# Einleitung

Hamcrest und AssertJ sind Klassenbibliotheken, die zur Verbesserung der Lesbarkeit und Ausdrucksstärke von Testklassen verwendet werden können. Bei beiden handelt es sich nicht um eigenständige Testing Frameworks, sondern sie stellen Ergänzungen zu JUnit und TestNG dar. Die Bibliotheken sind mit allen JVM-Sprachen kompatibel, die JUnit unterstützen, darunter Java, Kotlin, Scala und Groovy.

## 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 kleine 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 gleiches 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; daraus folgt, dass Hamcrest und AssertJ nicht die Mächtigkeit der JUnit-Assertions erhöhen, sondern nur die Usability, wie im Verlauf der vorliegenden Arbeit gezeigt werden soll.

## Hamcrest

Die Hamcrest-Bibliothek ist seit 2007 im öffentlichen Maven Repository [https://mvnrepository.com/artifact/org.hamcrest/hamcrest-core] verfügbar. Sie wurde 2012 als Bestandteil in JUnit 4.4 aufgenommen; JUnit 5 enthält Hamcrest nicht mehr, weist aber auf die Möglichkeit der Verwendung von "Third-party Assertion Libraries such as AssertJ, Hamcrest, Truth, etc." [https://junit.org/junit5/docs/5.0.0/user-guide/#writing-tests-assertions-third-party] ausdrücklich hin.

Der Quellcode vom Harmcrest wurde 2012 auf GitHub veröffentlicht [https://github.com/hamcrest/JavaHamcrest] und wurde seitdem lange nicht mehr gewartet. Erst 2018 wurde die Wartung des Projekts wiederaufgenommen und mit einigen Bugfixes und technischen Neuerungen als Version 2.0 released.

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

## AssertJ

AssertJ wurde 2013 auf GitHub und Maven Repository veröffentlicht und wurde seitdem durchgehend gewartet und monatlich released [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 Maven, Ivy und Gradle 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 [https://trends.google.de/trends/explore?date=all&q=hamcrest,assertj]. (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. Dafür gibt der Aufruf von assertThat ein Assertion-Object 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 "Fluent Interface" [https://www.martinfowler.com/bliki/FluentInterface.html] ist in moderneren Frameworks weit verbreitet und soll eine Domain Specific Language bilden, die intuitiv zu lesen und zu schreiben ist. Die Vor- und Nachteile dieses Stils werden in einem folgenden Kapitel bewertet.)

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.

## 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 gemeinsames Thema gewählt. Das Projekt ist auf GitHub verfügbar und ist ausführbar, enthält aber nur Beispielcode und keine fertige Shop-Software: <https://github.com/tbelmega/pcshop>

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

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.

## 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 nennt man dieses Programm einen automatisierten Test. Automatisierte Tests müssen von einer Person programmiert und nach jedem Durchlauf ebenfalls von einer Person (zusammengefasst) 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 Test

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

### Integration Test, Component Test

Ein Integrationstest ursprünglichen Sinne ist ein Test, der die Kommunikation 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 bereits anderweitig belegt ist.

### End-to-End Test, System Test, Functional Test, Use-Case-Test

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, wenn eine technisch getrennte Frontend-Anwendung existiert, die aus Nutzersicht Teil des Systems ist.

Die oben genannten Formen von Tests seien deshalb erwähnt, weil jede davon unter Verwendung von JUnit (und somit Hamcrest / AssertJ) implementiert werden kann. Die feingranularen Tests erfordern oft den Einsatz von Mocking-Frameworks wie Mockito zur Isolation des zu testenden Codes, die grobgranularen Formen nutzen zusätzliche Testbibliotheken wie Selenium für Tests über das Web-GUI (Graphical User Interface) oder das Spring TestRestTemplate für http-APIs. Tests über http-API oder das Web-GUI können aber auch mit anderen Mitteln als JUnit implementiert werden, zum Beispiel als Postman-Testsuite oder durch Bildvergleich-Werkzeuge.

Im Rahmen dieser Arbeit wird der Einfachheit halber von Unit Tests gesprochen, die meisten Aussagen lassen sich aber auf alle Formen des automatisierten Testens übertragen, falls nicht anders erläutert.

# Software Testing

Um die beiden vorgestellten Bibliotheken im wissenschaftlichen Sinne vergleichen zu können, ist es 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.

Eine vielzitierte Aussage über Software Testing stammt von Edsger W. Dijkstra auf der NATO-Konferenz zu Software Engineering 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 wahr und im Kontext der zitierten Diskussion über formale Programmkorrekheit 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.

Das offensichtliche Ziel 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 bezeichnet man das Ausführen von Test, 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 dazu 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. 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. Oder andersherum betrachtet: Mit Hilfe der Tests kann der Programmierer die Anzahl auftretender Fehler weit genug reduzieren, um mit Zuversicht weiter zu entwickeln (Beck 2002, S. 20).

Dieser Aspekt des Testens mutet wenig wissenschaftlich an, da er auf ein Gefühl des Entwicklers gegenüber seinem Programm abzielt anstelle der 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 Continous 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 Continous Deployment / Continous 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 das große Ganze eines Softwaresystems anschaulich beschreiben. Je detaillierter man aber versucht die Implementierung zu beschreiben, desto weniger hilfreich sind diese klassische Formen der Dokumentation, weil natürliche Sprache zur Ungenauigkeit neigt, der Autor der Dokumentation Teile missversteht oder 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.

Literaturverzeichnis

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

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

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.

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.

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.

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

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: Hrushikesha 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: Hrushikesha Mohanty, J.R. Mohanty und Arunkumar Balakrishnan (Hg.): Trends in Software Testing. Singapore: Springer Nature, S. 67–88.