Abbildung 10: Vergleich zweier Listen mit `equals`

Diese Variante sollte nicht verwendet werden. Der Test wird fehlschlagen, wenn die Elemente in den Listen eine unterschiedliche Reihenfolge aufweisen, obwohl die Reihenfolge für den Testfall keine Rolle spielen sollte. Auch bei Konvertierung der Listen in Sets, um die Reihenfolge zu ignorieren, bleibt ein Problem: Beim Fehlschlagen des Tests enthält die Fehlernachricht in Java lediglich die Adressen der beiden Objekte auf dem Heap. Kotlin überschreibt die `toString`-Methode der Collections

hilfreicherweise und gibt die Elemente der Sets aus (Abbildung 11), doch ab einer gewissen Länge kann der Entwickler nicht mehr mit dem Auge erkennen, inwiefern sich die erwartete und tatsächliche Liste unterscheiden.

```
java.lang.AssertionError:  
Expected :[Product(id=1, productName=Ryzen 7 2700X, manufacturer=AMD), Product(id=2, productName=Core i9 9900K, manufacturer=Intel)]  
Actual   :[Product(id=1, productName=Ryzen 7 2700X, manufacturer=AMD)]
```

Abbildung 11: Ausgabe der erwarteten und tatsächlichen Liste in Kotlin

Um dieses Problem unter Verwendung reiner JUnit-Bordmittel zu umgehen, gibt es eine Hand voll Möglichkeiten. So kann einzeln zugesichert werden, dass die erwarteten Elemente in der zurückgegebenen Liste enthalten sind (Abbildung 12). Um auszuschließen, dass die Liste unerwartet noch zusätzliche Elemente beinhaltet, muss dann außerdem die erwartete Länge der Liste geprüft werden.

```
// then  
assertTrue(products.contains(product1))  
assertTrue(products.contains(product2))  
assertEquals( expected: 2, products.size)
```

Abbildung 12: Zusicherung der einzelnen Elemente und der Länge

Auf diese Weise ist bei Fehlschlag zwar die Fehlernachricht nicht sehr aussagekräftig, aber die Zeilenangabe der ersten fehlgeschlagenen Assertion grenzt das Problem gut ein.

Besser wird die Fehlerausgabe, wenn der Entwickler jeder Assertion noch eine Erläuterung als Argument übergibt, die im Fehlerfall angezeigt wird (Abbildung 13).

```
// then  
assertTrue( message: "Loaded products should contain product 1", products.contains(product1))  
assertTrue( message: "Loaded products should contain product 2", products.contains(product2))  
assertEquals( expected: 2, products.size)
```

Abbildung 13: Eigene Fehlerausgaben für Assertions

```
java.lang.AssertionError: Loaded products should contain product 2  
    <2 internal calls>  
        at de.oncoding.pcshop.product.ProductRepositoryTestJUnit.loads all products from database (2) (ProductRepositoryTest.kt:79)  
    <8 internal calls>
```

Abbildung 14: Ausgabe eigener Fehlernachrichten

Individuelle Fehlernachrichten können präziser sein als jede generierte Fehlernachricht (Abbildung 14). Diese Möglichkeit steht bei allen Assertions offen, auch Hamcrest- und AssertJ-Assertions erlauben dies. Präzise Fehlernachrichten zu formulieren fordert aber einen gewissen Aufwand, der bei tausenden von Assertions in einer Codebasis durchaus ins Gewicht fällt. Daher ist es zu bevorzugen, wenn Bibliotheken wie Hamcrest und AssertJ ausreichend gute Fehlernachrichten automatisch generieren.

Die oben gezeigten Varianten der Assertion haben auch den Nachteil, dass die erforderliche Codemenge linear mit der Menge der Elemente in der Liste wächst, daher sind sie für längere Listen nicht praktikabel. Eine Alternative ist die Implementierung einer Helfermethode, die den gewünschten Vergleich ausführt (Abbildung 15).

```

@Test
fun `loads all products from database (3)`() {
    // given
    val product1 : Product = productRepository.save(Product(
        id = "1",
        productName = "Ryzen 7 2700X",
        manufacturer = "AMD"
    ))
    val product2 : Product = productRepository.save(Product(
        id = "2",
        productName = "Core i9 9900K",
        manufacturer = "Intel"
    ))

    // when
    val products : List<Product> = productRepository.findAll()

    // then
    assertTrue( message: "Loaded products should contain exactly product 1 and 2",
        containsExactlyInAnyOrder(products, product1, product2)
    )
}

```

Abbildung 15: Assertion mit eigener Vergleichsmethode

Auf diese Weise können beliebige komplexe Vergleiche vorgenommen werden, ohne den Test selbst lang werden zu lassen. Mit geeignet gewähltem Methodennamen ist die Lesbarkeit des Tests sehr gut. Die Helfermethoden sind idealerweise so generisch geschrieben, dass sie wiederverwendet werden können. Das gezeigte Beispiel leidet allerdings wieder an wenig hilfreichen Fehlerausgaben, da der Leser nicht erkennen kann, welches der erwarteten Elemente fehlt oder welches Element zu viel ist.

Ergänzt der Entwickler seine generischen Prüfmethoden um die Generierung präziser Fehlernachrichten, ist er bereits auf dem Weg zu erschaffen, was Hamcrest und AssertJ sind: Eine Assertion-Bibliothek wiederverwendbarer Matcher.

4.2 Vergleich der Lesbarkeit von komplexen Assertions

Abbildung 16 und Abbildung 17 zeigen den gleichen Test wie zuvor, aber umgesetzt mit Hamcrest bzw. AssertJ.

```

@Test
fun `loads all products from database`() {
    // given
    val product1 : Product = productRepository.save(Product(
        id = "1",
        productName = "Ryzen 7 2700X",
        manufacturer = "AMD"
    ))
    val product2 : Product = productRepository.save(Product(
        id = "2",
        productName = "Core i9 9900K",
        manufacturer = "Intel"
    ))

    // when
    val products : List<Product> = productRepository.findAll()

    // then
    assertThat(products, containsInAnyOrder(product1, product2))
}

```

Abbildung 16: containsInAnyOrder mit Hamcrest

```

@Test
fun `loads all products from database`() {
    // given
    val product1 : Product = productRepository.save(Product(
        id = "1",
        productName = "Ryzen 7 2700X",
        manufacturer = "AMD"
    ))
    val product2 : Product = productRepository.save(Product(
        id = "2",
        productName = "Core i9 9900K",
        manufacturer = "Intel"
    ))

    // when
    val products : List<Product> = productRepository.findAll()

    // then
    assertThat(products).containsExactlyInAnyOrder(product1, product2)
}

```

Abbildung 17: containsExactlyInAnyOrder mit AssertJ

In beiden Fällen ist festzustellen, dass sich die Assertion sehr gut lesen lässt: „Assert that ‚products‘ contains (exactly) in any order product1, product2“. Das Komma zwischen Erwartung und Matcher in Hamcrest irritiert im Lesefluss etwas, die schließende Klammer und der Punkt in AssertJ entsprechen mehr der wörtlichen Rede, aber der Unterschied ist als minimal zu bewerten.

Die Formulierung „exactly“ in AssertJ weist darauf hin, dass es sich nicht um eine einfache „contains“-Operation handelt. Es ist also keine hinreichende Bedingung, dass das Ergebnis alle erwarteten Elemente enthält; es darf auch keine zusätzlichen Elemente enthalten. AssertJ besitzt auch die einfache `containsInAnyOrder`-Methode, die weniger strikt ist.

Hamcrests `containsInAnyOrder` erlaubt keine zusätzlichen Elemente, der Name des Matchers ist also schwächer als die tatsächlich geprüfte Bedingung. Um das einfache „contains“ zu prüfen, besitzt Hamcrest den Matcher `hasItems`. Die Benennungen in AssertJ sind treffender, hier hat möglicherweise die jüngere Bibliothek einen Schönheitsfehler der älteren behoben. Dies betrifft aber nur genau den `containsInAnyOrder`-Fall, ansonsten haben Hamcrest-Matcher ebenfalls treffende Namen. Daher soll die Lesbarkeit der Assertions als Resultat dieser Betrachtung als gleichwertig angesehen werden.

4.3 Fehlernachrichten bei komplexen Assertions

Schlägt der oben gezeigte Hamcrest-Test fehl (Abbildung 16), lautet die Fehlermeldung (Abbildung 18) sinngemäß: Erwartet wurde ein „Iterable“ mit den Elementen Produkt1 und Produkt2, aber kein Element entspricht Produkt2.

```
java.lang.AssertionError:
Expected: iterable over [<Product(id=1, productName=Ryzen 7 2700X, manufacturer=AMD)>, <Product(id=2, productName=Core i9 9900K,
manufacturer=Intel)>] in any order
but: No item matches: <Product(id=2, productName=Core i9 9900K, manufacturer=Intel)> in [<Product(id=1, productName=Ryzen 7 2700X,
manufacturer=AMD)>]

] at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20) <2 internal calls>
at de.oncoding.poshop.product.ProductRepositoryTestHamcrest.loads all products from database(ProductRepositoryTest.kt:138)
] 20 internal calls
```

Abbildung 18: Fehlerausgabe von `containsInAnyOrder` (Hamcrest)

Der AssertJ-Test (Abbildung 17) zeigt dagegen die Fehlermeldung: „Erwartet wurde, dass [Produkt1] in beliebiger Reihenfolge exakt [Produkt1, Produkt2] enthält. Konnte aber [Produkt2] nicht finden.“ (Abbildung 19)

```
java.lang.AssertionError:
Expecting:
  <[Product(id=1, productName=Ryzen 7 2700X, manufacturer=AMD)]>
to contain exactly in any order:
  <[Product(id=1, productName=Ryzen 7 2700X, manufacturer=AMD),
    Product(id=2, productName=Core i9 9900K, manufacturer=Intel)]>
but could not find the following elements:
  <[Product(id=2, productName=Core i9 9900K, manufacturer=Intel)]>

at de.oncoding.poshop.product.ProductRepositoryTestAssertJ.loads all products from database(ProductRepositoryTest.kt:176)
```

Abbildung 19: Fehlerausgabe von `containsExactlyInAnyOrder` (AssertJ)

An der Hamcrest-Ausgabe irritiert zunächst kurz die Erwähnung des Iterable-Interfaces. Jüngere Java-Entwickler sind unter Umständen noch niemals Iteratoren und Iterables begegnet, da modernere Java-Versionen For-Each-Schleifen und Streams zur Verfügung stellen, sodass die sperrige Iterator-Syntax praktisch nicht mehr verwendet wird.

Abgesehen von diesem Detail ist die Ausgabe von AssertJ vor allem Aufgrund der Zeilenumbrüche besser lesbar. Außerdem enthält die Hamcrest-Ausgabe nur die erwartete Collection und die Differenz, AssertJ gibt zusätzlich noch die gesamte tatsächliche Collection aus. Bei umfangreicheren Collections lässt sich so schneller nachvollziehen, wie die Differenz zustande kommt; etwa wenn ein erwartetes Element zwar vorhanden ist, aber mit einem einzelnen abweichenden Wert, zum Beispiel einem abweichenden Zeitstempel.

Die Fehlerausgabe von Hamcrest ist insgesamt betrachtet also schon sehr hilfreich, die Ausgabe von AssertJ aber noch ein Stück weiterentwickelt.

4.4 Erwartete Exceptions

Oft soll auch getestet werden, dass der Produktivcode in einem Szenario eine bestimmte Exception wirft. Gerade im Test-First-Entwicklungsstil ist ein solcher Test die einzige Möglichkeit, um im Produktivcode überhaupt Exceptions auslösen zu können.

Der Test soll also fehlschlagen, falls die erwartete Exception nicht geworfen wird. Früher musste dazu ein try-catch-Block verwendet werden (Abbildung 20), der optisch vom eigentlichen Test ablenkt.

```
fun `throws JpaObjectRetrievalFailureException when entity not in db`() {  
    // given  
  
    // when  
    try {  
        productRepository.getOne( id: "non-existing-id")  
    } catch (e: Exception) {  
        // then  
        assertTrue(e is JpaObjectRetrievalFailureException)  
    }  
}
```

Abbildung 20: Erwartete Exception mit try-catch

Daher wurde in TestNG die Annotation für erwartete Exception eingeführt und mit Version 4.7 in JUnit 4 übernommen (Abbildung 21). Diese Schreibweise ist angenehm kompakt, beschränkt sich aber auf die Überprüfung des Typs der Exception.

```
@Test(expected = JpaObjectRetrievalFailureException::class)  
fun `throws JpaObjectRetrievalFailureException when entity not in db`() {  
    // given  
  
    // when  
    productRepository.getOne( id: "non-existing-id")  
  
    // then  
}
```

Abbildung 21: Erwartete Exception per Annotation

AssertJ bietet dagegen eine Assertion an, die den Aufruf des Produktivcodes als Lambda-Ausdruck entgegennimmt und dann verschiedene Zusicherungen über die geworfene Exception zulässt (Abbildung 22).

```
@Test
fun `throws JpaObjectRetrievalFailureException when entity not in db`() {
    // given

    // when
    assertThatThrownBy {
        productRepository.getOne( id: "non-existing-id")
    } // then
    }.isInstanceOf(JpaObjectRetrievalFailureException::class.java)
    .hasMessage("Entity not found")
}
```

Abbildung 22: Erwartete Exception in AssertJ

Um solche Zusicherungen wie den Inhalt der Exception-Message mit Hamcrest anzugeben, muss die Exception wie oben gezeigt über einen try-catch-Block aufgefangen und mit den üblichen Assertions geprüft werden. Hier zeigt sich, dass AssertJ als die modernere Bibliothek ein neues Java-Feature zur Verbesserung der Lesbarkeit einsetzt.

4.5 Nutzerfreundlichkeit und Lernkurve

Während JUnit selbst nur eine Hand voll Assertions zur Verfügung stellt, die der Entwickler schnell lernt und nicht vergisst, sind Hamcrest und AssertJ sehr umfangreich. Benötigt der Entwickler für einen Testfall in Hamcrest einen anderen als die meistgenutzten Matcher, muss er deshalb oft nach dem zu seiner Absicht passenden Matcher suchen – in der Dokumentation, im bestehenden Code oder in Internetforen wie Stackoverflow. Das unterbricht den Arbeitsfluss und verlangsamt die Entwicklung.

Die Fluent API von AssertJ bietet in dieser Hinsicht einen deutlichen Vorteil: In einer modernen Entwicklungsumgebung wie IntelliJ unterbreitet die Software dem Entwickler Vorschläge für verwendbare Methoden, abhängig vom Programmkontext an der Cursorposition (Introduction to AssertJ 2018).

Sobald der Punkt hinter die Assertion gesetzt wird, klappt eine Liste aller verfügbaren Methoden auf dem Assertion-Objekt auf. Da das Assertion-Objekt vom Datentyp des übergebenen Werts abhängt, handelt es sich nicht um eine Liste aller Methoden der Bibliothek, sondern nur von Methoden, die vom Typ her zum aktuellen Kontext passen.

Tippt der Entwickler dann einige Zeichen, so wird die Liste weiter gefiltert. Daher reicht eine grobe Ahnung des gewünschten Methodennamens aus, um die passende Assertion zu finden (Abbildung 23). Im vorliegenden Beispiel sucht der Entwickler nach Assertion-Methoden im Zusammenhang mit der Größe der Collection, und stößt auf `hasSize` und `hasSameSizeAs`.

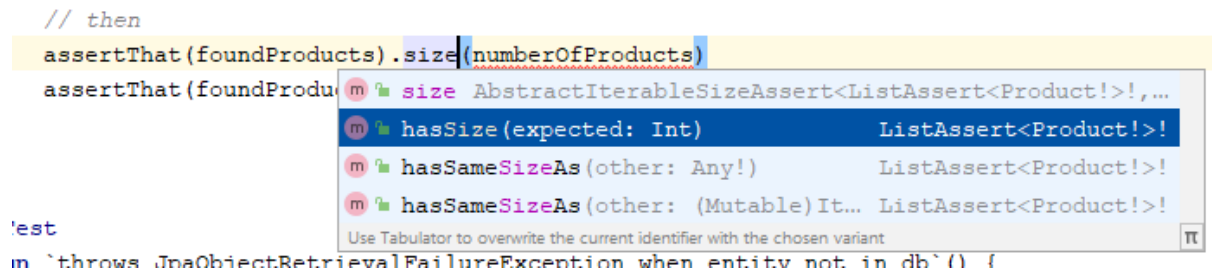


Abbildung 23: IntelliJ schlägt passende AssertJ-Methoden vor

Diese Hilfe kann die Entwicklungsumgebung bei Hamcrest-Matchern nicht bieten (Pinnegar), da die statischen Matcher-Methoden über zahlreiche Klassen der Bibliothek verteilt sind und nicht auf einem Objekt im Kontext aufgerufen werden.

4.6 Verfügbare Assertions

Da beide Bibliotheken einen ähnlichen Umfang an Matchern bzw. Assertion-Methoden anbieten, ist es überflüssig im Rahmen des Vergleichs sämtliche Bestandteile der Bibliotheken aufzulisten. Es soll reichen zu erwähnen, dass beide Bibliotheken neben einfachen Assertions zahlreiche Arten von Assertions mitbringen, z.B. Assertions für Listen, Klassen, Zahlen, Datumsobjekte, Nullwerte und Strings (Baeldung SRL. 2018). Ebenso bringen beide Bibliotheken die Umkehrungen all ihrer Assertions mit sowie die Möglichkeit, die Fehlerausgabe durch eigene Nachrichten zu überschreiben (Bushnev 2017) sowie eigene Matcher bzw. Assertion-Methoden zu schreiben, sollte die verwendete Bibliothek einen Spezialfall nicht abdecken.

5. Auswertung

Die vorangegangenen Beispiele haben Hamcrest und AssertJ unter den in Kapitel 2 entwickelten Gesichtspunkten gegenübergestellt.

In Sachen Lesbarkeit unterscheiden sich die beiden Bibliotheken nicht nennenswert. Beiden gelingt es, Assertions als flüssig lesbaren Satz zu formulieren, und die Namen der Matcher bzw. Assertionmethoden beschreiben deutlich ihren Zweck.

Die Fehlnachrichten von AssertJ sind aufgrund der Zeilenumbrüche besser strukturiert. Auch die Fehlnachrichten von Hamcrest enthalten für komplexe Vergleiche wesentlich bessere Informationen als JUnit selbst ermöglicht, aber AssertJ hat einen leichten Vorsprung.

Die Lernkurve von Hamcrest ist verhältnismäßig steil. Der Nutzer muss sich an die zahlreichen Matcher erinnern oder regelmäßig nachlesen, wie der passende Matcher für seine Absicht heißt. AssertJ besitzt den Vorteil der Fluent API, durch die eine ausgereifte Entwicklungsumgebung dem Entwickler die passende Assertion anbietet (Costigliola). Der Nutzer muss über AssertJ weniger im Kopf behalten und nachlesen, das Schreiben der Assertions fühlt sich flüssig und schnell an (AssertJ vs Hamcrest 2017). Dieser Aspekt ist der Hauptgrund, AssertJ anstelle von Hamcrest zu empfehlen (Phillip 2015) (Bushnev 2017).

```

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@ActiveProfiles("test")
@RunWith(SpringJUnit4ClassRunner::class)
class ProductE2ESpringBootTest {

    @Autowired
    lateinit var client: TestRestTemplate

    @Autowired
    lateinit var repository: ProductRepository

    @MockBean
    lateinit var authService: AuthService

    @MockBean
    lateinit var auditLoggerClient: AuditLoggerClient

    @Before
    @After
    fun cleanupDb() {
        repository.deleteAll()
    }

    @Test
    fun `create new product - success - return 201 CREATED`() {
        //given
        given(authService.isProductAdmin()).willReturn(value: true)
        val createProductRequest = CreateProductRequest(
            productName = "ROG Strix Z390-F Gaming Mainboard",
            manufacturer = "ASUS"
        )

        // when
        val response : ResponseEntity<Void!>! = client.postForEntity(
            url: "/api/v1/products",
            createProductRequest,
            Void::class.java
        )

        //then
        assertThat(response.statusCode).isEqualTo(HttpStatus.CREATED)
        assertThat(response.body).isNull()

        assertThat(repository.findByProductName(createProductRequest.productName)).hasSize(1)

        verify(auditLoggerClient).logProductCreated(createProductRequest)
    }
}

```

Abbildung 24: SpringBootTest mit TestRestTemplate, Mockito, DI-Container

Wie die Klasse ProductE2ESpringBootTest aus dem Beispielprojekt zeigt (Abbildung 24), spielen bei Tests allerdings weit mehr Faktoren als die verwendete Assertion-Bibliothek eine Rolle:

Der hier abgebildete Test lässt die Spring-Anwendung im Ganzen starten (SpringJUnit4ClassRunner) und holt über die Spring-Dependency-Injection Referenzen auf die in der Anwendung genutzten Instanzen der Komponenten (@Autowired) in die Testklasse. Das TestRestTemplate ermöglicht, Http-Requests aus dem Test an die auf zufälligem Port gestartete Anwendung zu schicken. Auf diese Weise wird hier das Verhalten der Anwendung für ein Szenario von REST-Schnittstelle bis zur Datenbank auf einmal getestet („End-to-End“). Aufrufe von Drittsystemen, die von der Anwendung im Produktivbetrieb gesendet werden, verhindert der Test, indem die betroffenen Client-Klassen in der Anwendung für den Test durch MockBeans ersetzt werden (@MockBean). Mit der Mockito-Bibliothek⁷ kann dann verifiziert werden, dass die MockBean auditLoggerClient während des Tests in der erwarteten Weise aufgerufen wurde.

Dieses Beispiel demonstriert, welche Technologieentscheidungen (z.B. Kotlin oder Java, Spring oder JavaEE, JUnit5 oder JUnit4 oder TestNG, Mockito oder EasyMock oder JMockit, Testcontainer oder Datenbankserver) außer der Assertion-Bibliothek ebenfalls Einfluss auf Kosten und Nutzen von Tests haben. Außerdem steht eine Reihe weiterer interessanter Testing-Technologien zur Verfügung, wie Pact-Tests⁸, Postman-Tests⁹, Selenium¹⁰ oder Cucumber¹¹, deren Einsatz aus Kosten-/Nutzensicht abgewogen werden will.

Dabei hat die verwendete Technologie nicht einmal den größten Einfluss auf die Testing-Praxis eines Softwareprojekts. Zu allererst kommt es auf die Erfahrung der Entwickler im Schreiben von Tests an, um hochwertige und wartbare Tests zu produzieren.

Die intrinsische Motivation der Entwickler spielt ebenfalls eine wichtige Rolle, denn Vorgaben wie „mindestens 80% Branch-Coverage“ können zwar vom Projektmanagement oder der Qualitätssicherung leicht ausgesprochen werden; ohne eigenes Bewusstsein der Entwickler für den Wert guter Tests führt dies aber nur dazu, dass viele Testfälle produziert werden, die aber den Produktivcode nur aufrufen um messbare Testabdeckung zu produzieren, ohne das beobachtete Verhalten sinnvoll mit erwartetem Verhalten zu vergleichen.

Um jungen Entwicklern Routine und Motivation für das Schreiben von Tests überhaupt zu vermitteln, muss zudem in der Organisation bereits eine Kultur bestehen, in der Tests als wertvolles Werkzeug und nicht als lästige Pflicht gelten.

In diesem Kontext ist offensichtlich, dass die Auswahl einer von zwei sehr ähnlichen Assertion-Bibliotheken kaum eine Rolle spielt. Wie in der vorliegenden Arbeit gezeigt wurde, ist AssertJ für ein neu beginnendes Projekt offenbar die bessere Wahl. Für eine bestehende Codebasis mit Hamcrest lohnt es sich dagegen nicht, auf AssertJ umzustellen, da die Kosten der Migration von den marginalen Vorzügen von AssertJ nicht aufgewogen werden.

⁷ <https://site.mockito.org/>

⁸ Ein Ansatz zur Integration von Client und Server über ausführbare Contracts. <https://docs.pact.io/>

⁹ Postman ist ein Http-Client zum manuellen Versenden von Requests an Server, der Entwickler kann aber auch das erwartete Ergebnis eines Requests definieren und dann eine Collection von Requests als Testsuite ausführen. <https://www.getpostman.com/>

¹⁰ Ein Testframework für Web-Oberflächen. Der Entwickler kann mit der Selenium-Bibliothek JUnit-Tests schreiben, die Nutzeraktionen im Browser emulieren und das Ergebnis überprüfen. <https://www.seleniumhq.org/>

¹¹ In Cucumber können Nicht-Techniker Akzeptanztests auf hoher Ebene definieren, die dann mit Skripts unterfüttert und ausführbar gemacht werden. <https://cucumber.io/>

Literaturverzeichnis

Ammann, Paul; Offutt, Jeff (2008): Introduction to Software Testing. New York: Cambridge University Press.

AssertJ vs Hamcrest (2017). Online verfügbar unter <http://randomthoughtsonjavaprogramming.blogspot.com/2017/03/assertj-vs-hamcrest.html>, zuletzt aktualisiert am 23.03.2017, zuletzt geprüft am 28.04.2019.

Baeldung SRL. (2018): Testing with Hamcrest. Online verfügbar unter <https://www.baeldung.com/java-junit-hamcrest-guide>, zuletzt aktualisiert am 30.04.2018, zuletzt geprüft am 28.04.2019.

Bechtold, Stefan; Brannen, Sam; Link, Johannes; Merdes, Matthias; Phillip, Marc; Stein, Christian: JUnit 5 User Guide. 3.4.1. Third-party Assertion Libraries. Online verfügbar unter <https://junit.org/junit5/docs/5.0.0/user-guide/#writing-tests-assertions-third-party>, zuletzt geprüft am 28.04.2019.

Beck, Kent (2002): Test-Driven Development. By Example. Boston: Addison-Wesley.

Bushnev, Yuri (2017): Hamcrest vs AssertJ Assertion Frameworks - Which One Should You Choose? Online verfügbar unter <https://www.blazemeter.com/blog/hamcrest-vs-assertj-assertion-frameworks-which-one-should-you-choose>, zuletzt aktualisiert am 19.09.2017, zuletzt geprüft am 28.04.2019.

Buxton, J. N.; Randell, B. (Hg.) (1970): Software Engineering Techniques. Report on a conference sponsored by the NATO science committee, Rome, 27.10.-31.10.1969. Online verfügbar unter <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF>, zuletzt geprüft am 01.05.2019.

Costigliola, Joel: AssertJ. Fluent assertions for java. Online verfügbar unter <http://joel-costigliola.github.io/assertj/>, zuletzt geprüft am 28.04.2019.

Engel, Cody (2018): Seven Principles of Great Unit Tests — Adapted For Android. Online verfügbar unter <https://proandroiddev.com/seven-principles-of-great-unit-tests-adapted-for-android-342515f98ef2>, zuletzt aktualisiert am 23.07.2018, zuletzt geprüft am 30.04.2019.

Fowler, Martin (2005): Fluent Interface. Online verfügbar unter <https://www.martinfowler.com/bliki/FluentInterface.html>, zuletzt aktualisiert am 20.12.2005, zuletzt geprüft am 13.05.2019.

Fowler, Martin (2013): Component Test. Online verfügbar unter <https://martinfowler.com/bliki/ComponentTest.html>, zuletzt aktualisiert am 22.04.2013, zuletzt geprüft am 02.05.2019.

Fowler, Martin (2014): Unit Test. Online verfügbar unter <https://martinfowler.com/bliki/UnitTest.html>, zuletzt aktualisiert am 05.05.2014, zuletzt geprüft am 01.05.2019.

Google Trends (2019): Hamcrest, AssertJ. Online verfügbar unter <https://trends.google.de/trends/explore?date=all&q=hamcrest,assertj>, zuletzt aktualisiert am 13.05.2019, zuletzt geprüft am 13.05.2019.

Hauer, Philipp (2019): Focus on Integration Tests Instead of Mock-Based Tests. Online verfügbar unter <https://phauer.com/2019/focus-integration-tests-mock-based-tests/>, zuletzt aktualisiert am 02.04.2019, zuletzt geprüft am 01.05.2019.

Introduction to AssertJ (2018). Online verfügbar unter <https://www.baeldung.com/introduction-to-assertj>, zuletzt aktualisiert am 04.06.2018, zuletzt geprüft am 30.04.2019.

Langr, Jeff (2015): Pragmatic Unit Tetsting. in Java 8 with JUnit. Unter Mitarbeit von Andy Hunt und Dave Thomas. Dallas, TX: The Pragmatic Programmers.

Martin, Robert C. (2009): Clean Code. Refactoring, Patterns, Testen und Techniken für sauberen Code. 1. Aufl.: mitp Verlagsd GmbH & Co. KG.

Massol, Vincent (2004): JUnit in Action. Unter Mitarbeit von Ted Husted. Greenwich, CT: Manning Publications Co.

Mishra, Pankhuri; Tripathi, Neeraj (2017): Tetsting as a Service. In: Hrushiksha Mohanty, J.R. Mohanty und Arunkumar Balakrishnan (Hg.): Trends in Software Testing. Singapore: Springer Nature, S. 149–176.

Moiz, Salman Abdul (2017): Uncertainty in Software Testing. In: Hrushiksha Mohanty, J.R. Mohanty und Arunkumar Balakrishnan (Hg.): Trends in Software Testing. Singapore: Springer Nature, S. 67–88.

Phillip, Marc (2015): Fluent Assertions. Ein Vergleich von Hamcrest, AssertJ und Truth. In: *JAVASpektrum* (6), S. 22–26. Online verfügbar unter https://www.sigs-datacom.de/uploads/tx_dmjournals/philipp_JS_06_15_gRfN.pdf, zuletzt geprüft am 28.04.2019.

Pinnegar, Michael: AssertJ – The only Java assertion framework you need. Online verfügbar unter <https://professional-practical-programmer.com/2016/06/26/assert-j/>, zuletzt geprüft am 28.04.2019.

Qusef, A; Bavota, G; Oliveto, R; Lucia, A; Binkley, D (2011): Scotch: test-to-code traceability using slicing and conceptual coupling. In: International Conference on Software Maintenance (27th : 2011 : Williamsburg, Va.) and Institute of Electrical and Electronics Engineer (Hg.): ICSM 2011 : proceedings of the 27th IEEE International Conference on Software Maintenance : Williamsburg, VA, USA : September 25-30, 2011 / [IEEE]. Proceedings of the 27th IEEE International Conference on Software Maintenance Software Maintenance (ICSM), 2011 27th IEEE International Conference on 2011 27th IEEE International Conference on Software Maintenance (ICSM) Twenty-Seventh IEEE International Conference on Software Maintenance. Piscataway, N.J., S. 63–72.

Samarthyam, Ganesh; Muralidharan, Manesh; Anna, Raghu Kalyan (2017): Understanding Test Debt. In: Hrushiksha Mohanty, J.R. Mohanty und Arunkumar Balakrishnan (Hg.): Trends in Software Testing. Singapore: Springer Nature, S. 1–18.

Vance, Stephen (2013): Quality Code. Software Testing Principles, Practices and Patterns. New Jersey: Addison-Wesley.

Vogel, Lars (2016): Using Hamcrest for testing - Tutorial. Online verfügbar unter <https://www.vogella.com/tutorials/Hamcrest/article.html>, zuletzt aktualisiert am 29.05.2016, zuletzt geprüft am 30.04.2019.